The Public Csound Reference Manual

CANONICAL VERSION 4.10

by Barry Vercoe, Media Lab MIT & contributors

Edited by John ffitch, Richard Boulanger, Jean Piché, & David Boothe

Copyright Notice

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

The original Hypertext Edition of the MIT Csound Manual was prepared for the World Wide Web by **Peter J. Nix** of the Department of Music at the University of Leeds and **Jean Piché** of the Faculté de musique de l'Université de Montréal. This Print Edition, in Adobe Acrobat format, and the current HTML Edition, are maintained by **David M. Boothe** of Lakewood Sound. The editors fully acknowledge the rights of the authors of the original documentation and programs, as set out above, and further request that this notice appear wherever this material is held.

Contributors

In addition to the core code developed by Barry Vercoe at MIT, a large part of the Csound code was modified, developed and extended by an independent group of programmers, composers and scientists. Copyright to this code is held by the respective authors:

Mike Berry
Eli Breder
Michael Casey
Michael Clark
Perry Cook
Sean Costello
Richard Dobson
Mark Dolson
Rasmus Ekman
Dan Ellis
Tom Erbe
John ffitch
Bill Gardner
Matt Ingalls
Richard Karpen
Victor Lazzarini
Allan Lee
David Macintyre
Gabriel Maldonado
Max Mathews
Hans Mikelson
Peter Neubäcker
Ville Pulkki
Marc Resibois
Rob Shaw
Paris Smaragdis
Greg Sullivan
Bill Verplank
Robin Whittle

This manual was compiled from the canonical Csound Manual sources maintained by John ffitch, Richard Boulanger, Jean Piché and David Boothe.

The Acrobat Edition of this manual was redesigned for the Csound version 4.10 release, in February & March, 2001. It is set in 10 pt. Trebuchet, from Microsoft Corporation. Headings are set Antique Olive from Hewlett-Packard Corporation. Syntax and code examples are set in Andale Mono from Monotype Corporation.

This page intentionally left blank.

Table of Contents

Finder

Symbols

Tags, Files and Extensions

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

This page intentionally left blank.

1 PREFACE

by Barry Vercoe, MIT Media Lab

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

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

The instruments in a Csound orchestra (**.orc**) are defined in a simple syntax that invokes complex audio processing routines. A score (**.sco**) passed to this orchestra contains numerically coded pitch and control information, in standard numeric score format. Although many users are content with this format, higher level score processing languages are often convenient.

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

Because it is written entirely in C, Csound is easily installed on any machine running Unix or C. At MIT it runs on VAX/DECstations under Ultrix 4.2, on SUNs under OS 4.1, SGI's under 5.0, on IBM PC's under DOS 6.2 and Windows 3.1, and on the Apple Macintosh under ThinkC 5.0. With this single language for defining the audio signal processing, and portable audio formats like AIFF and WAV, users can move easily from machine to machine.

The 1991 version added phase vocoder, FOF, and spectral data types. 1992 saw MIDI converter and control units, enabling Csound to be run from MIDI score-files and external keyboards. In 1994 the sound analysis programs (lpc, pvoc) were integrated into the main load module, enabling all Csound processing to be run from a single executable, and Cscore could pass scores

directly to the orchestra for iterative performance. The 1995 release introduced an expanded MIDI set with MIDI-based linseg, butterworth filters, granular synthesis, and an improved spectral-based pitch tracker. Of special importance was the addition of run-time event generating tools (Cscore and MIDI) allowing run-time sensing and response setups that enable interactive composition and experiment. It appeared that real-time software synthesis was now showing some real promise.

1.1 Where to Get Public Csound and the Csound Manual

Public Csound is available for download via anonymous ftp from :

- ftp://ftp.maths.bath.ac.uk/pub/dream

or

- ftp://ftp.musique.umontreal.ca/pub/mirrors/dream

The Acrobat Edition and HTML Edition of this manual is available for browser download from:

- http://www.lakewoodsound.com/csound

or via anonymous ftp from:

- ftp://ftp.csounds.com/manual

1.2 How to Install Csound

MACINTOSH

Detailed instructions for installing and configuring Csound on Macintosh systems may be obtained from:

- http://mitpress.mit.edu/e-books/csound/fpage/gs/mac/a/a.html

WINDOWS 95/98

Detailed instructions for installing and configuring Csound on Windows 95 or Windows 98 systems may be obtained from:

- http://mitpress.mit.edu/e-books/csound/fpage/gs/pc/pc.html

MS-DOS AND WINDOWS 3.X

Detailed instructions for installing and configuring Csound on MS-DOS or Windows 3.x systems may be obtained from:

- http://hem.passagen.se/rasmuse/PCinstal.htm

LINUX (DEVELOPERS' VERSION)

Introduction to the Developers' Linux Version

Building Csound for UNIX and Linux machines has been possible thanks to John Fitch's Csound.tar.gz source file kept at:

- ftp://ftp.maths.bath.ac.uk/pub/dream/newest

This source tree builds Csound on a variety of UNIX-type systems, including the NeXT, Sun's Solaris, SGI machines, and Intel-based Linux. It should be noted that John also maintains a Linux binary at the Bath repository. That version is built from his canonical sources.

In 1998 a group of developers prepared a new version of Csound for Linux. This version (often referred to as the "unofficial" distribution) aims to deliver a modern package for Linux users. It offers a variety of amenities specific to Linux systems, including these items:

- Enhanced makefile system
- 'autoconf' and 'configure' supported for site-specific build
- Support for Jaroslav Kysela's ALSA sound drivers
- Support for 64-bit Alpha systems
- Full MIDI and real-time audio support
- Builds shared library (libcsound.so) for greatly reduced memory footprint
- Includes Robin Whittle's random number generator
- Provided in various popular Linux distribution packaging formats
- Utilizes .csoundrc resource file
- Provides a high-priority scheduler for improved real-time i/o
- Includes support for full-duplex under the OSS/Free and OSS/Linux drivers
- CVS and bug-tracking system established for developers

This distribution's code base originates with the sources provided by John Fitch at the Bath site. Every effort is made to ensure compatibility with those sources at the opcode level, and users should have no trouble running most orc/sco files or .csd files made for Csound on other operating systems.

The makefile structure has been provided by Nicola Bernardini. He also maintains the CVS repository. Other features have been added by developers Ed Hall (Alpha port), Fred Floberg (scheduler), Robin Whittle, and Steve Kersten (full-duplex under OSS driver). RPM and DEB packages are sporadically available from Damien Miller and Guenter Geiger.

Building the developers' version is quite simple, using the familiar './configure; make depend; make; make install' command sequence. Instructions for compiling and installing Csound are provided with the package, along with other relevant documentation. A mail-list has been established for developers and users of this package, and a bug-tracking system has been set up by Damien Miller.

Preparing Linux Audio for Csound

As long as the basic Linux audio system is properly configured and installed, no special efforts need to be made in order to enjoy audio output from sound. The default real-time audio output device (devaudio) is defined as /dev/dsp in Csound itself, although other audio devices (/dev/audio, /dev/dspW) can be specified if so desired.

Using the Developers' Version

This version is designed to be opcode-compatible with any other version of Csound. However, some new options have been added which may require clarification.

Real-time audio output can be as simple as this:

```
csound -o devaudio -V 75 my.orc my.sco
```

The '-V' flag is a Linux-specific output volume control from Jonathan Mohr. Note that it will work only with the OSS/Free and OSS/Linux drivers.

Here we get a little more complicated:

```
csound --sched --ossin=/dev/dsp0 --ossout=/dev/dsp1 my.*
```

This example invokes Fred Floberg's high-priority scheduler (which will automatically disable graphics output) and Steve Kersten's support for full-duplex using either the OSS/Free driver included with the Linux kernel or the commercially available OSS/Linux driver. Linux users can use the asterisk as a wildcard for the orc/sco extensions. However, if you have my.orc, my.sco, and my.txt within the same directory the compiler will get confused and the wildcard won't work.

If more than one soundcard is present in the system, ALSA users have the option of choosing which card will function for either audio input or output. The command sequence then appears so:

```
csound --incard=1 --outcard=2 my.orc your.sco
```

The standard advice regarding audio buffer settings holds true for Linux as well as for any other version. If the audio output is choppy you may need to adjust the value for the '–b' flag which controls the sample frame size for the software audio buffer. The best setting will depend upon various aspects of your machine system, including CPU speed, memory limits, hard-disk performance, etc.

Supported options for MIDI include the '-Q' (MIDI output device) and '-K' (MIDI input port) flags from Gabriel Maldonado's DirectCsound. Here is an example which uses one of Gabriel's opcodes, requiring the use of a MIDI output port:

```
csound -Q0 -n my_moscil.orc my_moscil.sco
```

The '-Q0' flag selects the first available MIDI output device, '-n' cancels writing the output to disk.

It should be noted that, for Linux at least, in the opcode for this instrument (**moscil**) the sample rate determines the tempo of events. Setting the control rate (**kr**) to equal the sample rate (**sr**), **sr=kr** a higher sample rate will result in a slower performance. When **sr**=390000 (yes, you read that correctly, it's a sample rate of three hundred and ninety thousand) then the MIDI event performance output is approximately 60 BPM (beats per minute). At that sample rate a score tempo statement of 't 0 60'will actually mean 60 bpm. In essence, the sample rate acts as a restraint or throttle on the tempo of the MIDI event stream.

Using MIDI for real-time input is simple:

```
csound --sched -o devaudio -M/dev/midi my_midi_in.*
```

With correctly written orc/sco files this example will allow real-time control of Csound via whatever controlling device is hooked up to /dev/midi. If more than one MIDI device is present in the machine the user can specify which to use:

```
csound --sched -o devaudio -M/dev/midi01 my_midi_in.*
```

That sequence will select the second MIDI device for MIDI input.

Here we use a Type 0 standard MIDI file for the controlling input:

```
csound --sched -o devaudio -T -F/home/midfiles/my_type_0.mid
            my_cool.orc my_cool.sco
```

In these last two examples the score file provides only the necessary function tables and a place-holder to indicate how long Csound should stay active:

```
f1 0 8192 10 1   ; a sine wave
f0 240           ; stay active for 240 seconds
e
```

However, the '-T' flag will halt performance as soon as the end of the MIDI file is reached.

Availability

The Linux developers version of Csound is available in source and binary distributions. The main distribution sites are at AIMI in Italy:

- http://AIMI.dist.unige.it/AIMICSOUND/AIMICSOUND_home.html

and the ftp server for the Music Technology Department at Bowling Green State University in the USA:

- ftp://mustec.bgsu.edu/pub/linux

Developer Maurizio Umberto Puxeddu has also established a distribution point, though at this time it is version-specific and not browsable. For more information regarding his site, and for more information generally regarding Linux Csound, see this Web page:

- http://www.bright.net/~dlphilp/linux_csound.html

Credits

First thanks go to Barry Vercoe for creating Csound and allowing it to be freely and publicly distributed and to John Fitch for maintaining the canonical source packages (including his own build for Linux).

Special thanks go to the following persons for their development assistance and/or spiritual guidance:

- Paul Barton-Davis
- Nicola Bernardini
- Richard Boulanger
- Fred Floberg
- Ed Hall
- Steve Kersten

- Gabriel Maldonado
- Damien Miller
- Maurizio Umberto Puxeddu
- Larry Troxler
- Robin Whittle

My apologies to anyone I've left out. Please send corrections and emendations of this document to me at my email address below.

Dave Phillips
dlphilp@bright.net
September 1999

OTHER PLATFORMS

For information on availability of Csound for other platforms, see The Csound Frontpage:

- http://mitpress.mit.edu/e-books/csound/frontpage.html

1.3 How to use the Csound Manual

The Csound Manual is arranged as a *Reference* manual (not a tutorial), since that is the form the user will eventually find most helpful when inventing instruments. Csound can be a demanding experience at first. Hence it is highly advisable to peruse the tutorials included in the Supplement to this Manual. Once the basic concepts are grasped from the beginning tutorial, the reader might let himself into the remainder of the text by locating the information presented in the Reference entries that follow.

1.4 The Csound Mailing List

A Csound Mailing List exists to discuss Csound. It is run by John ffitch of Bath University, UK.

To have your name put on the mailing list send an empty message to:
`csound-subscribe@lists.bath.ac.uk`

Posts sent to
`csound@lists.bath.ac.uk`
go to *all* subscribed members of the list.

Bug Reports

Suspected bugs in the code may be submitted to the list.

This page intentionally left blank

2 SYNTAX OF THE ORCHESTRA

An orchestra statement in Csound has the format:

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

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

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

The remainder (result, opcode, and arguments) form the *basic statement*. This also is optional, i.e. a line may have only a label or comment or be entirely blank. If present, the basic statement must be complete on one line, and is terminated by a carriage return and line feed. Occasionally in this manual, a statement is divided between two lines. This is for printing convenience only, and does not apply to the HTML Edition. In orchestra files, statements must be complete on one line, without a carriage return or line feed before the end of the statement.

The opcode determines the operation to be performed; it usually takes some number of input values (or arguments, with a maximum value of about 800); and it usually has a result field variable to which it sends output values at some fixed rate. There are four possible rates:

- once only, at orchestra setup time (effectively a permanent assignment);
- once at the beginning of each note (at initialization (init) time: *i-rate*);
- once every performance-time control loop (perf-time control rate, or *k-rate*);
- once each sound sample of every control loop (perf-time audio rate, or *a-rate*).

2.1 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 a full path, 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, sound analysis SADIR, and include files for orchestra and score files INCDIR.

The search order is:

1. Soundfiles being written are placed in SFDIR (if it exists), else the current directory.
2. Soundfiles for reading are sought in the current directory, then SSDIR, then SFDIR.
3. Analysis control files for reading are sought in the current directory, then SADIR.
4. Files of code to be included in orchestra and score files (with **#include**) are sought first in the current directory, then in the same directory as the orchestra or score file (as appropriate), then finally INCDIR.

Beginning with Csound version 3.54, the file "csound.txt" contains the messages (in binary format) that Csound uses to provide information to the user during performance. This allows for the messages to be in any language, although the default is English. This file must be placed in the same directory as the Csound executable. Alternatively, this file may be stored in SFDIR, SSDIR, or SADIR. Unix users may also keep this file in "usr/local/lib/". The environment variable CSSTRNGS may be used to define the directory in which the database resides. This can be overridden with the **-j** command line option. (New in version 3.55)

2.2 Nomenclature

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 usually denoted by 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. With few exceptions, argument rates may not exceed the rate of the result. 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.

Thoughout this manual, the term "opcode" is used to indicate a command that usually produces an a-, k-, or i-rate output, and always forms the basis of a complete Csound orchestra statement. Items such as "**+**" or "**sin**(x)" or, "(a **>=** b **?** c **:** d)" are called "operators."

In the Csound orchestra, statements fall into twelve major categores, consisting of sixty-five sub-categories. Each is in a separate chapter of this manual. The categories (and corresponding chapter numbers) are as follows:

Orchestra Syntax

3: Orchestra Header Statements

4: Instrument Block Statements

5: Variable Initialization

Instrument Control

6: Instrument Invocation

7: Duration Control Statements

8: Real-time Performance Control

9: Time Reading

10: Clock Control

11: Sensing and Control

12: Conditional Values

13: Macros

14: Program Flow Control

15: Reinitialization

Mathematical Operations

16: Arithmetic and Logic Operations

17: Mathematical Functions

18: Trigonometric Functions

19: Amplitude Functions

20: Random Functions

21: Opcode Equivalents of Functions

Pitch Converters

22: Functions

23: Tuning Opcodes

MIDI Support

24: Converters

25: Controller Input

26: Slider Banks

27: Generic I/O

28: Note-on/Note-off

29: MIDI Message Output

30: Real-time Messages

31: MIDI Event Extenders

Signal Generators

32: Linear and Exponential Generators

33: Table Access

34: Phasors

35: Basic Oscillators

36: Dynamic Spectrum Oscillators

37: Additive Synthesis/Resynthesis

38: FM Synthesis

39: Sample Playback

40: Granular Synthesis

41: Waveguide Physical Modeling

42: Models and Emulations

43: STFT Resynthesis (Vocoding)

44: LPC Resynthesis

45: Random (Noise) Generators

Function Table Control

46: Tables Queries

47: Table Selection

48: Read/Write Operations

Signal Modifiers

49: Standard Filters

50: Specialized Filters

51: Envelope Modifiers

52: Amplitude Modifiers

53: Signal Limiters

54: Delay

55: Reverberation

56: Waveguides

57: Special Effects

58: Convolution and Morphing

59: Panning and Spatialization

60: Sample Level Operators

Zak Patch System

61: Zak Patch System

Operations Using Spectral Data Types

62: Operations Using Spectral Data Types

Signal Input and Output

63: Input

64: Output

65: File I/O

66: Sound File Queries

67: Printing and Display

2.3 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, e.g. 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. 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 symbols, value converters, arithmetic operations, and conditional values.

2.4 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-rate variables) or at the control rate (for k-rate variables). i- and k-variables are therefore useful for storing note parameter values, pitches, durations, slow-moving frequencies, vibratos, etc. a-rate variables, on the other hand, are arrays or vectors of information. Though renewed on the same perf-time control pass as k-rate variables, these array cells represent a finer resolution of time by dividing the control period into sample periods (see **ksmps**). a-rate variables are used to store and recall data changing at the audio sampling rate (e.g. output signals of oscillators, filters, etc.).

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

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

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

Type	When Renewable	Local	Global
reserved symbols	permanent	--	**r**symbol
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-rate 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.

2.5 Expressions

Expressions may be composed to any depth. 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.

3 ORCHESTRA SYNTAX: ORCHESTRA HEADER STATEMENTS

3.1 sr, kr, ksmps, nchnls

```
sr      =       iarg
kr      =       iarg
ksmps   =       iarg
nchnls  =       iarg
```

DESCRIPTION

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 *iarg* samples per second per channel. The default value is 44100.

kr = (optional) – set control rate to *iarg* samples per second. The default value is 4410.

ksmps = (optional) – set the number of samples in a control period to. This value must equal **sr/kr**. The default value is 10.

nchnls = (optional) – set number of channels of audio output to *iarg*. (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.

Beginning with Csound version 3.46, either **sr**, **kr**, or **ksmps** may be omitted. Csound will attempt to calculate the omitted value from the specified values, but it should evaluate to an integer.

EXAMPLE

```
sr = 10000
kr = 500
ksmps = 20
gi1 = sr/2.
ga init 0
itranspose = octpch(.01)
```

3.2 strset, pset

```
strset      iarg, “stringtext”
pset        con1, con2, con3,...
```

DESCRIPTION

Allow certain global parameters to be initialized at orchestra load time, rather than instrument initialization or performance time.

INITIALIZATION

iarg – numeric value to be associated with an alphanumeric string

con1, con2, etc. – preset values for a MIDI instrument

strset (optional) allows a string, such as a filename, to be linked with a numeric value. Its use is optional.

pset (optional) defines and initializes numeric arrays at orchestra load time. It may be used as an orchestra header statement (i.e. instrument 0) or within an instrument. When defined within an instrument, it is not part of its i-time or performance operation, and only one statement is allowed per instrument. These values are available as i-time defaults. When an instrument is triggered from MIDI it only gets p1 and p2 from the event, and p3, p4, etc. will receive the actual preset values.

EXAMPLES

The following statement, used in the orchestra header, will allow the numeric value 10 to substituted anywhere the soundfile *asound.wav* is called for.

```
strset 10, “asound.wav”
```

The example below illustrates **pset** as used within an instrument.

```
          instr 1
          pset          0,0,3,4,5,6     ; pfield substitutes
a1        oscil         10000, 440, p6
```

3.3 seed

seed ival

DESCRIPTION

Sets the global seed value for all **x-class noise generators**, as well as other opcodes that use a random call, such as **grain**. **rand**, **randi**, **randh**, **rnd**(x), and **birnd**(x) are not affected by **seed**.

INITIALIZATION

ival – value to be used as the random generator(s) seed value

PERFORMANCE

Use of **seed** will provide predictable results from an orchestra using with random generators, when required from multiple performances.

When specifying a seed value, *ival* should be an integer between 0 and 2^{32}. If *ival* = 0, the value of *ival* will be derived from the system clock.

3.4 ftgen

```
gir      ftgen      ifn, itime, isize, igen, iarga[, iargb...iargz]
```

DESCRIPTION

Generate a score function table from within the orchestra.

INITIALIZATION

gir – either a requested or automatically assigned table number above 100. If used within an instrument, may be local variable *ir*.

ifn – requested table number If *ifn* is zero, the number is assigned automatically and the value placed in *gir*. Any other value is used as the table number

itime – is ignored, but otherwise corresponds to p2 in the score **f statement**.

isize – table size. Corresponds to p3 of the score **f statement**.

igen – function table **GEN** routine. Corresponds to p4 of the score **f statement**.

iarga-iargz – function table arguments. Correspond to p5 through p*n* of the score **f statement**.

PERFORMANCE

This is equivalent to table generation in the score with the **f statement**.

AUTHOR

Barry Vercoe
MIT, Cambridge, Mass
1997

3.5 massign, ctrlinit

```
massign     ichnl, insnum
ctrlinit    ichnl, ictlno1, ival1[, ictlno2, ival2 [, ictlno3,
            ival3[,..ival32]]
```

DESCRIPTION

Initialize MIDI controllers for a Csound orchestra.

INITIALIZATION

ichnl – MIDI channel number

insnum – Csound orchestra instrument number

ictlno1, *ictlno2*, etc. – MIDI contoller numbers

ival1, *ival2*, etc. – initial value for corresponding MIDI contoller number

PERFORMANCE

massign assigns a MIDI channel number to a Csound instrument

ctrlinit sets initial values for a set of MIDI controllers.

AUTHORS

Barry Vercoe – Mike Berry
MIT, Cambridge, Mass
New in Csound version 3.47

This page intentionally left blank.

4 ORCHESTRA SYNTAX: INSTRUMENT BLOCK STATEMENTS

4.1 instr, endin

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

DESCRIPTION

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

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

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

endin – end the current instrument block.

Note:

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

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

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

This page intentionally left blank.

5 ORCHESTRA SYNTAX: VARIABLE INITIALIZATION

5.1 = , init, tival, divz

```
ir      =           iarg
kr      =           karg
ar      =           xarg
kr      init        iarg
ar      init        iarg
ir      tival
ir      divz        ia, ib, isubst
kr      divz        ka, kb, ksubst
ar      divz        xa, xb, ksubst
```

DESCRIPTION

= (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-rate variable, i.e., initialize the result. Note that **init** provides the only case of an init-time statement being permitted to write into a perf-time (k- or a-rate) result cell; the statement has no effect at perf-time.

tival – Put the value of the instrument's internal "tie-in" flag into the named i-rate 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.

This page intentionally left blank.

6 INSTRUMENT CONTROL: INSTRUMENT INVOCATION

6.1 schedule, schedwhen

schedule insnum, iwhen, idur [, p4, p5,…]
schedwhen ktrigger, kinst, kwhen, kdur [, p4, p5,…]

DESCRIPTION

Adds a new score event

INITIALIZATION

insnum – instrument number. Equivalent to p1 in a score **i statement**.

iwhen – start time of the new event. Equivalent to p2 in a score **i statement**.

idur – duration of event. Equivalent to p3 in a score **i statement**.

PERFORMANCE

ktrigger – trigger value for new event

schedule adds a new score event. The arguments, including options, are the same as in a score. The *iwhen* time (p2) is measured from the time of this event.

If the duration is zero or negative the new event is of MIDI type, and inherits the release sub-event from the scheduling instruction.

In the case of **schedwhen**, the event is only scheduled when the k-rate value *ktrigger* is first non-zero.

EXAMPLE

```
;; Double hit and 1sec separation
            instr 1
            schedule        2,  1, 0.5, p4, p5
 a1         shaker          p4, 60, 0.999, 0, 100, 0
            out             a1
            endin

            instr 2
 a1         marimba         p4, cpspch(p5), p6, p7, 2, 6.0, 0.05, 1, 0.1
            out             a1
            endin

            instr 3
 kr         table           kr, 1
            schedwhen       kr, 1, 0.25, 1, p4, p5
            endin
```

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
November, 1998 (New in Csound version 3.491)
Based on work by Gabriel Maldonado

6.2 schedkwhen

schedkwhen ktrigger, kmintim, kmaxnum, kinsnum, kwhen, kdur[, kp4, kp5, …]

DESCRIPTION

Adds a new score event generated by a k-rate trigger.

PERFORMANCE

ktrigger – triggers a new score event. If *ktrigger* = 0, no new event is triggered.

kmintim – minimum time between generated events, in seconds. If *kmintim* <= 0, no time limit exists. If the *kinsnum* is negative (to turn off an instrument), this test is bypassed.

kmaxnum – maximum number of simultaneous instances of instrument *kinsnum* allowed. If the number of extant instances of *kinsnum* is >= *kmaxnum*, no new event is generated. If *kmaxnum* is <= 0, it is not used to limit event generation. If the *kinsnum* is negative (to turn off an instrument), this test is bypassed.

kinsnum – instrument number. Equivalent to p1 in a score **i statement**.

kwhen – start time of the new event. Equivalent to p2 in a score **i statement**. Measured from the time of the triggering event. *kwhen* must be >= 0. If *kwhen* > 0, the instrument will not be initialized until the actual time when it should start performing.

kdur – duration of event. Equivalent to p3 in a score **i statement**. If *kdur* = 0, the instrument will only do an initialization pass, with no performance. If *kdur* is negative, a held note is initiated. (See **ihold** and **i statement**.)

kp4, kp5, etc. – Equivalent to p4, p5, etc., in a score **i statement**.

Note: While waiting for events to be triggered by **schedkwhen**, the performance must be kept going, or Csound may quit if no score events are expected. To guarantee continued performance, an **f0 statement** may be used in the score.

AUTHOR

Rasmus Ekman
EMS
Stockholm, Sweden
New in Csound version 3.59

6.3 turnon

turnon insnum[,itime]

DESCRIPTION

Activate an instrument, for an indefinite time.

INITIALIZATION

insnum – instrument number to be activated

itime – delay, in seconds, after which instrument *insnum* will be activated. Default is 0.

PERFORMANCE

turnon activates instrument *insnum* after a delay of *itime* seconds, or immediately if *itime* is not specified. Instrument is active until explicitly turned off. (See **turnoff**.)

---

7 INSTRUMENT CONTROL: DURATION CONTROL STATEMENTS

7.1 ihold, turnoff

```
ihold
turnoff
```

DESCRIPTION

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

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

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

EXAMPLE

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

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

This page intentionally left blank.

8 INSTRUMENT CONTROL: REAL-TIME PERFORMANCE CONTROL

8.1 active

```
ir      active      insnum
```

DESCRIPTION

Returns the number of active instances of an instrument.

INITIALIZATION

insnum – number of the instrument to be reported

PERFORMANCE

active returns the number of active instances of instrument number *insnum* at the time it is called.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
July, 1999
New in Csound version 3.57

8.2 cpuprc, maxalloc, prealloc

```
cpuprc      insnum, ipercent
maxalloc    insnum, icount
prealloc    insnum, icount
```

DESCRIPTION

Control allocation of cpu resources on a per-instrument basis, to optimize real-time output.

INITIALIZATION

insnum – instrument number

*ipercen*t – percent of cpu processing-time to assign. Can also be expressed as a fractional value.

icount – number of instrument allocations

PERFORMANCE

cpuprc sets the cpu processing-time percent usage of an instrument, in order to avoid buffer underrun in real-time performances, enabling a sort of polyphony theshold. The user must set *ipercent* value for each instrument to be activated in real-time. Assuming that the total theoretical processing time of the cpu of the computer is 100%, this percent value can only be defined empirically, because there are too many factors that contribute to limiting real-time polyphony in different computers.

For example, if *ipercent* is set to 5% for instrument 1, the maximum number of voices that can be allocated in real-time, is 20 (5% * 20 = 100%). If the user attempts to play a further note while the 20 previous notes are still playing, Csound inhibits the allocation of that note and will display the following warning message:

```
can't allocate last note because it exceeds 100% of cpu time
```

In order to avoid audio buffer underruns, it is suggested to set the maximum number of voices slightly lower than the real processing power of the computer. Sometimes an instrument can require more processing time than normal. If, for example, the instrument contains an oscillator which reads a table that doesn't fit in cache memory, it will be slower than normal. In addition, any program running concurrently in multitasking, can subtract processing power to varying degrees.

At the start, all instruments are set to a default value of *ipercent* = 0.0% (i.e. zero processing time or rather infinite cpu processing-speed). This setting is OK for deferred-time sessions.

maxalloc limits the number of allocations of an instrument. **prealloc** creates space for instruments but does not run them.

All instances of **cpuprc**, **maxalloc**, and **prealloc** must be defined in the header section, not in the instrument body.

EXAMPLE

```
sr     = 44100
kr     = 441
ksmps  = 100
nchnls = 2
  cpuprc 1, 2.5                  ; set instr 1 to 2.5% of processor power,
                                 ; i.e. maximum 40 voices (2.5% * 40 = 100%)
  cpuprc 2, 33.333          ; set instr 2 to 33.333% of processor power,
                                 ; i.e. maximum 3 voices (33.333% * 3 = 100%)

instr 1
...body...
endin

instr 2
....body...
endin
```

AUTHOR

Gabriel Maldonado
Italy
July, 1999
New in Csound version 3.57

This page intentionally left blank.

9 INSTRUMENT CONTROL: TIME READING

9.1 timek, times, timeinstk, timeinsts

```
ir      timek
kr      timek
ir      times
kr      times
kr      timeinstk
kr      timeinsts
```

DESCRIPTION

Opcodes to read absolute time since the start of the performance or of an instance of an instrument – in two formats.

PERFORMANCE

timek is for time in krate cycles. So with:

```
sr = 44100
kr = 6300
ksmps = 7
```

then after half a second, the **timek** opcode would report 3150. It will always report an integer.

Time in seconds is available with **times**. This would return 0.5 after half a second.

times and **timek** can also operate only at the start of the instance of the instrument. Both produce an i-rate variable (starting with i or gi) as their output.

timek and **times** can both produce a k-rate variable for output.

There are no input parameters.

timeinstk and **timeinsts** are similar to **timek** and **times**, except they return the time since the start of this instance of the instrument.

AUTHOR

Robin Whittle
Australia
May 1997

This page intentionally left blank.

10 INSTRUMENT CONTROL: CLOCK CONTROL

10.1 clockon, clockoff, readclock

```
          clockon          inum
          clockoff         inum
ir        readclock        inum
```

DESCRIPTION

Starts and stops one of a number of internal clocks, and reads the value of a clock.

INITIALIZATION

inum – the number of a clock. There are 32 clocks numbered 0 through 31. All other values are mapped to clock number 32.

ir – value at i-time, of the clock specified by *inum*

PERFORMANCE

Between a **clockon** and a **clockoff**, the CPU time used, is accumulated in the clock. The precision is machine dependent, but is the millisecond range on UNIX and Windows systems.

readclock reads the current value of a clock at initialization time.

Note: there is no way to zero a clock.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
July, 1999
New in Csound version 3.56

This page intentionally left blank.

11 INSTRUMENT CONTROL: SENSING AND CONTROL

11.1 pitch

```
koct,    pitch      asig, iupdte, ilo, ihi, idbthresh[, ifrqs, iconf,\\
kamp                istrt, iocts, iq, inptls, irolloff, iskip]
```

DESCRIPTION

Using the same techniques as **spectrum** and **specptrk, pitch** tracks the pitch of the signal in octave point decimal form, and amplitude in dB.

INITIALIZATION

iupdte – length of period, in seconds, that outputs are updated

ilo, ihi – range in which pitch is detected, expressed in octave point decimal

idbthresh – amplitude, expressed in decibels, necessary for the pitch to be detected. Once started it continues until it is 6 dB down.

ifrqs – number of divisions of an octave. Default is 12 and is limited to 120.

iconf – the number of conformations needed for an octave jump. Default is 10.

istrt – starting pitch for tracker. Default value is (*ilo* + *ihi*)/2.

iocts – number of octave decimations in spectrum. Default is 6.

iq – Q of analysis filters. Default is 10.

inptls – number of harmonics, used in matching. Computation time increases with the number of harmonics. Default is 4.

irolloff – amplitude rolloff for the set of filters expressed as fraction per octave. Values must be positive. Default is 0.6.

iskip – if non-zero, skips initialization

PERFORMANCE

pitch analyzes the input signal, *asig*, to give a pitch/amplitude pair of outputs, for the strongest frequency in the signal. The value is updated every *iupdte* seconds.

The number of partials and rolloff fraction can effect the pitch tracking, so some experimentation may be necessary. Suggested values are 4 or 5 harmonics, with rolloff 0.6, up to 10 or 12 harmonics with rolloff 0.75 for complex timbres, with a weak fundamental.

AUTHOR

John ffitch
University of Bath, Codemist Ltd.
Bath, UK
April, 1999
New in Csound version 3.54

11.2 pitchamdf

```
kcps,    pitchamdf   asig, imincps, imaxcps [, icps[, imedi[, idowns[,//
krms                 iexcps]]]]
```

DESCRIPTION

Follows the pitch of a signal based on the AMDF method (Average Magnitude Difference Function). Outputs pitch and amplitude tracking signals. The method is quite fast and should run in real-time. This technique usually works best for monophonic signals.

INITIALIZATION

imincps – estimated minimum frequency (expressed in Hz) present in the signal

imaxcps – estimated maximum frequency present in the signal

icps – estimated initial frequency of the signal. If 0, *icps* = (*imincps*+*imaxcps*) / 2. The default is 0.

imedi – size of median filter applied to the output *kcps*. The size of the filter will be *imedi**2+1. If 0, no median filtering will be applied. The default is 1.

idowns – downsampling factor for *asig*. Must be an integer. A factor of *idowns*>1 results in faster performance, but may result in worse pitch detection. Useful range is 1 – 4. The default is 1.

iexcps – how frequently pitch analysis is executed, expressed in Hz. If 0, *iexcps* is set to *imincps*. This is usually reasonable, but experimentation with other values may lead to better results. Default is 0.

PERFORMANCE

kcps – pitch tracking output

krms – amplitude tracking output

pitchamdf usually works best for monophonic signals, and is quite reliable if appropriate initial values are chosen. Setting *imincps* and *imaxcps* as narrow as possible to the range of the signal's pitch, results in better detedtion and performance.

Because this process can only detect pitch after an initial delay, setting *icps* close to the signal's real initial pitch prevents spurious data at the beginning.

The median filter prevents *kcps* from jumping. Experiment to determine the optimum value for *imedi* for a given signal.

Other initial values can usually be left at the default settings. Lowpass filtering of *asig* before passing it to **pitchamdf**, can improve preformance, especially with complex waveforms.

EXAMPLE

```
ginput     ftgen          1, 0, 0, -1, "input.wav", 0, 4, 0   ; input signal
giwave     ftgen          2, 0, 1024, 10, 1, 1, 1, 1   ; synth wave

           instr 1
asig       loscil         1, 1, ginput, 1      ; get input signal
                                               ; with original freq
asig       tone           asig, 1000    ; lowpass-filter
kcps, krms pitchamdf      asig, 150, 500, 200  ; extract pitch
                                               ; and envelope
asig1      oscil          krms, kcps, iwave    ; "resynthesize"
                                               ; with some waveform
           out            asig1
           endin
```

AUTHOR

Peter Neubäcker
Munich, Germany
August, 1999
New in Csound version 3.59

11.3 tempest

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

DESCRIPTION

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 Hz) 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 Hz.

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
```

11.4 follow

```
ar      follow      asig,  idt
```

DESCRIPTION

Envelope follower unit generator.

INITIALIZATION

idt – This is the period, in seconds, that the average amplitude of *asig* is reported. If the frequency of *asig* is low then *idt* must be large (more than half the period of *asig*)

PERFORMANCE

asig – This is the signal from which to extract the envelope.

EXAMPLE

```
k1      line    0, p3, 30000    ; Make k1 a simple envelope
a1      oscil   k1, 1000, 1     ; Make a simple signal using k1
ak1     follow  a1, .02         ; ak1 is now like k1
a2      oscil   ak1, 1000, 1    ; Make a simple signal using ak1
        out     a2              ; Both a1 and a2 are the same
```

To avoid zipper noise, by discontinuities produced from complex envelope tracking, a lowpass filter could be used, to smooth the estimated envelope.

AUTHOR

Paris Smaragdis
MIT, Cambridge
1995

11.5 trigger

```
kout     trigger     ksig, kthreshold, kmode
```

DESCRIPTION

Informs when a krate signal crosses a threshold.

PERFORMANCE

ksig – input signal

kthreshold – trigger threshold

kmode – can be 0 , 1 or 2

Normally **trigger** outputs zeroes: only each time *ksig* crosses *kthreshold* **trigger** outputs a 1. There are three modes of using *ktrig*:

- *kmode* = 0 – (down-up) *ktrig* outputs a 1 when current value of *ksig* is higher than *kthreshold,* while old value of *ksig* was equal to or lower than *kthreshold*.
- *kmode* = 1 – (up-down) *ktrig* outputs a 1 when current value of *ksig* is lower than *kthreshold* while old value of *ksig* was equal or higher than *kthreshold*.
- *kmode* = 2 – (both) *ktrig* outputs a 1 in both the two previous cases.

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.49

11.6 peak

```
kr      peak        ksig
kr      peak        asig
```

DESCRIPTION

These opcodes maintain the output k-rate variable as the peak absolute level so far received.

PERFORMANCE

kr – Output equal to the highest absolute value received so far. This is effectively an input to the opcode as well, since it reads *kr* in order to decide whether to write something higher into it.

ksig – k-rate input signal.

asig – a-rate input signal.

DEPRECATED NAME

Prior to Csound version 3.63, the k-rate version of **peak** was called **peakk**. **peak** is now used with either k- or a-rate input.

AUTHOR

Robin Whittle
Australia
May 1997

11.7 xyin, tempo

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

DESCRIPTION

Sense the cursor position in an output window. Apply tempo control to an uninterpreted score. When **xyin** is called the position of the mouse within the output window is used to reply to the request. This simple mechanism does mean that only one **xyin** can be used accurately at once. The position of the mouse is reported in the output window.

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 real-time 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
```

11.8 follow2

```
ar      follow2     asig, katt, krel
```

DESCRIPTION

A controllable envelope extractor using the algorithm attributed to Jean-Marc Jot.

PERFORMANCE

asig – the input signal whose envelope is followed

katt – the attack rate (60dB attack time in seconds)

krel – the decay rate (60dB decay time in seconds)

The output tracks the amplitude envelope of the input signal. The rate at which the output grows to follow the signal is controlled by the *katt*, and the rate at which it decreases in response to a lower amplitude, is controlled by the *krel*. This gives a smoother envelope than **follow**.

EXAMPLE

```
a1          follow2         ain, 0.01, .1
```

AUTHOR

John ffitch
University of Bath, Codemist Ltd.
Bath, UK
February, 2000
New in Csound version 4.03

11.9 setctrl, control

```
          setctrl     inum, kval, itype
kout      control     knum
```

DESCRIPTION

Configurable slider controls for realtime user input. Requires Winsound or TCL/TK.

setctrl sets a slider to a specific value, or sets a minimum or maximum range.

control reads a slider's value.

INITIALIZATION

inum – number of the slider to set

itype – type of value sent to the slider as follows:

- 1 – set the current value. Initial value is 0.
- 2 – set the minimum value. Default is 0.
- 3 – set the maximum value. Default is 127.
- 4 – set the label. (New in Csound version 4.09)

PERFORMANCE

kval – value to be sent to the slider

Calling **setctrl** or **control** will create a new slider on the screen. There is no theoretical limit to the number of sliders. Windows and TCL/TK use only integers for slider values, so the values may need rescaling. GUIs usually pass values at a fairly slow rate, so it may be advisable to pass the output of control through **port**.

EXAMPLE

```
 #define SLIDERNUM # 6 #
            instr 1
            kgoto           continue ; We don't want to configure sliders at k-
                                     ; rate!

 ; Set min=10, max=1000, actual=20
            setctrl         $SLIDERNUM., 20, 0
            setctrl         $SLIDERNUM., 10, 1
            setctrl         $SLIDERNUM., 1000, 2
continue:
 kcHz       control         $SLIDERNUM.  ; Read values with smoothing
 kcHz       port            kcHz, .02
 ; ... etc
            endin
```

AUTHOR

John ffitch
University of Bath, Codemist. Ltd.
Bath, UK
July, 2000
New in Csound version 4.06

11.10 button, checkbox

```
kr      button      inum
kr      checkbox    inum
```

DESCRIPTION

Sense on-screen controls. Needs Windows or TCL/TK.

INITIALIZATION

inum – the number of the button or checkbox. If it does not exist, it is made on-screen at initialization.

PERFORMANCE

If the button has been pushed since the last k-period, then return 1, otherwise return 0. If the checkbox is set (pushed) then return 1, if not, return 0.

EXAMPLE

Increase pitch while a checkbox is set, and extend duration for each push of a button.

```
            instr 1
kcps        =       cpsoct(p5)
k1          check   1
            if (k1 == 1) kcps = kcps * 1.1
a1          oscil   p4, kcps, 1
            out     a1

k2          button  1
            if (k2 == 1) p3 = p3 + 0.1
            endin
```

AUTHOR

John ffitch
University of Bath, Codemist Ltd.
Bath, UK
September, 2000
New in Csound version 4.08

11.11 sensekey

kr **sensekey**

DESCRIPTION

Returns the ASCII code of a key that has been pressed, or -1 if no key has been pressed.

PERFORMANCE

At release, this has not been properly verified, and seems not to work at all on Windows.

AUTHOR

John ffitch
University of Bath, Codemist. Ltd.
Bath, UK
October, 2000
New in Csound version 4.09

---

12 INSTRUMENT CONTROL: CONDITIONAL VALUES

12.1 >, <, >=, <=, ==, !=, ?

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

DESCRIPTION

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 **||**).

These are *operators* not *opcodes*. Therefore, they can be used within orchestra statements, but do not form complete statements themselves.

EXAMPLE

```
k2 = (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.

This page intentionally left blank.

13 INSTRUMENT CONTROL: MACROS

13.1 #define, $NAME, #undef

```
#define     NAME # replacement text #
#define     NAME(a' b' c') # replacement text #
$NAME.
#undef      NAME
```

DESCRIPTION

Macros are textual replacements which are made in the orchestra as it is being read. The macro system in Csound is a very simple one, and uses the characters # and $ to define and call macros. This can save typing, and can lead to a coherent structure and consistent style. This is similar to, but independent of, the macro system in the score language.

#define NAME – defines a simple macro. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Case is significant. This form is limiting, in that the variable names are fixed. More flexibility can be obtained by using a macro with arguments, described below.

#define NAME(*a' b' c'*) – defines a macro with arguments. This can be used in more complex situations. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Within the replacement text, the arguments can be substituted by the form: $A. In fact, the implementation defines the arguments as simple macros. There may be up to 5 arguments, and the names may be any choice of letters. Remember that case is significant in macro names.

$NAME. – calls a defined macro. To use a macro, the name is used following a $ character. The name is terminated by the first character which is neither a letter nor a number. If it is necessary for the name not to terminate with a space, a period, which will be ignored, can be used to terminate the name. The string, **$NAME.**, is replaced by the replacement text from the definition. The replacement text can also include macro calls.

#undef NAME – undefines a macro name. If a macro is no longer required, it can be undefined with **#undef NAME**.

INITIALIZATION

 # replacement text # – The replacement text is any character string (not containing a #) and can extend over multiple lines. The replacement text is enclosed within the # characters, which ensure that additional characters are not inadvertently captured.

PERFORMANCE

Some care is needed with textual replacement macros, as they can sometimes do strange things. They take no notice of any meaning, so spaces are significant. This is why, unlike the C programming language, the definition has the replacement text surrounded by # characters. Used carefully, this simple macro system is a powerful concept, but it can be abused.

EXAMPLES

Simple Macro

```
#define REVERB #ga = ga+a1
            out a1#

            instr 1
a1          oscil
$REVERB.
            endin

            instr 2
a1          repluck
$REVERB.
            endin
```

This will get expanded before compilation into:

```
            instr 1
a1          oscil
ga          =               ga+a1
            out a1
            endin

            instr 2
a1          repluck
 ga         =               ga+a1
            out             a1
            endin
```

Macro With Arguments

```
#define REVERB(A) #ga = ga+$A.
            out $A.#
            instr 1
  a1        oscil
  $REVERB(a1)
            endin

            instr 2
  a2        repluck
  $REVERB(a2)
            endin
```

This will get expanded before compilation into:

```
            instr 1
  a1        oscil
  ga        =               ga+a1
            out a1
            endin

            instr 2
  a2        repluck
  ga        =               ga+a2
            out a2
            endin
```

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

13.2 #include

#include "filename"

DESCRIPTION:

It is sometimes convenient to have the orchestra arranged in a number of files, for example with each instrument in a separate file. This style is supported by the **#include** facility which is part of the macro system. A line containing the text

```
#include "filename"
```

where the character " can be replaced by any suitable character. For most uses the double quote symbol will probably be the most convenient. The file name can include a full path.

This takes input from the named file until it ends, when input reverts to the previous input. There is currently a limit of 20 on the depth of included files and macros.

Another suggested use of **#include** would be to define a set of macros which are part of the composer's style.

An extreme form would be to have each instrument defines as a macro, with the instrument number as a parameter. Then an entire orchestra could be constructed from a number of **#include** statements followed by macro calls.

```
#include "clarinet"
#include "flute"
#include "bassoon"
$CLARINET(1)
$FLUTE(2)
$BASSOON(3)
```

It must be stressed that these changes are at the textual level and so take no cognizance of any meaning.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

This page intentionally left blank.

14 INSTRUMENT CONTROL: PROGRAM FLOW CONTROL

14.1 igoto, tigoto, kgoto, goto, if, timout

```
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
```

DESCRIPTION

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 under **reinit**).

EXAMPLE

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

This page intentionally left blank.

15 INSTRUMENT CONTROL: REINITIALIZATION

15.1 reinit, rigoto, rireturn

```
reinit      label
rigoto      label
rireturn
```

DESCRIPTION

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
          rireturn                    ; then resume perf
```

This page intentionally left blank.

16 MATHEMATICAL OPERATIONS: ARITHMETIC AND LOGIC OPERATIONS

16.1 -, +, &&, ||, *, /, ^, %

```
    -  a     (no rate restriction)
    +  a     (no rate restriction)
a   && b     (logical AND;  not audio-rate)
a   || b     (logical OR;   not audio-rate)
a   +  b     (no rate restriction)
a   -  b     (no rate restriction)
a   *  b     (no rate restriction)
a   /  b     (no rate restriction)
a   ^  b     (b not audio-rate)
a   %  b     (no rate restriction)
```

DESCRIPTION

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.

The operator ^ raises *a* to the *b* power. *b* may not be audio-rate. Use with caution as precedence may not work correctly. See Section 5.2. New in Csound version 3.493.

The operator % returns the value of *a* reduced by *b*, so that the result, in absolute value, is that of the absolute value of *b*, by repeated subtraction. This is the same as modulus function in integers. New in Csound version 3.50.

---

17 MATHEMATICAL OPERATIONS: MATHEMATICAL FUNCTIONS

17.1 int, frac, i, abs, exp, log, log10, sqrt

```
int(x)          (init- or control-rate args only)
frac(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)
log10(x)        (no rate restriction)
sqrt(x)         (no rate restriction)
```

DESCRIPTION

Where the argument within the parentheses may be an expression. These functions 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*.

i(*x*) – returns an init-type equivalent of the argument (k-rate)

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).

log10(*x*) – returns the base 10 log of *x* (*x* positive only).

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

Note that for **log, log10,** and **sqrt** the argument value is restricted.

17.2 powoftwo, logbtwo

```
powoftwo(x) (init-rate or control-rate args only)
logbtwo(x)  (init-rate or control-rate args only)
```

DESCRIPTION

Power-of-two operations.

PERFORMANCE

powoftwo() function returns 2 ^ *x* and allows positive and negatives numbers as argument. The range of values admitted in **powoftwo**() is -5 to +5 allowing a precision more fine than one cent in a range of ten octaves. If a greater range of values is required, use the slower opcode **pow**.

logbtwo() returns the logarithm base two of *x*. The range of values admitted as argument is .25 to 4 (i.e. from -2 octave to +2 octave response). This function is the inverse of **powoftwo**().

These functions are fast, because they read values stored in tables. Also they are very useful when working with tuning ratios. They work at i- and k-rate.

AUTHORS

Gabriel Maldonado
Italy
June, 1998

John ffitch
University of Bath, Codemist, Ltd.
Bath, UK
July, 1999
New in Csound version 3.57

18 MATHEMATICAL OPERATIONS: TRIGONOMETRIC FUNCTIONS

18.1 sin, cos, tan, sininv, cosinv, taninv, sinh, cosh, tanh

```
sin(x)              (no rate restriction)
cos(x)              (no rate restriction)
tan(x)              (no rate restriction)
sininv(x)           (no rate restriction)
cosinv(x)           (no rate restriction)
taninv(x)           (no rate restriction)
sinh(x)             (no rate restriction)
cosh(x)             (no rate restriction)
tanh(x)             (no rate restriction)
```

DESCRIPTION

Where the argument within the parentheses may be an expression. These functions perform trigonometric conversions. The result can then be a term in a further expression.

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

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

tan (*x*) – returns the tangent of *x*.

sininv(*x*) – returns the arcsine of *x*.

cosinv(*x*) – returns the arcosine of *x*.

taninv(x) – returns the arctangent of *x*.

sinh(*x*) – returns the hyperbolic sine of *x*.

cosh(*x*) – returns the hyperbolic cosine of *x*.

tanh (x*)* – returns the hyperbolic tangent of *x* .

This page intentionally left blank.

19 MATHEMATICAL OPERATIONS: AMPLITUDE FUNCTIONS

19.1 dbamp, ampdb dbfsamp, ampdbfs

```
dbamp(x)          (init- or control-rate args only)
ampdb(x)          (no rate restriction)

dbfsamp(x)        (init- or control-rate args only)
ampdbfs(x)        (no rate restriction)
```

DESCRIPTION

Where the argument within the parentheses may be an expression. These functions perform conversions between raw amplitude values and their decibel equivelents. The result can then be a term in a further expression.

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

ampdb(*x*) – returns the amplitude equivalent of the decibel value *x*. Thus:

60 dB = 1000
66 dB = 1995.262
72 dB = 3891.07
78 dB = 7943.279
84 dB = 15848.926
90 dB = 31622.764

dbfsamp(x) – returns the decibel equivalent, relative to full scale amplitude, of the raw amplitude *x*. Full scale is assumed to be 16 bit. New is Csound version 4.10.

ampdbfs(x) – returns the amplitude equivalent of the decibel value *x*, which is relative to full scale amplitude. Full scale is assumed to be 16 bit. New is Csound version 4.10.

This page intentionally left blank.

20 MATHEMATICAL OPERATIONS: RANDOM FUNCTIONS

20.1 rnd, birnd

```
rnd(x)          (init- or control-rate args only)
birnd(x)        (init- or control-rate args only)
```

DESCRIPTION

Where the argument within the parentheses may be an expression. These value converters sample a global random sequence, but do not reference **seed**. The result can be a term in a further expression.

PERFORMANCE

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*. **rnd** and **birnd** obtain values from a global pseudo-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

AUTHOR

Barry Vercoe
MIT
Cambridge, Massachusetts
1997

This page intentionally left blank.

21 MATHEMATICAL FUNCTIONS: OPCODE EQUIVALENTS OF FUNCTIONS

21.1 sum

```
ar      sum         asig1, asig2[, asig3...asigN]
```

DESCRIPTION

Sums any number of a-rate signals.

PERFORMANCE

asig1, etc. – a-rate signals to be summed (mixed or added).

AUTHOR

Gabriel Maldonado
Italy
April, 1999
New in Csound version 3.54

21.2 product

```
ar      product     asig1, asig2[, asig3...asigN]
```

DESCRIPTION

Multiplies any number of a-rate signals.

PERFORMANCE

asig1, etc. – a-rate signals to be multiplied.

AUTHOR

Gabriel Maldonado
Italy
April, 1999
New in Csound version 3.54

21.3 pow

```
ir      pow         iarg, ipow
ir      =           iarg ^ ipow
kr      pow         karg, kpow[, inorm]
ar      pow         aarg, kpow[, inorm]
```

DESCRIPTION

Computes *xarg* to the power of *kpow* (or *ipow*) and scales the result by *inorm*.

INITIALIZATION

inorm – The number to divide the result (default to 1). This is especially useful if you are doing powers of a- or k- signals where samples out of range are extremely common!

iarg – i-rate base

ipow – i-rate exponent

PERFORMANCE

karg – k-rate base.

kpow – k-rate exponent

aarg – a-rate base.

EXAMPLES

```
i2t2      pow         2,2                 ; Computes 2^2.
kline     line        0, 1, 4
kexp      pow         kline, 2, 4
```

This feeds a linear function to **pow** and scales that to the line's peak value. The output will be an exponential curve with the same range as the input line.

```
iamp      pow         10, 2
a1        oscil       iamp, 100, 1
a2        pow         a1, 2, iamp
          out         a2
```

This will output a sine with its negative part folded over the amplitude axis. The peak value will be iamp = 10^2 = 100.

The first line could also be written:

```
i2t2  =    2 ^ 2
```

Use ^ with caution in arithmetical statements, as the precedence may not be correct. This operator is new as of Csound version 3.493.

DEPRECATED NAMES

pow was originally three opcodes called **ipow**, **kpow**, and **apow**. As of Csound version 3.48 those names are deprecated, and the three seperate opcodes replaced by **pow**.

AUTHOR

Paris Smaragdis
MIT, Cambridge
1995

21.4 taninv2

```
ir        taninv2     ix, iy
kr        taninv2     kx, ky
ar        taninv2     ax, ay
```

DESCRIPTION

Returns the arctangent of *iy*/*ix*, *ky*/*kx*, or *ay*/*ax*. If either *x* or *y* is zero, **taninv2** returns zero.

INITIALIZATION

ix, iy – values to be converted

PERFORMANCE

kx, ky – control rate signals to be converted

ax, ay – audio rate signals to be converted

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

21.5 mac, maca

```
ar      mac       asig1, ksig1, asig2[, ksig2, asig3, ...asigN, ksigN]
ar      maca      asig1, asig2[, asig3, asig4, asig5, ...asigN]
```

DESCRIPTION

Multiply and accumulate k- and/or a-rate signals.

PERFORMANCE

ksig1, etc. – k-rate input signals

asig1, etc. – a-rate input signals

mac multiplies and accumulates a- and k-rate signals. It is equivalent to:

```
ar = asig1 + ksig1*asig2 + ksig2+asig3 + ...
```

maca multiplies and accumulates a-rate signals only. It is equivalent to:

```
ar = asig1 + asig2*asig3 + asig4+asig5 + ...
```

AUTHOR

John ffitch
University of Bath, Codemist, Ltd.
Bath, UK
May, 1999
New in Csound version 3.55

This page intentionally left blank.

22 PITCH CONVERTERS: FUNCTIONS

22.1 octpch, pchoct, cpspch, octcps, cpsoct

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)

DESCRIPTION

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

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

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

Name	Abbreviation
octave point pitch-class (8ve.pc)	*pch*
octave point decimal	*oct*
cycles per second	*cps* (Hz)

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

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

```
cpspch(8.09)
```

will convert the pitch argument 8.09 to its *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.

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

This page intentionally left blank.

23 PITCH CONVERTERS: TUNING OPCODES

23.1 cps2pch, cpsxpch

```
icps     cps2pch     ipch, iequal
icps     cpsxpch     ipch, iequal, irepeat, ibase
```

DESCRIPTION

Converts a pitch-class notation into cycles-per-second (Hz) for equal divisions of the octave (for **cps2pch**) or for equal divisions of any interval. There is a restriction of no more than 100 equal divisions.

INITIALIZATION

ipch – Input number of the form 8ve.pc, indicating an 'octave' and which note in the octave.

iequal – if positive, the number of equal intervals into which the 'octave' is divided. Must be less than or equal to 100. If negative, is the number of a table of frequency multipliers.

irepeat – Number indicating the interval which is the 'octave.' The integer 2 corresponds to octave divisions, 3 to a twelfth, 4 is two octaves, and so on. This need not be an integer, but must be positive.

ibase – The frequency which corresponds to pitch 0.0

Note:

The following are essentially the same

```
ia          =           cpspch(8.02)
ib          cps2pch     8.02, 12
ic          cpsxpch     8.02, 12, 2, 1.02197503906
```

These are opcodes not functions.

Negative values of *ipch* are allowed, but not negative *irepeat*, *iequal* or *ibase*.

EXAMPLE

```
  inote     cps2pch     p5, 19          ; convert oct.pch to
                                        ; cps in 19ET
  inote     cpsxpch     p5, 12, 3, 261.62561 ; Pierce scale centered
                                        ; on middle A
  inote     cpsxpch     p5, 21, 4, 16.35160062496 ; 10.5ET scale
The use of a table allows exotic scales by mapping frequencies in a table. For example
the table:
  f2 0 16 -2 1 1.1 1.2 1.3 1.4 1.6 1.7 1.8 1.9
can be used with:
  ip        cps2pch     p4, -2
to get a 10 note scale of unequal divisions.
```

AUTHOR

John ffitch
University of Bath/Codemist Ltd. Gabriel Maldonado
Italy
1998 (New in Csound version 3.492)
Bath, UK
1997

---

24 MIDI SUPPORT: CONVERTERS

24.1 notnum, veloc, cpsmidi, cpsmidib, octmidi, octmidib, pchmidi, pchmidib, ampmidi, aftouch, pchbend, midictrl

```
ival    notnum
ival    veloc       [ilow, ihigh]
icps    cpsmidi
icps    cpsmidib    [irange]
kcps    cpsmidib    [irange]
ioct    octmidi
ioct    octmidib    [irange]
koct    octmidib    [irange]
ipch    pchmidi
ipch    pchmidib    [irange]
kpch    pchmidib    [irange]
iamp    ampmidi     iscal[, ifn]
kaft    aftouch     [imin[, imax]]
ibend   pchbend     [imin[, imax]]
kbend   pchbend     [imin[, imax]]
ival    midictrl    inum[imin[, imax]]
kval    midictrl    inum[imin[, imax]]
```

DESCRIPTION

Get a value from the MIDI event that activated this instrument, or from a continuous MIDI controller, and convert it to a locally useful format.

INITIALIZATION

iscal – i-time scaling factor.

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.

inum, ictlno – MIDI controller number

initial – the *initial* value of the controller

ilow, ihigh – low and high ranges for mapping

irange – the pitch bend range in semitones

ichnl – the MIDI channel

imin, imax – set minimum and maximum limits on values obtained

PERFORMANCE

notnum, **veloc** – get the MIDI byte value (0 – 127) denoting the note number or velocity of the current event.

cpsmidi, **octmidi**, **pchmidi** – get the note number of the current MIDI event, expressed in cps, oct, or pch units for local processing.

cpsmidib, **octmidib**, **pchmidib** – get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in cps, oct, or pch units. Available as an i-time value or as a continuous k-rate 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*.

aftouch, **pchbend** – get the current after-touch, or pitch-bend value for this channel, rescaled to the range 0 – *iscal*. 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.

midictrl – get the current value (0 – 127) of a specified MIDI controller.

AUTHOR

Barry Vercoe – Mike Berry
MIT – Mills
May 1997

24.2 cpstmid

icps **cpstmid** ifn

DESCRIPTION

This unit is similar to **cpsmidi**, but allows fully customized micro-tuning scales.

INITIALIZATION

ifn – function table containing the parameters (*numgrades*, *interval*, *basefreq*, *basekeymidi*) and the tuning ratios.

PERFORMANCE

Init-rate only

cpsmid requires five parameters. The first, *ifn,* is the function table number of the tuning ratios, and the other parameters must be stored in the function table itself. The function table *ifn* should be generated by **GEN2,** with normalization inhibited. The first four values stored in this function are:

1. *numgrades* – the number of grades of the micro-tuning scale

2. *interval* – the frequency range covered before repeating the grade ratios,for example 2 for one octave, 1.5 for a fifth etc.

3. *basefreq* – the base frequency of the scale in Hz

4. *basekeymidi* – the MIDI note number to which *basefreq* is assigned unmodified

After these four values, the user can begin to insert the tuning ratios. For example, for a standard 12 note scale with the base frequency of 261 Hz assigned to the key number 60, the corresponding **f** statement in the score to generate the table should be:

```
;        numgrades        basefreq              tuning-ratios (equal temp)
;                     interval         basekeymidi
f1 0 64 -2   12      2       261        60        1  1.059463094359  1.122462048309
1.189207115003 ..etc...
```

Another example with a 24 note scale with a base frequency of 440 assigned to the key number 48, and a repetition interval of 1.5:

```
;        numgrades        basefreq              tuning-ratios .......
;                  interval         basekeymidi
f1 0 64 -2   24      1.5     440        48        1   1.01  1.02  1.03   ..etc...
```

AUTHOR

Gabriel Maldonado
Italy
1998 (New in Csound version 3.492)

This page intentionally left blank.

25 MIDI SUPPORT: CONTROLLER INPUT

25.1 initc7, initc14, initc21

```
initc7      ichan, ictlno, ivalue
initc14     ichan, ictlno1, ictlno2, ivalue
initc21     ichan, ictlno1, ictlno2, ictlno3, ivalue
```

DESCRIPTION

Initializes MIDI controller *ictlno* with *ivalue*

INITIALIZATION

ichan – MIDI channel

ictlno – controller number (**initc7**)

ictlno1 – most significant byte controller number

ictlno2 – in **initc14** least significant byte controller number; in **initc21** Medium Significant Byte controller number

ictlno3 – least significant byte controller number

ivalue – floating point value (must be within 0 to 1)

PERFORMANCE

initc7, **initc14**, **initc21** can be used together with both **midicXX** and **ctrlXX** opcodes for initializing the first controller's value. *ivalue* argument must be set with a number within 0 to 1. An error occurs if it is not. Use the following formula to set *ivalue* according with **midicXX** and **ctrlXX** min and max range:

```
ivalue = (initial_value – min) / (max – min)
```

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.47

25.2 midic7, midic14, midic21, ctrl7, ctrl14, ctrl21

```
idest    midic7      ictlno, imin, imax [, ifn]
kdest    midic7      ictlno, kmin, kmax [, ifn]
idest    midic14     ictlno1, ictlno2, imin, imax [, ifn]
kdest    midic14     ictlno1, ictlno2, kmin, kmax [, ifn]
idest    midic21     ictlno1, ictlno2, ictlno3, imin, imax [, ifn]
kdest    midic21     ictlno1, ictlno2, ictlno3, kmin, kmax [, ifn]
idest    ctrl7       ichan, ictlno, imin, imax [,ifn]
kdest    ctrl7       ichan, ictlno, kmin, kmax [,ifn]
idest    ctrl14      ichan, ictlno1, ictlno2, imin, imax [,ifn]
kdest    ctrl14      ichan, ictlno1, ictlno2, kmin, kmax [,ifn]
idest    ctrl21      ichan, ictlno1, ictlno2, ictlno3, imin, imax [,ifn]
kdest    ctrl21      ichan, ictlno1, ictlno2, ictlno3, kmin, kmax [,ifn]
```

DESCRIPTION

Allow precise MIDI input controller signal.

INITIALIZATION

idest – output signal

ictlno – MIDI controller number (1-127)

ictlno1 – most-significant byte controller number (1-127)

ictlno2 – in **midic14**: least-significant byte controller number (1-127); in **midic21**: mid-significant byte controller number (1-127)

ictlno3 – least-significant byte controller number (1-127)

imin – user-defined minimum floating-point value of output

imax – user-defined maximum floating-point value of output

ifn (optional) – table to be read when indexing is required. Table must be normalized. Output is scaled according to *imax* and *imin* val.

PERFORMANCE

kdest – output signal

kmin – user-defined minimum floating-point value of output

kmax – user-defined maximum floating-point value of output

midic7 (i- and k-rate 7 bit MIDI control) allows floating point 7 bit MIDI signal scaled with a minimum and a maximum range. It also allows optional non-interpolated table indexing. In **midic7** minimum and maximum values can be varied at k-rate. **midic14** (i- and k-rate 14 bit MIDI control) do the same as the above with 14 bit precision. **midic21** (i- and k-rate 21 bit MIDI control) do the same as the above with 21 bit precision. **midic14** and **midic21** can use optional interpolated table indexing. They require two or three MIDI controllers as input.

ctrl7, ctrl14, ctrl21 are very similar to midicXX opcodes the only differences are:

- **ctrl**XX UGs can be included in score oriented instruments without Csound crashes.
- They need the additional parameter *ichan* containing the MIDI channel of the controller. MIDI channel is the same for all the controller used in a single **ctrl14** or **ctrl21** opcode.

DEPRECATED NAMES

The opcode names **imidic7**, **imidic14**, **imidic21**, **ictrl7**, **ictrl14**, and **ictrl21** have been deprecated in Csound version 3.52. Instead use **imidic7**, **imidic14**, **imidic21**, **ictrl7**, **ictrl14**, and **ictrl21**, respectively, with i-rate outputs.

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.47

25.3 chanctrl

```
ival     chanctrl    ichnl, ictlno[,ilow,ihigh]
kval     chanctrl    ichnl, ictlno[,ilow,ihigh]
```

DESCRIPTION

Get the current value of a controller and optionally map it onto specified range.

INITIALIZATION

ichnl – the MIDI channel

ictlno – the MIDI controller number

ilow, ihigh – low and high ranges for mapping

AUTHOR

Mike Berry
Mills College
May 1997

26 MIDI Support: Slider Banks

26.1 slider8, slider16, slider32, slider64, slider8f, slider16f, slider32f, slider64f, s16b14, s32b14

```
i1, ..., i8     slider8     ichan, ictlnum1, imin1, imax1, ifn1, ....,\\
                            ictlnum8, imin8, imax8, ifn8
k1, ..., k8     slider8     ichan, ictlnum1, imin1, imax1, init1, ifn1,\\
                            ...,  ictlnum8, imin8, imax8, init8, ifn8
i1, ..., i16    slider16    ichan, ictlnum1,  imin1,  imax1,  ifn1, ....,\\
                            ictlnum16, imin16, imax16, ifn16
k1, ..., k16    slider16    ichan, ictlnum1,  imin1,  imax1,  init1, ifn1,\\
                            ...., ictlnum16, imin16, imax16, init16, ifn16
i1, ..., i32    slider32    ichan, ictlnum1,  imin1,  imax1,  ifn1, ....,\\
                            ictlnum32, imin32, imax32, ifn32
k1, ..., k32    slider32    ichan, ictlnum1,  imin1,  imax1,  init1, fn1,\\
                            ...., ictlnum32, imin32, imax32, init32, ifn32
i1, ..., i64    slider64    ichan, ictlnum1,  imin1,  imax1,  ifn1, ....,\\
                            ictlnum64, imin64, imax64, ifn64
k1, ..., k64    slider64    ichan, ictlnum1,  imin1,  imax1,  init1,  ifn1,\\
                            ...., ictlnum64, imin64, imax64, init64, ifn64
k1, ..., k8     slider8f    ichan, ictlnum1, imin1, imax1, init1, ifn1,\\
                            icutoff1, ...., ictlnum8, imin8, imax8, init8,\\
                            ifn8, icutoff8
k1, ..., k16    slider16f   ichan, ictlnum1,  imin1,  imax1,  init1, ifn1,\\
                            icutoff1, .... ,ictlnum16, imin16, imax16,\\
                            init16, ifn16, icutoff16
k1, ..., k32    slider32f   ichan, ictlnum1,  imin1,  imax1,  init1,  ifn1,\\
                            icutoff1, .... , ictlnum32, imin32, imax32,\\
                            init32, ifn32, icutoff32
k1, ..., k64    slider64f   ichan, ictlnum1,  imin1,  imax1,  init1,  ifn1,\\
                            icutoff1, .... , ictlnum64, imin64, imax64,\\
                            init64, ifn64, icutoff64
i1, ..., i16    s16b14      ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1,\\
                            initvalue1, ifn1, ....., ictlno_msb16,\\
                            ictlno_lsb16, imin16, imax16, initvalue16, ifn16
k1, ..., k16    s16b14      ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1,\\
                            ifn1, ictlno_msb16, ictlno_lsb16, imin16,\\
                            imax16, ifn16
i1, ..., i32    s32b14      ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1,\\
                            initvalue1, ifn1, ....., ictlno_msb32,\\
                            ictlno_lsb32, imin32, imax32, initvalue32, ifn32
k1, ..., k32    s32b14      ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1,\\
                            ifn1, ...., ictlno_msb32, ictlno_lsb32, imin32,\\
                            imax32, ifn32
```

Description

MIDI slider control banks

INITIALIZATION

i1 ... *i64* – output values

ichan – MIDI channel (1-16)

ictlnum1 ... *ictlnum64* – MIDI control number

ictlno_msb1 .... *ictlno_msb32* – MIDI control number (most significant byte)

ictlno_lsb1 .... *ictlno_lsb32* – MIDI control number (least significant byte)

imin1 ... *imin64* – minimum values for each controller

imax1 ... *imax64* – maximum values for each controller

init1 ... *init64* – initial value for each controller

ifn1 ... *ifn64* – function table for conversion for each controller

icutoff1 ... *icutoff64* – low-pass filter cutoff frequency for each controller

PERFORMANCE

k1 ... *k64* – output values

sliderN and **sliderNf** are banks of MIDI controllers, useful when using MIDI mixer such as Kawai MM-16 or others for changing whatever sound parameter in real-time. The raw MIDI control messages at the input port are converted to agree with *iminN* and *imaxN*, and an initial value can be set. Also, an optional non-interpolated function table with a custom translation curve is allowed, useful for enabling exponential response curves.

When no function table translation is required, set the *ifnN* value to 0, else set *ifnN* to a valid function table number. When table translation is enabled (i.e. setting *ifnN* value to a non-zero number referring to an already allocated function table), *initN* value should be set equal to *iminN* or *imaxN* value, else the initial output value will not be the same as specified in *initN* argument.

slider8 allows a bank of 8 different MIDI control message numbers, **slider16** does the same with a bank of 16 controls, and so on.

sliderNf filter the signal before output, for eliminating discontinuities due to the low resolution of the MIDI (7 bit); the cutoff frequency can be set separately for each controller (suggested range: .1 to 5 Hz).

As the input and output arguments are many, you can split the line using '\' (backslash) character (new in 3.47 version) to improve the readability. Using these opcodes is considerably more efficient than using the separate ones (**ctrl7** and **tonek**) when more controllers are required.

In the i-rate version of **sliderN**, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.

sNb14 opcode is the 14-bit version of this bank of controllers.

Warning: sliderNf opcodes do not output the required initial value immediately, but only after some k-cycles, because the filter slightly delays the output.

DEPRECATED NAMES

The opcode names **islider8**, **islider16**, **islider32**, **islider64**, **is16b14**, and **is32b14** have been deprecated as of Csound version 3.52. Use **slider8**, **slider16**, **slider32**, **slider64**, **s16b14**, and **s32b14**, respectively, for i-rate output.

AUTHOR

Gabriel Maldonado
Italy
December 1998 (New in Csound version 3.50)

This page intentionally left blank.

27 MIDI SUPPORT: GENERIC I/O

27.1 midiin

`kstatus, kchan, kdata1, kdata2` **midiin**

DESCRIPTION

Returns a generic MIDI message received by the MIDI IN port

PERFORMANCE

kstatus – the type of MIDI message. Can be:

- 128 (note off),
- 144 (note on),
- 160 (polyphonic aftertouch),
- 176 (control change),
- 192 (program change),
- 208 (channel aftertouch),
- 224 (pitch bend)
- 0 if no MIDI messages are pending in the MIDI IN buffer.

kchan – MIDI channel (1-16)

kdata1, kdata2 – message-dependent data values

midiin has no input arguments, because it reads at the MIDI in port implicitly. It works at k-rate. Normally (i.e., when no messages are pending) *kstatus* is zero, only when MIDI data are present in the MID IN buffer, is *kstatus* set to the type of the relevant messages.

AUTHOR

Gabriel Maldonado
Italy
1998 (New in Csound version 3.492)

27.2 midiout

midiout kstatus, kchan, kdata1, kdata2

DESCRIPTION

Sends a generic MIDI message to the MIDI OUT port

PERFORMANCE

kstatus – the type of MIDI message. Can be:

- 128 (note off),
- 144 (note on),
- 160 (polyphonic aftertouch),
- 176 (control change),
- 192 (program change),
- 208 (channel aftertouch),
- 224 (pitch bend)
- 0 when no MIDI messages must be sent to the MIDI OUT port.

kchan – MIDI channel (1-16)

kdata1, kdata2 – message-dependent data values

midiout has no output arguments, because it sends a message to the MIDI OUT port implicitly. It works at k-rate. It sends a MIDI message only when *kstatus* is non-zero.

Warning: Normally *kstatus* should be set to 0. Only when the user intends to send a MIDI message, can it be set to the corresponding message type number.

AUTHOR

Gabriel Maldonado
Italy
1998 (New in Csound version 3.492)

28 MIDI SUPPORT: NOTE-ON/NOTE-OFF

28.1 noteon, noteoff, noteondur, noteondur2

```
noteon     ichn, inum, ivel
noteoff    ichn, inum, ivel
noteondur  ichn, inum, ivel, idur
noteondur2 ichn, inum, ivel, idur
```

DESCRIPTION

Send note-on and note-off messages to the MIDI OUT port.

INITIALIZATION

ichn – MIDI channel number (0-15)

inum – note number (0-127)

ivel – velocity (0-127)

PERFORMANCE

noteon (i-rate note on) and **noteoff** (i-rate note off) are the simplest MIDI OUT opcodes. **noteon** sends a MIDI noteon message to MIDI OUT port, and **noteoff** sends a noteoff message. A **noteon** opcode must always be followed by an **noteoff** with the same channel and number inside the same instrument, otherwise the note will play endlessly. These **noteon** and **noteoff** are useful only when introducing a timout statement to play a non-zero duration MIDI note. For most purposes it is better to use **noteondur** and **noteondur2**.

noteondur and **noteondur2** (i-rate note on with duration) send a noteon and a noteoff MIDI message both with the same channel, number and velocity. Noteoff message is sent after *idur* seconds are elapsed by the time **noteondur** was active.

noteondur differs from **noteondur2** in that **noteondur** truncates note duration when current instrument is deactivated by score or by real-time playing, while **noteondur2** will extend performance time of current instrument until *idur* seconds have elapsed. In real-time playing it is suggested to use **noteondur** also for undefined durations, giving a large *idur* value.

Any number of **noteondur** or **noteondur2** opcodes can appear in the same Csound instrument, allowing chords to be played by a single instrument.

NAME CHANGES

Prior to Csound version 3.52 (February, 1999), these opcodes were called **ion**, **ioff**, **iondur**, and **iodur2**. **ondur** and **ondur2** changed to **noteondur** and **noteondur2** in Csound version 3.53.

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.47

28.2 moscil, midion

```
moscil      kchn, knum, kvel, kdur, kpause
midion      kchn, knum, kvel
```

DESCRIPTION

Send a stream of note-on and note-off messages to the MIDI OUT port.

PERFORMANCE

kchn – MIDI channel number (0-15)

knum – note number (0-127)

kvel – velocity (0-127)

kdur – note duration in seconds

kpause – pause duration after each noteoff and before new note in seconds

moscil and **midion** are the most powerful MIDI OUT opcodes. **moscil** (MIDI oscil) plays a stream of notes of *kdur* duration. Channel, pitch, velocity, duration and pause can be controlled at k-rate, allowing very complex algorithmically generated melodic lines. When current instrument is deactivated, the note played by current instance of **moscil** is forcedly truncated.

midion (k-rate note on) plays MIDI notes with current *kchn*, *knum* and *kvel*. These arguments can be varied at k-rate. Each time the MIDI converted value of any of these arguments changes, last MIDI note played by current instance of **midion** is immediately turned off and a new note with the new argument values is activated. This opcode, as well as **moscil**, can generate very complex melodic textures if controlled by complex k-rate signals.

Any number of **moscil** or **midion** opcodes can appear in the same Csound instrument, allowing a counterpoint-style polyphony within a single instrument.

DEPRECATED NAMES

midion was originally called **kon.** As of Csound version 3.493, that name is deprecated. **midion** should be used instead of **kon**.

AUTHOR

Gabriel Maldonado
Italy
May 1997 (**moscil** new in Csound version 3.47)

28.3 midion2

midion2 kchn, knum, kvel, ktrig

DESCRIPTION

Sends noteon and noteoff messages to the MIDI out port when triggered by a value different than zero.

PERFORMANCE

kchn – MIDI channel

knum – MIDI note number

kvel – note velocity

ktrig – trigger input signal (normally 0)

Similar to **midion**, this opcode sends noteon and noteoff messages to the MIDI out port, but only when *ktrig* is non-zero. This opcode is can work together with the output of the **trigger** opcode.

AUTHOR

Gabriel Maldonado
Italy
1998 (New in Csound version 3.492)

29 MIDI SUPPORT: MIDI MESSAGE OUTPUT

29.1 outic, outkc, outic14, outkc14, outipb, outkpb, outiat, outkat, outipc, outkpc, outipat, outkpat

```
outic    ichn, inum, ivalue, imin, imax
outkc    kchn, knum, kvalue, kmin, kmax
outic14  ichn, imsb, ilsb, ivalue, imin, imax
outkc14  kchn, kmsb, klsb, kvalue, kmin, kmax

outipb   ichn, ivalue, imin, imax
outkpb   kchn, kvalue, kmin, kmax
outiat   ichn, ivalue, imin, imax
outkat   kchn, kvalue, kmin, kmax
outipc   ichn, iprog, imin, imax
outkpc   kchn, kprog, kmin, kmax

outipat  ichn, inotenum, ivalue, imin, imax
outkpat  kchn, knotenum, kvalue, kmin, kmax
```

DESCRIPTION

Send a single Channel message to the MIDI OUT port.

PERFORMANCE

ichn, *kchn* – MIDI channel number (0-15)

inum, *knum* – controller number (0-127 for example 1 = ModWheel; 2 = BreathControl etc.)

ivalue, *kvalue* – floating point value

imin, *kmin* – minimum floating point value (converted in MIDI integer value 0)

imax, *kmax* – maximum floating point value (converted in MIDI integer value 127 (7 bit) or 16383 (14 bit))

imsb, *kmsb* – most significant byte controller number when using 14 bit parameters

ilsb, *klsb* – least significant byte controller number when using 14 bit parameters

iprog, *kprog* – program change number in floating point

inotenum, *knotenum* – MIDI note number (used in polyphonic aftertouch messages)

outic and **outkc** (i- and k-rate MIDI controller output) send controller messages to MIDI OUT device. **outic14** and **outkc14** (i and k-rate MIDI 14 bit controller output) send a pair of controller messages. These opcodes can drive 14 bit parameters on MIDI instruments that recognize them. The first control message contains the most significant byte of *i(k)value* argument while the second message contains the less significant byte. *i(k)msb* and *i(k)lsb* are the number of the most and less significant controller.

outipb and **outkpb** (i- and k-rate pitch bend output) send pitch bend messages.

outiat and **outkat** (i- and k-rate aftertouch output) send aftertouch messages. **outiat** and **outkat** (i- and k-rate aftertouch output) send aftertouch messages.

outipc and **outkpc** (i- and k-rate program change output) send program change messages.

outipat and **outkpat** (i- and k-rate polyphonic aftertouch output) send polyphonic aftertouch messages.

These opcodes can drive a different value of a parameter for each note currently active. They work only with MIDI instruments which recognize them.

N.B. All these opcodes can scale the *i(k)value* floating-point argument according with *i(k)max* and *i(k)min* values. For example, setting *i(k)min* = 1.0 and *i(k)max* = 2.0, when *i(k)value* argument receives a 2.0 value, the opcode will send a 127 value to MIDI OUT device, while when receiving a 1.0 it will send a 0 value. i-rate opcodes send their message once during instrument initialization. k-rate opcodes send a message each time the MIDI converted value of argument *i(k)value* changes.

DEPRECATED NAMES

Prior to Csound version 3.52, these opcodes were named **ioutc**, **koutc**, **ioutc14**, **koutc14**, **ioutpb, koutpb**, **ioutat**, **koutat**, **ioutpc**, **koutpc**, **ioutpat**, and **koutpat**. The current names were adopted with version 3.52 (February, 1999) to avoid name space pollution.

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.47

29.2 nrpn

nrpn kchan, kparmnum, kparmvalue

DESCRIPTION

Sends a NPRN (Non Registered Parameter Number) message to the MIDI OUT port each time one of the input arguments changes.

PERFORMANCE

kchan – MIDI channel

kparmnum – number of NRPN parameter

kparmvalue – value of NRPN parameter

This opcode sends new message when the MIDI translated value of one of the input arguments changes. It operates at k-rate. Useful with the MIDI instruments that recognize NRPNs (for example with the newest sound-cards with internal MIDI synthesizer such as SB AWE32, AWE64, GUS etc. in which each patch parameter can be changed during the performance via NRPN)

AUTHOR

Gabriel Maldonado
Italy
1998 (New in Csound version 3.492)

29.3 mdelay

mdelay kstatus, kchan, kd1, kd2, kdelay

DESCRIPTION

A MIDI delay opcode.

PERFORMANCE

kstatus – status byte of MIDI message to be delayed

kchan – MIDI channel (1-16)

kd1 – first MIDI data byte

kd2 – second MIDI data byte

kdelay – delay time in seconds

Each time that *kstatus* is other than zero, **mdelay** outputs a MIDI message to the MIDI out port after *kdelay* seconds. This opcode is useful in implementing MIDI delays. Several instances of **mdelay** can be present in the same instrument with different argument values, so complex and colorful MIDI echoes can be implemented. Further, the delay time can be changed at k-rate.

AUTHOR

Gabriel Maldonado
Italy
November, 1998 (New in Csound version 3.492)

30 MIDI Support: Real-time Messages

30.1 mclock, mrtmsg

mclock ifreq
mrtmsg imsgtype

Description

Send system real-time messages to the MIDI OUT port.

Initialization

ifreq – clock message frequency rate in Hz

imsgtype – type of real-time message:

- 1 sends a START message (0xFA)
- 2 sends a CONTINUE message (0xFB)
- 0 sends a STOP message (0xFC)
- -1 sends a SYSTEM RESET message (0xFF)
- -2 sends an ACTIVE SENSING message (0xFE)

Performance

mclock (MIDI clock) sends a MIDI CLOCK message (0xF8) every 1/*ifreq* seconds. So *ifreq* is the frequency rate of CLOCK message in Hz.

mrtmsg (MIDI real-time message) sends a real-time message once, in init stage of current instrument. *imsgtype* parameter is a flag to indicate the message type (see above).

Author

Gabriel Maldonado
Italy
New in Csound version 3.47

This page intentionally left blank.

31 MIDI Support: Event Extenders

31.1 xtratim, release

```
          xtratim     iextradur
kflag     release
```

Description

Extend the duration of real-time generated events and handle their extra life (see also **linenr**).

Initialization

iextradur – additional duration of current instrument instance

Performance

xtratim extends current MIDI-activated note duration of *iextradur* seconds after the corresponding note-off message has deactivated current note itself. This opcode has no output arguments.

release outputs current note state. If current note is in the **release** stage (i.e. if its duration has been extended with **xtratim** opcode and if it has only just deactivated), *kflag* output argument is set to 1, else (in sustain stage of current note) is set to 0. These two opcodes are useful for implementing complex release-oriented envelopes.

Example

```
          instr 1 ;allows complex ADSR envelope with MIDI events
inum      notnum
icps      cpsmidi
iamp      ampmidi         4000
;
;------- complex envelope block ------
          xtratim         1 ;extra-time, i.e. release dur
krel      init            0
krel      release           ;outputs release-stage flag (0 or 1 values)
if (krel  .5) kgoto rel     ;if in release-stage goto release section
;
;************ attack and sustain section ***********
kmp1      linseg          0, .03, 1, .05, 1, .07, 0, .08, .5, 4, 1, 50, 1
kmp       =               kmp1*iamp
          kgoto done
;
;--------- release section --------
rel:
kmp2      linseg          1, .3, .2, .7, 0
kmp       =               kmp1*kmp2*iamp
done:
;------
   a1     oscili          kmp, icps, 1
          out             a1
          endin
```

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.47

---

32 SIGNAL GENERATORS: LINEAR AND EXPONENTIAL GENERATORS

32.1 line, expon, linseg, linsegr, expseg, expsegr, expsega

```
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
ar        expsega     ia, idur1, ib[, idur2, ic[...]]
```

DESCRIPTION

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

INITIALIZATION

ia- starting value. Zero is illegal for exponentials.

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

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

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

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 cause the last-defined segment to continue on in the same direction.

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.

expsega is almost identical to **expseg**, but more precise when defining segments with very short durations (i.e., in a percussive attack phase) at audio rate. Note that **expseg** does not operate correctly at audio rate when segments are shorter than a k-period. In this situation, **expsega** should be used instead of **expseg**.

EXAMPLE

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

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

AUTHOR

Gabriel Maldonado (**expsega**)
Italy
June, 1998
New in Csound version 3.57

32.2 adsr, madsr, xadsr, mxadsr

```
kr      adsr        iatt, idec, islev, irel[, idel]
kr      madsr       iatt, idec, islev, irel[, idel]
kr      xadsr       iatt, idec, islev, irel[, idel]
kr      mxadsr      iatt, idec, islev, irel[, idel]
ar      adsr        iatt, idec, islev, irel[, idel]
ar      madsr       iatt, idec, islev, irel[, idel]
ar      xadsr       iatt, idec, islev, irel[, idel]
ar      mxadsr      iatt, idec, islev, irel[, idel]
```

DESCRIPTION

Calculates the classical ADSR envelope

INITIALIZATION

iatt – duration of attack phase

idec – duration of decay

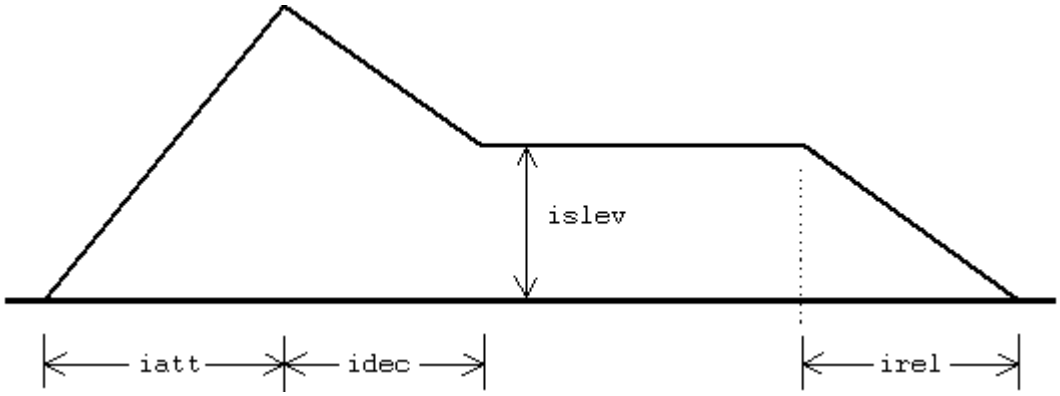
islev – level for sustain phase

irel – duration of release phase

idel – period of zero before the envelope starts

PERFORMANCE

The envelope is the range 0 to 1 and may need to be scaled further. The envelope may be described as:



The length of the sustain is calculated from the length of the note. This means **adsr** is not suitable for use with MIDI events. The opcode **madsr** uses the **linsegr** mechanism, and so can be used in MIDI applications. The opcodes **xadsr** and **mxadsr** are identical to **adsr** and **madsr**, respectively, except they use exponential, rather than linear, line segments. **adsr** and **madsr** new in Csound version 3.49. **xadsr** and **mxadsr** new in Csound version 3.51.

32.3 transeg

```
kr      transeg     ibeg, idur, itype, ival
ar      transeg     ibeg, idur, itype, ival
```

DESCRIPTION

Constructs a user-definable envelope.

INITIALIZATION

ibeg – starting value

ival – value after dur seconds

idur – duration in seconds of segment

itype – if 0, a straight line is produced. If non-zero, then transeg creates the following curve, for *n* steps:

```
ibeg+(ivalue-ibeg)*(1-exp(i*itype/(n-1)))/(1-exp(itype))
```

PERFORMANCE

If *itype* > 0, there is a slowly rising, fast decaying (convex) curve, while if *itype* < 0, the curve is fast rising, slowly decaying (concave). See also **GEN16**.

AUTHOR

John ffitch
University of Bath, Codemist. Ltd.
Bath, UK
October, 2000
New in Csound version 4.09

33 SIGNAL GENERATORS: TABLE ACCESS

33.1 table, tablei, table3, oscil1, oscil1i, osciln

```
ir      table       indx, ifn[, ixmode[, ixoff[, iwrap]]]
ir      tablei      indx, ifn[, ixmode[, ixoff[, iwrap]]]
ir      table3      indx, ifn[, ixmode[, ixoff[, iwrap]]]
kr      table       kndx, ifn[, ixmode[, ixoff[, iwrap]]]
kr      tablei      kndx, ifn[, ixmode[, ixoff[, iwrap]]]
kr      table3      kndx, ifn[, ixmode[, ixoff[, iwrap]]]
ar      table       andx, ifn[, ixmode[, ixoff[, iwrap]]]
ar      tablei      andx, ifn[, ixmode[, ixoff[, iwrap]]]
ar      table3      andx, ifn[, ixmode[, ixoff[, iwrap]]]
kr      oscil1      idel, kamp, idur, ifn
kr      oscil1i     idel, kamp, idur, ifn
ar      osciln      kamp, ifrq, ifn, itimes
```

DESCRIPTION

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) – index data mode. The default value is 0.

- 0 = raw index
- 1 = normalized (0 to 1)

ixoff (optional) – amount by which index 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 index flag. The default value is 0.

- 0 = nowrap (index < 0 treated as index=0; index tablesize sticks at index=size)
- 1 = wraparound

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 on behalf of init, control or audio indices. These indices can be raw entry 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 index is likely to be full scale, or if interpolation is being used, the table should have an extended guard point. **table** indexed by a periodic phasor (see **phasor**) will simulate an oscillator.

oscil1 accesses values by sampling once through the function table at a rate determined by *idur*. For the first *idel* seconds, the point of scan will reside at the first location of the table; it will then begin moving through the table at a constant rate, reaching the end in

another *idur* seconds; from that time on (i.e. after *idel* + *idur* seconds) it will remain pointing at the last location. Each value obtained from sampling is then multiplied by an amplitude factor *kamp* before being written into the result. Because **oscil1** is an interpolating opcode, the table it reads should have a guard point.

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 index is used to interpolate between adjacent table entries. The smoothness gained by interpolation is at some small cost in execution time (see also **oscili**, etc.), but the interpolating and non-interpolating units are otherwise interchangeable. Note that when **tablei** uses a periodic index whose modulo *n* is less than the power of 2 table length, the interpolation process requires that there be an (*n*+ 1)th table value that is a repeat of the 1st (see **f Statement** in score). **table3** is experimental, and is identical to **tablei**, except that it uses cubic interpolation. (New in Csound version 3.50.)

---

34 SIGNAL GENERATORS: PHASORS

34.1 phasor

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

DESCRIPTION

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 *kcps* or *xcps* 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
```

34.2 phasorbnk

```
kr      phasorbnk  kcps, kndx, icnt[, iphs]
ar      phasorbnk  xcps, kndx, icnt[, iphs]
```

DESCRIPTION

Produce an arbitrary number of normalized moving phase values, accessable by an index.

34.2.1 INITIALIZATION

icnt – maximum number of phasors to be used.

iphs – initial phase, expressed as a fraction of a cycle (0 to 1). If -1 initialization is skipped. If *iphas*>1 each phasor will be initialized with a random value.

34.2.2 PERFORMANCE

kndx – index value to access individual phasors

For each independent phasor, an internal phase is successively accumulated in accordance with the *kcps* or *xcps* frequency to produce a moving phase value, normalized to lie in the range 0 <= phs < 1. Each individual phasor is accessed by index *kndx*.

This phasor bank can be used inside a k-rate loop to generate multiple independent voices, or together with the **adsynt** opcode to change parameters in the tables used by **adsynt**.

34.2.3 EXAMPLE

Generate multiple voices with independent partials. This example is better with **adsynt**. See also the example under **adsynt**, for k-rate use of **phasorbnk**.

```
  giwave     ftgen          1, 0, 1024, 10, 1    ; generate a sinewave table

             instr 1
  icnt       =              10                   ; generate 10 voices
  asum       =              0             ; empty output buffer
  kindex     =              0                    ; reset loop index
loop:                                     ; loop executed every k-cycle

  kcps       =              (kindex+1)*100 + 30          ; non-harmonic partials
  aphas      phasorbnk      kcps, kindex, icnt           ; get phase for each voice
  asig       table          aphas, giwave, 1             ; and read wave from table
  asum       =              asum + asig          ; accumulate output

  kindex     =              kindex + 1
  if (kindex < icnt)        kgoto loop           ; do loop

             out            asum*3000
             endin
```

AUTHOR

Peter Neubäcker
Munich, Germany
August, 1999
New in Csound version 3.58

---

35 SIGNAL GENERATORS: BASIC OSCILLATORS

35.1 oscil, oscili, oscil3

```
kr      oscil       kamp, kcps, ifn[, iphs]
kr      oscili      kamp, kcps, ifn[, iphs]
kr      oscil3      kamp, kcps, ifn[, iphs]
ar      oscil       xamp, xcps, ifn[, iphs]
ar      oscili      xamp, xcps, ifn[, iphs]
ar      oscil3      xamp, xcps, ifn[, iphs]
```

DESCRIPTION

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

INITIALIZATION

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

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

PERFORMANCE

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

oscili differs from **oscil** 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. **oscil3** is experimental, and is identical to **oscili**, except that it uses cubic interpolation. (New in Csound version 3.50.)

EXAMPLE

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

35.2 poscil, poscil3

```
ar      poscil      kamp, kcps, ifn [,iphs]
kr      poscil      kamp, kcps, ifn [,iphs]
ar      poscil3     kamp, kcps, ifn [,iphs]
kr      poscil3     kamp, kcps, ifn [,iphs]
```

DESCRIPTION

High precision oscillators. **poscil3** uses cubic interpolation.

INITIALIZATION

ifn – function table number

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

PERFORMANCE

ar – output signal

kamp – amplitude

kcps – frequency

poscil (precise oscillator) is the same as **oscili**, but allows much more precise frequency control, especially when using long tables and low frequency values. It uses floating-point table indexing, instead of integer math, like **oscil** and **oscili**. It is only a bit slower than **oscili**.

AUTHORS

Gabriel Maldonado (**poscil**)
Italy
1998 (New in Csound version 3.52)

John ffitch (**poscil3**)
University of Bath/Codemist Ltd.
Bath, UK
February, 1999 (New in Csound version 3.52)

35.3 lfo

```
kr      lfo         kamp, kcps[, itype]
ar      lfo         kamp, kcps[, itype]
```

DESCRIPTION

A low frequency oscillator of various shapes.

INITIALIZATION

itype -- determine the waveform of the oscillator. Default is 0.

- 0: sine
- 1: triangles
- 2: square (bipolar)
- 3: square (unipolar)
- 4: saw-tooth
- 5: saw-tooth(down)

The sine wave is implemented as a 4096 table and linear interpolation. The others are calculated.

PERFORMANCE

kamp – amplitude of output

kcps – frequency of oscillator

EXAMPLE

```
            instr 1
kp          lfo             10, 5, 4
ar          oscil           p4, p5+kp, 1
            out             ar
            endin
```

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
November, 1998 (New in Csound version 3.491)

This page intentionally left blank.

36 SIGNAL GENERATORS: DYNAMIC SPECTRUM OSCILLATORS

36.1 buzz, gbuzz

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

DESCRIPTION

Output is a set of harmonically related cosine partials.

INITIALIZATION

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

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

PERFORMANCE

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. New in Csound version 3.57, *knh* defaults to one. If *knh* is negative, the absolute value is used.

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

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

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

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

36.2 vco

```
ar      vco         kamp, kfqc[, iwave][, ipw][, ifn][, imaxd]
```

Description

Implementation of a band limited, analog modeled oscillator, based on integration of band limited impulses. **vco** can be used to simulate a variety of analog wave forms. Last four arguments were made optional in Csound version 4.10.

Initialization

iwave (optional) – determines the waveform :

- 1: sawtooth
- 2: Square/PWM
- 3: triangle/Saw/Ramp

iwave defaults to 1.

ipw (optional) – determines the pulse width when *iwave* is set to 2, and determines Saw/Ramp character when *iwave* is set to 3. The value of *ipw* should be between 0 and 1. A value of .5 will generate a square wave or a triangle wave depending on *iwave*. Default is 1.

ifn (optional) – the table number of a of a stored sine wave. Default is 1.

imaxd (optional) – is the maximum delay time. A time of 1/ifqc may be required for the pwm and triangle waveform. To bend the pitch down this value must be as large as 1/(minimum frequency). Default is 1.

Performance

kamp – determines the amplitude

kfqc – is the frequency of the wave

Example

```
          instr 10
idur      =             p3 ; Duration
iamp      =             p4 ; Amplitude
ifqc      =             cpspch(p5)      ; Frequency
iwave     =             p6              ; Selected wave form 1=Saw,
                                          ; 2=Square/PWM, 3=Tri/Saw-Ramp-Mod
isine     =             1
imaxd     =             1/ifqc*2 ; Allows pitch bend down of two octaves
kpw1      oscil         .25, ifqc/200, 1
kpw       =             kpw1 + .5
asig      vco           iamp, ifqc, iwave, kpw, 1, imaxd
          outs          asig, asig ; Output and amplification
          endin
```

```
f1 0 65536 10 1

  ; Sta Dur Amp   Pitch Wave
  i10 0 2   20000 5.00  1
  i10 + .   .     . 2
  i10 . .   .     . 3
  i10 . 2   20000 7.00  1
  i10 . .   .     . 2
  i10 . .   .     . 3
  i10 . 2   20000 9.00  1
  i10 . .   .     . 2
  i10 . .   .     . 3
  i10 . 2   20000 11.00 1
  i10 . .   .     .     2
  i10 . .   .     .     3
  e
```

AUTHOR

Hans Mikelson
December, 1998 (New in Csound version 3.50)

36.3 mpulse

```
ar      mpulse      kamp, kfreq[, ioffset]
```

DESCRIPTION

Generate a set of impulses. of amplitude *kamp* at frequency *kfreq*. The first impulse is after a delay of *ioffset* seconds. The value of *kfreq* is read only after an impulse, so it is the interval to the next impulse at the time of an impulse.

INITIALIZATION

ioffset – the delay before the first impulse. If it is negative, the value is taken as the number of samples, otherwise it is in seconds. Default is zero.

PERFORMANCE

kamp – amplitude of the impulses generated

kfreq – frequency of the impulse train

After the initial delay, an impulse of *kamp* amplitude is generated as a single sample. Immediately after generating the impulse, the time of the next one is calculated. If *kfreq* is zero, there is an infinite wait to the next impulse. If *kfreq* is negative, the frequency is counted in samples rather than seconds.

EXAMPLE

Generate a set of impulses at 10 a second, after a delay of 0.05s

```
            instr       1
a1          mpulse      32000, 0.1, 0.05
            out         a1
            endin
```

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
September, 2000 (New in Csound version 4.08)

This page intentionally left blank.

37 SIGNAL GENERATORS: ADDITIVE SYNTHESIS/RESYNTHESIS

37.1 adsyn

```
ar      adsyn       kamod, kfmod, ksmod, ifilcod
```

DESCRIPTION

Output is an additive set of individually controlled sinusoids using an oscillator bank.

INITIALIZATION

ifilcod – integer or character-string denoting a control-file derived from analysis of an audio signal. An integer denotes the suffix of a file *adsyn.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. 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. Memory usage depends on the size of the file involved, which are read and held entirely in memory during computation but are shared by multiple calls.

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 **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 (limit removed in version 3.47) 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 frequency and *ksmod* modifying the *speed* with which the millisecond breakpoint 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.

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.

37.2 adsynt

```
ar      adsynt      kamp, kcps, iwfn, ifreqfn, iampfn, icnt[, iphs]
```

DESCRIPTION

Performs additive synthesis with an arbitrary number of partials, not necessarily harmonic.

INITIALIZATION

iwfn – table containing a waveform, usually a sine. Table values are not interpolated for performance reasons, so larger tables provide better quality.

ifreqfn – table containing frequency values for each partial. *ifreqfn* may contain beginning frequency values for each partial, but is usually used for generating parameters at runtime with **tablew**. Frequencies must be relative to *kcps*. Size must be at least *icnt*.

iampfn – table containing amplitude values for each partial. *iampfn* may contain beginning amplitude values for each partial, but is usually used for generating parameters at runtime with **tablew**. Amplitudes must be relative to *kamp*. Size must be at least *icnt*.

icnt – number of partials to be generated

iphs – initial phase of each oscillator, if *iphs* = -1, initialization is skipped. If *iphs* > 1, all phases will be initialized with a random value.

PERFORMANCE

kamp – amplitude of note

kcps – base frequency of note. Partial frequencies will be relative to *kcps*.

Frequency and amplitude of each partial is given in the two tables provided. The purpose of this opcode is to have an instrument generate synthesis parameters at k-rate and write them to global parameter tables with the **tablew** opcode.

EXAMPLES

These two instruments perform additive synthesis. The output of each sounds like a Tibetan bowl. The first one is static, as parameters are only generated at init-time. In the second one, parameters are continuously changed.

```
gifrqs     ftgen          2, 0, 32, 7, 0, 32, 0       ; generate two emty tables
                                    ; for adsynt
giamps     ftgen          3, 0, 32, 7, 0, 32, 0       ; for freqency and amp
                                    ; parameters

           instr 1              ; generates parameters at init time
icnt       =              10           ; generate 10 voices
index      =              0            ; init loop index
```

```
loop:                                   ; loop only executed at
                                        ; init time
  ifreq     pow           index + 1, 1.5       ; define non-harmonic partials
  iamp      =             1 / (index+1) ; define amplitudes
            tableiw       ifreq, index, gifrqs ; write to tables
            tableiw       iamp, index, giamps  ; used by adsynt
  index     =             index + 1
  if        (index < icnt) igoto loop    ; do loop

  asig      adsynt        5000, 150, giwave, gifrqs, giamps, icnt
            out           asig
endin


            instr 2                     ; generates paramaters
                                        ; every k-cycle
  icnt      =             10            ; generate 10 voices
  kindex    =             0             ; reset loop index
loop:                                   ; loop executed every
                                        ; k-cycle
  kspeed    pow           kindex + 1, 1.6      ; generate lfo for
                                        ; frequencies
  kphas     phasorbnk     kspeed * 0.7, kindex, icnt   ; individual phase for each
voice
  klfo      table         kphas, giwave, 1
  kdepth    pow           1.4, kindex   ; arbitrary parameter twiddling...
  kfreq     pow           kindex + 1, 1.5
  kfreq     =             kfreq + klfo*0.006*kdepth
            tablew        kfreq, kindex, gifrqs       ; write freqs to table for
                                        ; adsynt

  kspeed    pow           kindex + 1, 0.8      ; generate lfo for amplitudes
  kphas     phasorbnk     kspeed*0.13, kindex, icnt, 2 ; individual phase for
                                        ; each voice
  klfo      table         kphas, giwave, 1
  kamp      pow           1 / (kindex + 1), 0.4       ; arbitrary parameter
                                        ; twiddling...
  kamp      =             kamp * (0.3+0.35*(klfo+1))
            tablew        kamp, kindex, giamps ; write amps to table for
                                        ; adsynt

  kindex    =             kindex + 1
  if        (kindex < icnt) kgoto loop   ; do loop
  giwave    ftgen         1, 0, 1024, 10, 1    ; generate a sinewave
                                        ; table
  asig      adsynt        5000, 150, giwave, gifrqs, giamps, icnt
            out           asig
endin
```

AUTHOR

Peter Neubäcker
Munich, Germany
August, 1999
New in Csound version 3.58

37.3 hsboscil

```
ar      hsboscil    kamp, ktone, kbrite, ibasfreq, iwfn, ioctfn \\
                    [, ioctcnt[, iphs]]
```

DESCRIPTION

An oscillator which takes tonality and brightness as arguments, relative to a base frequency.

INITIALIZATION

ibasfreq – base frequency to which tonality and brighness are relative

iwfn – function table of the waveform, usually a sine

ioctfn – function table used for weighting the octaves, usually something like:

```
f1 0  1024  -19  1  0.5  270  0.5
```

ioctcnt – number of octaves used for brightness blending. Must be in the range 2 to 10. Default is 3.

iphs – initial phase of the oscillator. If *iphs* = -1, initialization is skipped.

PERFORMANCE

kamp – amplitude of note

ktone – cyclic tonality parameter relative to *ibasfreq* in logarithmic octave, range 0 to 1, values > 1 can be used, and are internally reduced to **frac**(*ktone*).

kbrite – brightness parameter relative to *ibasfreq*, achieved by weighting *ioctcnt* octaves. It is scaled in such a way, that a value of 0 corresponds to the orignal value of *ibasfreq*, 1 corresponds to one octave above *ibasfreq*, -2 corresponds to two octaves below *ibasfreq*, etc. *kbrite* may be fractional.

hsboscil takes tonality and brightness as arguments, relative to a base frequency (*ibasfreq*). Tonality is a cyclic parameter in the logarithmic octave, brightness is realized by mixing multiple weighted octaves. It is useful when tone space is understood in a concept of polar coordinates.

Making *ktone* a line, and *kbrite* a constant, produces Risset's glissando.

Oscillator table *iwfn* is always read interpolated. Performance time requires about *ioctcnt* * **oscili**.

EXAMPLES

```
  giwave     ftgen      1, 0, 1024, 10, 1, 1, 1, 1         ; synth wave
  giblend    ftgen      2, 0, 1024, -19, 1, 0.5, 270, 0.5  ; blending window

             instr 1        ; endless glissando
  ktona      line       0,10,1
  asig       hsboscil   10000, ktona, 0, 200, giwave, giblend, 5
             out        asig
endin

             instr 2        ; MIDI instrument: all octaves sound alike,
  itona      octmidi          ; velocity is mapped to brightness
  ibrite     ampmidi    3
  ibase      =          cpsoct(6)
  kenv       expon      20000, 1, 100
  asig       hsboscil   kenv, itona, ibrite, ibase, giwave, giblend, 5
           out        asig
endin
```

AUTHOR

Peter Neubäcker
Munich, Germany
August, 1999
New in Csound version 3.58

38 SIGNAL GENERATORS: FM SYNTHESIS

38.1 foscil, foscili

```
ar      foscil      xamp, kcps, xcar, xmod, kndx, ifn[, iphs]
ar      foscili     xamp, kcps, xcar, xmod, kndx, ifn[, iphs]
```

DESCRIPTION

Basic frequency modulated oscillators.

INITIALIZATION

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

iphs (optional) – initial phase of waveform in table *ifn*, 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.

38.1.1 PERFORMANCE

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* * *xmod*. For integral values of *xcar* and *xmod*, the perceived fundamental will be the minimum positive value of *kcps* * (xcar – n * xmod), n = 1,1,2,...* 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. Previous to version 3.50, *xcar* and *xmod* could be k-rate only.

foscili differs from **foscil** 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.

38.2 fmvoice

```
ar      fmvoice     kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate, ifn1, \\
                    ifn2, ifn3, ifn4, ivibfn
```

DESCRIPTION

FM Singing Voice Synthesis

INITIALIZATION

ifn1, ifn2, ifn3,ifn3 -- Tables, usually of sinewaves.

PERFORMANCE

kamp – Amplitude of note.

kfreq – Frequency of note played.

kvowel -- the vowel being sung, in the range 0-64

ktilt -- the spectral tilt of the sound in the range 0 to 99

kvibamt -- Depth of vibrato

kvibrate -- Rate of vibrato

EXAMPLE

```
k1          line            0, p3, 64
a1          fmvoice         31129.60, 110, k1, 0, 0.005, 6, 1,1,1,1,1
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

38.3 fmbell, fmrhode, fmwurlie, fmmetal, fmb3, fmpercfl

```
a1      fmbell      kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2,\\
                    ifn3, ifn4, ivfn
a1      fmrhode     kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2,\\
                    ifn3, ifn4, ivfn
a1      fmwurlie    kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2,\\
                    ifn3, ifn4, ivfn
a1      fmmetal     kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2,\\
                    ifn3, ifn4, ivfn
a1      fmb3        kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2,\\
                    ifn3, ifn4, ivfn
a1      fmpercfl    kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2,\\
                    ifn3, ifn4, ivfn
```

DESCRIPTION

A family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

INITIALIZATION

All these opcodes take 5 tables for initialization. The first 4 are the basic inputs and the last is the low frequency oscillator (LFO) used for vibrato. The last table should usually be a sine wave.

For the other opcodes the initial waves should be as in the table:

	ifn1	ifn2	ifn3	ifn4
fmbell	sinewave	sinewave	sinewave	sinewave
fmrhode	sinewave	sinewave	sinewave	fwavblnk
fmwurlie	sinewave	sinewave	sinewave	fwavblnk
fmmetal	sinewave	twopeaks	twopeaks	sinewave
fmb3	sinewave	sinewave	sinewave	sinewave
fmpercfl	sinewave	sinewave	sinewave	sinewave

The sounds produced are then:

fmbell Tubular Bell
fmrhode Fender Rhodes Electric Piano
fmwurlie Wurlitzer Electric Piano
fmmetal "Heavy Metal"
fmb3 Hammond B3 organ
fmpercfl Percussive Flute

PERFORMANCE

kamp – Amplitude of note.

kfreq – Frequency of note played.

kc1, kc2 -- Controls for the synthesizer, as in the table:

kc1	kc2		Algorithm
fmbell	Mod index 1	Crossfade of two outputs	5
fmrhode	Mod index 1	Crossfade of two outputs	5
fmwurlie	Mod index 1	Crossfade of two outputs	5
fmmetal	Total mod index	Crossfade of two modulators	3
fmb3	Total mod index	Crossfade of two modulators	4
fmpercfl	Total mod index	Crossfade of two modulators	4

kvdepth -- Vibrator depth
kvrate -- Vibrator rate

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

39 Signal Generators: Sample Playback

39.1 loscil, loscil3

```
ar[,ar2] loscil      xamp, kcps, ifn[, ibas[,imod1,ibeg1,iend1 \\
                       [, imod2,ibeg2,iend2]]]
ar[,ar2] loscil3     xamp, kcps, ifn[, ibas[,imod1,ibeg1,iend1 \\
                       [, imod2,ibeg2,iend2]]]
```

Description

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 Hz of the recorded sound. This optionally overrides the frequency given in the AIFF file, but is required if the file did not contain one. The default value is 261.626 Hz, i.e. middle C. (New in Csound 4.03).

imod1, imod2 (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.

loscil3 is experimental. It is identical to **loscil**, except that it uses cubic interpolation. New in Csound version 3.50.

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, .01 ;at noteoff, extend by 50 ms.
asig      loscil        kamp, icps, ifno, cpsoct(ibas/12. + 3)
```

39.2 lposcil, lposcil3

```
ar      lposcil     kamp, kfreqratio, kloop, kend, ifn [,iphs]
ar      lposcil3    kamp, kfreqratio, kloop, kend, ifn [,iphs]
```

DESCRIPTION

Read sampled sound (mono or stereo) from a table, with optional sustain and release looping, and high precision. **lposcil3** uses cubic interpolation.

INITIALIZATION

ifn – function table number

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

PERFORMANCE

kamp – amplitude

kfreqratio – multiply factor of table frequency (for example: 1 = original frequency, 1.5 = a fifth up , .5 = an octave down)

kloop – loop point (in samples)

kend – end loop point (in samples)

lposcil (looping precise oscillator) allows varying at k-rate, the starting and ending point of a sample contained in a table (**GEN01**). This can be useful when reading a sampled loop of a wavetable, where repeat speed can be varied during the performance.

AUTHORS

Gabriel Maldonado (**lposcil**)
Italy
1998 (New in Csound version 3.52)

John ffitch (**lposcil3**)
University of Bath/Codemist Ltd.
Bath, UK
February, 1999 (New in Csound version 3.52)

39.3 sfload, sfplist, sfilist, sfpassign, sfpreset, sfplay, sfplaym, sfinstr, sfinstrm

```
ir        sfload      "filename"
          sfpassign   istartndx, ifilhandle
ir        sfpreset    iprog, ibank, ifilhandle, iprendx

          sfplist     ifilhandle
          sfilist     ifilhandle

a1, a2    sfplay      ivel, inotnum, xamp, xfreq, iprendx [, iflag]
`a1       sfplaym     ivel, inotnum, xamp, xfreq, iprendx [, iflag]

a1, a2    sfinstr     ivel, inotnum, xamp, xfreq, instrnum, ifilhandle [,
                      iflag]
a1        sfinstrm    ivel, inotnum, xamp, xfreq, instrnum, ifilhandle [,
                      iflag]
```

DESCRIPTION

Csound support for the SoundFont2 (SF2) sample file format. These opcodes allow management the sample-structure of SF2 files. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format can be found in the Appendix.

Note that **sfload**, **sfpassign**, and **sfpreset** are placed in the header section of a Csound orchestra.

INITIALIZATION

ir – output to be used by other SF2 opcodes. For **sfload**, *ir* is *ifilhandle*. For **sfpreset**, *ir* is *iprendx*.

filename – name of the SF2 file, with its complete path. It must be typed within double-quotes. Use "/" to separate directories. This applies to DOS and Windows as well, where using a backslash will generate an error.

ifilhandle – unique number generated by **sfload** opcode to be used as an identifier for a SF2 file. Several SF2 files can be loaded and activated at the same time.

istartndx – starting index preset by the user in bulk preset assignments (see below).

iprendx – preset index

iprog – program number of a bank of presets in a SF2 file

ibank – number of a specific bank of a SF2 file

ivel – velocity value

inotnum – MIDI note number value

iflag – flag regarding the behavior of *xfreq* and *inotnum*

instrnum – number of an instrument of a SF2 file.

PERFORMANCE

xamp – amplitude correction factor

xfreq – frequency value or frequency multiplier, depending by *iflag*. When *iflag* = 0, **xfreq** is a multiplier of a the default frequency, assigned by SF2 preset to the *inotenum* value. When *iflag* = 1, **xfreq** is the absolute frequency of the output sound, in Hz. Default is 0.

sfload loads an entire SF2 file into memory. It returns a file handle to be used by other opcodes. Several instances of **sfload** can placed in the header section of an orchestra, allowing use of more than one SF2 file in a single orchestra.

sfplist prints a list of all presets of a previously loaded SF2 file to the console.

sfilist prints a list of all instruments of a previously loaded SF2 file to the console.

sfpassign assigns all presets of a previously loaded SF2 file to a sequence of progressive index numbers, to be used later with the opcodes **sfplay** and **sfplaym**. *istartndx* specifies the starting index number. Any number of **sfpassign** instances can be placed in the header section of an orchestra, each one assigning presets belonging to different SF2 files. The user must take care that preset index numbers of different SF2 files do not overlap.

sfpreset assigns an existing preset of a previously loaded SF2 file to an index number, to be used later with the opcodes **sfplay** and **sfplaym**. The user must previously know the program and the bank numbers of the preset in order to fill the corresponding arguments. Any number of **sfpreset** instances can be placed in the header section of an orchestra, each one assigning a different preset belonging to the same (or different) SF2 file to different index numbers.

sfplay plays a preset, generating a stereo sound. *ivel* does not directly affect the amplitude of the output, but informs **sfplay** about which sample should be chosen in multi-sample, velocity-split presets.

When *iflag* = 0, *inotnum* sets the frequency of the output according to the MIDI note number used, and *xfreq* is used as a multiplier. When *iflag* = 1, the frequency of the output, is set directly by *xfreq*. This allows the user to use any kind of micro-tuning based scales. However, this method is designed to work correctly only with presets tuned to the default equal temperament. Attempts to use this method with a preset already having non-standard tunings, or with drum-kit-based presets, could give unexpected results.

Adjustment of the amplitude can be done by varying the *xamp* argument, which acts as a multiplier.

Notice that both *xamp* and *xfreq* can use k-rate as well as a-rate signals. Both arguments must use variables of the same rate, or **sfplay** will not work correctly. *iprendx* must contain the number of a previously assigned preset, or Csound will crash.

sfplaym is a mono version of **sfplay**. It should be used with mono preset, or with the stereo presets in which stereo output is not required. It is faster than **sfplay**.

sfinstr plays an SF2 instrument instead of a preset (an SF2 instrument is the base of a preset layer). *instrnum* specifies the instrument number, and the user must be sure that the specified number belongs to an existing instrument of a determinate soundfont bank. Notice that both *xamp* and *xfreq* can operate at k-rate as well as a-rate, but both arguments must work at the same rate.

sfinstrm plays is a mono version of **sfinstr**. This is the fastest opcode of the SF2 family.

These opcodes only support the sample structure of SF2 files. The modulator structure of the SoundFont2 format is not supported in Csound. Any modulation or processing to the sample data is left to the Csound user, bypassing all restrictions forced by the SF2 standard.

AUTHOR

Gabriel Maldonado
Italy
May, 2000 (New in Csound Version 4.06)

---

40 SIGNAL GENERATORS: GRANULAR SYNTHESIS

40.1 fof, fof2

```
ar      fof     xamp, xfund, xform, koct, kband, kris, kdur, kdec,\\
                iolaps, ifna, ifnb, itotdur[, iphs[, ifmode]]
ar      fof2    xamp, xfund, xform, koct, kband, kris, kdur, kdec,\\
                iolaps, ifna, ifnb, itotdur, kphs, kgliss
```

DESCRIPTION

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.

fof2 implements k-rate incremental indexing into *ifna* function with each successive burst.

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.

In the **fof2** implementation, *kphs* allows k-rate indexing of function table *ifna* with each successive burst, making it suitable for time-warping applications. Values of for *kphs* are normalized from 0 to 1, 1 being the end of the function table *ifna*. *kgliss* – sets the end pitch of each grain relative to the initial pitch, in octaves. Thus *kgliss* = 2 means that the grain ends two octaves above its initial pitch, while *kgliss* = -5/3 has the grain ending a perfect major sixth below. **Note**: There are no optional parameters in **fof2**

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.

40.2 fog

```
ar      fog         xamp, xdens, xtrans, xspd, koct, kband, kris, kdur,\\
                    kdec, iolaps, ifna, ifnb, itotdur[, iphs[, itmode]]
```

DESCRIPTION

Audio output is a succession of grains derived from data in a stored function table *ifna*. The local envelope of these grains and their timing is based on the model of **fof** synthesis and permits detailed control of the granular synthesis.

INITIALIZATION

iolaps – number of pre-located spaces needed to hold overlapping grain data. Overlaps are density dependent, and the space required depends on the maximum value of *xdens* kdur*. Can be over-estimated at no computation cost. Uses less than 50 bytes of memory per *iolaps*.

ifna, *ifnb* – table numbers of two stored functions. The first is the data used for granulation, usually from a soundfile (GEN01). The second is a rise shape, used forwards and backwards to shape the grain rise and decay; this is normally a sigmoid (GEN19) but may be linear (GEN07).

itotdur – total time during which this **fog** will be active. Normally set to p3. No new grain 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.

itmode (optional) – transposition mode. If zero, each grain keeps the *xtrans* value it was launched with. if non-zero, each is influenced by *xtrans* continuously. The default value is 0.

PERFORMANCE

xamp – amplitude factor. Amplitude is also dependent on the number of overlapping grains, the interaction of the rise shape (*ifnb*) and the exponential decay (*kband*), and the scaling of the grain waveform (*ifna*). The actual amplitude may therefore exceed *xamp*.

xdens – density. The frequency of grains per second.

xtrans – transposition factor. The rate at which data from the stored function table *ifna* is read within each grain. This has the effect of transposing the original material. A value of 1 produces the original pitch. Higher values transpose upwards, lower values downwards. Negative values result in the function table being read backwards.

xspd – speed. The rate at which successive grains advance through the stored function table *ifna*. *xspd* is in the form of an index (0 to 1) to *ifna*. This determines the movement of a pointer used as the starting point for reading data within each grain. (*xtrans* determines the rate at which data is read starting from this pointer.)

koct – octaviation index. The operation of this parameter is identical to that in **fof**.

kband, *kris*, *kdur*, *kdec* – grain envelope shape. These parameters determine the exponential decay (*kband*), and the rise (*kris*), overall duration (*kdur*,) and decay (*kdec*) times of the grain envelope. Their operation is identical to that of the local envelope parameters in **fof**.

The Csound **fog** generator is by Michael Clarke, extending his earlier work based on IRCAM's fof algorithm.

EXAMPLE

```
  ;p4        = transposition factor
  ;p5        = speed factor
  ;p         = function table for grain data

;scaling to reflect sample rate and table length
  i1         =           sr/ftlen(p6)
  a1         phasor      i1*p5 ;index for speed
  a2         fog         5000, 100, p4, a1, 0, 0, , .01, .02, .01, 2, p6, 1, p3, 0, 1
```

AUTHOR

Michael Clark
Huddersfield
May 1997

40.3 grain

```
ar        grain       xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn,\\
                      iwfn, imgdur [, igrnd]
```

DESCRIPTION

Generates granular synthesis textures.

INITIALIZATION

igfn – The ftable number of the grain waveform. This can be just a sine wave or a sampled sound.

iwfn – Ftable number of the amplitude envelope used for the grains (see also **GEN20**).

imgdur – Maximum grain duration in seconds. This the biggest value to be assigned to *kgdur*.

igrn – (optional) if non-zero, turns off grain offset randomness. This means that all grains will begin reading from the beginning of the *igfn* table. If zero (the default), grains will start reading from random *igfn* table positions.

PERFORMANCE

xamp – Amplitude of each grain.

xpitch – Grain pitch. To use the original frequency of the input sound, use the formula: sndsr / ftlen(*igfn*) where sndsr is the original sample rate of the *igfn* sound.

xdens – Density of grains measured in grains per second. If this is constant then the output is synchronous granular synthesis, very similar to **fof**. If *xdens* has a random element (like added noise), then the result is more like asynchronous granular synthesis.

kampoff – Maximum amplitude deviation from *kamp*. This means that the maximum amplitude a grain can have is *kamp* + *kampoff* and the minimum is *kamp*. If *kampoff* is set to zero then there is no random amplitude for each grain.

kpitchoff – Maximum pitch deviation from *kpitch* in Hz. Similar to *kampoff*.

kgdur – Grain duration in seconds. The maximum value for this should be declared in *imgdur*. If *kgdur* at any point becomes greater than *imgdur*, it will be truncated to *imgdur*.

The grain generator is based primarily on work and writings of Barry Truax and Curtis Roads.

EXAMPLE

A texture with gradually shorter grains and wider amp and pitch spread

```
;;;;;;;;;;;;;;; graintest.orc
instr 1
    insnd = 10
    ibasfrq = 32000 / ftlen(insnd)   ; Use original sample rate of insnd file
    kamp   expseg 8000, p3/2, 8000, p3/2, 16000
    kpitch line ibasfrq, p3, ibasfrq * .8
    kdens  line 600, p3, 200
    kaoff  line 0, p3, 5000
    kpoff  line 0, p3, ibasfrq * .5
    kgdur  line .4, p3, .1
    imaxgdur =  .5
```

```
  ar  grain kamp, kpitch, kdens, kaoff, kpoff, kgdur, insnd, 5, imaxgdur, 0.0
 out ar
endin
;;;;;;;;;;;;;;; graintest.sco
f5  0 512  20 2                  ; Hanning window
f10 0 65536 1  “Sound.wav” 0 0 0
i1 0 10
e
```

AUTHOR

Paris Smaragdis
MIT
May 1997

40.4 granule

```
asig     granule     xamp, ivoice, iratio, imode, ithd, ifn, ipshift,\\
                     igskip, igskip_os, ilength, kgap, igap_os, kgsize,\\
                     igsize_os, iatt, idec [,iseed[,ipitch1[,ipitch2\\
                     [,ipitch3[,ipitch4[,ifnenv]]]]]]
```

DESCRIPTION

The **granule** unit generator is more complex than **grain**, but does add new possibilities.

granule is a Csound unit generator which employs a wavetable as input to produce granularly synthesized audio output. Wavetable data may be generated by any of the GEN subroutines such as GEN01 which reads an audio data file into a wavetable. This enable a sampled sound to be used as the source for the grains. Up to 128 voices are implemented internally. The maximum number of voices can be increased by redefining the variable MAXVOICE in the grain4.h file. **granule** has a build-in random number generator to handle all the random offset parameters. Thresholding is also implemented to scan the source function table at initialization stage. This facilitates features such as skipping silence passage between sentences.

The characteristics of the synthesis are controlled by 22 parameters. *xamp* is the amplitude of the output and it can be either audio rate or control rate variable.

PERFORMANCE

xamp – amplitude.

ivoice – number of voices.

iratio – ratio of the speed of the gskip pointer relative to output audio sample rate. e.g. 0.5 will be half speed.

imode – +1 grain pointer move forward (same direction of the gskip pointer), -1 backward (oppose direction to the gskip pointer) or 0 for random.

ithd – threshold, if the sampled signal in the wavetable is smaller then *ithd*, it will be skipped.

ifn – function table number of sound source.

ipshift – pitch shift control. If *ipshift* is 0, pitch will be set randomly up and down an octave. If *ipshift* is 1, 2, 3 or 4, up to four different pitches can be set amount the number of voices defined in *ivoice*. The optional parameters *ipitch1*, *ipitch2*, *ipitch3* and *ipitch4* are used to quantify the pitch shifts.

igskip – initial skip from the beginning of the function table in sec.

igskip_os – gskip pointer random offset in sec, 0 will be no offset.

ilength – length of the table to be used starting from *igskip* in sec.

kgap – gap between grains in sec.

igap_os – gap random offset in % of the gap size, 0 gives no offset.

kgsize – grain size in sec.

igsize_os – grain size random offset in % of grain size, 0 gives no offset.

iatt – attack of the grain envelope in % of grain size.

idec – decade of the grain envelope in % of grain size.

[iseed] – optional, seed for the random number generator, default is 0.5.

[ipitch1], [ipitch2], [ipitch3], [ipitch4]- optional, pitch shift parameter, used when *ipshift* is set to 1, 2, 3 or 4. Time scaling technique is used in pitch shift with linear interpolation between data points. Default value is 1, the original pitch.

EXAMPLE

```
; Orchestra file:
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
             instr 1
;
  k1         linseg   0,0.5,1,(p3-p2-1),1,0.5,0
  a1         granule  p4*k1,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,\
                        p16,p17,p18,p19,p20,p21,p22,p23,p24
  a2         granule  p4*k1,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,\
                        p16,p17,p18,p19, p20+0.17,p21,p22,p23,p24
             outs     a1,a2
             endin

; Score file:
; f statement read sound file sine.aiff in the SFDIR
; directory into f-table 1
f1      0 524288 1 “sine.aiff” 1 0
i1      0 10 2000 64 0.5 0 0 1 4 0 0.005 10 0.01 50 0.02 50 30 30 0.39 \
        1 1.42 0.29 2
e
```

The above example reads a sound file called sine.aiff into wavetable number 1 with 524,288 samples. It generates 10 seconds of stereo audio output using the wavetable. In the orchestra file, all parameters required to control the synthesis are passed from the score file. A linseg function generator is used to generate an envelope with 0.5 second of linear attack and decay. Stereo effect is generated by using different seeds for the two **granule** function calls. In the example, 0.17 is added to p20 before passing into the second **granule** call to ensure that all of the random offset events are different from the first one.

In the score file, the parameters are interpreted as:

```
p5  (ivoice) the number of voices is set to 64
p6  (iratio) is set to 0.5, it scan the wavetable at half of the speed
      of the audio output rate
p7  (imode) is set to 0, the grain pointer only move forward
p8  (ithd) is set to 0, skipping the thresholding process
p9  (ifn) is set to 1, function table number 1 is used
p10 (ipshift) is set to 4, four different pitches are going to be
      generated
p11 (igskip) is set to 0 and p12 (igskip_os) is set to 0.005, no
      skipping into the wavetable and a 5 mSec random offset is used
p13 (ilength) is set to 10, 10 seconds of the wavetable is to be used
p14 (kgap) is set to 0.01 and p15 (igap_os) is set to 50, 10 mSec gap
      with 50% random offset is to be used
p16 (kgsize) is set to 0.02 and p17 (igsize_os) is set to 50, 20 mSec
      grain with 50% random offset is used
p18 (iatt) and p19 (idec) are set to 30, 30% of linear attack and
      decade is applied to the grain
p20 (iseed) seed for the random number generator is set to 0.39
p21 - p 24 are pitches set to 1 which is the original pitch, 1.42
      which is a 5th up, 0.29 which is a 7th down and finally 2 which is
      an octave up.
```

AUTHOR

Allan Lee
Belfast
1996

40.5 sndwarp, sndwarpst

```
ar[,ac]              sndwarp   xamp, xtimewarp, xresample, ifn1, ibeg,\\
                               iwsize, irandw, ioverlap, ifn2, itimemode

ar1,ar2[,ac1,ac2]    sndwarpst xamp, xtimewarp, xresample, ifn1, ibeg,\\
                               iwsize, irandw, ioverlap, ifn2, itimemode
```

DESCRIPTION

sndwarp reads sound samples from a table and applies time-stretching and/or pitch modification. Time and frequency modification are independent from one another. For example, a sound can be stretched in time while raising the pitch! The window size and overlap arguments are important to the result and should be experimented with. In general they should be as small as possible. For example, start with *iwsize*=sr/10 and *ioverlap*=15. Try *irandw=iwsize**.2. If you can get away with less overlaps, the program will be faster. But too few may cause an audible flutter in the amplitude. The algorithm reacts differently depending upon the input sound and there are no fixed rules for the best use in all circumstances. But with proper tuning, excellent results can be achieved.

INITIALIZATION

ifn1 – the number of the table holding the sound samples which will be subjected to the **sndwarp** processing. GEN01 is the appropriate function generator to use to store the sound samples from a pre-existing soundfile.

ibeg – the time in seconds to begin reading in the table (or soundfile). When *itimemode* is non- zero, the value of *xtimewarp* is offset by *ibeg*.

iwsize – the window size in samples used in the time scaling algorithm.

irandw – the bandwidth of a random number generator. The random numbers will be added to *iwsize*.

ioverlap – determines the density of overlapping windows.

ifn2 – a function used to shape the window. It is usually used to create a ramp of some kind from zero at the beginning and back down to zero at the end of each window. Try using a half a sine (i.e.: f1 0 16384 9 .5 1 0) which works quite well. Other shapes can also be used.

PERFORMANCE

ar – single channel of output from the **sndwarp** unit generator while *ar1* and *ar2* are the stereo (left and right) outputs from **sndwarpst**. **sndwarp** assumes that the function table holding the sampled signal is a mono one while **sndwarpst** assumes that it is stereo. This simply means that **sndwarp** will index the table by single-sample frame increments and **sndwarpst** will index the table by a two-sample frame increment. The user must be aware then that if a mono signal is used with sndwarpst or a stereo one with **sndwarp**, time and pitch will be altered accordingly.

ac – in **sndwarp** and *ac1*, *ac2* in **sndwarpst**, are single layer (no overlaps), unwindowed versions of the time and/or pitch altered signal. They are supplied in order to be able to balance the amplitude of the signal output, which typically contains many overlapping and windowed versions of the signal, with a clean version of the time-scaled and pitch-shifted signal. The **sndwarp** process can cause noticeable changes in amplitude, (up and down), due to a time differential between the overlaps when time-shifting is being done. When used with a balance unit, *ac*, *ac1*, *ac2* can greatly enhance the quality of sound. They are

optional, but note that in **sndwarpst** they must both be present in the syntax (use both or neither). An example of how to use this is given below.

xamp – the value by which to scale the amplitude (see note on the use of this when using *ac*, *ac1*, *ac2*).

xtimewarp – determines how the input signal will be stretched or shrunk in time. There are two ways to use this argument depending upon the value given for *itimemode*. When the value of *itimemode* is 0, xitimewarp will scale the time of the sound. For example, a value of 2 will stretch the sound by 2 times. When *itimemode* is any non-zero value then *xtimewarp* is used as a time pointer in a similar way in which the time pointer works in lpread and **pvoc**. An example below illustrates this. In both cases, the pitch will not be altered by this process. Pitch shifting is done independently using *xresample*.

xresample – the factor by which to change the pitch of the sound. For example, a value of 2 will produce a sound one octave higher than the original. The timing of the sound, however, will not be altered.

EXAMPLE

The below example shows a slowing down or stretching of the sound stored in the stored table (*ifn1*). Over the duration of the note, the stretching will grow from no change from the original to a sound which is ten times "slower" than the original. At the same time the overall pitch will move upward over the duration by an octave.

```
iwindfun=1
isampfun=2
ibeg=0
iwindsize=2000
iwindrand=400
ioverlap=10
awarp   line    1, p3, 10
aresamp line    1, p3, 2
kenv    line    1, p3, .1
asig    sndwarp kenv,awarp,aresamp,isampfun,ibeg,iwindsize,iwindrand, \\
                ioverlap,iwindfun,0
```

Now, here's an example using *xtimewarp* as a time pointer and using stereo:

```
itimemode     =         1
atime         line      0, p3, 10
ar1, ar2       sndwarpst kenv, atime, aresamp, sampfun, ibeg, \\
                         iwindsize, iwindrand, ioverlap, \\
                         iwindfun, itimemode
```

In the above, atime advances the time pointer used in the **sndwarp** from 0 to 10 over the duration of the note. If p3 is 20 then the sound will be two times slower than the original. Of course you can use a more complex function than just a single straight line to control the time factor.

Now the same as above but using the balance function with the optional outputs:

```
asig,acmp   sndwarp  1,awarp,aresamp,isampfun,ibeg,iwindsize,iwindrand,\\
                          ioverlap,iwindfun,itimemode
abal        balance asig, acmp

asig1,asig2,acmp1,acmp2 sndwarpst 1, atime, aresamp, sampfun,\\
                                  ibeg, iwindsize, iwindrand, \\
                                  ioverlap, iwindfun, itimemode
abal1       balance asig1, acmp1
abal2       balance asig2, acmp2
```

In the above two examples notice the use of the balance unit. The output of balance can then be scaled, enveloped, sent to an out or outs, and so on. Notice that the amplitude arguments to **sndwarp** and **sndwarpst** are "1" in these examples. By scaling the signal after the **sndwarp** process, *abal*, *abal1*, and *abal2* should contain signals that have nearly

the same amplitude as the original input signal to the **sndwarp** process. This makes it much easier to predict the levels and avoid samples out of range or sample values that are too small.

More advice: Only use the stereo version when you really need to be processing a stereo file. It is somewhat slower than the mono version and if you use the balance function it is slower again. There is nothing wrong with using a mono **sndwarp** in a stereo orchestra and sending the result to one or both channels of the stereo output!

AUTHOR

Richard Karpen
Seattle, Wash
1997

---

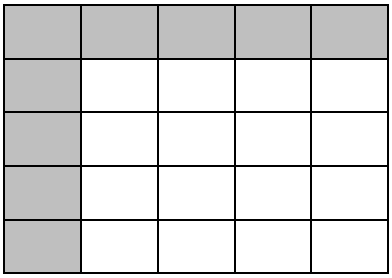
41 SIGNAL GENERATORS: SCANNED SYNTHESIS

Scanned synthesis is a variant of physical modeling, where a network of masses connected by springs is used to generate a dynamic waveform. The opcode **scanu** defines the mass/spring network and sets it in motion. The opcode **scans** follows a predefined path (trajectory) around the network and outputs the detected waveform. Several **scans** instances may follow different paths around the same network.

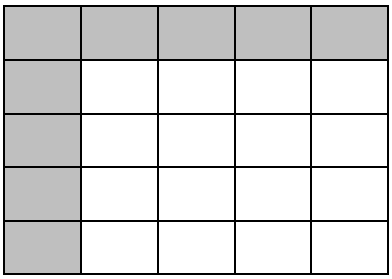
These are highly efficient mechanical modelling algorithms for both synthesis and sonic animation via algorithmic processing. They should run in real-time. Thus, the output is useful either directly as audio, or as controller values for other parameters.

The Csound implementation adds support for a scanning path or matrix. Essentially, this offers the possibility of reconnecting the masses in different orders, causing the signal to propagate quite differently. They do not necessarily need to be connected to their direct neighbors. Essentially, the matrix has the effect of "molding" this surface into a radically different shape.

To produce the matrices, the table format is straightforward. For example, for 4 masses we have the following grid describing the possible connections:



Whenever two masses are connected, the point they define is 1. If two masses are not connected, then the point they define is 0. For example, a unidirectional string has the following connections: (1,2), (2,3), (3,4). If it is bidirectional, it also has (2,1), (3,2), (4,3)). For the unidirectional string, the matrix appears:



The above table format of the connection matrix is for conceptual convenience only. The actual values shown in te table are obtained by **scans** from an ASCII file using **GEN23**. The actual ASCII file is created from the table model row by row. Therefore the ASCII file for the example table shown above becomes:

```
0100001000010000
```

This marix example is very small and simple. In practice, most scanned synthesis instruments will use many more masses than four, so their matrices will be much larger and more complex. See the example in the **scans** documentation.

Please note that the generated dynamic wavetables are very unstable. Certain values for masses, centering, and damping can cause the system to "blow up" and the most interesting sounds to emerge from your loudspeakers!

The supplement to this manual contains a tutorial on scanned synthesis. The tutorial, examples, and other information on scanned synthesis is available from the Scanned Synthesis page at cSounds.com (http://www.csounds.com/scanned).

Scanned synthesis developed by Bill Verplank, Max Mathews and Rob Shaw at Interval Research between 1998 and 2000.

41.1 scanu

scanu `init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, kstif, kcentr, kdamp, ileft, iright, kpos, kstrngth, ain, idisp, id`

DESCRIPTION

Compute the waveform and the wavetable for use in scanned synthesis.

INITIALIZATION

init – the initial position of the masses. If this is a negative number, then the absolute of *init* signifies the table to use as a hammer shape. If *init* > 0, the length of it should be the same as the intended mass number, otherwise it can be anything.

ifnvel – the ftable that contains the initial velocity for each mass. It should have the same size as the intended mass number.

ifnmass – ftable that contains the mass of each mass. It should have the same size as the intended mass number.

ifnstif – ftable that contains the spring stiffness of each connection. It should have the same size as the square of the intended mass number. The data ordering is a row after row dump of the connection matrix of the system.

ifncentr – ftable that contains the centering force of each mass. It should have the same size as the intended mass number.

ifndamp – the ftable that contains the damping factor of each mass. It should have the same size as the intended mass number.

ileft – If *init* < 0, the position of the left hammer (*ileft* = 0 is hit at leftmost, *ileft* = 1 is hit at rightmost).

iright – If *init* < 0, the position of the right hammer (*iright* = 0 is hit at leftmost, *iright* = 1 is hit at rightmost).

idisp – If 0, no display of the masses is provided.

id – If positive, the ID of the opcode. This will be used to point the scanning opcode to the proper waveform maker. If this value is negative, the absolute of this value is the wavetable on which to write the waveshape. That wavetable can be used later from an other opcode to generate sound. The initial contents of this table will be destroyed.

PERFORMANCE

kmass – scales the masses

kstif – scales the spring stiffness

kcentr – scales the centering force

kdamp – scales the damping

kpos – position of an active hammer along the string (*kpos* = 0 is leftmost, *kpos* = 1 is rightmost). The shape of the hammer is determined by *init* and the power it pushes with is *kstrngth*.

kstrngth – power that the active hammer uses

ain – audio input that adds to the velocity of the masses. Amplitude should not be too great.

EXAMPLE

For an example, see the documentation on **scans**.

AUTHOR

Paris Smaragdis
MIT Media Lab
Boston, Massachussetts USA
March, 2000 (New in Csound version 4.05)

41.2 scans

```
ar      scans      kamp, kfreq, ifn, id[, iorder]
```

DESCRIPTION

Generate audio output using scanned synthesis.

INITIALIZATION

ifn – ftable containing the scanning trajectory. This is a series of numbers that contains addresses of masses. The order of these addresses is used as the scan path. It should not contain values greater than the number of masses, or negative numbers. See the introduction to the scanned synthesis section.

id – ID number of the **scanu** opcode's waveform to use

iorder (optional) – order of interpolation used internally. It can take any value in the range 1 to 4, and defaults to 4, which is quartic interpolation. The setting of 2 is quadratic and 1 is linear. The higher numbers are slower, but not necessarily better.

PERFORMANCE

kamp – output amplitude. Note that the resulting amplitude is also dependent on instantaneous value in the wavetable. This number is effectively the scaling factor of the wavetable.

kfreq – frequency of the scan rate

EXAMPLE

Here is a simple example of scanned synthesis. The user must supply the matrix file "string-128". This file, as well as several other matrices, is available in a zipped file from the Scanned Synthesis page at cSounds.com (http://www.csounds.com/scanned).

```
;orchestra ---------------------------------------
            sr              =     44100
            kr              =     4410
            ksmps           =     10
            nchnls          =     1

            instr   1
  a0        =               0
            scanu           1,.01, 6, 2, 3, 4, 5, 2, .1, .1, -.01, .1, .5, 0, 0, a0,
1, 2

  a1        scans           ampdb(p4), cpspch(p5), 7,       2
            out             a1
            endin

;score -------------------------------------------

; Initial condition
f1 0 128 7 0 64 1 64 0

; Masses
f2 0 128 -7 1 128 1

; Spring matrices
f3 0 16384 -23 "string-128"

; Centering force
f4  0 128 -7 0 128 2
```

```
; Damping
f5 0 128 -7 1 128 1

; Initial velocity
f6 0 128 -7 0 128 0

; Trajectories
f7 0 128 -5 .001 128 128

; Note list
i1 0  10  86 6.00
i1 11 14  86 7.00
i1 15 20  86 5.00
e
```

AUTHOR

Paris Smaragdis
MIT Media Lab
Boston, Massachussetts USA
March, 2000 (New in Csound version 4.05)

---

42 SIGNAL GENERATORS: WAVEGUIDE PHYSICAL MODELING

42.1 pluck

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

DESCRIPTION

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

INITIALIZATION

icps – intended pitch value in Hz, 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.

42.2 wgpluck

```
ar      wgpluck     icps, iamp, kpick, iplk, idamp, ifilt, axcite
```

DESCRIPTION

A high fidelity simulation of a plucked string, using interpolating delay-lines.

INITIALIZATION

icps – frequency of plucked string

iamp – amplitude of string pluck

iplk – point along the string, where it is plucked, in the range of 0 to 1. 0 = no pluck

idamp – damping of the note. This controls the overall decay of the string. The greater the value of idamp1, the faster the decay. Negative values will cause an increase in output over time.

ifilt – control the attenuation of the filter at the bridge. Higher values cause the higher harmonics to decay faster.

PERFORMANCE

kpick – proportion of the way along the point to sample the output

axcite – signal which excites the string

A string of frequency *icps* is plucked with amplitude *iamp* at point *iplk*. The decay of the virtual string is controlled by *idamp* and *ifilt* which simulate the bridge. The oscillation is sampled at the point *kpick*, and excited by the signal *axcite*.

EXAMPLES

The following example produces a moderately long note with rapidly decaying upper partials:

```
            instr 1
   axcite   oscil          1, 1, 1
   apluck   wgpluck        220, 120, .5, 0, 10, 1000, axcite
            out            apluck
            endin
whereas the following produces a shorter, brighter note:
            instr 1
   axcite   oscil          1, 1, 1
   apluck   wgpluck        220, 120, .5, 0, 30, 10, axcite
            out            apluck
            endin
```

42.3 repluck, wgpluck2

```
ar       wgpluck2    iplk, xam, icps, kpick, krefl
ar       repluck     iplk, xam, icps, kpick, krefl, axcite
```

DESCRIPTION

wgpluck2 is an implementation of the physical model of the plucked string, with control over the pluck point, the pickup point and the filter. **repluck** is the same operation, but with an additional audio signal, *axcite*, used to excite the 'string'. Both opcodes are based on the Karplus-Strong algorithms.

INITIALIZATION

icps – The string plays at *icps* pitch.

iplck – The point of pluck is *iplk*, which is a fraction of the way up the string (0 to 1). A pluck point of zero means no initial pluck.

PERFORMANCE

xamp – Amplitude of note.

kpick – Proportion of the way along the string to sample the output.

kabsp – absorption coefficient at the bridge where 1 means total absorption and 0 is no absorption.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
1997

42.4 wgbow

```
ar      wgbow      kamp, kfreq, kpres, krat, kvibf, kvamp, ifn[, iminfreq]
```

DESCRIPTION

Audio output is a tone similar to a bowed string, using a physical model developed from Perry Cook, but re-coded for Csound.

INITIALIZATION

ifn – table of shape of vibrato, usually a sine table, created by a function

iminfreq – lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial *kfreq*. If *iminfreq* is negative, initialization will be skipped.

PERFORMANCE

A note is played on a string-like instrument, with the arguments as below.

kamp – Amplitude of note.

kfreq – Frequency of note played.

kpres – a parameter controlling the pressure of the bow on the string. Values should be about 3. The useful range is approximately 1 to 5.

krat – the position of the bow along the string. Usual playing is about 0.127236. The suggested range is 0.025 to 0.23.

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

EXAMPLE

```
kv      linseg      0, 0.5, 0, 1, 1, p3-0.5, 1
a1      wgbow       31129.60, 440, 3.0, 0.127236, 6.12723, kv*0.01, 1
        out         a1
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

42.5 wgflute

```
ar      wgflute     kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp,\\
                    ifn[, iminfreq[, kjetrf[, kendrf]]]
```

DESCRIPTION

Audio output is a tone similar to a flute, using a physical model developed from Perry Cook, but re-coded for Csound.

INITIALIZATION

iatt – time in seconds to reach full blowing pressure. 0.1 seems to correspond to reasonable playing.

idetk – time in seconds taken to stop blowing. 0.1 is a smooth ending

ifn – table of shape of vibrato, usually a sine table, created by a function

iminfreq – lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial kfreq. If *iminfreq* is negative, initialization will be skipped.

PERFORMANCE

kamp – Amplitude of note.

kfreq – Frequency of note played. While it can be varied in performance, I have not tried it.

kjet – a parameter controlling the air jet. Values should be positive, and about 0.3. The useful range is approximately 0.08 to 0.56.

kngain – amplitude of the noise component, about 0 to 0.5

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

kjetrf – amount of reflection in the breath jet that powers the flute. Default value is 0.5.

kendrf – reflection coefficient of the breath jet. Default value is 0.5. Both *ijetrf* and *iendrf* are used in the calculation of the pressure differential.

EXAMPLE

```
a1          wgflute         31129.60, 440, 0.32, 0.1, 0.1, 0.15, 5.925, 0.05, 1
            out             a1
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

42.6 wgbrass

```
ar      wgbrass     kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn[, iminfreq]
```

DESCRIPTION

Audio output is a tone related to a brass instrument, using a physical model developed from Perry Cook, but re-coded for Csound. [NOTE: This is rather poor, and at present uncontrolled. Needs revision, and possibly more parameters].

INITIALIZATION

iatt -- time taken to reach full pressure

ifn – table of shape of vibrato, usually a sine table, created by a function

iminfreq – lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial *kfreq*. If *iminfreq* is negative, initialization will be skipped.

PERFORMANCE

A note is played on a brass-like instrument, with the arguments as below.

kamp – Amplitude of note.

kfreq – Frequency of note played.

ktens – lip tension of the player. Suggested value is about 0.4

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

EXAMPLE

```
a1      wgbrass     31129.60, 440, 0.1, 6.137, 0.05, 1
        out         a1
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

42.7 wgclar

```
ar      wgclar      kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, \\
                    kvamp, ifn[, iminfreq]
```

DESCRIPTION

Audio output is a tone similar to a clarinet, using a physical model developed from Perry Cook, but re-coded for Csound.

INITIALIZATION

iatt – time in seconds to reach full blowing pressure. 0.1 seems to correspond to reasonable playing. A longer time gives a definite initial wind sound.

idetk – time in seconds taken to stop blowing. 0.1 is a smooth ending

ifn – table of shape of vibrato, usually a sine table, created by a function

iminfreq – lowest frequency at which the instrument will play. If it is omitted it is taken to be the same as the initial kfreq. If *iminfreq* is negative, initialization will be skipped.

PERFORMANCE

A note is played on a clarinet-like instrument, with the arguments as below.

kamp – Amplitude of note.

kfreq – Frequency of note played.

kstiff – a stiffness parameter for the reed. Values should be negative, and about -0.3. The useful range is approximately -0.44 to -0.18.

kngain – amplitude of the noise component, about 0 to 0.5

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

EXAMPLE

```
a1    wgclar      31129.60, 440, -0.3, 0.1, 0.1, 0.2, 5.735, 0.1, 1
      out         a1
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

42.8 wgbowedbar

```
ar      wgbowedbar  kamp, kfreq, kpos, kbowpres, kgain[, kconst, ktvel,
                    kbowpos, ilow]
```

DESCRIPTION

A physical model of a bowed bar, belonging to the Perry Cook family of waveguide instruments.

INITIALIZATION

ilow (optional) – lowest frequency required

PERFORMANCE

kamp – amplitude of signal

kfreq – frequency of signal

kpos – position of the bow on the bar, in the range 0 to 1

kbowpres – pressure of the bow (as in **wgbowed**)

kgain – gain of filter. A value of about 0.809 is suggested.

kconst (optional) – an integration constant. Default is zero.

ktvel (optional) – either 0 or 1. When *ktvel* = 0, the bow velocity follows an ADSR style trajectory. When *ktvel* = 1, the value of the bow velocity decays in an exponentially.

kbowpos (optional) – the position on the bow, which affects the bow velocity trajectory.

EXAMPLE

```
;orchestra ----------------

            instr 1
;           pos             =     [0, 1]
;           bowpress        =     [1, 10]
;           gain            =     [0.8, 1]
;           intr            =     [0,1]
;           trackvel        =     [0, 1]
;           bowpos          =     [0, 1]

  kb        line            0.5, p3, 0.1
  kp        line            0.6, p3, 0.7
  kc        line            1, p3, 1

  a1        wgbowedbar      p4, cpspch(p5), kb, kp, 0.995, p6, 0, kc, 50

            out             a1
            endin

;score -------------------

  i1        0   3 32000 7.00 0

  e
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 4.07

---

43 SIGNAL GENERATORS: MODELS AND EMULATIONS

43.1 moog

```
a1      moog        kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn,\\
                    iwfn, ivfn
```

DESCRIPTION

An emulation of a mini-Moog synthesizer.

INITIALIZATION

iafn, iwfn, ivfn – three table numbers containing the attack waveform (unlooped), the main looping wave form, and the vibrato waveform. The files *mandpluk.aiff* and *impuls20.aiff* are suitable for the first two, and a sine wave for the last.

PERFORMANCE

kamp – Amplitude of note.

kfreq – Frequency of note played.

kfiltq – Q of the filter, in the range 0.8 to 0.9

kfiltrate – rate control for the filter in the range 0 to 0.0002

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

43.2 shaker

```
ar      shaker      kamp, kfreq, kbeans, kdamp, ktimes[, idecay]
```

DESCRIPTION

Audio output is a tone related to the shaking of a maraca or similar gourd instrument. The method is a physically inspired model developed from Perry Cook, but re-coded for Csound.

INITIALIZATION

idecay – If present indicates for how long at the end of the note the shaker is to be damped. The default value is zero.

PERFORMANCE

A note is played on a maraca-like instrument, with the arguments as below.

kamp – Amplitude of note.

kfreq – Frequency of note played.

kbeans – The number of beans in the gourd. A value of 8 seems suitable,

kdamp -- The damping value of the shaker. Values of 0.98 to 1 seems suitable, with 0.99 a reasonable default.

ktimes -- Number of times shaken.

The argument *knum* was redundant, so was removed in version 3.49.

EXAMPLE

```
a1          shaker          31129.60, 440, 8, 0.999, 100, 0
            outs            a1, a1
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

43.3 marimba, vibes

```
ar      marimba     kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, \\
                    idec[, idoubles[, itriples]]
ar      vibes       kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, \\
                    idec
```

DESCRIPTION

Audio output is a tone related to the striking of a wooden or metal block as found in a marimba or vibraphone. The method is a physical model developed from Perry Cook, but re-coded for Csound.

INITIALIZATION

ihrd – the hardness of the stick used in the strike. A range of 0 to 1 is used. 0.5 is a suitable value.

ipos – where the block is hit, in the range 0 to 1.

imp – a table of the strike impulses. The file "marmstk1.wav" is a suitable function from measurements, and can be loaded with a **GEN01** table.

ivfn – shape of vibrato, usually a sine table, created by a function

idec – time before end of note when damping is introduced

idoubles – percentage of double strikes. Default is 40%.

itriples – percentage of triple strikes. Default is 20%.

PERFORMANCE

kamp – Amplitude of note.

kfreq – Frequency of note played.

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

EXAMPLE

```
a1    marimba     31129.60, 440, 0.5, 0.561, 2, 6.0, 0.05, 1, 0.1
a2    vibes       31129.60, 440, 0.5, 0.561, 2, 4.0, 0.2, 1, 0.1a1
      outs        a1, a2
```

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

43.4 mandol

```
ar      mandol      kamp, kfreq, kpluck, kdetune, kgain, ksize, ifn\\
                    [, iminfreq]
```

DESCRIPTION

An emulation of a mandolin.

INITIALIZATION

ifn -- table number containing the pluck wave form. The file mandpluk.aiff is suitable for this.

iminfreq -- Lowest frequency to be played on the note. If it is omitted it is taken to be the same as the initial kfreq.

PERFORMANCE

kamp – Amplitude of note.

kfreq – Frequency of note played.

kpluck – The pluck position, in range 0 to 1. Suggest 0.4.

kgain – the loopgain of the model, in the range 0.97 to 1.

kdetune – The proportional detuning between the two strings. Suggested range 1 and 0.9.

ksize – The size of the body of the mandolin. Range 0 to 2.

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

43.5 gogobel

```
ar        gogobel     kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn
```

DESCRIPTION

Audio output is a tone related to the striking of a cow bell or similar. The method is a physical model developed from Perry Cook, but re-coded for Csound.

INITIALIZATION

ihrd -- the hardness of the stick used in the strike. A range of 0 to 1 is used. 0.5 is a suitable value.

ipos -- where the block is hit, in the range 0 to 1.

imp – a table of the strike impulses. The file "*marmstk1.wav*" is a suitable function from measurements, and can be loaded with a **GEN01** table.

ivfn – shape of vibrato, usually a sine table, created by a function.

PERFORMANCE

A note is played on a cowbell-like instrument, with the arguments as below.

kamp – Amplitude of note.

kfreq – Frequency of note played.

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

EXAMPLE

```
a1    gogobel    31129.60, 440, p4, 0.561, 3, 6.0, 0.3, 1
      outs        a1, a2
```

NAME CHANGE

Prior to Csound version 3.52 (February, 1999), this opcode was called **agogobel**.

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

43.6 voice

```
ar      voice      kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn
```

DESCRIPTION

An emulation of a human voice.

INITIALIZATION

ifn, ivfn – two table numbers containing the carrier wave form and the vibrato waveform. The files "impuls20.aiff", "ahh.aiff", "eee.aiff", or "ooo.aiff" are suitable for the first of these, and a sine wave for the second. These files are available from:

```
ftp://ftp.maths.bath.ac.uk/pub/dream/documentation/sounds/modelling
/
```

PERFORMANCE

kamp – Amplitude of note.

kfreq – Frequency of note played. It can be varied in performance.

kphoneme – an integer in the range 0 to 16, which select the formants for the sounds:

"eee","ihh","ehh","aaa",
"ahh","aww","ohh","uhh",
"uuu","ooo","rrr","lll",
"mmm","nnn","nng","ngg".

At present the phonemes
"fff","sss","thh","shh",
"xxx","hee","hoo","hah",
"bbb","ddd","jjj","ggg",
"vvv","zzz","thz","zhh"

are not available (!)

kform – Gain on the phoneme. values 0.0 to 1.2 recommended.

kvibf – frequency of vibrato in Hertz. Suggested range is 0 to 12

kvamp – amplitude of the vibrato

AUTHOR

John ffitch (after Perry Cook)
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.47

43.7 lorenz

```
ax, ay, az  lorenz  ks, kr, kb, kh, ix, iy, iz, iskip
```

DESCRIPTION

Implements the Lorenz system of equations. The Lorenz system is a chaotic-dynamic system which was originally used to simulate the motion of a particle in convection currents and simplified weather systems. Small differences in initial conditions rapidly lead to diverging values. This is sometimes expressed as the butterfly effect. If a butterfly flaps its wings in Australia, it will have an effect on the weather in Alaska. This system is one of the milestones in the development of chaos theory. It is useful as a chaotic audio source or as a low frequency modulation source.

INITIALIZATION

ix, *iy*, *iz* – the initial coordinates of the particle

iskip – used to skip generated values. If *iskip* is set to 5, only every fifth value generated is output. This is useful in generating higher pitched tones.

PERFORMANCE

ksv – the Prandtl number or sigma

krv – the Rayleigh number

kbv – the ratio of the length and width of the box in which the convection currents are generated

kh – the step size used in approximating the differential equation. This can be used to control the pitch of the systems. Values of .1-.001 are typical.

The equations are approximated as follows:

```
x = x + h*(s*(y - x))
y = y + h*(-x*z + r*x - y)
z = z + h*(x*y - b*z)
```

The historical values of these parameters are:

```
ks = 10
kr = 28
kb = 8/3
```

EXAMPLE

```
                instr 20
 ksv = p4
 krv = p5
 kbv = p6

 ax, ay, az   lorenz   ksv, krv, kbv, .01, .6, .6, .6, 1
                endin

;score
  ;   start dur S   R   V
  i20 5     1   10  28  2.667
e
```

AUTHOR

Hans Mikelson
February 1999
(New in Csound version 3.53)

43.8 planet

```
ax, ay, az          planet  kmass1, kmass2, ksep, ix, iy, iz, ivx, \\
                            ivy, ivz, idelta, ifriction
```

DESCRIPTION

planet simulates a planet orbiting in a binary star system. The outputs are the x, y and z coordinates of the orbiting planet. It is possible for the planet to achieve escape velocity by a close encounter with a star. This makes this system somewhat unstable.

INITIALIZATION

ix, iy, iz – the initial x, y and z coordinates of the planet

ivx, ivy, ivz – the initial velocity vector components for the planet.

idelta – the step size used to approximate the differential equation.

ifriction – a value for friction, which can used to keep the system from blowing up

PERFORMANCE

kmass1 – the mass of the first star

kmass2 – the mass of the second star

ksep – determines the distance between the two stars

ax, ay, az – the output x, y, and z coordinates of the planet

EXAMPLE

```
  instr 1
idur          =             p3
iamp          =             p4
km1           =             p5
km2           =             p6
ksep          =              p7
ix            =             p8
iy            =             p9
iz            =             p10
ivx           =             p11
ivy           =             p12
ivz           =             p13
ih            =             p14
ifric         =             p15

kamp          linseg        0, .002, iamp, idur-.004, iamp, .002, 0

ax,ay,az planet             km1, km2, ksep, ix, iy, iz, ivx, ivy, ivz, ih, ifric

              outs          ax*kamp, ay*kamp

              endin

;   Sta  Dur  Amp   M1  M2  Sep  X   Y  Z  VX  VY  VZ   h   Frict
i1  0    1    5000  .5  .35 2.2  0  .1  0  .5  .6  -.1  .5  -0.1
i1  +    .    .     .5  0   0    0  .1  0  .5  .6  -.1  .5   0.1
i1  .    .    .     .4  .3  2    0  .1  0  .5  .6  -.1  .5   0.0
i1  .    .    .     .3  .3  2    0  .1  0  .5  .6  -.1  .5   0.1
i1  .    .    .     .25 .3  2    0  .1  0  .5  .6  -.1  .5   1.0
i1  .    .    .     .2  .5  2    0  .1  0  .5  .6  -.1  .1   1.0
```

AUTHOR

Hans Mikelson
December 1998
New in Csound version 3.50

43.9 cabasa, crunch, sekere, sandpaper, stix

```
ar      cabasa      iamp, idettack[, knum, kdamp, kmaxshake]
ar      crunch      iamp, idettack[, knum, kdamp, kmaxshake]
ar      sekere      iamp, idettack[, knum, kdamp, kmaxshake]
ar      sandpaper   iamp, idettack[, knum, kdamp, kmaxshake]
ar      stix        iamp, idettack[, knum, kdamp, kmaxshake]
```

DESCRIPTION

Semi-physical models of various percussion sounds.

INITIALIZATION

iamp – Amplitude of output. Note: As these instruments are stochastic, this is only a approximation.

idettack – period of time over which all sound is stopped

PERFORMANCE

knum – The number of beads, teeth, bells, timbrels, etc. If, zero the default value is:

```
cabasa      = 512.0000
crunch      =   7.0000
sekere      =  64.0000
sandpaper   = 128.0000
stix        =  30.0000
```

kdamp – the damping factor of the instrument. The value is used as an adjustment around the defaults, with 1 being no damping. If zero, the default values are used. The defaults are:

```
cabasa      =   0.9970
crunch      =   0.99806
sekere      =   0.9990
sandpaper   =   0.9990
stix        =   0.9980
```

kmaxshake – amount of energy to add back into the system. The value should be in range 0 to 1.

EXAMPLE

```
;orchestra ---------------
  sr      =           44100
  kr      =            4410
  ksmps   =              10
  nchnls  =               1
  gknum      init       0           ;initialize optional arguments
  gkdamp     init       0           ;   for use with all instruments
  gkmaxshake init       0
          instr 01                  ;an example of a cabasa
  a1      cabasa      p4, 0.01, gknum, gkdamp, gkmaxshake
          out         a1
          endin
          instr 02                  ;an example of a crunch
  a1      crunch      p4, 0.01, gknum, gkdamp, gkmaxshake
          out         a1
          endin
          instr 03                  ;an example of a sekere
```

```
  a1      sekere      p4, 0.01, gknum, gkdamp, gkmaxshake
          out         a1
          endin
          instr 04                ;an example of sandpaper blocks
  a1      line        2, p3, 2    ;preset amplitude increase
  a2      sandpaper   p4, 0.01, gknum, gkdamp, gkmaxshake
  a3      product     a1, a2      ;increase amplitude
          out         a3
          endin
          instr 05                ;an example of stix
  a1      line        20, p3, 20  ;preset amplitude increase
  a2      stix        p4, 0.01, gknum, gkdamp, gkmaxshake
  a3      product     a1, a2      ;increase amplitude
          out         a3
          endin
;score -------------------
   i1 0 1 26000
   i2 2 1 26000
   i3 4 1 26000
   i4 6 1 26000
   i5 8 1 26000
   e
```

AUTHOR

John ffitch
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 4.07

43.10 guiro, tambourine, bamboo, dripwater, sleighbells

```
ar          guiro       iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1]
ar          tambourine  iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1,
                        kfreq2]
ar          bamboo      iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1,
                        kfreq2]
ar          dripwater   iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1,
                        kfreq2]
ar          sleighbells iamp, idettack[, knum, kdamp, kmaxshake, kfreq, kfreq1,
                        kfreq2]
```

DESCRIPTION

Semi-physical models of various percussion sounds.

INITIALIZATION

iamp – Amplitude of output. Note: As these instruments are stochastic, this is only a approximation.

idettack – period of time over which all sound is stopped

PERFORMANCE

knum – The number of beads, teeth, bells, timbrels, etc. If, zero the default value is:

```
guiro       = 128.0000
tambourine  =  32.0000
bamboo      =   1.2500
dripwater   =  10.0000
sleighbells =  32.0000
```

kdamp – the damping factor of the instrument. The value is used as an adjustment around the defaults, with 1 being no damping. If zero, the default values are used. The defaults are:

```
guiro       =   1.0000
tambourine  =   0.9985
bamboo      =   0.9999
dripwater   =   0.9950
sleighbells =   0.9994
```

kmaxshake – amount of energy to add back into the system. The value should be in range 0 to 1.

kfreq – the main resonant frequency. The default values are:

```
guiro       = 2500.0000
tambourine  = 2300.0000
bamboo      = 2800.0000
dripwater   =  450.0000
sleighbells = 2500.0000
```

kfreq1 – the first resonant frequency. The default values are:

```
tambourine  = 5600.0000
bamboo      = 2240.0000
dripwater   =  600.0000
sleighbells = 5300.0000
```

kfreq2 – the second resonant frequency. The default values are:

```
tambourine  = 8100.0000
bamboo      = 3360.0000
dripwater   =  750.0000
sleighbells = 6500.0000
```

EXAMPLE

```
;orchestra ---------------
  sr      =           44100
  kr      =            4410
  ksmps   =              10
  nchnls  =               1
          instr 01                    ;example of a guiro
  a1      guiro       p4, 0.01
          out         a1
          endin
          instr 02                    ;example of a tambourine
  a1      tambourine  p4, 0.01
          out         a1
          endin
          instr 03                    ;example of bamboo
  a1      bamboo      p4, 0.01
          out         a1
          endin
          instr 04                    ;example of a water drip
  a1      line        5, p3, 5        ;preset an amplitude boost
  a2      dripwater   p4, 0.01, 0, .9
  a3      product     a1, a2          ;increase amplitude
          out         a3
          endin
          instr 05                    ;an example of sleighbells
  a1      sleighbells p4, 0.01
          out         a1
          endin
;score -------------------
   i1 0 1 20000
   i2 2 1 20000
   i3 4 1 20000
   i4 6 1 20000
   i5 8 1 20000
   e
```

AUTHOR

John ffitch
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 4.07

This page intentionally left blank.

44 SIGNAL GENERATORS: STFT RESYNTHESIS (VOCODING)

44.1 pvoc, vpvoc

```
ar      pvoc      ktimpnt, kfmod, ifilcod [, ispecwp, iextractmode,\\
                  ifreqlim, igatefn]

ar      vpvoc     ktimpnt, kfmod, ifile[, ispecwp[, ifn]]
```

DESCRIPTION

Output is an additive set of individually controlled sinusoids, using phase vocoder resynthesis.

INITIALIZATION

ifilcod – integer or character-string denoting a control-file derived from analysis of an audio signal. An integer denotes the suffix of a file *pvoc.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in the one given by the environment variable **SADIR** (if defined).**pvoc** control contains breakpoint amplitude and frequency envelope values organized for fft resynthesis. Memory usage depends on the size of the file involved, which is 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.

iextractmode (optional) – determines if spectral extraction will be carried out, and if so, whether components that have changes in frequency below *ifreqlim* or above *ifreqlim* will be discarded. A value for *iextractmode* of 1 will cause **pvadd** to synthesize only those components where the frequency difference between analysis frames is greater than *ifreqlim*. A value of 2 for *iextractmode* will cause **pvadd** to synthesize only those components where the frequency difference between frames is less than *ifreqlim*. The default values for *iextractmode* and *ifreqlim* are 0, in which case a simple resynthesis will be done. See examples under **pvadd** for how to use spectral extraction.

igatefn – is the number of a stored function which will be applied to the amplitudes of the analysis bins before resynthesis takes place. If *igatefn* is greater than 0, the amplitudes of each bin will be scaled by *igatefn* through a simple mapping process. First, the amplitudes of all of the bins in all of the frames in the entire analysis file are compared to determine the maximum amplitude value. This value is then used create normalized amplitudes as indeces into the stored function *igatefn*. The maximum amplitude will map to the last point in the function. An amplitude of 0 will map to the first point in the function. Values between 0 and 1 will map accordingly to points along the function table. See examples under **pvadd** for how to use amplitude gating.

ifn (optional) – optional function table containing control information for **vpvoc**. If *ifn* = 0, control is derived internally from a previous **tableseg** or **tablexseg** unit. Default is 0. (New in Csound version 3.59)

PERFORMANCE

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 orignally written by Dan Ellis. It is based in part on the system of Mark Dolson, but the pre-analysis concept is new. The spectral extraction and amplitude gating (new in Csound version 3.56) were added by Richard Karpen based on functions in SoundHack by Tom Erbe.

vpvoc is identical to **pvoc** except that it takes the result of a previous **tableseg, tablexseg** and uses the resulting function table (passed internally to the **vpvoc**), as an envelope over the magnitudes of the analysis data channels. Optionally, a table specified by *ifn*, may be used. The result is spectral enveloping. The function size used in the t**ableseg** should be *framesize/2,* where framesize is the number of bins in the phase vocoder analysis file that is being used by the **vpvoc**. Each location in the table will be used to scale a single analysis bin. By using different functions for *ifn1*, *ifn2*, etc.. in the **tableseg**, the spectral envelope becomes a dynamically changing one. See also **tableseg** and **tablexseg**.

EXAMPLE

The following example using **vpvoc**, shows the use of functions such as

```
f 1 0 256 5 .001 128 1 128 .001
f 2 0 256  5 1 128 .001 128 1
f 3 0 256  7 1 256 1
```

to scale the amplitudes of the separate analysis bins.

```
ktime   line            0, p3,3 ; time pointer, in seconds, into file
        tablexseg       1, p3*.5, 2, p3*.5, 3
apv     vpvoc           ktime,1, “pvoc.file”
```

The result would be a time-varying "spectral envelope" applied to the phase vocoder analysis data. Since this amplifies or attenuates the amount of signal at the frequencies that are paired with the amplitudes which are scaled by these functions, it has the effect of applying very accurate filters to the signal. In this example the first table would have the effect of a band- pass filter , gradually be band-rejected over half the note's duration, and then go towards no modification of the magnitudes over the second half.

AUTHORS

Dan Ellis (**pvoc**)

Richard Karpen (**vpvoc**)
Seattle, Washngton
1997

44.2 pvread, pvbufread, pvinterp, pvcross, tableseg, tablexseg

```
kfr, kamp  pvread      ktimpnt,  ifile, ibin
           pvbufread   ktimpnt, ifile
  ar       pvinterp    ktimpnt, kfmod, ifile, kfreqscale1, kfreqscale2,\\
                       kampscale1, kampscale2, kfreqinterp, kampinterp
  ar       pvcross     ktimpnt, kfmod, ifile, kamp1, kamp2[, ispecwp]
           tableseg    ifn1, idur1, ifn2[, idur2, ifn3[...]]
           tablexseg   ifn1, idur1, ifn2[, idur2, ifn3[...]]
```

DESCRIPTION

pvread reads from a **pvoc** file and returns the frequency and amplitude from a single analysis channel or bin. The returned values can be used anywhere else in the Csound instrument. For example, one can use them as arguments to an oscillator to synthesize a single component from an analyzed signal or a bank of pvreads can be used to resynthesize the analyzed sound using additive synthesis by passing the frequency and magnitude values to a bank of oscillators.

pvbufread reads from a pvoc file and makes the retrieved data available to any following **pvinterp** and **pvcross** units that appear in an instrument before a subsequent **pvbufread** (just as **lpread** and **lpreson** work together). The data is passed internally and the unit has no output of its own. **pvinterp** and **pvcross** allow the interprocessing of two phase vocoder analysis files prior to the resynthesis which these units do also. Both of these units receive data from one of the files from a previously called **pvbufread** unit. The other file is read by the **pvinterp** and/or **pvcross** units. Since each of these units has its own time-pointer the analysis files can be read at different speeds and directions from one another. **pvinterp** does not allow for the use of the *ispecwp* process as with the **pvoc** and **vpvoc** units.

pvinterp interpolates between the amplitudes and frequencies, on a bin by bin basis, of two phase vocoder analysis files (one from a previously called **pvbufread** unit and the other from within its own argument list), allowing for user defined transitions between analyzed sounds. It also allows for general scaling of the amplitudes and frequencies of each file separately before the interpolated values are calculated and sent to the resynthesis routines. The *kfmod* argument in **pvinterp** performs its frequency scaling on the frequency values after their derivation from the separate scaling and subsequent interpolation is performed so that this acts as an overall scaling value of the new frequency components.

pvcross applies the amplitudes from one phase vocoder analysis file to the data from a second file and then performs the resynthesis. The data is passed, as described above, from a previously called **pvbufread** unit. The two k-rate amplitude arguments are used to scale the amplitudes of each files separately before they are added together and used in the resynthesis (see below for further explanation). The frequencies of the first file are not used at all in this process. This unit simply allows for cross-synthesis through the application of the amplitudes of the spectra of one signal to the frequencies of a second signal. Unlike **pvinterp**, **pvcross** does allow for the use of the *ispecwp* as in **pvoc** and **vpvoc**.

tableseg and **tablexseg** are like **linseg** and **expseg** but interpolate between values in a stored function tables. The result is a new function table passed internally to any following **vpvoc** which occurs before a subsequent **tableseg** or **tablexseg** (much like **lpread**/**lpreson** pairs work). The uses of these are described below under **vpvoc**.

INITIALIZATION

ifile – the **pvoc** number (n in pvoc.n) or the name in quotes of the analysis file made using pvanal. (See **pvoc**.)

ibin – the number of the analysis channel from which to return frequency in Hz and magnitude.

ifn1, *ifn2*, *ifn3*, etc. – function table numbers for **tableseg** and **tablexseg**. *ifn1*, *ifn2*, and so on, must be the same size.

idur1, *idur2*, etc. – durations for **tableseg** and t**ablexseg,** during which interpolation from one table to the next will take place.

PERFORMANCE

kfreq, kamp – outputs of the **pvread** unit. These values, retrieved from a phase vocoder analysis file, represent the values of frequency and amplitude from a single analysis channel specified in the *ibin* argument. Interpolation between analysis frames is performed at k-rate resolution and dependent of course upon the rate and direction of *ktimpnt*.

ktimpnt, *kfmod*, *ispecwp* – used for **pvread** exactly the same as for **pvoc** (see above description of **pvinterp** for its special use of *kfmod*).

kfreqscale1, *kfreqscale2*, *kampscale1*, *kampscale2* – used in **pvinterp** to scale the frequencies and amplitudes stored in each frame of the phase vocoder analysis file. *kfreqscale1* and *kampscale1* scale the frequencies and amplitudes of the data from the file read by the previously called **pvbufread** (this data is passed internally to the pvinterp unit). *kfreqscale2* and *kampscale2* scale the frequencies and amplitudes of the file named by ifile in the **pvinterp** argument list and read within the **pvinterp** unit. By using these arguments it is possible to adjust these values before applying the interpolation. For example, if file1 is much louder than file2, it might be desirable to scale down the amplitudes of file1 or scale up those of file2 before interpolating. Likewise one can adjust the frequencies of each to bring them more in accord with one another (or just the opposite, of course!) before the interpolation is performed.

kfreqinterp, kampinterp – used in **pvinterp** to determine the interpolation distance between the values of one phase vocoder file and the values of a second file. When the value of *kfreqinterp* is 0, the frequency values will be entirely those from the first file (read by the **pvbufread**), post scaling by the *kfreqscale1* argument. When the value of *kfreqinterp* is 1 the frequency values will be those of the second file (read by the **pvinterp** unit itself), post scaling by kfreqscale2. When *kfreqinterp* is between 0 and 1 the frequency values will be calculated, on a bin, by bin basis, as the percentage between each pair of frequencies (in other words, *kfreqinterp*=.5 will cause the frequencies values to be half way between the values in the set of data from the first file and the set of data from the second file). *kampinterp1* and *kampinterp2* work in the same way upon the amplitudes of the two files. Since these are k-rate arguments, the percentages can change over time making it possible to create many kinds of transitions between sounds.

EXAMPLE

The example below shows the use **pvread** to synthesize a single component from a phase vocoder analysis file. It should be noted that the kfreq and kamp outputs can be used for any kind of synthesis, filtering, processing, and so on.

```
ktime      line          0, p3, 3
krefq,kamp pvread        ktime, "pvoc.file", 7 :read data from 7th analysis bin\
asig       oscili        kamp, kfreq, 1  ; finction 1 is a stored sine
```

The example below shows an example using **pvbufread** with **pvinterp** to interpolate between the sound of an oboe and the sound of a clarinet. The value of *kinterp* returned by a linseg is used to determine the timing of the transitions between the two sounds. The interpolation of frequencies and amplitudes are controlled by the same factor in this example, but for other effects it might be interesting to not have them synchronized in this way. In this example the sound will begin as a clarinet, transform into the oboe and then return again to the clarinet sound. The value of *kfreqscale2* is 1.065 because the oboe in this case is a semitone higher in pitch than the clarinet and this brings them approximately to the same pitch. The value of *kampscale2* is .75 because the analyzed clarinet was somewhat louder than the analyzed oboe. The setting of these two parameters make the transition quite smooth in this case, but such adjustments are by no means necessary or even advocated.

```
ktime1     line       0, p3, 3.5 ; used as index in the “oboe.pvoc” file
ktime2     line       0, p3, 4.5 ; used as index in the “clar.pvoc” file
kinterp    linseg    1, p3*.15, 1, p3*.35, 0, p3*.25, 0, p3*.15, 1, p3*.1, 1
           pvbufread ktime1, “oboe.pvoc”
apv        pvinterp  ktime2,1,”clar.pvoc”,1,1.065,1,.75,1-kinterp,1-kinterp
```

Below is an example using **pvbufread** with **pvcross**. In this example the amplitudes used in the resynthesis gradually change from those of the oboe to those of the clarinet. The frequencies, of course, remain those of the clarinet throughout the process since **pvcross** does not use the frequency data from the file read by **pvbufread**.

```
ktime1  line           0, p3, 3.5 ; used as index in the “oboe.pvoc” file
ktime2  line           0, p3, 4.5 ; used as index in the “clar.pvoc” file
kcross  expon         .001, p3, 1
        pvbufread      ktime1, “oboe.pvoc”
apv     pvcross        ktime2, 1, “clar.pvoc”, 1-kcross, kcross
```

AUTHOR

Richard Karpen
Seattle, Wash
1997

44.3 pvadd

```
ar      pvadd      ktimpnt, kfmod, ifilcod, ifn, ibins[, ibinoffset, \\
                   ibinincr, iextractmode, ifreqlim, igatefn]
```

DESCRIPTION

pvadd reads from a **pvoc** file and uses the data to perform additive synthesis using an internal array of interpolating oscillators. The user supplies the wave table (usually one period of a sine wave), and can choose which analysis bins will be used in the re-synthesis.

INITIALIZATION

ifilcod – integer or character-string denoting a control-file derived from analysis of an audio signal. An integer denotes the suffix of a file *pvoc.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. If not fullpath, the file is sought first in the current directory, then in the one given by the environment variable **SADIR** (if defined). **pvoc** control files contain 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**).

ifn – table number of a stored function containing a sine wave

ibins – number of bins that will be used in the resynthesis (each bin counts as one oscillator in the re-synthesis)

ibinoffset (optional) – is the first bin used (it is optional and defaults to 0).

ibinincr (optional) – sets an increment by which **pvadd** counts up from *ibinoffset* for *ibins* components in the re-synthesis (see below for a further explanation).

iextractmode (optional) – determines if spectral extraction will be carried out and if so whether components that have changes in frequency below *ifreqlim* or above *ifreqlim* will be discarded. A value for *iextractmode* of 1 will cause **pvadd** to synthesize only those components where the frequency difference between analysis frames is greater than *ifreqlim*. A value of 2 for *iextractmode* will cause **pvadd** to synthesize only those components where the frequency difference between frames is less than *ifreqlim*. The default values for *iextractmode* and *ifreqlim* are 0, in which case a simple resynthesis will be done. See examples below.

igatefn (optional) – is the number of a stored function which will be applied to the amplitudes of the analysis bins before resynthesis takes place. If *igatefn* is greater than 0 the amplitudes of each bin will be scaled by *igatefn* through a simple mapping process. First, the amplitudes of all of the bins in all of the frames in the entire analysis file are compared to determine the maximum amplitude value. This value is then used create normalized amplitudes as indeces into the stored function *igatefn*. The maximum amplitude will map to the last point in the function. An amplitude of 0 will map to the first point in the function. Values between 0 and 1 will map accordingly to points along the function table.This will be made clearer in the examples below.

PERFORMANCE

ktimpnt and *kfmod* are used in the same way as in **pvoc**.

EXAMPLES

```
ktime line  0, p3, p3
asig  pvadd ktime, 1, "oboe.pvoc", 1, 100, 2
```

In the above, *ibins* is 100 and *ibinoffset* is 2. Using these settings the resynthesis will contain 100 components beginning with bin #2 (bins are counted starting with 0). That is, resynthesis will be done using bins 2-101 inclusive. It is usually a good idea to begin with bin 1 or 2 since the 0th and often 1st bin have data that is neither necessary nor even helpful for creating good clean resynthesis.

```
ktime line  0, p3, p3
asig  pvadd ktime, 1, "oboe.pvoc", 1, 100, 2, 2
```

The above is the same as the previous example with the addition of the value 2 used for the optional *ibinincr* argument. This result will still result in 100 components in the resynthesis, but **pvadd** will count through the bins by 2 instead of by 1. It will use bins 2, 4, 6, 8, 10, and so on. For *ibins*=10, *ibinoffset*=10, and *ibinincr*=10, **pvadd** would use bins 10, 20, 30, 40, up to and including 100.

Below is an example using spectral extraction. In this example *iextractmode* is one and *ifreqlim* is 9. This will cause pvadd to synthesize only those bins where the frequency deviation, averaged over 6 frames, is greater than 9.

```
ktime line  0, p3, p3
asig  pvadd ktime, 1,  "oboe.pvoc", 1, 100, 2, 2, 1, 9
```

If *iextractmode* were 2 in the above, then only those bins with an average frequency deviation of less than 9 would be synthesized. If tuned correctly, this technique can be used to separate the pitched parts of the spectrum from the noisy parts. In practice, this depends greatly on the type of sound, the quality of the recording and digitization, and also on the analysis window size and frame increment.

Next is an example using amplitude gating. The last 2 in the argument list stands for f2 in the score.

```
asig  pvadd ktime, 1,  "oboe.pvoc", 1, 100, 2, 2, 0, 0, 2
```

Suppose the score for the above were to contain:

```
f2 0 512 7 0 256 1 256 1
```

Then those bins with amplitudes of 50% of the maximum or greater would be left unchanged, while those with amplitudes less than 50% of the maximum would be scaled down. In this case the lower the amplitude the more severe the scaling down would be. But suppose the score contains:

```
f2 0 512 5 1 512 .001
```

In this case, lower amplitudes will be left unchanged and greater ones will be scaled down, turning the sound "upside-down" in terms of the amplitude spectrum! Functions can be arbitrarily complex. Just remember that the normalized amplitude values of the analysis are themselves the indeces into the function.

Finally, both spectral extraction and amplitude gating can be used together. The example below will synthesize only those components that with a frequency deviation of less than 5Hz per frame and it will scale the amplitudes according to F2.

```
asig  pvadd ktime, 1,  "oboe.pvoc", 1, 100, 1, 1, 2, 5, 2
```

USEFUL HINTS:

By using several **pvadd** units together, one can gradually fade in different parts of the resynthesis, creating various "filtering" effects. The author uses **pvadd** to synthesis one bin at a time to have control over each separate component of the re-synthesis.

If any combination of *ibins*, *ibinoffset*, and *ibinincr*, creates a situation where **pvadd** is asked to used a bin number greater than the number of bins in the analysis, it will just use all of the available bins, and give no complaint. So to use every bin just make *ibins* a big number (i.e. 2000).

Expect to have to scale up the amplitudes by factors of 10-100, by the way.

AUTHOR

Richard Karpen
Seattle, Wash
1998 (New in Csound version 3.48, additional arguments version 3.56)

---

45 SIGNAL GENERATORS: LPC RESYNTHESIS

45.1 lpread, lpreson, lpfreson

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

DESCRIPTION

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 and four parameter values) derived from n-pole linear predictive spectral analysis of a source audio signal. An integer denotes the suffix of a file *lp.m*; a character-string (in double quotes) gives a filename, optionally a full pathname. 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 Hz.
```

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* (between 0 and 1) derived during predictive analysis reflects the deterministic/random nature of the analyzed source. This will emerge low for pitched (periodic) material and higher for noisy material. The transition from voiced to unvoiced speech, for example, produces an error signal value of about .001. During synthesis, the error signal value can be used to determine the nature of the **lpreson** driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a

mix of the two. In normal speech resynthesis, the pitched input to **lpreson** is a wideband periodic signal or pulse train derived from a unit such as **buzz**, and the non-pitched source is usually derived from **rand**. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

lpfreson is a formant shifted **lpreson**, in which *kfrqratio* is the (Hz) 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.

45.2 lpslot, lpinterp

```
lpslot      islot
lpinterpol  islot1, islot2, kmix
```

DESCRIPTION

Interpolate between two lpc analysis files.

INITIALIZATION

islot – number of slot to be selected [0<*islot*<20]

lpslot selects the slot to be use by further lp opcodes. This is the way to load and reference several analysis at the same time.

islot1 – slot where the first analysis was stored

islot2 – slot where the second analysis was stored

kmix – mix value between the two analysis. Should be between 0 and 1. 0 means analysis 1 only. 1 means analysis 2 only. Any value in between will produce interpolation between the filters.

lpinterp computes a new set of poles from the interpolation between two analysis. The poles will be stored in the current **lpslot** and used by the next **lpreson** opcode.

EXAMPLE

Here is a typical orc using the opcodes:

```
    ipower  init      50000          ; Define sound generator
    ifreq   init      440
    asrc    buzz      ipower,ifreq,10,1

    ktime   line      0,p3,p3        ; Define time lin
            lpslot    0              ; Read square data poles
krmsr,krmso,kerr,kcps lpread  ktime,"square.pol"
            lpslot  1                ; Read triangle data poles
krmsr,krmso,kerr,kcps lpread  ktime,"triangle.pol"
    kmix    line      0,p3,1         ; Compute result of mixing
            lpinterp  0,1,kmix       ; and balance power
    ares    lpreson   asrc
    aout    balance   ares,asrc
            out       aout
```

AUTHOR

Mark Resibois
Brussels
1996

This page intentionally left blank.

46 SIGNAL GENERATORS: RANDOM (NOISE) GENERATORS

46.1 rand, randh, randi

```
kr      rand        xamp [, iseed[, isize[, ioffset]]]
kr      randh       kamp, kcps[, iseed[, isize[, ioffset]]]
kr      randi       kamp, kcps[, iseed[, isize[, ioffset]]]
ar      rand        xamp [, iseed[, isize[, ioffset]]]
ar      randh       xamp, xcps[, iseed[, isize[, ioffset]]]
ar      randi       xamp, xcps[, iseed[, isize[, ioffset]]]
```

DESCRIPTION

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

INITIALIZATION

iseed (optional) – seed value for the recursive pseudo-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. A value greater than 1 will obtain the seed value from the system clock. (New in Csound version 4.10.) The default seed value is .5.

isize – if zero, a 16 bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

PERFORMANCE

koffset (optional) – a base value added to the random result. New in Csound version 4.03.

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

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

EXAMPLE

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

46.2 x-class noise generators

```
ir      linrand     krange
kr      linrand     krange
ar      linrand     krange

ir      trirand     krange
kr      trirand     krange
ar      trirand     krange

ir      exprand     krange
kr      exprand     krange
ar      exprand     krange

ir      bexprnd     krange
kr      bexprnd     krange
ar      bexprnd     krange

ir      cauchy      kalpha
kr      cauchy      kalpha
ar      cauchy      kalpha

ir      pcauchy     kalpha
kr      pcauchy     kalpha
ar      pcauchy     kalpha

ir      poisson     klambda
kr      poisson     klambda
ar      poisson     klambda

ir      gauss       krange
kr      gauss       krange
ar      gauss       krange

ir      weibull     ksigma, ktau
kr      weibull     ksigma, ktau
ar      weibull     ksigma, ktau

ir      betarand    krange, kalpha, kbeta
kr      betarand    krange, kalpha, kbeta
ar      betarand    krange, kalpha, kbeta

ir      unirand     krange
kr      unirand     krange
ar      unirand     krange
```

DESCRIPTION

All of the following opcodes operate in i-, k- and a-rate.

linrand *krange* – Linear distribution random number generator. *krange* is the range of the random numbers (0 – *krange*). Outputs only positive numbers.

trirand *krange* – Same as above only outputs both negative and positive numbers.

exprand *krange* – Exponential distribution random number generator. *krange* is the range of the random numbers (0 – *krange*). Outputs only positive numbers.

bexprnd *krange* – Same as above, only extends to negative numbers too with an exponential distribution.

cauchy *kalpha* -Cauchy distribution random number generator. *kalpha* controls the spread from zero (big *kalpha* = big spread). Outputs both positive and negative numbers.

pcauchy *kalpha* – Same as above, outputs positive numbers only.

poisson *klambda* – Poisson distribution random number generator. *klambda* is the mean of the distribution. Outputs only positive numbers.

gauss *krange* – Gaussian distribution random number generator. *krange* is the range of the random numbers (-*krange* – 0 – *krange*). Outputs both positive and negative numbers.

weibull *ksigma*, *ktau* – Weibull distribution random number generator. *ksigma* scales the spread of the distribution and *ktau*, if greater than one numbers near *ksigma* are favored, if smaller than one small values are favored and if t equals 1 the distribution is exponential. Outputs only positive numbers.

betarand *krange*, *kalpha*, *kbeta* – Beta distribution random number generator. *krange* is the range of the random numbers (0 – *krange*). If *kalpha* is smaller than one, smaller values favor values near 0. If *kbeta* is smaller than one, smaller values favor values near *krange*. If both *kalpha* and *kbeta* equal one we have uniform distribution. If both *kalpha* and *kbeta* are greater than one we have a sort of Gaussian distribution. Outputs only positive numbers.

unirand *krange* – Uniform distribution random number generator. *krange* is the range of the random numbers (0 – *krange*).

For more detailed explanation of these distributions, see:

- C. Dodge – T.A. Jerse 1985. Computer music. Schirmer books. pp.265 – 286
- D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 – 379.

EXAMPLE

```
a1  trirand    32000          ; Audio noise with triangle distribution
k1  cauchy     10000          ; Control noise with Cauchy dist.
i1  betarand   30000, .5, .5 ; i-time random value, beta dist.
```

DEPRECATED NAMES

These opcode names originally started with i, k, or a to denote the rate at which the opcode operated. These names are deprecated as of Csound version 3.49. The current form should now be used; the previous form will not work.

AUTHOR

Paris Smaragdis
MIT, Cambridge
1995

46.3 pinkish

```
ar      pinkish         xin[, imethod, inumbands, iseed, iskip]
```

DESCRIPTION

Generates approximate pink noise (-3dB/oct response) by one of two different methods:

- a multirate noise generator after Moore, coded by Martin Gardner
- a filter bank designed by Paul Kellet

INITIALIZATION

imethod (optional) – selects filter method:

- 0 = Gardner method (default).
- 1 = Kellet filter bank.
- 2 = A somewhat faster filter bank by Kellet, with less accurate response.

inumbands (optional) – only effective with Gardner method. The number of noise bands to generate. Maximum is 32, minimum is 4. Higher levels give smoother spectrum, but above 20 bands there will be almost DC-like slow fluctuations. Default value is 20.

iseed (optional) – only effective with Gardner method. If non-zero, seeds the random generator. If zero, the generator will be seeded from current time. Default is 0.

iskip (optional) – if non-zero, skip (re)initialization of internal state (useful for tied notes). Default is 0.

PERFORMANCE

xin – for Gardner method: k- or a-rate amplitude. For Kellet filters: normally a-rate uniform random noise from rand (31-bit) or unirand, but can be any a-rate signal. The output peak value varies widely (±15%) even over long runs, and will usually be well below the input amplitude. Peak values may also occasionally overshoot input amplitude or noise.

pinkish attempts to generate pink noise (i.e., noise with equal energy in each octave), by one of two different methods.

The first method, by Moore & Gardner, adds several (up to 32) signals of white noise, generated at octave rates (sr, sr/2, sr/4 etc). It obtains pseudo-random values from an internal 32-bit generator. This random generator is local to each opcode instance and seedable (similar to **rand**).

The second method is a lowpass filter with a response approximating -3dB/oct. If the input is uniform white noise, it outputs pink noise. Any signal may be used as input for this method. The high quality filter is slower, but has less ripple and a slightly wider operating frequency range than less computationally intense versions. With the Kellet filters, seeding is not used.

The Gardner method output has some frequency response anomalies in the low-mid and high-mid frequency ranges. More low-frequency energy can be generated by increasing the number of bands. It is also a bit faster. The refined Kellet filter has very smooth spectrum, but a more limited effective range. The level increases slightly at the high end of the spectrum.

EXAMPLE

Kellet-filtered noise for a tied note (*iskip* is non-zero).

```
            instr 1
  awhite    unirand       2.0
  awhite    =             awhite - 1.0            ; Normalize to +/-1.0
  apink     pinkish       awhite, 1, 0, 0, 1
            out           apink * 30000
            endin
```

AUTHORS

Phil Burke
John ffitch
University of Bath/Codemist Ltd.
Bath, UK
May, 2000 (New in Csound version 4.06)

46.4 noise

```
ar      noise      xamp, kbeta
```

DESCRIPTION

A white noise generator with an IIR lowpass filter.

PERFORMANCE

xamp – amplitude of final output

kbeta – beta of the lowpass filter. Should be in the range of 0 to 1.

The filter equation is:

```
y_n = sqrt(1-beta^2) * x_n + beta Y_(n-1)
```

where *x_n* is white noise.

AUTHOR

John ffitch
University of Bath, Codemist. Ltd.
Bath, UK
December, 2000
New in Csound version 4.10

47 FUNCTION TABLE CONTROL: TABLE QUERIES

47.1 ftlen, ftlptim, ftsr, nsamp

```
ftlen(x)        (init-rate args only)
ftlptim(x)      (init-rate args only)
ftsr(x)         (init-rate args only)
nsamp(x)        (init-rate args only)
```

DESCRIPTION

Where the argument within the parentheses may be an expression. These value converters return information about a stored function table. The result can be a term in a further expression.

PERFORMANCE

ftlen(*x*) – returns the size (number of points, excluding guard point) of stored function table number *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. Note that **ftlen** will always return a power-of-2 value, i.e. the function table guard point (see **f Statement**) is not included. As of Csound version 3.53, **ftlen** works with deferred function tables (see **GEN01**).

ftlptim(*x*) – returns the loop segment start-time (in seconds) of stored function table number *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.

ftsr(*x*) – returns the sampling-rate of a **GEN01** or **GEN22** generated table. The sampling-rate is determined from the header of the original file. If the original file has no header, or the table was not created by these two GENs **ftsr** returns 0. New in Csound version 3.49.

nsamp(*x*) – returns the number of samples loaded into stored function table number *x* by **GEN01** or **GEN23**. This is useful when a sample is shorter than the power-of-two function table that holds it. New in Csound version 3.49.

AUTHORS

Barry Vercoe
MIT
Cambridge, Massachusetts
1997

Gabriel Maldonado (**ftsr**, **nsamp**)
Italy
October, 1998

47.2 tableng

```
ir       tableng     ifn
kr       tableng     kfn
```

DESCRIPTION

Interrogates a function table for length.

47.2.1 INITIALIZATION

ifn – Table number to be interrogated

47.2.2 PERFORMANCE

kfn – Table number to be interrogated

tableng returns the length of the specified table. This will be a power of two number in most circumstances. It will not show whether a table has a guardpoint or not. It seems this information is not available in the table's data structure. If the specified table is not found, then 0 will be returned.

Likely to be useful for setting up code for table manipulation operations, such as **tablemix** and **tablecopy**.

NAME CHANGES

As of Csound version 3.52, the name of the opcode **itablegpw** has been changed to **tableng**.

AUTHOR

Robin Whittle
Australia
May 1997

48 FUNCTION TABLE CONTROL: TABLE SELECTION

48.1 tablekt, tableikt

```
kr      tablekt     kndx, kfn[, ixmode[, ixoff[, iwrap]]]
ar      tablekt     xndx, kfn[, ixmode[, ixoff[, iwrap]]]
kr      tableikt    kndx, kfn[, ixmode[, ixoff[, iwrap]]]
ar      tableikt    xndx, kfn[, ixmode[, ixoff[, iwrap]]]
```

DESCRIPTION

k-rate control over table numbers.

The standard Csound opcodes **table** and **tablei**, when producing a k- or a-rate result, can only use an init-time variable to select the table number. **tablekt** and **tableikt** accept k-rate control as well as i-time. In all other respects they are similar to the original opcodes.

INITIALIZATION

indx – Index into table, either a positive number range

ifn – Table number. Must be >= 1. Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (**GEN01**) then an error will result and the instrument will be de-activated.

ixmode – if 0, *xndx* and *ixoff* ranges match the length of the table. if non-zero *xndx* and *ixoff* have a 0 to 1 range. Default is 0

ixoff – if 0, total index is controlled directly by *xndx,* i.e. the indexing starts from the start of the table. If non-zero, start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* !=0). Default is 0.

iwrap – if *iwrap* = 0, Limit mode: when total index is below 0, then final index is 0.Total index above table length results in a final index of the table length – high out of range total indexes stick at the upper limit of the table. If *iwrap* !=0, Wrap mode: total index is wrapped modulo the table length so that all total indexes map into the table. For instance, in a table of length 8, *xndx* = 5 and *ixoff* = 6 gives a total index of 11, which wraps to a final index of 3. Default is 0.

PERFORMANCE

kndx – Index into table, either a positive number range

xndx – matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0)

kfn – Table number. Must be >= 1. Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (**GEN01**) then an error will result and the instrument will be de-activated.

AUTHOR

Robin Whittle
Australia
1997

49 FUNCTION TABLE CONTROL: READ/WRITE OPERATIONS

49.1 tableiw, tablew, tablewkt

```
tableiw     isig, indx, ifn[, ixmode[, ixoff[, iwgmode]]]
tablew      ksig, kndx, ifn[, ixmode[, ixoff[, iwgmode]]]
tablew      asig, andx, ifn[, ixmode[, ixoff[, iwgmode]]]
tablewkt    ksig, kndx, kfn[, ixmode[, ixoff[, iwgmode]]]
tablewkt    asig, andx, kfn[, ixmode[, ixoff[, iwgmode]]]
```

DESCRIPTION

These opcodes operate on existing function tables, changing their contents. **tableiw** is used when all inputs are init time variables or constants and you only want to run it at the initialization of the instrument. **tablew** is for writing at k- or at a-rates, with the table number being specified at init time. **tablewkt** is the same, but uses a k-rate variable for selecting the table number. The valid combinations of variable types are shown by the first letter of the variable names.

INITIALIZATION

isig, *ksig*, *asig*- The value to be written into the table.

indx, *kndx*, *andx* – Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0)

ifn, *kfn* – Table number. Must be = 1. Floats are rounded down to an integer. If a table number does not point to a valid table, or the table has not yet been loaded (**GEN01**) then an error will result and the instrument will be de-activated.

ixmode – Default is 0.

- 0 : xndx and ixoff ranges match the length of the table.
- !=0 : xndx and ixoff have a 0 to 1 range.

ixoff – Default is 0.

- 0: Total index is controlled directly by *xndx*. i.e. the indexing starts from the start of the table.
- !=0: Start indexing from somewhere else in the table. Value must be positive and less than the table length (*ixmode* = 0) or less than 1 (*ixmode* !=0).

iwgmode – Default is 0.

- 0: Limit mode
- 1: Wrap mode
- 2: Guardpoint mode.

PERFORMANCE

Limit mode (0)

Limit the total index (*ndx* + *ixoff*) to between 0 and the guard point. For a table of length 5, this means that locations 0 to 3 and location 4 (the guard point) can be written. A negative total index writes to location 0. Total indexes 4 write to location 4.

Wrap mode (1)

Wrap total index value into locations 0 to E, where E is one less than either the table length or the factor of 2 number which is one less than the table length. For example, wrap into a 0 to 3 range – so that total index 6 writes to location 2.

Guardpoint mode (2)

The guardpoint is written at the same time as location 0 is written – with the same value.

This facilitates writing to tables which are intended to be read with interpolation for producing smooth cyclic waveforms. In addition, before it is used, the total index is incremented by half the range between one location and the next, before being rounded down to the integer address of a table location.

Normally (*igwmode* = 0 or 1) for a table of length 5 – which has locations 0 to 3 as the main table and location 4 as the guard point, a total index in the range of 0 to 0.999 will write to location 0. ("0.999" means just less than 1.0.) 1.0 to 1.999 will write to location 1, etc. A similar pattern holds for all total indexes 0 to 4.999 (*igwmode* = 0) or to 3.999 (*igwmode* = 1). *igwmode* = 0 enables locations 0 to 4 to be written – with the guardpoint (4) being written with a potentially different value from location 0.

With a table of length 5 and the *iwgmode* = 2, then when the total index is in the range 0 to 0.499, it will write to locations 0 and 4. Range 0.5 to 1.499 will write to location 1 etc. 3.5 to 4.0 will also write to locations 0 and 4.

This way, the writing operation most closely approximates the results of interpolated reading. Guard point mode should only be used with tables that have a guardpoint.

Guardpoint mode is accomplished by adding 0.5 to the total index, rounding to the next lowest integer, wrapping it modulo the factor of two which is one less than the table length, writing the table (locations 0 to 3 in our example) and then writing to the guard point if index == 0.

tablew has no output value. The last three parameters are optional and have default values of 0.

Caution with k-rate table numbers :

The following notes also apply to the tablekt and tableikt opcodes which can now have their table number changed at k-rate.

At k-rate or a-rate, if a table number of < 1 is given, or the table number points to a non-existent table, or to one which has a length of 0 (it is to be loaded from a file later) then an error will result and the instrument will be deactivated. *kfn* and *afn* must be initialized at the appropriate rate using **init**. Attempting to load an i-rate value into *kfn* or *afn* will result in an error.

CHANGED NAME

As of Csound version 3.52, the opcode name **itablew** is changed to **tableiw**.

AUTHOR

Robin Whittle
Australia
May 1997

49.2 tablegpw, tablemix, tablecopy, tableigpw, tableimix, tableicopy

```
tablegpw    kfn
tablemix    kdft, kdoff, klen, ks1ft, ks1off, ks1g, ks2ft, \\
            ks2off, ks2g
tablecopy   kdft, ksft
tableigpw   ifn
tableimix   idft, idoff, ilen, is1ft, is1off, is1g, is2ft, \\
            is2off, is2g
tableicopy  idft, isft
```

DESCRIPTION

These opcodes allow tables to be copied and mixed.

INITIALIZATION

ifn – Function table number

PERFORMANCE

kfn – Function table number

kdft – Destination function table number

kdoff – Offset to start writing from. Can be negative.

kdft – Number of destination function table.

ksft – Number of source function table.

klen – Number of write operations to perform. Negative means work backwards.

ks1ft, *ks2ft* – Source function tables. These can be the same as the destination table, if care is exercised about direction of copying data.

ks1off, *ks2off* –Offsets to start reading from in source tables.

ks1g, *ks2g* – Gains to apply when reading from the source tables. The results are added and the sum is written to the destination table.

tablgpw – For writing the table's guard point, with the value which is in location 0. Does nothing if table does not exist.

Likely to be useful after manipulating a table with **tablemix** or **tablecopy.**

tablemix – This opcode mixes from two tables, with separate gains into the destination table. Writing is done for *klen* locations, usually stepping forward through the table – if *klen* is positive. If it is negative, then the writing and reading order is backwards – towards lower indexes in the tables. This bi-directional option makes it easy to shift the contents of a table sideways by reading from it and writing back to it with a different offset.

If *klen* is 0, no writing occurs. Note that the internal integer value of *klen* is derived from the ANSI C floor() function – which returns the next most negative integer. Hence a fractional negative *klen* value of -2.3 would create an internal length of 3, and cause the copying to start from the offset locations and proceed for two locations to the left.

The total index for table reading and writing is calculated from the starting offset for each table, plus the index value, which starts at 0 and then increments (or decrements) by 1 as mixing proceeds.

These total indexes can potentially be very large, since there is no restriction on the offset or the *klen*. However each total index for each table is ANDed with a length mask (such as 0000 0111 for a table of length 8) to form a final index which is actually used for reading or writing. So no reading or writing can occur outside the tables. This is the same as "wrap" mode in table read and write. These opcodes do not read or write the guardpoint. If a table has been rewritten with one of these, then if it has a guardpoint which is supposed to contain the same value as the location 0, then call **tablegpw** afterwards.

The indexes and offsets are all in table steps – they are not normalized to 0 – 1. So for a table of length 256, *klen* should be set to 256 if all the table was to be read or written.

The tables do not need to be the same length – wrapping occurs individually for each table.

tablecopy – Simple, fast table copy opcodes. Takes the table length from the destination table, and reads from the start of the source table. For speed reasons, does not check the source length – just copies regardless – in "wrap" mode. This may read through the source table several times. A source table with length 1 will cause all values in the destination table to be written to its value.

tablecopy cannot read or write the guardpoint. To read it use **table**, with *ndx* = the table length. Likewise use table write to write it.

To write the guardpoint to the value in location 0, use **tablegpw**.

This is primarily to change function tables quickly in a real-time situation.

NAME CHANGES

As of Csound version 3.52, the names of the opcodes **itablegpw**, **itablemix**, and **itablecopy**, have been changed to **tableigpw**, **tableimix**, and **tableicopy**, respectively.

AUTHOR

Robin Whittle
Australia
May 1997

49.3 tablera, tablewa

```
ar       tablera     kfn, kstart, koff
kstart   tablewa     kfn, asig, koff
```

DESCRIPTION

These opcodes read and write tables in sequential locations to and from an a-rate variable. Some thought is required before using them. They have at least two major, and quite different, applications which are discussed below.

INITIALIZATION

ar – a-rate destination for reading ksmps values from a table.

kfn – i- or k-rate number of the table to read or write.

kstart – Where in table to read or write.

asig – a-rate signal to read from when writing to the table.

koff – i- or k-rate offset into table. Range unlimited – see explanation at end of this section.

PERFORMANCE

In one application, these are intended to be used in pairs, or with several **tablera** opcodes before a **tablewa** – all sharing the same *kstart* variable.

These read from and write to sequential locations in a table at audio rates, with ksmps floats being written and read each cycle.

tablera starts reading from location *kstart*. **tablewa** starts writing to location *kstart*, and then writes to *kstart* with the number of the location one more than the one it last wrote. (Note that for **tablewa**, *kstart* is both an input and output variable.) If the writing index reaches the end of the table, then no further writing occurs and zero is written to *kstart*.

For instance, if the table's length was 16 (locations 0 to 15), and ksmps was 5. Then the following steps would occur with repetitive runs of the **tablewa** opcode, assuming that *kstart* started at 0.

```
Run no. Initial  Final    locations written
        kstart   kstart
1       0        5        0 1 2 3 4
2       5        10       5 6 7 8 9
3       10       15       10 11 12 13 14
4       15       0        15
```

This is to facilitate processing table data using standard a-rate orchestra code between the **tablera** and **tablewa** opcodes. They allow all Csound k-rate operators to be used (with caution) on a-rate variables – something that would only be possible otherwise by **ksmps** = 1, downsamp and upsamp.

Several cautions:

- The k-rate code in the processing loop is really running at a-rate, so time dependant functions like **port** and **oscil** work faster than normal – their code is expecting to be running at k-rate.
- This system will produce undesirable results unless the ksmps fits within the table length. For instance a table of length 16 will accommodate 1 to 16 samples, so this example will work with **ksmps** = 1 to 16.
- Both these opcodes generate an error and deactivate the instrument if a table with length < **ksmps** is selected. Likewise an error occurs if *kstart* is below 0 or greater than the highest entry in the table – if *kstart* = table length.
- *kstart* is intended to contain integer values between 0 and (table length – 1). Fractional values above this should not affect operation but do not achieve anything useful.
- These opcodes are not interpolating, and the *kstart* and *koff* parameters always have a range of 0 to (table length – 1) – not 0 to 1 as is available in other table read/write opcodes. *koff* can be outside this range but it is wrapped around by the final AND operation.
- These opcodes are permanently in wrap mode. When *koff* is 0, no wrapping needs to occur, since the *kstart*++ index will always be within the table's normal range. *koff* != 0 can lead to wrapping.
- The offset does not affect the number of read/write cycles performed, or the value written to *kstart* by **tablewa**.
- These opcodes cannot read or write the guardpoint. Use **tablegpw** to write the guardpoint after manipulations have been done with **tablewa**.

EXAMPLES

```
kstart   =       0

lab1:
  atemp  tablera ktabsource, kstart, 0  ; Read 5 values from table into an
                                        ; a-rate variable.

  atemp  =       log(atemp)      ; Process the values using a-rate
                                        ; code.

  kstart tablewa ktabdest, atemp, 0     ; Write it back to the table

if ktemp  0 goto lab1                   ; Loop until all table locations
                                        ; have been processed.
```

The above example shows a processing loop, which runs every k-cycle, reading each location in the table ktabsource, and writing the log of those values into the same locations of table *ktabdest*.

This enables whole tables, parts of tables (with offsets and different control loops) and data from several tables at once to be manipulated with a-rate code and written back to another (or to the same) table. This is a bit of a fudge, but it is faster than doing it with k-rate table read and write code.

Another application is:

```
  kzero = 0
  kloop = 0

  kzero tablewa 23, asignal, 0   ; ksmps a-rate samples written
                                 ; into locations 0 to (ksmps -1) of table 23.

  lab1: ktemp table kloop, 23    ; Start a loop which runs ksmps times,
                                        ; in which each cycle processes one of
   [ Some code to manipulate ]   ; table 23's values with k-rate orchestra
   [ the value of ktemp. ] ; code.
```

```
 tablew ktemp, kloop, 23 ; Write the processed value to the table.

kloop = kloop + 1                    ; Increment the kloop, which is both the
                                     ; pointer into the table and the loop
if kloop < ksmps goto lab1     ; counter. Keep looping until all values
                                     ; in the table have been processed.

asignal   tablera 23, 0, 0     ; Copy the table contents back
                                     ; to an a-rate variable.
```

koff – This is an offset which is added to the sum of *kstart* and the internal index variable which steps through the table. The result is then ANDed with the lengthmask (000 0111 for a table of length 8 – or 9 with guardpoint) and that final index is used to read or write to the table. *koff* can be any value. It is converted into a long using the ANSI floor() function so that -4.3 becomes -5. This is what we would want when using offsets which range above and below zero.

Ideally this would be an optional variable, defaulting to 0, however with the existing Csound orchestra read code, such default parameters must be init time only. We want k-rate here, so we cannot have a default.

This page intentionally left blank.

50 SIGNAL MODIFIERS: STANDARD FILTERS

50.1 port, portk, tone, tonek, atone, atonek, reson, resonk, areson, aresonk

```
kr      port        ksig, ihtim[, isig]
kr      portk       ksig, khtim[, isig]
kr      tonek       ksig, khp[, iskip]
kr      atonek      ksig, khp[, iskip]
kr      resonk      ksig, kcf, kbw[, iscl, iskip]
kr      aresonk     ksig, kcf, kbw[, iscl, iskip]
ar      tone        asig, khp[, iskip]
ar      atone       asig, khp[, iskip]
ar      reson       asig, kcf, kbw[, iscl, iskip]
ar      areson      asig, kcf, kbw[, iscl, iskip]
```

DESCRIPTION

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.

iskip – 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 (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. With **portk**, the half-time can be varied at the control rate.

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

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

atone, **areson** are filters whose transfer functions are the complements of **tone** and **reso**n. **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. This property is particularly useful for controlled mixing of different sources (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**.

50.2 tonex, atonex, resonx

```
ar      tonex       asig, khp[, inumlayer, iskip]
ar      atonex      asig, khp[, inumlayer, iskip]
ar      resonx      asig, kcf, kbw[, inumlayer, iscl, iskip]
```

DESCRIPTION

tonex, **atonex** and **resonx** are equivalent to filters consisting of more layers of **tone**, **atone** and **reson**, with the same arguments, serially connected. Using a stack of a larger number of filters allows a sharper cutoff. They are faster than using a larger number instances in a Csound orchestra of the old opcodes, because only one initialization and 'k' cycle are needed at time, and the audio loop falls entirely inside the cache memory of processor.

INITIALIZATION

inumlayer – number of elements in the filter stack.. Default value is 4.

iskip – 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 (see **balance**). The default value is 0.

PERFORMANCE

asig – input signal

khp – the response curve's half-power point. Half power is defined as peak power / root 2.

kcf – the center frequency of the filter, or frequency position of the peak response.

kbw – bandwidth of the filter (the Hz difference between the upper and lower half-power points)

AUTHOR

Gabriel Maldonado (adapted by John ffitch)
Italy
New in Csound version 3.49

50.3 resonr, resonz

```
ar      resonr    asig, kcf, kbw[,iscl, iskip]
ar      resonz    asig, kcf, kbw[,iscl, iskip]
```

DESCRIPTION

Implementations of a second-order, two-pole two-zero bandpass filter with variable frequency response.

INITIALIZATION

The optional initialization variables for **resonr** and **resonz** are identical to the i-time variables for **reson**.

iskip – 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 (see **balance**). The default value is 0.

PERFORMANCE

resonr and **resonz** are variations of the classic two-pole bandpass resonator (**reson**). Both filters have two zeroes in their transfer functions, in addition to the two poles. **resonz** has its zeroes located at z = 1 and z = -1. **resonr** has its zeroes located at +sqrt(*R*) and -sqrt(*R*), where *R* is the radius of the poles in the complex z-plane. The addition of zeroes to **resonr** and **resonz** results in the improved selectivity of the magnitude response of these filters at cutoff frequencies close to 0, at the expense of less selectivity of frequencies above the cutoff peak.

resonr and **resonz** are very close to constant-gain as the center frequency is swept, resulting in a more efficient control of the magnitude response than with traditional two-pole resonators such as **reson**.

resonr and **resonz** produce a sound that is considerably different from **reson**, especially for lower center frequencies; trial and error is the best way of determining which resonator is best suited for a particular application.

asig – input signal to be filtered

kcf – cutoff or resonant frequency of the filter, measured in Hz

kbw – bandwidth of the filter (the Hz difference between the upper and lower half-power points)

EXAMPLE

```
; Orchestra file for resonant filter sweep of a sawtooth-like waveform.
; The outputs of reson, resonr, and resonz are scaled by coefficients
; specified in the score, so that each filter can be heard on its own
; from the same instrument.

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

           instr 1

idur     =        p3
ibegfreq =        p4      ; beginning of sweep frequency
iendfreq =        p5      ; ending of sweep frequency
ibw      =        p6      ; bandwidth of filters in Hz
ifreq    =        p7      ; frequency of gbuzz that is to be filtered
iamp     =        p8      ; amplitude to scale output by
ires     =        p9      ; coefficient to scale amount of reson in output
iresr    =        p10     ; coefficient to scale amount of resonr in output
iresz    =        p11     ; coefficient to scale amount of resonz in output

; Frequency envelope for reson cutoff
kfreq    linseg   ibegfreq, idur * .5, iendfreq, idur * .5, ibegfreq

; Amplitude envelope to prevent clicking
kenv     linseg   0, .1, iamp, idur - .2, iamp, .1, 0

; Number of harmonics for gbuzz scaled to avoid aliasing
iharms   =        (sr*.4)/ifreq

asig     gbuzz    1, ifreq, iharms, 1, .9, 1      ; “Sawtooth” waveform
ain      =        kenv * asig                     ; output scaled by amp
                                                  ; envelope
ares     reson    ain, kfreq, ibw, 1
aresr    resonr   ain, kfreq, ibw, 1
aresz    resonz   ain, kfreq, ibw, 1

         out      ares * ires + aresr * iresr + aresz * iresz

           endin

; Score file for above
f1 0 8192 9 1 1 .25                  ; cosine table for gbuzz generator

i1  0 10 1 3000 200 100 4000 1 0 0    ; reson  output with bw = 200
i1 10 10 1 3000 200 100 4000 0 1 0    ; resonr output with bw = 200
i1 20 10 1 3000 200 100 4000 0 0 1    ; resonz output with bw = 200
i1 30 10 1 3000  50 200 8000 1 0 0    ; reson  output with bw = 50
i1 40 10 1 3000  50 200 8000 0 1 0    ; resonr output with bw = 50
i1 50 10 1 3000  50 200 8000 0 0 1    ; resonz output with bw = 50
e
```

TECHNICAL HISTORY

resonr and **resonz** were originally described in an article by Julius O. Smith and James B. Angell [1]. Smith and Angell recommended the **resonz** form (zeros at +1 and -1) when computational efficiency was the main concern, as it has one less multiply per sample, while **resonr** (zeroes at + and – the square root of the pole radius R) was recommended for situations when a perfectly constant-gain center peak was required.

Ken Steiglitz, in a later article [2], demonstrated that **resonz** had constant gain at the true peak of the filter, as opposed to **resonr**, which displayed constant gain at the pole angle. Steiglitz also recommended **resonz** for its sharper notches in the gain curve at zero and Nyquist frequency. Steiglitz's recent book [3] features a thorough technical discussion of **reson** and **resonz**, while Dodge and Jerse's textbook [4] illustrates the differences in the response curves of **reson** and **resonz**.

REFERENCES

1. 1. Smith, Julius O. and Angell, James B., "A Constant-Gain Resonator Tuned by a Single Coefficient," *Computer Music Journal*, vol. 6, no. 4, pp. 36-39, Winter 1982.

2. 2. Steiglitz, Ken, "A Note on Constant-Gain Digital Resonators," *Computer Music Journal*, vol. 18, no. 4, pp. 8-10, Winter 1994.

3. 3. Ken Steiglitz, *A Digital Signal Processing Primer, with Applications to Digital Audio and Computer Music*. Addison-Wesley Publishing Company, Menlo Park, CA, 1996.

4. 4. Dodge, Charles and Jerse, Thomas A., *Computer Music: Synthesis, Composition, and Performance*. New York: Schirmer Books, 1997, 2nd edition, pp. 211-214.

AUTHOR

Sean Costello
Seattle, Washington
1999

New in Csound version 3.55

50.4 resony

```
ar      resony      asig, kbf, kbw, inum, ksep[, isepmode, iscl, iskip]
```

DESCRIPTION

A bank of second-order bandpass filters, connected in parallel.

INITIALIZATION

inum – number of filters

isepmode – if *isepmode* = 0, the separation of center frequencies of each filter is generated logarithmically (using octave as unit of measure). If *isepmode* != 0, the separation of center frequencies of each filter is generated linearly (using Hertz). 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. **balance**). The default value is 0.

iskip – 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.

PERFORMANCE

asig – audio input signal

kbf – base frequency, i.e. center frequency of lowest filter in Hz

kbw – bandwidth in Hz

ksep – separation of the center frequency of filters in octaves

resony is a bank of second-order bandpass filters, with k-rate variant frequency separation, base frequency and bandwidth, connected in parallel (i.e. the resulting signal is a mix of the output of each filter). The center frequency of each filter depends of *kbf* and *ksep* variables. The maximum number of filters is set to 100.

EXAMPLE

In this example the global variable *gk1* modifies *kbf*, *gk2* modifies *kbw*, *gk3 inum*, *gk4 ksep*, and *gk5* the main volume.

```
            instr       1
a1          soundin     "myfile.aif"
a2          resony      a1, gk1, gk2, i(gk3), gk4, 2
            out         a2 * gk5
            endin
```

AUTHOR

Gabriel Maldonado
Italy
1999
New in Csound version 3.56

50.5 lowres, lowresx

```
ar      lowres      asig, kcutoff, kresonance [,iskip]
ar      lowresx     asig, kcutoff, kresonance [, inumlayer, iskip]
```

DESCRIPTION

lowres is a resonant lowpass filter. **lowresx** is equivalent to more layer of **lowres**, with the same arguments, serially connected.

INITIALIZATION

inumlayer – number of elements in a **lowresx** stack. Default value is 4. There is no maximum.

iskip – initial disposition of 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

asig – input signal

kcutoff – filter cutoff frequency point

kresonance – resonance amount

lowres is a resonant lowpass filter derived from a Hans Mikelson orchestra. This implementation is much faster than implementing it in Csound language, and it allows **kr** lower than **sr**. *kcutoff* is not in Hz and *kresonance* is not in dB, so experiment for the finding best results.

lowresx is equivalent to more layer of **lowres**, with the same arguments, serially connected. Using a stack of a larger number of filters allows a sharper cutoff. This is faster than using a larger number of instances of **lowres** in a Csound orchestra, because only one initialization and k cycle are needed at time, and the audio loop falls entirely inside the cache memory of processor. Based on an orchestra by Hans Mikelson.

AUTHOR

Gabriel Maldonado (adapted by John ffitch)
Italy
New in Csound version 3.49

50.6 vlowres

```
ar      vlowres     asig, kfco, kres, iord, ksep
```

DESCRIPTION

A bank of filters in which the cutoff frequency can be separated under user control

INITIALIZATION

iord – total number of filters (1 to 10)

PERFORMANCE

asig – input signal

kfco – frequency cutoff (not in Hz)

ksep – frequency cutoff separation for each filter

vlowres (variable resonant lowpass filter) allows a variable response curve in resonant filters. It can be thought of as a bank of lowpass resonant filters, each with the same resonance, serially connected. The frequency cutoff of each filter can vary with the *kcfo* and *ksep* parameters.

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.49

50.7 lowpass2

```
ar      lowpass2    asig, kcf, kq[, iskip]
```

DESCRIPTION

Implementation of resonant second-order lowpass filter.

INITIALIZATION

iskip – initial disposition of 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

asig – input signal to be filtered

kcf – cutoff or resonant frequency of the filter, measured in Hz

kq – Q of the filter, defined, for bandpass filters, as bandwidth/cutoff. *kq* should be between 1 and 500

lowpass2 is a second order IIR lowpass filter, with k-rate controls for cutoff frequency (*kcf*) and Q (*kq*). As *kq* is increased, a resonant peak forms around the cutoff frequency, transforming the lowpass filter response into a response that is similar to a bandpass filter, but with more low frequency energy. This corresponds to an increase in the magnitude and "sharpness" of the resonant peak. For high values of *kq*, a scaling function such as **balance** may be required. In practice, this allows for the simulation of the voltage-controlled filters of analog synthesizers, or for the creation of a pitch of constant amplitude while filtering white noise.

EXAMPLE

```
; Orchestra file for resonant filter sweep of a sawtooth-like waveform.
  sr     = 44100
  kr     = 2205
  ksmps  = 20
  nchnls = 1

             instr 1

idur         =              p3
ifreq        =              p4
iamp         =              p5 * .5
iharms       =              (sr*.4) / ifreq

; Sawtooth-like waveform
asig         gbuzz          1, ifreq, iharms, 1, .9, 1

; Envelope to control filter cutoff
kfreq        linseg         1, idur * 0.5, 5000, idur * 0.5, 1

afilt        lowpass2       asig, kfreq, 30
```

```
; Simple amplitude envelope
kenv        linseg          0, .1, iamp, idur -.2, iamp, .1, 0
  out       asig * kenv

  endin


; Score file for above
f1 0 8192 9 1 1 .25

i1 0 5 100 1000
i1 5 5 200 1000
e
```

AUTHOR

Sean Costello
Seattle, Washington
August, 1999
New in Csound version 4.0

50.8 biquad, rezzy, moogvcf

```
ar      biquad      asig, kb0, kb1, kb2, ka0, ka1, ka2[, iskip]

ar      rezzy       asig, xfco, xres[, imode]

ar      moogvcf     asig, xfco, xres[, iscale]
```

DESCRIPTION

Implementation of a sweepable general purpose filter and two sweepable, resonant low-pass filters.

INITIALIZATION

iskip (optional) – if non-zero, initialization will be skipped. Default value 0. (New in Csound version 3.50)

imode (optional) – if zero **rezzy** is low-pass, if nonzero, high-pass. Default value is 0. (New in Csound version 3.50)

iscale (optional) – internal scaling factor. Use if *asig* is not in the range +/-1. Input is first divided by *iscale*, then output is multiplied *iscale*. Default value is 1. (New in Csound version 3.50)

PERFORMANCE

asig – input signal

xfco – filter cut-off frequency in Hz. As of version 3.50, may i-,k-, or a-rate.

xres – amount of resonance. For **rezzy**, values of 1 to 100 are typical. Resonance should be one or greater. For **moogvcf**, self-oscillation occurs when *xres* is approximately one. As of version 3.50, may i-,k-, or a-rate.

biquad is a general purpose biquadratic digital filter of the form:

```
a0*y(n) + a1*y[n-1] + a2*y[n-2] = b0*x[n] + b1*x[n-1] + b2*x[n-2]
```

This filter has the following frequency response:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + b2*Z^{-2}}{a0 + a1*Z^{-1} + a2*Z^{-2}}$$

This type of filter is often encountered in digital signal processing literature. It allows six user-defined k-rate coefficients.

rezzy is a resonant low-pass filter created empirically by Hans Mikelson.

moogvcf is a digital emulation of the Moog diode ladder filter configuration. This emulation is based loosely on the paper "Analyzing the Moog VCF with Considerations for Digital Implementation" by Stilson and Smith (CCRMA). This version was originally coded in Csound by Josep Comajuncosas. Some modifications and conversion to C were done by Hans Mikelson.

Note: This filter requires that the input signal be normalized to one.

EXAMPLES

```
;biquad example
kfcon       =           *3.14159265*kfco/sr
kalpha      =           -2*krez*cos(kfcon)*cos(kfcon)+krez*krez*cos(2*kfcon)
kbeta       =           *krez*sin(2*kfcon)-2*krez*cos(kfcon)*sin(kfcon)
kgama       =           +cos(kfcon)
km1         =           *kgama+kbeta*sin(kfcon)
km2         =           *kgama-kbeta*sin(kfcon)
kden        =           (km1*km1+km2*km2)
kb0         =           .5*(kalpha*kalpha+kbeta*kbeta)/kden
kb1         =           kb0
kb2         =           0
ka0         =           1
ka1         =           -2*krez*cos(kfcon)
ka2         =           krez*krez
ayn         biquad      axn, kb0, kb1, kb2, ka0, ka1, ka2
  outs      ayn*iamp/2, ayn*iamp/2

;   Sta  Dur  Amp    Pitch Fco   Rez
i14 8.0  1.0  20000  6.00  1000  .8
i14 +    1.0  20000  6.03  2000  .95


;rezzy example
kfco        expseg      100+.01*ifco, .2*idur, ifco+100, .5*idur, ifco*.1+100,
.3*idur, .001*ifco+100
apulse1     buzz        1,ifqc, sr/2/ifqc, 1 ; Avoid aliasing
asaw        integ       apulse1
axn         =           asaw-.5
ayn         rezzy       axn, kfco, krez
  outs      ayn*iamp, ayn*iamp

;    Sta  Dur  Amp    Pitch  Fco   Rez
i10  0.0  1.0  20000  6.00   1000  2
i10  +    1.0  20000  6.03   2000  10


;moogvcf example
apulse1     buzz        1,ifqc, sr/2/ifqc, 1 ; Avoid aliasing
asaw        integ       apulse1
ax          =           asaw-.5
ayn         moogvcf     ax, kfco, krez
  outs      ayn*iamp, ayn*iamp

;     Sta  Dur  Amp    Pitch  Fco   Rez
i11   4.0  1.0  20000  6.00   1000  .4
i11   +    1.0  20000  6.03   2000  .7
```

AUTHOR

Hans Mikelson
October 1998
New in Csound version 3.49

50.9 svfilter

```
alow, ahigh, aband  svfilter  asig, kcf, kq[, iscl]
```

DESCRIPTION

Implementation of a resonant second order filter, with simultaneous lowpass, highpass and bandpass outputs.

INITIALIZATION

iscl – coded scaling factor, similar to that in **reson**. A non-zero value signifies a peak response factor of 1, i.e. all frequencies other than *kcf* are attenuated in accordance with the (normalized) response curve. A zero value signifies no scaling of the signal, leaving that to some later adjustment (see **balance**). The default value is 0.

PERFORMANCE

svfilter is a second order state-variable filter, with k-rate controls for cutoff frequency and Q. As Q is increased, a resonant peak forms around the cutoff frequency. **svfilter** has simultaneous lowpass, highpass, and bandpass filter outputs; by mixing the outputs together, a variety of frequency responses can be generated. The state-variable filter, or "multimode" filter was a common feature in early analog synthesizers, due to the wide variety of sounds available from the interaction between cutoff, resonance, and output mix ratios. **svfilter** is well suited to the emulation of "analog" sounds, as well as other applications where resonant filters are called for.

asig – Input signal to be filtered.

kcf – Cutoff or resonant frequency of the filter, measured in Hz.

kq – Q of the filter, which is defined (for bandpass filters) as bandwidth/cutoff. *kq* should be in a range between 1 and 500. As *kq* is increased, the resonance of the filter increases, which corresponds to an increase in the magnitude and "sharpness" of the resonant peak. When using **svfilter** without any scaling of the signal (where *iscl* is either absent or 0), the volume of the resonant peak increases as Q increases. For high values of Q, it is recommended that *iscl* be set to a non-zero value, or that an external scaling function such as **balance** is used.

svfilter is based upon an algorithm in Hal Chamberlin's *Musical Applications of Microprocessor*s (Hayden Books, 1985).

EXAMPLE

```
; Orchestra file for resonant filter sweep of a sawtooth-like waveform.
; The separate outputs of the filter are scaled by values from the score,
; and are mixed together.
sr     = 44100
kr     = 2205
ksmps  = 20
nchnls = 1


           instr 1

idur     = p3
ifreq    = p4
iamp     = p5
ilowamp  = p6              ; determines amount of lowpass output in signal
ihighamp = p7              ; determines amount of highpass output in signal
```

```
ibandamp = p8              ; determines amount of bandpass output in signal
iq       = p9              ; value of q

iharms   =        (sr*.4) / ifreq

asig     gbuzz     1, ifreq, iharms, 1, .9, 1          ; Sawtooth-like
                                                       ; waveform
kfreq    linseg    1, idur * 0.5, 4000, idur * 0.5, 1  ; Envelope to control
                                                       ; filter cutoff

alow, ahigh, aband   svfilter  asig, kfreq, iq

aout1    =         alow * ilowamp
aout2    =         ahigh * ihighamp
aout3    =         aband * ibandamp
asum     =         aout1 + aout2 + aout3
kenv     linseg    0, .1, iamp, idur -.2, iamp, .1, 0  ; Simple amplitude
                                                       ; envelope
         out       asum * kenv

            endin


; Score file for above
f1 0 8192 9 1 1 .25

i1  0 5 100 1000 1 0 0  5  ; lowpass sweep
i1  5 5 200 1000 1 0 0 30  ; lowpass sweep, octave higher, higher q
i1 10 5 100 1000 0 1 0  5  ; highpass sweep
i1 15 5 200 1000 0 1 0 30  ; highpass sweep, octave higher, higher q
i1 20 5 100 1000 0 0 1  5  ; bandpass sweep
i1 25 5 200 1000 0 0 1 30  ; bandpass sweep, octave higher, higher q
i1 30 5 200 2000 .4 .6  0  ; notch sweep - notch formed by combining
                           ; highpass and lowpass outputs
e
```

AUTHOR

Sean Costello
Seattle, Washington
1999
New in Csound version 3.55

50.10 hilbert

ar1, ar2 **hilbert** asig

DESCRIPTION

An IIR implementation of a Hilbert transformer.

PERFORMANCE

asig – input signal

ar1 – cosine output of *asig*

ar2 – sine output of *asig*

hilbert is an IIR filter based implementation of a broad-band 90 degree phase difference network. The input to **hilbert** is an audio signal, with a frequency range from 15 Hz to 15 kHz. The outputs of **hilbert** have an identical frequency response to the input (i.e. they sound the same), but the two outputs have a constant phase difference of 90 degrees, plus or minus some small amount of error, throughout the entire frequency range. The outputs are in quadrature.

hilbert is useful in the implementation of many digital signal processing techniques that require a signal in phase quadrature. *ar1* corresponds to the cosine output of **hilbert**, while *ar2* corresponds to the sine output. The two outputs have a constant phase difference throughout the audio range that corresponds to the phase relationship between cosine and sine waves.

Internally, **hilbert** is based on two parallel 6th-order allpass filters. Each allpass filter implements a phase lag that increases with frequency; the difference between the phase lags of the parallel allpass filters at any given point is approximately 90 degrees.

Unlike an FIR-based Hilbert transformer, the output of **hilbert** does not have a linear phase response. However, the IIR structure used in **hilbert** is far more efficient to compute, and the nonlinear phase response can be used in the creation of interesting audio effects, as in the second example below.

EXAMPLES

The first example implements frequency shifting, or single sideband amplitude modulation. Frequency shifting is similar to ring modulation, except the upper and lower sidebands are separated into individual outputs. By using only one of the outputs, the input signal can be "detuned," where the harmonic components of the signal are shifted out of harmonic alignment with each other, e.g. a signal with harmonics at 100, 200, 300, 400 and 500 Hz, shifted up by 50 Hz, will have harmonics at 150, 250, 350, 450, and 550 Hz.

```
sr      = 44100
kr      = 4410
ksmps   = 10
nchnls  = 2

                instr 1

idur          =         p3
ibegshift     =         p4      ; initial amount of frequency shift-
                                ; can be positive or negative
iendshift     =         p5      ; final amount of frequency shift-
                                ; can be positive or negative
```

```
kfreq         linseg    ibegshift, idur, iendshift ; A simple envelope
                                                   ; for determining
                                                   ; the amount of
                                                   ; frequency shift.


ain           soundin   "supertest.wav" ; Use the sound of your choice.


areal, aimag  hilbert     ain   ; Phase quadrature output derived from
                                        ; input signal.


asin          oscili  1, kfreq, 1       ; Quadrature oscillator.
acos          oscili  1, kfreq, 1, .25


amod1         =       areal * acos      ; Trigonometric identity-
                                        ; see references for further
                                        ; details.
amod2         =       aimag * asin



                                          ; Both sum and difference
                                          ; frequencies can be
                                          ; output at once.
aupshift      =       (amod1 + amod2) * 0.7  ; aupshift corresponds to
                                             ; the sum frequencies, while
adownshift    =       (amod1 - amod2) * 0.7  ; adownshift corresponds to
                                             ; the difference frequencies.
                                          ; Notice that the adding of
                                          ; the two together is
                                          ; identical to the output of
                                          ; ring modulation.

              outs    aupshift, aupshift
                endin

; a simple score
 f1 0 16384 10 1                ; sine table for quadrature oscillator
 i1 0 29 0 200                  ; starting with no shift, ending with all
                                ; frequencies shifted up by 200 Hz.
 i1 30 29 0 -200                ; starting with no shift, ending with all
                                ; frequencies shifted up by 200 Hz.
 e
```

The second example is a variation of the first, but with the output being fed back into the input. With very small shift amounts (i.e. between 0 and +-6 Hz), the result is a sound that has been described as a "barberpole phaser" or "Shepard tone phase shifter. Several notches appear in the spectrum, and are constantly swept in the direction opposite that of the shift, producing a filtering effect that is reminiscent of Risset's "endless glissando."

```
sr      = 44100
kr      = 44100                ; kr MUST be set to sr for "barberpole" effect
ksmps   = 1
nchnls  = 2

                instr 2

afeedback     init    0         ; initialization of feedback

idur          =       p3
ibegshift     =       p4        ; initial amount of frequency shift -
                                ; can be positive or negative
iendshift     =       p5        ; final amount of frequency shift - can be
                                ; positive or negative
ifeed         =       p6        ; amount of feedback - the higher the
                                ; number, the more pronounced
                                ; the effect. Experiment to see at what
                                ; point oscillation occurs
                                ; (often a factor of 1.4 is the maximum
                                ; feedback before oscillation).

kfreq         linseg  ibegshift, idur, iendshift
```

```
ain           soundin "supertest.wav"
areal, aimag  hilbert ain + afeedback
asin          oscili  1, kfreq, 1
acos          oscili  1, kfreq, 1, .25

amod1         =       areal * acos
amod2         =       aimag * asin

aupshift      =       (amod1 + amod2) * 0.7
adownshift    =       (amod1 - amod2) * 0.7

afeedback     =       (amod1 - amod2) * .5 * ifeed ; feedback taken from
                                                   ; downshift output

              outs    aupshift, aupshift

                 endin

; a simple score
f1 0 16384 10 1           ; sine table for quadrature oscillator
i2 0  29 -.3 -.3 1.4      ; upwards sweep, at a rate of .3 times a second,
                          ; lots of feedback
i2 30 30  .1  .1 1.4      ; downwards sweep, .3 times a second,
                          ; lots of feedback
i2 60 29   5  -5 1.4      ; sweep goes from .3 time a second,
                          ; descending in pitch,
                          ; to .3 times a second ascending in pitch, with a
                          ; large amount of feedback.
e
```

TECHNICAL HISTORY

The use of phase-difference networks in frequency shifters was pioneered by Harald Bode [1]. Bode and Bob Moog provide an excellent description of the implementation and use of a frequency shifter in the analog realm [2]. This would be an excellent first source for those that wish to explore the possibilities of single sideband modulation. Bernie Hutchins provides more applications of the frequency shifter, as well as a detailed technical analysis [3]. A recent paper by Scott Wardle [4] describes a digital implementation of a frequency shifter, as well as some unique applications.

REFERENCES

1. H. Bode, "Solid State Audio Frequency Spectrum Shifter." AES Preprint No. 395 (1965).
2. H. Bode and R.A. Moog, "A High-Accuracy Frequency Shfiter for Professional Audio Applications." *Journal of the Audio Engineering Society*, July/August 1972, vol. 20, no. 6, p. 453.
3. B. Hutchins. *Musical Engineer's Handbook* (Ithaca, NY: Electronotes, 1975), ch. 6a.
4. S. Wardle, "A Hilbert-Transformer Frequency Shifter for Audio." Available online at http://www.iua.upf.es/dafx98/papers/.

AUTHOR

Sean Costello
Seattle, Washington
1999

New in Csound version 3.55

50.11 butterhp, butterlp, butterbp, butterbr

```
ar      butterhp    asig, kfreq [,iskip]
ar      butterlp    asig, kfreq [,iskip]
ar      butterbp    asig, kfreq, kband [,iskip]
ar      butterbr    asig, kfreq, kband [,iskip]
```

DESCRIPTION

Implementations of second-order high-pass, low-pass, band-pass and band-reject Butterworth filters. Note: these opcodes can also be written **butlp, buthp, butbp, butbr.**

PERFORMANCE

These filters are Butterworth second-order IIR filters. They are slightly slower than the original filters in Csound, but they offer an almost flat passband and very good precision and stopband attenuation.

asig – Input signal to be filtered.

kfreq – Cutoff or center frequency for each of the filters. In the case of **butterbp** and **butterbr**, the center *kfreq* is the intervalic, not the mathematical center.

kband – Bandwidth of the bandpass and bandreject filters.

iskip – Skip initialization if present and non zero

EXAMPLE

```
asig    rand      10000            ; White noise signal
alpf    butterlp  asig, 1000       ; cutting frequencies above1K
ahpf    butterhp  asig, 500        ; passing frequencies above 500Hz
abpf    butterbp  asig, 1000, 2000 ; passing 2 octaves: 500 to 2000 Hz
abrf    butterbr  asig, 200, 150   ; cutting 2 octaves: 50 to 200 Hz
```

AUTHOR

Paris Smaragdis
MIT, Cambridge
1995

50.12 filter2, zfilter2

```
ar      filter2     asig, iM,iN,ib0,ib1,..., ibM,ia1,ia2,...,iaN
kr      filter2     ksig, iM,iN,ib0,ib1,...,ibM,ia1,ia2,...,iaN
ar      zfilter2    asig, kdamp,kfreq,iM,iN,ib0,ib1,...,ibM,ia1,ia2,...,iaN
```

DESCRIPTION

General purpose custom filter with time-varying pole control. The filter coefficients implement the following difference equation:

```
(1)*y(n)=b0*x[n]+b1*x[n-1]+...+bM*x[n-M]-a1*y[n-1]-...-aN*y[n-N]
```

the system function for which is represented by:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + \ldots + bM*Z^{-M}}{1 + a1*Z^{-1} + \ldots + aN*Z^{-N}}$$

INITIALIZATION

At initialization the number of zeros and poles of the filter are specified along with the corresponding zero and pole coefficients. The coefficients must be obtained by an external filter-design application such as Matlab and specified directly or loaded into a table via **GEN01**. With **zfilter2,** the roots of the characteristic polynomials are solved at initialization so that the pole-control operations can be implemented efficiently.

PERFORMANCE

The **filter2** opcodes perform filtering using a transposed form-II digital filter lattice with no time-varying control. **zfilter2** uses the additional operations of radial pole-shearing and angular pole-warping in the Z plane.

Pole shearing increases the magnitude of poles along radial lines in the Z-plane. This has the affect of altering filter ring times. The k-rate variable *kdamp* is the damping parameter. Positive values (0.01 to 0.99) increase the ring-time of the filter (hi-Q), negative values (-0.01 to -0.99) decrease the ring-time of the filter, (lo-Q).

Pole warping changes the frequency of poles by moving them along angular paths in the Z plane. This operation leaves the shape of the magnitude response unchanged but alters the frequencies by a constant factor (preserving 0 and p). The k-rate variable *kfreq* determines the frequency warp factor. Positive values (0.01 to 0.99) increase frequencies toward p and negative values (-0.01 to -0.99) decrease frequencies toward 0.

Since **filter2** implements generalized recursive filters, it can be used to specify a large range of general DSP algorithms. For example, a digital waveguide can be implemented for musical instrument modeling using a pair of **delayr** and **delayw** opcodes in conjunction with the **filter2** opcode.

EXAMPLE

A first-order linear-phase lowpass linear-phase FIR filter operating on a k-rate signal:

```
k1   filter2 ksig, 2, 0, 0.5, 0.5   ;; k-rate FIR filter
```

A controllable second-order IIR filter operating on an a-rate signal:

```
; IIR filter
  a1   zfilter2 asig, kdamp, kfreq, 1, 2, 1, ia1, ia2 ; controllable a-rate
```

DEPRECATED NAMES

The k-rate version of **filter2** was originally called **kfilter2**. As of Csound version 3.493, this name is deprecated. **filter2** should be used instead of **kfilter2**. The opcode determines its rate from the output argument.

AUTHOR

Michael A. Casey
MIT
Cambridge, Mass.
1997

50.13 lpf18

```
ar      lpf18       asig, kfco, kres, kdist
```

DESCRIPTION

Implementation of a 3 pole sweepable resonant lowpass filter.

PERFORMANCE

kfco – the filter cut-off frequency in Hz. Should be in the range 0 to sr/2.

kres – the amount of resonance. Self-oscillation occurs when *kres* is approximately 1. Should usually be in the range 0 to 1, however, values slightly greater than 1 are possible for more sustained oscillation and an "overdrive" effect.

kdist – amount of distortion. *kdist* = 0 gives a clean output. *kdist* > 0 adds **tanh**() distortion controlled by the filter parameters, in such a way that both low cutoff and high resonance increase the distortion amount. Some experimentation is encouraged.

lpf18 is a digital emulation of a 3 pole (18 dB/oct.) lowpass filter capable of self-oscillation with a built-in distortion unit. It is really a 3-pole version of **moogvcf**, retuned, recalibrated and with some performance improvements. The tuning and feedback tables use no more than 6 adds and 6 multiplies per control rate. The distortion unit, itself, is based on a modified **tanh**() function driven by the filter controls.

Note: This filter requires that the input signal be normalized to one.

AUTHOR

Josep M Comajuncosas
Spain
December, 2000
New in Csound version 4.10

50.14 tbvcf

```
ar      tbvcf      asig, xfco, xres, kdist, kasym
```

DESCRIPTION

This opcode attempts to model some of the filter characteristics of a Roland TB303 voltage-controlled filter. Euler's method is used to approximate the system, rather than traditional filter methods. Cutoff frequency, Q, and distortion are all coupled. Empirical methods were used to try to unentwine, but frequency is only approximate as a result. Future fixes for some problems with this opcode may break existing orchestras relying on this version of **tbvcf**.

PERFORMANCE

asig – input signal. Should be normalized to ±1.

xfco – filter cutoff frequency. Optimum range is 10,000 to 1500. Values below 1000 may cause problems.

xres – resonance or Q. Typically in the range 0 to 2.

kdist – amount of distortion. Typical value is 2. Changing kdist significantly from 2 may cause odd interaction with xfco and xres.

kasym – asymmetry of resonance. Typically in the range 0 to 1.

EXAMPLE

```
;-------------------------------------------------------
; TBVCF Test
; Coded by Hans Mikelson December, 2000
;-------------------------------------------------------
  sr      =  44100   ; Sample rate
  kr      =  4410   ; Kontrol rate
  ksmps   =  10   ; Samples/Kontrol period
  nchnls  =  2        ; Normal stereo
  zakinit    50, 50

          instr 10

  idur    =          p3                                ; Duration
  iamp    =          p4                                ; Amplitude
  ifqc    =          cpspch(p5)                        ; Pitch to frequency
  ipanl   =          sqrt(p6)                          ; Pan left
  ipanr   =          sqrt(1-p6)                        ; Pan right
  iq      =          p7
  idist   =          p8
  iasym   =          p9

  kdclck  linseg     0, .002, 1, idur-.004, 1, .002, 0 ; Declick envelope

  kfco    expseg     10000, idur, 1000                 ; Frequency envelope

  ax      vco        1, ifqc, 2, 1                     ; Square wave
  ay      tbvcf      ax, kfco, iq, idist, iasym        ; TB-VCF
  ay      buthp      ay/1, 100                         ; Hi-pass

          outs       ay*iamp*ipanl*kdclck, ay*iamp*ipanr*kdclck
          endin
```

```
f1 0 65536 10 1
  ; TeeBee Test
  ;   Sta  Dur  Amp    Pitch Pan  Q    Dist1 Asym
  i10 0    0.2  32767  7.00  .5   0.0  2.0   0.0
  i10 0.3  0.2  32767  7.00  .5   0.8  2.0   0.0
  i10 0.6  0.2  32767  7.00  .5   1.6  2.0   0.0
  i10 0.9  0.2  32767  7.00  .5   2.4  2.0   0.0
  i10 1.2  0.2  32767  7.00  .5   3.2  2.0   0.0
  i10 1.5  0.2  32767  7.00  .5   4.0  2.0   0.0
  i10 1.8  0.2  32767  7.00  .5   0.0  2.0   0.25
  i10 2.1  0.2  32767  7.00  .5   0.8  2.0   0.25
  i10 2.4  0.2  32767  7.00  .5   1.6  2.0   0.25
  i10 2.7  0.2  32767  7.00  .5   2.4  2.0   0.25
  i10 3.0  0.2  32767  7.00  .5   3.2  2.0   0.25
  i10 3.3  0.2  32767  7.00  .5   4.0  2.0   0.25
  i10 3.6  0.2  32767  7.00  .5   0.0  2.0   0.5
  i10 3.9  0.2  32767  7.00  .5   0.8  2.0   0.5
  i10 4.2  0.2  32767  7.00  .5   1.6  2.0   0.5
  i10 4.5  0.2  32767  7.00  .5   2.4  2.0   0.5
  i10 4.8  0.2  32767  7.00  .5   3.2  2.0   0.5
  i10 5.1  0.2  32767  7.00  .5   4.0  2.0   0.5
  i10 5.4  0.2  32767  7.00  .5   0.0  2.0   0.75
  i10 5.7  0.2  32767  7.00  .5   0.8  2.0   0.75
  i10 6.0  0.2  32767  7.00  .5   1.6  2.0   0.75
  i10 6.3  0.2  32767  7.00  .5   2.4  2.0   0.75
  i10 6.6  0.2  32767  7.00  .5   3.2  2.0   0.75
  i10 6.9  0.2  32767  7.00  .5   4.0  2.0   0.75
  i10 7.2  0.2  32767  7.00  .5   0.0  2.0   1.0
  i10 7.5  0.2  32767  7.00  .5   0.8  2.0   1.0
  i10 7.8  0.2  32767  7.00  .5   1.6  2.0   1.0
  i10 8.1  0.2  32767  7.00  .5   2.4  2.0   1.0
  i10 8.4  0.2  32767  7.00  .5   3.2  2.0   1.0
  i10 8.7  0.2  32767  7.00  .5   4.0  2.0   1.0
```

AUTHOR

Hans Mikelson
December, 2000 – January, 2001
New in Csound 4.10

51 SIGNAL MODIFIERS: SPECIALIZED FILTERS

51.1 nlfilt

```
ar      nlfilt      ain, ka, kb, kd, kL, kC
```

DESCRIPTION

Implements the filter

```
Y{n} =a Y{n-1} + b Y{n-2} + d Y^2{n-L} + X{n} – C
```

described in Dobson and Fitch (ICMC'96)

EXAMPLE

Non-linear effect:

```
a = b = 0
d = 0.8, 0.9, 0.7
C = 0.4, 0.5, 0.6
L = 20
```

This affects the lower register most but there are audible effects over the whole range. We suggest that it may be useful for coloring drums, and for adding arbitrary highlights to notes

Low Pass with non-linear:

```
a = 0.4
b = 0.2
d = 0.7
C = 0.11
L = 20, ... 200
```

There are instability problems with this variant but the effect is more pronounced of the lower register, but is otherwise much like the pure comb. Short values of **L** can add attack to a sound.

High Pass with non-linear: The range of parameters are

```
a = 0.35
b = -0.3
d = 0.95
C = 0,2, ... 0.4
L = 200
```

High Pass with non-linear: The range of parameters are

```
a = 0.7
b = -0.2, ... 0.5
d = 0.9
C = 0.12, ... 0.24
L = 500, 10
```

The high pass version is less likely to oscillate. It adds scintillation to medium-high registers. With a large delay **L** it is a little like a reverberation, while with small values there appear to be formant-like regions. There are arbitrary color changes and resonances as the pitch changes. Works well with individual notes.

Warning: The "useful" ranges of parameters are not yet mapped.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
1997

51.2 pareq

```
ar      pareq      asig, kc, iv, iq, imode
```

DESCRIPTION

Implementation of Zoelzer's parametric equalizer filters, with some modifications by the author.

The formula for the low shelf filter is:

```
omega = 2*pi*f/sr
K = tan(omega/2)

b0 = 1 + sqrt(2*V)*K + V*K^2
b1 = 2*(V*K^2 - 1)
b2 = 1 - sqrt(2*V)*K + V*K^2

a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2
```

The formula for the high shelf filter is:

```
omega = 2*pi*f/sr
K = tan((pi-omega)/2)

b0 = 1 + sqrt(2*V)*K + V*K^2
b1 = -2*(V*K^2 - 1)
b1 = 1 - sqrt(2*V)*K + V*K^2

a0 = 1 + K/Q + K^2
a1 = -2*(K^2 - 1)
a2 = 1 - K/Q + K^2
```

The formula for the peaking filter is:

```
omega = 2*pi*f/sr
K = tan(omega/2)

b0 = 1 + V*K/2 + K^2
b1 = 2*(K^2 - 1)
b2 = 1 - V*K/2 + K^2

a0 = 1 + K/Q + K^2
a1 = 2*(K^2 - 1)
a2 = 1 - K/Q + K^2
```

INITIALIZATION

iv – amount of boost or cut. Positive values give boost, negative values give cut.

iq – Q of the filter (sqrt(.5) is no resonance)

imode – operating mode

- 0 = Peaking
- 1 = Low Shelving
- 2 = High Shelving

PERFORMANCE

kc – center frequency in peaking mode, corner frequency in shelving mode.

asig – the incoming signal

EXAMPLE

```
        instr 15
ifc     =     p4            ; Center / Shelf
iq      =     p5            ; Quality factor sqrt(.5) is no resonance
iv      =     ampdb(p6)     ; Volume Boost/Cut
imode   =     p7            ; Mode 0=Peaking EQ, 1=Low Shelf, 2=High Shelf
kfc     linseg  ifc*2, p3, ifc/2
asig    rand  5000                    ; Random number source for testing
aout    pareq asig, kfc, iv, iq, imode ; Parametric equalization
        outs  aout, aout              ; Output the results
endin

; SCORE:
;   Sta  Dur  Fcenter  Q        Boost/Cut(dB)  Mode
i15 0    1    10000   .2          12             1
i15 +    .    5000    .2          12             1
i15 .    .    1000    .707       -12             2
i15 .    .    5000    .1         -12             0
e
```

AUTHOR

Hans Mikelson
December, 1998
(New in Csound version 3.50)

51.3 dcblock

```
ar      dcblock     asig[, ig]
```

DESCRIPTION

Implements the DC blocking filter

Y[i] = X[i] – X[i-1] + (igain * Y[i=1])

Based on work by Perry Cook.

INITIALIZATION

igain – the gain of the filter, which defaults to 0.99

PERFORMANCE

ain – audio signal input

AUTHOR

John ffitch
University of Bath, Codemist Ltd.
Bath, UK
New in Csound version 3.49

This page intentionally left blank.

52 SIGNAL MODIFIERS: ENVELOPE MODIFIERS

52.1 linen, linenr, envlpx, envlpxr

```
kr      linen      kamp, irise, idur, idec
ar      linen      xamp, irise, idur, idec
kr      linenr     kamp, irise, idec, iatdec
ar      linenr     xamp, irise, idec, iatdec
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]]
```

DESCRIPTION

linen – apply a straight line rise and decay pattern to an input amp signal.

envlpx – apply an envelope consisting of 3 segments:

- stored function rise shape
- modified exponential pseudo steady state
- exponential decay.

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

INITIALIZATION

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

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

idec – decay time in seconds. Zero means no decay. An *idec* > *idur* will cause a truncated 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. Note that this attenuation is not by fixed rate (as in a piano), but is sensitive to a note's duration. However, if *iatss* is negative (or if steady state < 4 k-periods) a fixed attenuation rate of *abs*(*iatss*) per second will be used. 0 is illegal.

iatdec – attenuation factor by which the closing steady state value is 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 negative 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**). If **linen** rise and decay periods overlap then both modifications will be in effect for that time; in **envlpx** that will cause a truncated decay. If the overall duration *idur* is exceeded in performance, the final decay will continue on in the same direction, going negative for **linen** but tending asymptotically to zero in **envlpx**.

linenr is unique within Csound in containing a **note-off sensor** and **release time extender**. When it senses either a score event termination or a MIDI noteoff, it will immediately extend the performance time of the current instrument by *idec* seconds, then execute an exponential decay towards the factor *iatdec*. For two or more units in an instrument, extension is by the greatest *idec*.

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.

MULTIPLE "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*.

53 SIGNAL MODIFIERS: AMPLITUDE MODIFIERS

53.1 rms, gain, balance

```
kr      rms         asig[, ihp, iskip]
ar      gain        asig, krms[, ihp, iskip]
ar      balance     asig, acomp[, ihp, iskip]
```

DESCRIPTION

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 Hz) of a special internal low-pass filter. The default value is 10.

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

PERFORMANCE

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

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 (e.g., in passing through a filter bank) can be restored by matching it with, for instance, its own source. It should be noted that **gain** and **balance** provide amplitude modification only – output signals are not altered in any other respect.

EXAMPLE

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

53.2 dam

```
ar      dam         asig, kthreshold, icomp1, icomp2, irtime, iftime
```

DESCRIPTION

This opcode dynamically modifies a g*ain* value applied to the input sound '*ain*' by comparing its power level to a given threshold level. The signal will be compressed/expanded with different factors regarding that it is over or under the threshold.

INITIALIZATION

icomp1 – compression ratio for upper zone.

icomp2 – compression ratio for lower zone.

irtime – gain rise time in seconds. Time over which the gain factor is allowed to raise of one unit.

iftime – gain fall time in seconds. Time over which the gain factor is allowed to decrease of one unit.

PERFORMANCE

asig – input signal

kthreshold – level of input signal which acts as the threshold. Can be changed at k-time (e.g. for ducking)

Note on the compression factors: A compression ratio of one leaves the sound unchanged. Setting the ratio to a value smaller than one will compress the signal (reduce its volume) while setting the ratio to a value greater than one will expand the signal (augment its volume).

AUTHOR

Marc Resibois
Belgium
1997

53.3 clip

```
ar      clip        asig, imeth, ilimit[, iarg]
```

DESCRIPTION

Clips an a-rate signal to a predefined limit, in a "soft" manner, using one of three methods.

INITIALIZATION

imeth – selects the clipping method. The default is 0. The methods are:

- 0 = Bram de Jong method (default)
- 1 = sine clipping
- 2 = tanh clipping

ilimit – limiting value

iarg (optional) – when *imeth* = 0, indicates the point at which clipping starts, in the range 0 – 1. Not used when *imeth* = 1 or *imeth* = 2. Default is 0.5.

PERFORMANCE

asig – a-rate input signal

The Bram de Jong method (*imeth* = 0) applies the algorithm:

```
|x| > a:          f(x) = sin(x) * (a+(x-a)/(1+((x-a)/(1-a))^2
|x| > 1:          f(x) = sin(x) * (a+1)/2
```

This method requires that *asig* be normalized to 1.

The second method (*imeth* = 1) is the sine clip:

```
|x|  < limit:    f(x) = limit * sin(π*x/(2*limit))
                 f(x) = limit * sin(x)
```

The third method (*imeth* = 0) is the tanh clip:

```
|x|  < limit:    f(x) = limit * tanh(x/limit)/tanh(1)
                 f(x) = limit * sin(x)
```

Note: Method 1 appears to be non-functional at release of Csound version 4.07.

EXAMPLE

```
a1          soundin
a2          oscil          25000, 1
asig        clip           a1+a2, 0, 30000, .75
            out            asig
```

AUTHOR

John ffitch
University of Bath, Codemist Ltd.
Bath, UK
August, 2000
New in Csound version 4.07

This page intentionally left blank.

54 SIGNAL MODIFIERS: SIGNAL LIMITERS

54.1 limit, mirror, wrap

```
ir      limit       isig, ilow, ihigh
kr      limit       ksig, klow, khigh
ar      limit       asig, klow, khigh

ir      wrap        isig, ilow, ihigh
kr      wrap        ksig, klow, khigh
ar      wrap        asig, klow, khigh

ir      mirror      isig, ilow, ihigh
kr      mirror      ksig, klow, khigh
ar      mirror      asig, klow, khigh
```

DESCRIPTION

Wraps the signal in various ways.

INITIALIZATION

isig – input signal

ilow – low threshold

ihigh – high threshold

PERFORMANCE

xsig – input signal

xlow – low threshold

xhigh – high threshold

limit sets lower and upper limits on the *xsig* value they process. If *xhigh* is lower than *xlow*, then the output will be the average of the two – it will not be affected by *xsig*. **mirror** "reflects" the signal that exceeds the low and high thresholds. **wrap** wraps-around the signal that exceeds the low and high thresholds.

These opcodes are useful in several situations, such as table indexing or for clipping and modeling i-rate, k-rate or a-rate signals. **wrap** is also useful for wrap-around of table data when the maximum index is not a power of two (see **table** and **tablei**). Another use of **wrap** is in cyclical event repeating, with arbitrary cycle length.

AUTHORS

Gabriel Maldonado (**wrap**, **mirror**)
Italy
New in Csound version 3.49

Robin Whittle (**limit**)
Australia
New in Csound version 3.46

---

55 SIGNAL MODIFIERS: DELAY

55.1 delayr, delayw, delay, delay1

```
ar      delayr      idlt[, iskip]
        delayw      asig
ar      delay       asig, idlt[, iskip]
ar      delay1      asig[, iskip]
```

DESCRIPTION

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

INITIALIZATION

idlt – requested delay time in seconds. This can be as large as available memory will permit. The space required for n seconds of delay is 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.

iskip (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**).

delayr/**delayw** pairs may be interleaved. Beginning another **delayr**/**delayw** pair before terminating a previous pair is no longer excluded. For the interleaved pairs, the first **delayr** unit is associated with the first **delayw** unit, the second **delayr** unit with the second **delayw** unit, and so on. In this way, it is possible to implement cross-coupled feedback that is completed within the same control-rate cycle. See Example 2. (This feature added in Csound version 3.57 by Jens Groh and John ffitch).

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.

EXAMPLES

Example 1:

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

Example 2:

```
ainput1      =              .....
ainput2      =              .....

;Read delayed signal, first delayr instance:
adly1        delayr         0.11

;Read delayed signal, second delayr instance:
adly2        delayr         0.07

;Do some cross-coupled manipulation:
afdbk1       =              0.7 * adly1 + 0.7 * adly2 + ainput1
afdbk2       =              -0.7 * adly1 + 0.7 * adly2 + ainput2

;Feed back signal associated with first delayr instance:
  delayw     afdbk1

;Feed back signal associated with second delayr instance:
  delayw     afdbk2
  outs       adly1, adly2
```

55.2 deltap, deltapi, deltapn, deltap3

```
ar       deltap      kdlt
ar       deltapi     xdlt
ar       deltapn     xnumsamps
ar       deltap3     xdlt
```

DESCRIPTION

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. **deltapn** is identical to **deltapi**, except delay time is specified in number of samples, instead of seconds (Hans Mikelson). **deltap3** is experimental, and uses cubic interpolation. (New in Csound version 3.50.)

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**.

delayr/**delayw** pairs may be interleaved. To associate a delay tap unit with a specific **delayr** unit, it not only has to be located between that **delayr** and the appropriate **delayw** unit, but must also precede any following **delayr** units. See Example 2. (This feature added in Csound version 3.57 by Jens Groh and John ffitch).

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

EXAMPLES

Example 1:

```
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
```

Example 2:

```
ainput1      =              .....
ainput2      =              .....
kdlyt1       =              .....
kdlyt2       =              .....

;Read delayed signal, first delayr instance:
adump        delayr         4.0
adly1        deltap         kdlyt1       ;associated with first delayr instance

;Read delayed signal, second delayr instance:
adump        delayr         4.0
adly2        deltap         kdlyt2               ; associated with second delayr
instance

;Do some cross-coupled manipulation:
afdbk1       =              0.7 * adly1 + 0.7 * adly2 + ainput1
afdbk2       =              -0.7 * adly1 + 0.7 * adly2 + ainput2

;Feed back signal, associated with first delayr instance:
  delayw     afdbk1

;Feed back signal, associated with second delayr instance:
  delayw     afdbk2
  outs       adly1, adly2
```

55.3 multitap

```
ar      multitap    asig, itime1, igain1, itime2, igain2 . . .
```

DESCRIPTION

Multitap delay line implementation.

INITIALIZATION

The arguments *itime* and *igain* set the position and gain of each tap.

The delay line is fed by asig.

EXAMPLE

```
a1      oscil       1000, 100, 1
a2      multitap    a1, 1.2, .5, 1.4, .2
        out         a2
```

This results in two delays, one with length of 1.2 and gain of .5, and one with length of 1.4 and gain of .2.

AUTHOR

Paris Smaragdis
MIT, Cambridge
1996

55.4 vdelay, vdelay3

```
ar      vdelay      asig, adel, imaxdel  [, iskip]
ar      vdelay3     asig, adel, imaxdel  [, iskip]
```

DESCRIPTION

This is an interpolating variable time delay, it is not very different from the existing implementation (**deltapi**), it is only easier to use. **vdelay3** is experimental, and is the same as vdelay, except that it uses cubic interpolation. (New in Version 3.50.)

INITIALIZATION

imaxdel – Maximum value of delay in milliseconds. If *adel* gains a value greater than *imaxdel* it is folded around *imaxdel*. This should not happen.

iskip – Skip initialization if present and non-zero

PERFORMANCE

With this unit generator it is possible to do Doppler effects or chorusing and flanging.

asig – Input signal.

adel – Current value of delay in milliseconds. Note that linear functions have no pitch change effects. Fast changing values of *adel* will cause discontinuities in the waveform resulting noise.

EXAMPLE

```
f1 0 8192 10 1
ims  =       100             ; Maximum delay time in msec
a1   oscil   10000, 1737, 1  ; Make a signal
a2   oscil   ims/2, 1/p3, 1  ; Make an LFO
a2   =       a2 + ims/2      ; Offset the LFO so that it is positive
a3   vdelay  a1, a2, ims     ; Use the LFO to control delay time
     out     a3
```

Two important points here. First, the delay time must be always positive. And second, even though the delay time can be controlled in k-rate, it is not advised to do so, since sudden time changes will create clicks.

AUTHOR

Paris Smaragdis
MIT, Cambridge
1995

56 SIGNAL MODIFIERS: REVERBERATION

56.1 comb, alpass, reverb

```
ar      comb        asig, krvt, ilpt[, iskip][, insmps]
ar      alpass      asig, krvt, ilpt[, iskip][, insmps]
ar      reverb      asig, krvt[, iskip]
```

DESCRIPTION

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

INITIALIZATION

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

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

insmps (optional) – if non-zero, loop time is in samples instead of seconds. Default is zero. New in Csound version 4.10.

PERFORMANCE

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

A standard **reverb** unit is composed of four **comb** filters in parallel followed by two **alpass** units in series. Loop times are set for optimal "natural room response." Core storage requirements for this unit are proportional only to the sampling rate, each unit requiring approximately 3K words for every 10KC. The **comb, alpass, delay, tone** and other Csound units provide the means for experimenting with alternate reverberator designs

Since output from the standard **reverb** will begin to appear only after 1/20 second or so of delay, and often with less than three-fourths of the original power, it is normal to output both the source and the reverberated signal. If *krvt* is inadvertently set to a non-positive number, *krvt* will be reset automatically to 0.01. (New in Csound version 4.07.) Also, since the reverberated sound will persist long after the cessation of source events, it is normal to put **reverb** in a separate instrument to which sound is passed via a *global variable*, and to leave that instrument running throughout the performance.

EXAMPLE

```
ga1       init          0          ; init an audio receiver/mixer
          instr         1          ; instr (there may be many copies)
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
```

56.2 reverb2, nreverb

```
ar      reverb2     asig, ktime, khdif [,iskip]
ar      nreverb     asig, ktime, khdif [,iskip] [,inumCombs, ifnCombs]\\
                    [, inumAlpas, ifnAlpas]
```

DESCRIPTION

This is a reverberator consisting of 6 parallel comb-lowpass filters being fed into a series of 5 allpass filters. **nreverb** replaces **reverb2** (version 3.48) and so both opcodes are identical.

INITIALISATION

iskip – Skip initialization if present and non zero

inumCombs – number of filter constants in comb filter. If omitted, the values default to the **nreverb** constants. New in Csound version 4.09.

ifnCombs – function table with *inumCombs* comb filter time values, followed the same number of gain values. The ftable should not be rescaled (use negative fgen number). Positive time values are in seconds. The time values are converted internally into number of samples, then set to the next greater prime number. If teh time is negative, it is interpreted directly as time in sample frames, and no processing is done (except negation). New in Csound version 4.09.

inumAlpas, *ifnAlpas* – same as *inumCombs* and *ifnCombs*, for allpass filter. New in Csound version 4.09.

PERFORMANCE

The output of **nreverb** (and **reverb2**) is zeroed on the first perfomance pass. The input signal *asig* is reverberated for *ktime* seconds. The parameter *khdif* controls the high frequency diffusion amount. The values of *khdif* should be from 0 to 1. If *khdif* is set to 0 the all the frequencies decay with the same speed. If *khdif* is 1, high frequencies decay faster than lower ones. If *ktime* is inadvertently set to a non-positive number, *ktime* will be reset automatically to 0.01. (New in Csound version 4.07.)

As of Csound version 4.09, **nreverb** may read any number of comb and allpass filter from an ftable.

EXAMPLES

This results in a 2.5 sec reverb with faster high frequency attenuation:

```
a1   oscil   10000, 100, 1
a2   reverb2 a1, 2.5, .3
     out     a1 + a2 * .2
```

This illustrates the use of an ftable for filter constants:

```
;Orchestra:
            instr 1
 a1         soundin         "neopren.wav"
 a2         nreverb         a1, 1.5, .75, 0, 8, 71, 4, 72
            out             a1 + a2 * .4
            endin

;Score:
  ; freeverb time constants, as direct (negative) sample, with arbitrary gains
  f71 0 16   -2  -1116 -1188 -1277 -1356 -1422 -1491 -1557 -1617  0.8  0.79  0.78 \\
             0.77  0.76  0.75  0.74  0.73
  f72 0 16   -2  -556 -441 -341 -225  0.7  0.72  0.74  0.76

  i1   0     7
  e
```

AUTHORS

Paris Smaragdis (**reverb2**)
MIT, Cambridge
1995

Richard Karpen (**nreverb**)
Seattle, Wash
1998

56.3 nestedap

```
ar      nestedap    asig, imode, imaxdel, idel1, igain1[, idel2, igain2\\
                    [, idel3, igain3]]
```

DESCRIPTION

Three different nested all-pass filters, useful for implementing reverbs.

INITIALIZATION

imode – operating mode of the filter:

1 = simple all-pass filter
2 = single nested all-pass filter
3 = double nested all-pass filter

idel1, *idel2*, *idel3* – delay times of the filter stages. Delay times are in seconds and must be greater than zero. *idel1* must be greater than the sum of *idel2* and *idel3*.

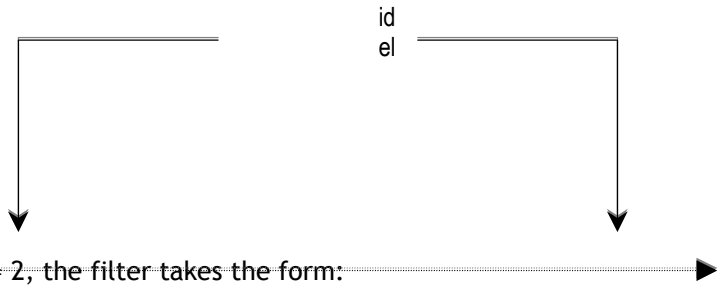
igain1, *igain2*, *igain3* – gain of the filter stages.

imaxdel – will be necessary if k-rate delays are implemented. Not currently used.

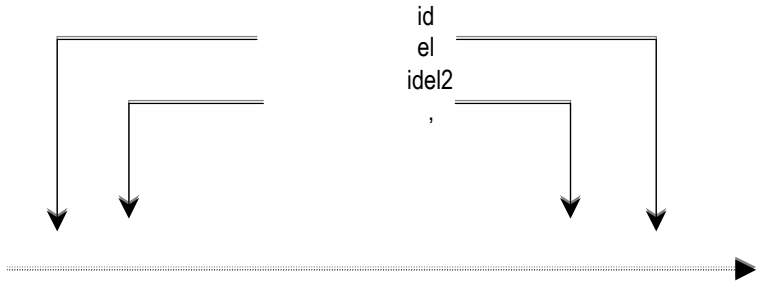
PERFORMANCE

asig – input signal

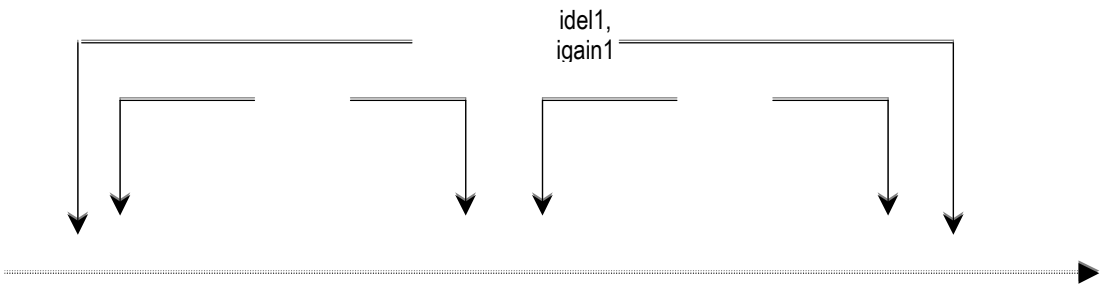
If *imode* = 1, the filter takes the form:



If *imode* = 2, the filter takes the form:



If *imode* = 3, the filter takes the form:



EXAMPLE

```
            instr 5
  insnd   =  p4
  gasig   diskin      insnd, 1
            endin

            instr 10
  imax      =           1
  idel1     =           p4
  igain1    =           p5
  idel2     =           p6
  igain2    =           p7
  idel3     =           p8
  igain3    =           p9
  idel4     =           p10
  igain4    =           p11
  idel5     =           p12
  igain5    =           p13
  idel6     =           p14
  igain6    =           p15
  afdbk     init        0

  aout1     nestedap    gasig+afdbk*.4, 3, imax, idel1, igain1, idel2,\\   igain2,
idel3,    igain3

  aout2     nestedap    aout1, 2, imax, idel4, igain4, idel5, igain5

  aout      nestedap    aout2, 1, imax, idel6, igain6

  afdbk     butterlp    aout, 1000

            outs        gasig+(aout+aout1)/2, gasig-(aout+aout1)/2

  gasig     =           0

            endin

;Score

f1 0 8192 10 1

; Diskin
;   Sta  Dur  Soundin
i5  0    3    1

; Reverb
;   St  Dur  Del1 Gn1  Del2  Gn2  Del3  Gn3  Del4  Gn4  Del5  Gn5  Del6  Gn6
i10 0   4    97   .11  23   .07   43   .09   72    .2   53    .2   119   .3
e
```

AUTHOR

Hans Mikelson
February 1999
New in Csound version 3.53

56.4 babo

```
a1, a2   babo        asig, ksrcx, ksrcy, ksrcz, irx, iry, irz[, idiff\\
                     [, ifno]]
```

DESCRIPTION

babo stands for **ba**ll-within-the-**bo**x. It is a physical model reverberator based on a paper by Davide Rocchesso "The Ball within the Box: a sound-processing metaphor", Computer Music Journal, Vol 19, N.4, pp.45-47, Winter 1995.

The resonator geometry can be defined, along with some response characteristics, the position of the listener within the resonator, and the position of the sound source.

INITIALIZATION

irx, iry, irz – the coordinates of the geometry of the resonator (length of the edges in meters)

idiff – is the coefficient of diffusion at the walls, which regulates the amount of diffusion (0-1, where 0 = no diffusion, 1 = maximum diffusion - default: 1)

ifno – expert values function: a function number that holds all the additional parameters of the resonator. This is typically a GEN2--type function used in non-rescaling mode. They are as follows:

- *decay* – main decay of the resonator (default: 0.99)
- *hydecay* – high frequency decay of the resonator (default: 0.1)
- *rcvx, rcvy, rcvz* – the coordinates of the position of the receiver (the listener) (in meters; 0,0,0 is the resonator center)
- *rdistance* – the distance in meters between the two pickups (your ears, for example - default: 0.3)
- *direct* – the attenuation of the direct signal (0-1, default: 0.5)
- *early_diff* – the attenuation coefficient of the early reflections (0-1, default: 0.8)

PERFORMANCE

asig – the input signal

ksrcx, ksrcy, ksrcz – the virtual coordinates of the source of sound (the input signal). These are allowed to move at k-rate and provide all the necessary variations in terms of response of the resonator.

EXAMPLES

Orchestra File - Simple usage

```
                ; minimal babo instrument
                ;
                instr 1
                ix     =p5          ; x position of source
                iy     =p6          ; y position of source
                iz     =p7          ; z position of source
                ixsize =p8          ; width  of the resonator
                iysize =p9          ; depth  of the resonator
                izsize =p10         ; height of the resonator

        ainput  soundin p4

        al,ar   babo    ainput*0.9, ix, iy, iz, ixsize, iysize, izsize

                outs    al,ar
```

endin

Score File - Simple Usage

```
       ; simple babo usage:
       ;
       ;p4     : soundin number
       ;p5     : x position of source
       ;p6     : y position of source
       ;p7     : z position of source
       ;p1     : width  of the resonator
       ;p12    : depth  of the resonator
       ;p13    : height of the resonator
       ;
       i1  0 10 1  6  4  3     14.39 11.86 10
       ;           ^^^^^^^     ^^^^^^^^^^^^^
       ;           |||||||     +++++++++++++: optimal room dims according to
       ;           |||||||                    Milner and Bernard JASA 85(2), 1989
       ;           +++++++++: source position
       e
```

Orchestra File - Expert usage

```
                  ; full blown babo instrument with movement
                  ;
                  instr 2
                  ixstart=p5          ; start x position of source (left-right)
                  ixend  =p8          ; end   x position of source
                  iystart=p6          ; start y position of source (front-back)
                  iyend  =p9          ; end   y position of source
                  izstart=p7          ; start z position of source (up-down)
                  izend  =p10         ; end   z position of source
                  ixsize =p11         ; width  of the resonator
                  iysize =p12         ; depth  of the resonator
                  izsize =p13         ; height of the resonator
                  idiff  =p14         ; diffusion coefficient
                  iexpert=p15         ; power user values stored in this function

       ainput     soundin p4
       ksource_x  line    ixstart, p3, ixend
       ksource_y  line    iystart, p3, iyend
       ksource_z  line    izstart, p3, izend

       al,ar      babo    ainput*0.9, ksource_x, ksource_y, ksource_z,
ixsize, iysize, izsize, idiff, iexpert

                  outs    al,ar

                  endin
```

Score File - Expert Usage

```
       ; full blown instrument
       ;p5         : start x position of source (left-right)
       ;p6         : end   x position of source
       ;p7         : start y position of source (front-back)
       ;p8         : end   y position of source
       ;p9         : start z position of source (up-down)
       ;p10        : end   z position of source
       ;p11        : width  of the resonator
       ;p12        : depth  of the resonator
       ;p13        : height of the resonator
       ;p14        : diffusion coefficient
       ;p15        : power user values stored in this function

       ;         decay  hidecay rx ry rz rdistance direct early_diff
       f1  0 8 -2  0.95   0.95   0  0  0    0.3     0.5      0.8  ; brighter
       f2  0 8 -2  0.95   0.5    0  0  0    0.3     0.5      0.8  ; default (to be set
as)
       f3  0 8 -2  0.95   0.01   0  0  0    0.3     0.5      0.8  ; darker
       f4  0 8 -2  0.95   0.7    0  0  0    0.3     0.1      0.4  ; to hear the effect
of diffusion
       f5  0 8 -2  0.9    0.5    0  0  0    0.3     2.0      0.98 ; to hear the
movement
       f6  0 8 -2  0.99   0.1    0  0  0    0.3     0.5      0.8  ; default vals
       ;        ^
       ;         ----- gen. number: negative to avoid rescaling
```

```
i2   10 10 1  6  4  3   6   4  3  14.39  11.86  10   1   6 ; defaults
i2   +   4 2  6  4  3   6   4  3  14.39  11.86  10   1   1 ; hear brightness 1
i2   +   4 2  6  4  3  -6  -4  3  14.39  11.86  10   1   2 ; hear brightness 2
i2   +   4 2  6  4  3  -6  -4  3  14.39  11.86  10   1   3 ; hear brightness 3
i2   +   3 2  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 0.0 4 ; hear diffusion 1
i2   +   3 2  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 1.0 4 ; hear diffusion 2
i2   +   4 2 12  4  3 -12  -4 -3  24.39  21.86  20   1   5 ; hear movement
;
i2   +   4 1  6  4  3   6   4  3  14.39  11.86   10  1   1 ; hear brightness 1
i2   +   4 1  6  4  3  -6  -4  3  14.39  11.86   10  1   2 ; hear brightness 2
i2   +   4 1  6  4  3  -6  -4  3  14.39  11.86   10  1   3 ; hear brightness 3
i2   +   3 1  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 0.0 4 ; hear diffusion 1
i2   +   3 1  .6 .4 .3 -.6 -.4 .3  1.439  1.186  1.0 1.0 4 ; hear diffusion 2
i2   +   4 1 12  4  3 -12  -4 -3  24.39  21.86   20  1   5 ; hear movement
;            ^^^^^^^^^^^^^^^^^^^  ^^^^^^^^^^^^^^^^^^  ^   ^
;            ||||||||||||||||||||  ||||||||||||||||||  |-: expert values function
;            ||||||||||||||||||||  ||||||||||||||||||  +--: diffusion
;            ||||||||||||||||||||  optimal room dims according
                                   to Milner and Bernard JASA 85(2), 1989
;            ||||||||||||||||||||
;            --------------------: source position start and end
e
```

AUTHORS

Paolo Filippi
Padova, Italy
1999

Nicola Bernardini
Rome, Italy
2000

New in Csound version 4.09

This page intentionally left blank.

57 SIGNAL MODIFIERS: WAVEGUIDES

57.1 wguide1, wguide2

```
ar      wguide1     asig, xfreq, xcutoff, kfeedback;
ar      wguide2     asig, xfreq1, xfreq2, kcutoff1, kcutoff2, kfeedback1,\
                    kfeedback2
```

DESCRIPTION

Simple waveguide blocks

PERFORMANCE

asig – the input of excitation noise

xfreq – the frequency (i.e. the inverse of delay time) Changed to x-rate in Csound version 3.59.

kcutoff1, kcutoff2 – the filter cutoff frequency in Hz

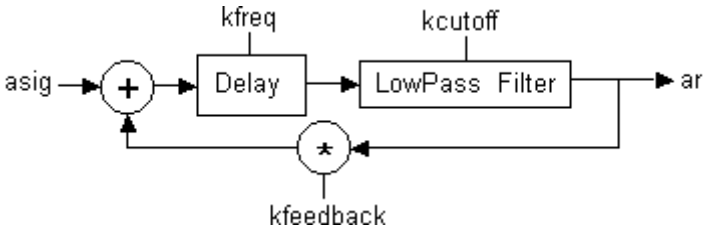
xcutoff – the filter cutoff frequency in Hz. Changed to x-rate for **wguide1** in Csound version 3.59.

kfeedback – the feedback factor

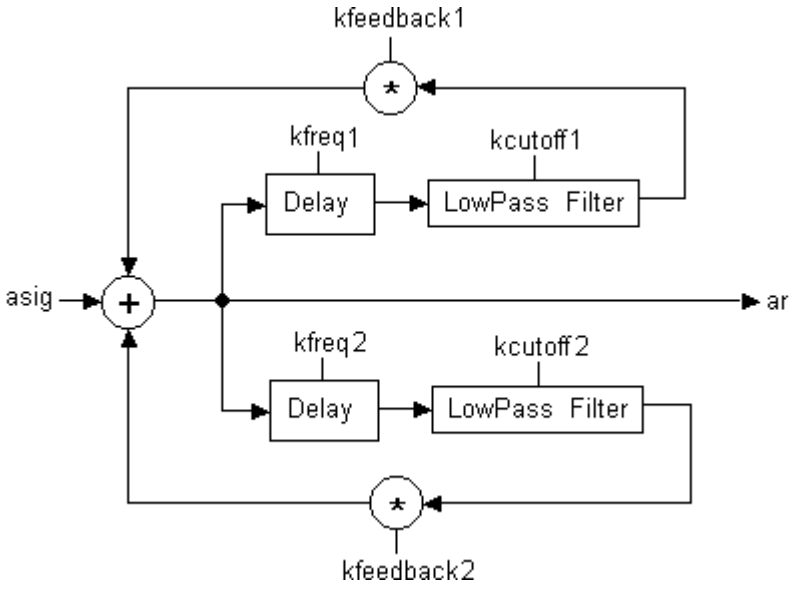
wguide1 is the most elemental waveguide model, consisting of one delayline and one first-order lowpass filter.

wguide2 is a model of beaten plate consisting of two parallel delaylines and two first-order lowpass filters. The two feedback lines are mixed and sent to the delay again each cycle.

Implementing waveguide algorithms as opcodes, instead of as orc instr, allows the user to set **kr** different than **sr**, allowing better performance particularly when using real-time.



wguide1



wguide2

AUTHOR

Gabriel Maldonado
Italy
October, 1998 (New in Csound version 3.49)

57.2 streson

```
ar      streson     asig, kfr, ifdbgain
```

An audio signal is modified by a string resonator with variable fundamental frequency.

INITIALIZATION

ifdbgain – feedback gain, between 0 and 1, of the internal delay line. A value close to 1 creates a slower decay and a more pronounced resonance. Small values may leave the input signal unaffected. Depending on the filter frequency, typical values are > .9.

PERFORMANCE

streson passes the input *asig* through a network composed of comb, low-pass and all-pass filters, similar to the one used in some versions of the Karplus-Strong algorithm, creating a string resonator effect. The fundamental frequency of the "string" is controlled by the k-rate variable *kfr*. This opcode can be used to simulate sympathetic resonances to an input signal.

streson is an adaptation of the StringFlt object of the SndObj Sound Object Library developed by the author.

AUTHOR

Victor Lazzarini
Music Department
National University of Ireland, Maynooth
Maynooth, Co. Kildare
1998 (New in Csound version 3.494)

This page intentionally left blank.

58 SIGNAL MODIFIERS: SPECIAL EFFECTS

58.1 harmon

```
ar      harmon      asig, kestfrq, kmaxvar, kgenfreq1, kgenfreq2, imode,\\
                    iminfrq, iprd
```

DESCRIPTION

Analyze an audio input and generate harmonizing voices in synchrony.

INITIALIZATION

imode – interpreting mode for the generating frequency inputs *kgenfreq1*, *kgenfreq2*. 0: input values are ratios with respect to the audio signal analyzed frequencies. 1: input values are the actual requested frequencies in Hz.

iminfrq – the lowest expected frequency (in Hz) 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 analyzed 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 Hz), 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
```

AUTHOR

Barry Vercoe
MIT, Cambridge, Mass
1997

58.2 flanger

```
ar      flanger         asig, adel, kfeedback[, imaxd ]
```

DESCRIPTION

A user controlled flanger.

INITIALIZATION

imaxd(optional) – maximum delay in seconds (needed for initial memory allocation)

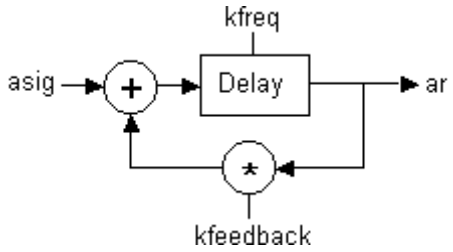
PERFORMANCE

asig – input signal

adel – delay in seconds

kfeedback – feedback amount (in normal tasks this should not exceed 1, even if bigger values are allowed)

This unit is useful for generating choruses and flangers. The delay must be varied at a-rate connecting *adel* to an oscillator output. Also the feedback can vary at k-rate. This opcode is implemented to allow **kr** different than **sr** (else delay could not be lower than **ksmps**) enhancing real-time performance. This unit is very similar to **wguide1**, the only difference is **flanger** does not have the lowpass filter.



flanger

AUTHOR

Gabriel Maldonado
Italy
New in Csound version 3.49

58.3 distort1

```
ar      distort1    asig[, kpregain[, kpostgain[, kshape1[, kshape2]]]]
```

DESCRIPTION

Implementation of modified hyperbolic tangent distortion. **distort1** can be used to generate wave shaping distortion based on a modification of the **tanh** function.

$$aout = \frac{exp(asig * (pregain + shape1)) - exp(asig*(pregain+shape2))}{exp(asig*pregain) + exp(-asig*pregain)}$$

PERFORMANCE

asig – is the input signal.

kpregain – determines the amount of gain applied to the signal before waveshaping. A value of 1 gives slight distortion.

kpostgain – determines the amount of gain applied to the signal after waveshaping.

kshape1 – determines the shape of the positive part of the curve. A value of 0 gives a flat clip, small positive values give sloped shaping.

kshape2 – determines the shape of the negative part of the curve.

All arguments except *asig*, were made optional in Csound version 3.52.

EXAMPLE

```
gadist  init    0

        instr 1
iamp    =       p4
ifqc    =       cpspch(p5)
asig    pluck   iamp, ifqc, ifqc, 0, 1
gadist  =       gadist + asig
        endin

        instr 50
kpre    init    p4
kpost   init    p5
kshap1  init    p6
kshap2  init    p7
aout    distort1 gadist, kpre, kpost, kshap1, kshap2
        outs    aout, aout
gadist  =       0
        endin

;   Sta  Dur  Amp    Pitch
i1  0.0  3.0  10000  6.00
i1  0.5  2.5  10000  7.00
i1  1.0  2.0  10000  7.07
i1  1.5  1.5  10000  8.00

;   Sta  Dur  PreGain PostGain Shape1 Shape2
i50 0    3    2       1        0      0
e
```

AUTHOR

Hans Mikelson
December 1998 (New in Csound version 3.50)

58.4 phaser1, phaser2

```
ar      phaser1     asig, kfreq, iord, kfeedback[, iskip]
ar      phaser2     asig, kfreq, kq, iord, imode, ksep, kfeedback
```

DESCRIPTION

An implementation of *iord* number of first-order (**phaser1**) or second-order (**phaser2**) allpass filters in series.

INITIALIZATION

iord – the number of allpass stages in series. For **phaser1**, these are first-order filters, and *iord* can range from 1 to 4999. For **phaser2**, these are second-order filters, and *iord* can range from 1 to 2499. With higher orders, the computation time increases.

iskip – used to control 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.

imode – used in calculation of notch frequencies.

PERFORMANCE

kfreq – frequency (in Hz) of the filter(s). For **phaser1**, this is the frequency at which each filter in the series shifts its input by 90 degrees. For **phaser2**, this is the center frequency of the notch of the first allpass filter in the series. This frequency is used as the base frequency from which the frequencies of the other notches are derived.

kq – Q of each notch. Higher Q values result in narrow notches. A Q between 0.5 and 1 results in the strongest "phasing" effect, but higher Q values can be used for special effects.

kfeedback – amount of the output which is fed back into the input of the allpass chain. With larger amounts of feedback, more prominent notches appear in the spectrum of the output. *kfeedback* must be between -1 and +1. for stability.

ksep – scaling factor used, in conjunction with *imode*, to determine the frequencies of the additional notches in the output spectrum.

phaser1 implements *iord* number of first-order allpass sections, serially connected, all sharing the same coefficient. Each allpass section can be represented by the following difference equation:

```
y(n) = C * x(n) + x(n-1) - C * y(n-1)
```

where `x(n)` is the input, `x(n-1)` is the previous input, `y(n)` is the output, `y(n-1)` is the previous output, and C is a coefficient which is calculated from the value of *kfreq*, using the bilinear z-transform.

By slowly varying *kfreq*, and mixing the output of the allpass chain with the input, the classic "phase shifter" effect is created, with notches moving up and down in frequency. This works best with *iord* between 4 and 16. When the input to the allpass chain is mixed with the output, 1 notch is generated for every 2 allpass stages, so that with *iord* = 6, there will be 3 notches in the output. With higher values for *iord*, modulating *kfreq* will result in a form of nonlinear pitch modulation.

phaser2 implements *iord* number of second-order allpass sections, connected in series. The use of second-order allpass sections allows for the precise placement of the frequency, width, and depth of notches in the frequency spectrum. *iord* is used to directly determine the number of notches in the spectrum; e.g. for *iord* = 6, there will be 6 notches in the output spectrum.

There are two possible modes for determining the notch frequencies. When *imode* = 1, the notch frequencies are determined the following function:

```
frequency of notch N = kbf + (ksep * kbf * N-1)
```

For example, with *imode* = 1 and *ksep* = 1, the notches will be in harmonic relationship with the notch frequency determined by *kfreq* (i.e. if there are 8 notches, with the first at 100 Hz, the next notches will be at 200, 300, 400, 500, 600, 700, and 800 Hz). This is useful for generating a "comb filtering" effect, with the number of notches determined by *iord*. Different values of *ksep* allow for inharmonic notch frequencies and other special effects. *ksep* can be swept to create an expansion or contraction of the notch frequencies. A useful visual analogy for the effect of sweeping *ksep* would be the bellows of an accordion as it is being played – the notches will be seperated, then compressed together, as *ksep* changes.

When *imode* = 2, the subsequent notches are powers of the input parameter *ksep* times the initial notch frequency specified by *kfreq*. This can be used to set the notch frequencies to octaves and other musical intervals. For example, the following lines will generate 8 notches in the output spectrum, with the notches spaced at octaves of *kfreq*:

```
aphs       phaser2        ain, kfreq, 0.5, 8, 2, 2, 0
aout       =              ain + aphs
```

When *imode* = 2, the value of *ksep* must be greater than 0. *ksep* can be swept to create a compression and expansion of notch frequencies (with more dramatic effects than when *imode* = 1).

EXAMPLES

```
; Orchestra for demonstration of phaser1 and phaser2
  sr     = 44100
  kr     = 4410
  ksmps  = 10
  nchnls = 1

             instr 1                              ; demonstration of phase shifting
                                                  ; abilities of phaser1.
                                          ; Input mixed with output of
                                          ; phaser1 to generate notches.
                                          ; Shows the effects of different
                                          ; iorder values on the sound
  idur       =              p3
  iamp       =              p4 * .05
  iorder     =              p5            ; number of 1st-order stages in
                                          ; phaser1 network.
                                          ; Divide iorder by 2 to get the
                                          ; number of notches.
  ifreq      =              p6            ; frequency of modulation of
                                          ; phaser1
  ifeed      =              p7            ; amount of feedback for phaser1

  kamp       linseg         0, .2, iamp, idur - .2, iamp, .2, 0

  iharms     =              (sr*.4) / 100

  asig       gbuzz          1, 100, iharms, 1, .95, 2 ; "Sawtooth" waveform

; modulation oscillator for phaser1 ugen.
  kfreq      oscili         5500, ifreq, 1
  kmod       =              kfreq + 5600
```

```
aphs        phaser1         asig, kmod, iorder, ifeed
            out             (asig + aphs) * iamp

            endin

            instr 2                     ; demonstration of phase shifting
                                        ; abilities of phaser2. Input
                                        ; mixed with output of phaser2 to
                                        ; generate notches. Demonstrates
                                        ; the interaction of imode and
                                        ; ksep.
idur        =               p3
iamp        =               p4 * .04
iorder      =               p5          ; number of 2nd-order stages in
                                        ; phaser2 network

ifreq       =               p6          ; not used
ifeed       =               p7          ; amount of feedback for phaser2
imode       =               p8          ; mode for frequency scaling
isep        =               p9          ; used with imode to determine
                                        ; notch frequencies

kamp        linseg          0, .2, iamp, idur - .2, iamp, .2, 0
iharms      =               (sr*.4) / 100

asig        gbuzz           1, 100, iharms, 1, .95, 2 ; "Sawtooth" waveform

; exponentially decaying function, to control notch frequencies
kline       expseg          1, idur, .005
aphs        phaser2         asig, kline * 2000, .5, iorder, imode, isep, ifeed
            out             (asig + aphs) * iamp
            endin

; score file for above
f1 0 16384 9 .5 -1 0 ; inverted half-sine, used
; for modulating phaser1 frequency
f2 0 8192 9 1 1 .25 ; cosine wave for gbuzz


; phaser1
i1 0 5 7000 4 .2 .9
i1 6 5 7000 6 .2 .9
i1 12 5 7000 8 .2 .9
i1 18 5 7000 16 .2 .9
i1 24 5 7000 32 .2 .9
i1 30 5 7000 64 .2 .9

; phaser2, imode=1
i2 37 10 7000 8 .2 .9 1 .33
i2 48 10 7000 8 .2 .9 1 2

; phaser2, imode=2
i2 60 10 7000 8 .2 .9 2 .33
i2 72 10 7000 8 .2 .9 2 2
e
```

TECHNICAL HISTORY

A general description of the differences between flanging and phasing can be found in Hartmann [1]. An early implementation of first-order allpass filters connected in series can be found in Beigel [2], where the bilinear z-transform is used for determining the phase shift frequency of each stage. Cronin [3] presents a similar implementation for a four-stage phase shifting network. Chamberlin [4] and Smith [5] both discuss using second-order allpass sections for greater control over notch depth, width, and frequency.

REFERENCES

1. Hartmann, W.M. "Flanging and Phasers." *Journal of the Audio Engineering Society*, Vol. 26, No. 6, pp. 439-443, June 1978.

2. Beigel, Michael I. "A Digital 'Phase Shifter' for Musical Applications, Using the Bell Labs (Alles-Fischer) Digital Filter Module." *Journal of the Audio Engineering Society*, Vol. 27, No. 9, pp. 673-676,September 1979.

3. Cronin, Dennis. "Examining Audio DSP Algorithms." *Dr. Dobb's Journal*, July 1994, p. 78-83.

4. Chamberlin, Hal. *Musical Applications of Microprocessors*. Second edition. Indianapolis, Indiana: Hayden Books, 1985.

5. Smith, Julius O. "An Allpass Approach to Digital Phasing and Flanging." *Proceedings of the 1984 ICMC*, p. 103-108.

AUTHOR

Sean Costello
Seattle, Washington

1999
New in Csound version 4.0

This page intentionally left blank.

59 SIGNAL MODIFIERS: CONVOLUTION AND MORPHING

59.1 convolve

```
ar1[,ar2]           convolve      ain, ifilcod, ichannel
[,ar3][,ar4]
```

DESCRIPTION

Output is the convolution of signal *ain* and the impulse response contained in *ifilcod*. If more than one output signal is supplied, each will be convolved with the same impulse response. Note that it is considerably more efficient to use one instance of the operator when processing a mono input to create stereo, or quad, outputs. Note: this opcode can also be written **convle**.

INITIALIZATION

ifilcod – integer or character-string denoting an impulse response data file. An integer denotes the suffix of a file *convolve.m*; a character string (in double quotes) gives a filename, optionally a full pathname. If not a fullpath, the file is sought first in the current directory, then in the one given by the environment variable SADIR (if defined). The data file contains the Fourier transform of an impulse response. Memory usage depends on the size of the data file, which is read and held entirely in memory during computation, but which is shared by multiple calls.

PERFORMANCE

convolve implements Fast Convolution. The output of this operator is delayed with respect to the input. The following formulas should be used to calculate the delay:

```
For (1/kr) <= IRdur:
        Delay = ceil(IRdur * kr) / kr
For (1/kr)  IRdur:
        Delay = IRdur * ceil(1/(kr*IRdur))
Where:
        kr  = Csound control rate
        IRdur = duration, in seconds, of impulse response
        ceil(n) = smallest integer not smaller than n
```

One should be careful to also take into account the initial delay, if any, of the impulse response. For example, if an impulse response is created from a recording, the soundfile may not have the initial delay included. Thus, one should either ensure that the soundfile has the correct amount of zero padding at the start, or, preferably, compensate for this delay in the orchestra. (the latter method is more efficient). To compensate for the delay in the orchestra, subtract the initial delay from the result calculated using the above formula(s), when calculating the required delay to introduce into the 'dry' audio path.

For typical applications, such as reverb, the delay will be in the order of 0.5 to 1.5 seconds, or even longer. This renders the current implementation unsuitable for real time applications. It could conceivably be used for real time filtering however, if the number of taps is small enough.

The author intends to create a higher-level operator at some stage, that would mix the wet & dry signals, using the correct amount of delay automatically.

EXAMPLE

Create frequency domain impulse response file:

```
c:\ Csound -Ucvanal l1_44.wav l1_44.cv
```

Determine duration of impulse response. For high accuracy, determine the number of sample frames in the impulse response soundfile, and then compute the duration with:

```
duration = (sample frames)/(sample rate of soundfile)
```

This is due to the fact that the SNDINFO utility only reports the duration to the nearest 10ms. If you have a utility that reports the duration to the required accuracy, then you can simply use the reported value directly.

```
c:\ sndinfo l1_44.wav
length = 60822 samples, sample rate = 44100

Duration = 60822/44100 = 1.379s.
```

Determine initial delay, if any, of impulse response. If the impulse response has not had the initial delay removed, then you can skip this step. If it has been removed, then the only way you will know the initial delay is if the information has been provided separately. For this example, let's assume that the initial delay is 60ms. (0.06s)

Determine the required delay to apply to the dry signal, to align it with the convolved signal:

```
If kr = 441:
      1/kr = 0.0023, which is <= IRdur (1.379s), so:
      Delay1  = ceil(IRdur * kr) / kr
              = ceil(608.14) / 441
              = 609/441
              = 1.38s

Accounting for the initial delay:
      Delay2  = 0.06s
Total delay   = delay1 - delay2
              = 1.38 - 0.06
              = 1.32s
```

Create .orc file, e.g.:

```
; Simple demonstration of CONVOLVE operator, to apply reverb.
       sr = 44100
       kr = 441
       ksmps = 100
       nchnls = 2
       instr   1
imix = 0.22 ; Wet/dry mix. Vary as desired.
            ; NB: 'Small' reverbs often require a much higher
            ; percentage of wet signal to sound interesting. 'Large'
            ; reverbs seem require less. Experiment! The wet/dry mix is
            ; very important - a small change can make a large difference.
ivol = 0.9  ; Overall volume level of reverb. May need to adjust
            ; when wet/dry mix is changed, to avoid clipping.
idel = 1.32 ; Required delay to align dry audio with output of convolve.
            ; This can be automatically calculated within the orc file,
            ; if desired.
```

```
adry            soundin "anechoic.wav"      ; input (dry) audio
awet1,awet2     convolve adry,"l1_44.cv"    ; stereo convolved (wet) audio
adrydel         delay   (1-imix)*adry,idel  ; Delay dry signal, to align it with
                                            ; convolved signal. Apply level
                                            ; adjustment here too.
                outs    ivol*(adrydel+imix*awet1),ivol*(adrydel+imix*awet2)
                                            ; Mix wet & dry signals, and output
        endin
```

AUTHOR

Greg Sullivan
1996

59.2 cross2

```
ar      cross2      ain1, ain2, isize, ioverlap, iwin, kbias
```

DESCRIPTION

This is an implementation of cross synthesis using FFT's.

INITIALIZATION

isize – This is the size of the FFT to be performed. The larger the size the better the frequency response but a sloppy time response.

ioverlap – This is the overlap factor of the FFT's, must be a power of two. The best settings are 2 and 4. A big overlap takes a long time to compile.

iwin – This is the ftable that contains the window to be used in the analysis.

PERFORMANCE

ain1 – The stimulus sound. Must have high frequencies for best results.

ain2 – The modulating sound. Must have a moving frequency response (like speech) for best results.

kbias – The amount of cross synthesis. 1 is the normal, 0 is no cross synthesis.

EXAMPLES

```
a1   oscil   10000, 1, 1
a2   rand    10000
a3   cross2  a2, a1, 2048, 4, 2, 1
     out     a3
```

If ftable one is a speech sound, this will result in speaking white noise.
ftable 2 must be a window function (**GEN20**).

AUTHOR

Paris Smaragdis
MIT, Cambridge
1997

60 SIGNAL MODIFIERS: PANNING AND SPATIALIZATION

60.1 pan

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

DESCRIPTION

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 (inscribed) 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
```

60.2 locsig, locsend

```
a1, a2                locsig        asig, kdegree, kdistance, kreverbsend
a1, a2, a3, a4        locsig        asig, kdegree, kdistance, kreverbsend
a1, a2                locsend
a1, a2, a3, a4        locsend
```

DESCRIPTION

locsig takes an input signal and distributes it among 2 or 4 channels using values in degrees to calculate the balance between adjacent channels. It also takes arguments for distance (used to attenuate signals that are to sound as if they are some distance further than the loudspeaker itself), and for the amount the signal that will be sent to reverberators. This unit is based upon the example in the Charles Dodge/Thomas Jerse book, *Computer Music*, page 320.

locsend depends upon the existence of a previously defined **locsig**. The number of output signals must match the number in the previous **locsig**. The output signals from **locsend** are derived from the values given for distance and reverb in the **locsig** and are ready to be sent to local or global reverb units (see example below). The reverb amount and the balance between the 2 or 4 channels are calculated in the same way as described in the Dodge book (an essential text!).

PERFORMANCE

kdegree – value between 0 and 360 for placement of the signal in a 2 or 4 channel space configured as: a1=0, a2=90, a3=180, a4=270 (kdegree=45 would balanced the signal equally between a1 and a2). **locsig** maps *kdegree* to sin and cos functions to derive the signal balances (i.e.: asig=1, kdegree=45, a1=a2=.707).

kdistance – value >= 1 used to attenuate the signal and to calculate reverb level to simulate distance cues. As *kdistance* gets larger the sound should get softer and somewhat more reverberant (assuming the use of **locsend** in this case).

kreverbsend – the percentage of the direct signal that will be factored along with the distance and degree values to derive signal amounts that can be sent to a reverb unit such as reverb, or reverb2.

EXAMPLE

```
asig  some audio signal
kdegree                     line  0, p3, 360
kdistance    line           1, p3, 10
a1, a2, a3, a4              locsig        asig, kdegree, kdistance, .1
ar1, ar2, ar3, ar4          locsend
ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4
             outq                 a1, a2, a3, a4
                            endin

             instr 99       ; reverb instrument
a1           reverb2        ga1, 2.5, .5
a2           reverb2        ga2, 2.5, .5
a3           reverb2        ga3, 2.5, .5
a4           reverb2        ga4, 2.5, .5
  outq       a1, a2, a3, a4
ga1 = 0
ga2 = 0
ga3 = 0
ga4 = 0
```

In the above example, the signal, asig, is sent around a complete circle once during the duration of a note while at the same time it becomes more and more "distant" from the listeners' location. **locsig** sends the appropriate amount of the signal internally to **locsend**. The outputs of the **locsend** are added to global accumulators in a common Csound style and the global signals are used as inputs to the reverb units in a separate instrument.

locsig is useful for quad and stereo panning as well as fixed placed of sounds anywhere between two loudspeakers. Below is an example of the fixed placement of sounds in a stereo field.

```
            instr 1
a1, a2      locsig          asig, p4, p5, .1
ar1, ar2    locsend
ga1 = ga1+ar1
ga2 = ga2+ar2
  outs      a1, a2
            endin

            instr 99
  ; reverb....
            endin
```

A few notes:

```
  ;place the sound in the left speaker and near
  i1 0 1 0 1
  ;place the sound in the right speaker and far
  i1 1 1 90 25
  ;place the sound equally between left and right and in the middle ground distance
  i1 2 1 45 12
  e
```

The next example shows a simple intuitive use of the distance value to simulate Doppler shift. The same value is used to scale the frequency as is used as the distance input to **locsig**.

```
kdistance    line           1, p3, 10
kfreq         =   (ifreq * 340) / (340 + kdistance)
asig                        oscili        iamp, kfreq, 1
kdegree                     line  0, p3, 360
a1, a2, a3, a4              locsig        asig, kdegree, kdistance, .1
ar1, ar2, ar3, ar4          locsend
```

AUTHOR

Richard Karpen
Seattle, Wash
1998 (New in Csound version 3.48)

60.3 space, spsend, spdist

```
a1, a2, a3, a4        space        asig, ifn, ktime, kreverbsend [,kx, ky]
a1, a2, a3, a4        spsend
k1                    spdist       ifn, ktime, [,kx, ky]
```

DESCRIPTION

space takes an input signal and distributes it among 4 channels using Cartesian xy coordinates to calculate the balance of the outputs. The xy coordinates can be defined in a separate text file and accessed through a Function statement in the score using **GEN28**, or they can be specified using the optional *kx*, *ky* arguments. There advantages to the former are:

- A graphic user interface can be used to draw and edit the trajectory through the Cartesian plane
- The file format is in the form time1 X1 Y1 time2 X2 Y2 time3 X3 Y3 allowing the user to define a time-tagged trajectory.

space then allows the user to specify a time pointer (much as is used for **pvoc**, **lpread** and some other units) to have detailed control over the final speed of movement.

spsend depends upon the existence of a previously defined **space**. The output signals from **spsend** are derived from the values given for XY and reverb in the **space** and are ready to be sent to local or global reverb units (see example below).

spdist uses the same xy data as **space**, also either from a text file using **GEN28** or from x and y arguments given to the unit directly. The purpose of this unit is to make available the values for distance that are calculated from the xy coordinates. In the case of **space** the xy values are used to determine a distance which is used to attenuate the signal and prepare it for use in **spsend**. But it is also useful to have these values for distance available to scale the frequency of the signal before it is sent to the **space** unit.

PERFORMANCE

The configuration of the XY coordinates in space places the signal in the following way:

- a1 is -1, 1
- a2 is 1, 1
- a3 is -1, -1
- a4 is 1, -1.

This assumes a loudspeaker set up as a1 is left front, a2 is right front, a3 is left back, a4 is right back. Values greater than 1 will result in sounds being attenuated as if in the distance. **space** considers the speakers to be at a distance of 1; smaller values of XY can be used, but **space** will not amplify the signal in this case. It will, however balance the signal so that it can sound as if it were within the 4 speaker **space**. x=0, y=1, will place the signal equally balanced between left and right front channels, x=y=0 will place the signal equally in all 4 channels, and so on. Although there must be 4 output signal from **space**, it can be used in a 2 channel orchestra. If the XYs are kept so that Y>=1, it should work well to do panning and fixed localization in a stereo field.

ifn – number of the stored function created using **GEN28**. This function generator reads a text file which contains sets of three values representing the xy coordinates and a time-tag for when the signal should be placed at that location. The file should look like:

```
0       -1       1
1        1       1
2        4       4
2.1     -4      -4
3       10      -10
5       -40      0
```

If that file were named "move" then the **GEN28** call in the score would like:

```
f1 0 0 28 "move"
```

GEN28 takes 0 as the size and automatically allocates memory. It creates values to 10 milliseconds of resolution. So in this case there will be 500 values created by interpolating X1 to X2 to X3 and so on, and Y1 to Y2 to Y3 and so on, over the appropriate number of values that are stored in the function table. In the above example, the sound will begin in the left front, over 1 second it will move to the right front, over another second it move further into the distance but still in the left front, then in just 1/10th of a second it moves to the left rear, a bit distant. Finally over the last .9 seconds the sound will move to the right rear, moderately distant, and it comes to rest between the two left channels (due west!), quite distant. Since the values in the table are accessed through the use of a time-pointer in the **space** unit, the actual timing can be made to follow the file's timing exactly or it can be made to go faster or slower through the same trajectory. If you have access to the GUI that allows one to draw and edit the files, there is no need to create the text files manually. But as long as the file is ASCII and in the format shown above, it doesn't matter how it is made!

Important: If *ifn* is 0 then **space** will take its values for the xy coordinates from kx and ky.

ktime – index into the table containing the xy coordinates. If used like:

```
 ktime line 0, 5, 5
a1, a2, a3, a4 space asig, 1, ktime, ...
```

with the file "move" described above, the speed of the signal's movement will be exactly as described in that file. However:

```
 ktime line 0, 10, 5
```

the signal will move at half the speed specified. Or in the case of:

```
 ktime line 5, 15, 0
```

the signal will move in the reverse direction as specified and 3 times slower! Finally:

```
 ktime line 2, 10, 3
```

will cause the signal to move only from the place specified in line 3 of the text file to the place specified in line 5 of the text file, and it will take 10 seconds to do it.

kreverbsend – the percentage of the direct signal that will be factored along with the distance as derived from the XY coordinates to calculate signal amounts that can be sent to reverb units such as reverb, or reverb2.

kx, ky – when *ifn* is 0, **space** and **spdist** will use these values as the XY coordinates to localize the signal. They are optional and both default to 0.

EXAMPLE

```
                          instr 1
asig       some audio signal
ktime      line           0, p3, p10
a1,a2,a3,a4 space         asig,1, ktime, .1
ar1,ar2,ar3,ar4 spsend

ga1 = ga1+ar1
ga2 = ga2+ar2
ga3 = ga3+ar3
ga4 = ga4+ar4

                          outq  a1, a2, a3, a4
                          endin

instr 99 ; reverb instrument

a1         reverb2        ga1, 2.5, .5
a2         reverb2        ga2, 2.5, .5
a3         reverb2        ga3, 2.5, .5
a4         reverb2        ga4, 2.5, .5

                          outq  a1, a2, a3, a4
ga1 = 0
ga2 = 0
ga3 = 0
ga4 = 0
```

In the above example, the signal, *asig*, is moved according to the data in Function #1 indexed by *ktime*. **space** sends the appropriate amount of the signal internally to **spsend**. The outputs of the **spsend** are added to global accumulators in a common Csound style and the global signals are used as inputs to the reverb units in a separate instrument.

space can useful for quad and stereo panning as well as fixed placed of sounds anywhere between two loudspeakers. Below is an example of the fixed placement of sounds in a stereo field using XY values from the score instead of a function table.

```
                          instr 1
...
a1,a2,a3,a4               space         asig, 0, 0, .1, p4, p5
ar1,ar2,ar3,ar4           spsend

ga1 = ga1+ar1
ga2 = ga2+ar2
           outs a1, a2
                          endin

                          instr 99 ; reverb....
....
                          endin
```

A few notes: p4 and p5 are the X and Y values

```
;place the sound in the left speaker and near
i1 0 1 -1 1
;place the sound in the right speaker and far
i1 1 1 45 45
;place the sound equally between left and right and in the middle ground distance
i1 2 1 0 12
e
```

The next example shows a simple intuitive use of the distance values returned by **spdist** to simulate Doppler shift.

```
ktime                   line  0, p3, 10
kdist                   spdist        1, ktime
kfreq      = (ifreq * 340) / (340 + kdist)
asig                    oscili        iamp, kfreq, 1

a1, a2, a3, a4          space         asig, 1, ktime, .1
ar1, ar2, ar3, ar4      spsend
```

The same function and time values are used for both **spdist** and **space**. This insures that the distance values used internally in the **space** unit will be the same as those returned by **spdist** to give the impression of a Doppler shift!

AUTHOR

Richard Karpen
Seattle, Wash
1998 (New in Csound version 3.48)

60.4 hrtfer

```
aLeft, aRight  hrtfer    asig, kAz, kElev, "HRTFcompact"
```

DESCRIPTION

Output is binaural (headphone) 3D audio.

INITIALIZATION

kAz – azimuth value in degrees. Positive values represent position on the right, negative values are positions on the left.

kElev – elevation value in degrees. Positive values represent position above horizontal, negative values are positions above horizontal.

At present, the only file which can be used with **hrtfer** is HRTFcompact. It must be passed to the opcode as the last argument within quotes as shown above.

HRTFcompact my be obtained via anonymous ftp from:

```
ftp://ftp.maths.bath.ac.uk/pub/dream/utilities/Analysis/HRTFcompact
```

PERFORMANCE

These unit generators place a mono input signal in a virtual 3D space around the listener by convolving the input with the appropriate HRTF data specified by the opcode's azimuth and elevation values. **hrtfer** allows these values to be k-values, allowing for dynamic spatialization. **hrtfer** can only place the input at the requested position because the HRTF is loaded in at i-time (remember that currently, Csound has a limit of 20 files it can hold in memory, otherwise it causes a segmentation fault). The output will need to be scaled either by using balance or by multiplying the output by some scaling constant.

Note – the sampling rate of the orchestra must be 44.1kHz. This is because 44.1kHz is the sampling rate at which the HRTFs were measured. In order to be used at a different rate, the HRTFs would need to be re-sampled at the desired rate.

EXAMPLE

```
kaz             linseg  0, p3, -360  ; move the sound in circle
kel             linseg  -40, p3, 45  ; around the listener, changing
                                     ; elevation as its turning
asrc            soundin "soundin.1"
aleft,aright    hrtfer  asrc, kaz, kel, "HRTFcompact"
aleftscale      =       aleft * 200
arightscale     =       aright * 200
                outs    aleftscale, arightscale
```

AUTHORS

Eli Breder & David MacIntyre
Montreal
1996

60.5 vbaplsinit, vbap4, vbap8, vbap16, vbap4move, vbap8move, vbap16move, vbapz, vbapzmove

```
                        vbaplsinit idim, ilsnum, idir1, idir2,...

ar1, ar2, ar3, ar4      vbap4      asig, iazim, ielev, ispread

ar1, ar2, ar3, ar4,     vbap8      asig, iazim, ielev, ispread
ar5, ar6, ar7, ar8

ar1, ar2, ar3, ar4,     vbap16     asig, iazim, ielev, ispread
ar5, ar6, ar7, ar8,
ar9, ar10, ar11, ar12,
ar13, ar14, ar15, ar16

ar1, ar2, ar3, ar4      vbap4move  asig, ispread, ifldnum, ifld1, ifld2,...

ar1, ar2, ar3, ar4,     vbap8move  asig, ispread, ifldnum, ifld1, ifld2,...
ar5, ar6, ar7, ar8

ar1, ar2, ar3, ar4,     vbap16move asig, ispread, ifldnum, ifld1, ifld2,...
ar5, ar6, ar7, ar8,
ar9, ar10, ar11, ar12,
ar13, ar14, ar15, ar16

                        vbapz      inumchnls, istartndx, asig, iazim,
                                   ielev, ispread

                        vbapzmove  inumchnls, istartndx, ispread, ifldnum,
                                   ifld1, ifld2,...
```

DESCRIPTION

Distribute an audio signal among 2 to 16 output channels or write it to a ZAK array, all with localization control.

INITIALIZATION

idim – dimensionality of loudspeaker array. Either 2 or 3.

ilsnum – number of loudspeakers. In two dimensions, the number can vary from 2 to 16. In three dimensions, the number can vary from 3 and 16.

idir1, *idir2*, etc. – directions of loudspeakers. Number of directions must be less than or equal to 16. In two-dimensional loudspeaker positioning, *idir*n is the azimuth angle respective to *n*th channel. In three-dimensional loudspeaker positioning, fields are the azimuth and elevation angles of each loudspeaker consequently (*azi1*, *ele1*, *azi2*, *ele2*, etc.).

iazim – azimuth angle of the virtual source

ielev – elevation angle of the virtual source

ispread – spreading of the virtual source (range 0 – 100). If value is zero, conventional amplitude panning is used. When *ispread* is increased, the number of loudspeakers used in panning increases. If value is 100, the sound is applied to all loudspeakers.

ifldnum – number of fields (absolute value must be 2 or larger). If *ifldnum* is positive, the virtual source movement is a polyline specified by given directions. Each transition is performed in an equal time interval. If *ifldnum* is negative, specified angular velocities are applied to the virtual source during specified relative time intervals (see below).

ifld1, ifld2, etc. – azimuth angles or angular velocities, and relative durations of movement phases (see below).

inumchnls – number of channels to write to the ZA array. Must be in the range 2 – 256.

istartndx – first index or position in the ZA array to use

PERFORMANCE

asig – audio signal to be panned

vbap4, **vbap8**, and **vbap16** take an input signal, *asig*, and distribute it among 2 to 16 outputs, according to the controls *iazim* and *ielev*, and the configured loudspeaker placement. If *idim* = 2, *ielev* is set to zero. The distribution is performed using Vector Base Amplitude Panning (VBAP – See reference). VBAP distributes the signal using loudspeaker data configured with **vbaplsinit**. The signal is applied to, at most, two loudspeakers in 2-D loudspeaker configurations, and three loudspeakers in 3-D loudspeaker configurations. If the virtual source is panned outside the region spanned by loudspeakers, the nearest loudspeakers are used in panning.

vbap4move, **vbap8move**, and **vbap16move** allow the use of moving virtual sources. If *ifldnum* is positive, the fields represent directions of virtual sources and equal times, *iazi1*, [*iele1*,] *iazi2*, [*iele2*,], etc. The position of the virtual source is interpolated between directions starting from the first direction and ending at the last. Each interval is interpolated in time that is fraction total_time / number_of_intervals of the duration of the sound event.

If *ifldnum* is negative, the fields represent angular velocities and equal times. The first field is, however, the starting direction, *iazi1*, [*iele1*,] *iazi_vel1*, [*iele_vel1*,] *iazi_vel2*, [*iele_vel2*,] .... Each velocity is applied to the note that is fraction total_time / number_of_velocities of the duration of the sound event. If the elevation of the virtual source becomes greater than 90 degrees or less than 0 degrees, the polarity of angular velocity is changed. Thus the elevational angular velocity produces a virtual source that moves up and down between 0 and 90 degrees.

The opcodes **vbapz** and **vbapzmove** are the multiple channel analogs of the above opcodes, working an *inumchnls* and using a ZAK array for output.

EXAMPLE

2-D panning example with stationary virtual sources:

```
            sr              =     4100
            kr              =     441
            ksmps           =     100
            nchnls          =     4
            vbaplsinit            2, 6,  0, 45, 90, 135, 200, 245, 290, 315

            instr 1
asig        oscil           20000, 440, 1
a1,a2,a3,a4,a5,a6,a7,a8     vbap8  asig, p4, 0, 20 ;p4 = azimuth
```

```
  ;render twice with alternate outq statements
  ;  to obtain two 4 channel .wav files:

            outq     a1,a2,a3,a4
;           outq     a5,a6,a7,a8
            endin
```

REFERENCE

Ville Pulkki: "Virtual Sound Source Positioning Using Vector Base Amplitude Panning"
Journal of the Audio Engineering Society,
1997 June, Vol. 45/6, p. 456.

AUTHORS

Ville Pulkki
Sibelius Academy Computer Music Studio
Laboratory of Acoustics and Audio Signal Processing
Helsinki University of Technology
Helsinki, Finland
May, 2000 (New in Csound version 4.06)

John ffitch (**vbapz**, **vbabzmove**)
University of Bath/Codemist Ltd.
Bath, UK
May, 2000 (New in Csound version 4.06)

61 SIGNAL MODIFIERS: SAMPLE LEVEL OPERATORS

61.1 samphold, downsamp, upsamp, interp, integ, diff

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

DESCRIPTION

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.

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

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

PERFORMANCE

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

61.2 ntrpol

```
ir      ntrpol      isig1, isig2, ipoint [, imin, imax]
kr      ntrpol      ksig1, ksig2, kpoint [, imin, imax]
ar      ntrpol      asig1, asig2, kpoint [, imin, imax]
```

DESCRIPTION

Calculates the weighted mean value (i.e. linear interpolation) of two input signals

INITIALIZATION

imin – minimum xpoint value (optional, default 0)

imax – maximum xpoint value (optional, default 1)

PERFORMANCE

xsig1, *xsig2* – input signals

xpoint – interpolation point between the two values

ntrpol opcode outputs the linear interpolation between two input values. *xpoint* is the distance of evaluation point from the first value. With the default values of *imin* and *imax*, (0 and 1) a zero value indicates no distance from the first value and the maximum distance from the second one. With a 0.5 value, **ntrpol** will output the mean value of the two inputs, indicating the exact half point between *xsig1* and *xsig2*. A 1 value indicates the maximum distance from the first value and no distance from the second one. The range of *xpoint* can be also defined with *imin* and *imax* to make its management easier.

These opcodes are useful for crossfading two signals.

AUTHOR

Gabriel Maldonado
Italy
October, 1998 (New in Csound version 3.49)

61.3 fold

```
ar      fold        asig, kincr
```

DESCRIPTION

Adds artificial foldover to an audio signal.

PERFORMANCE

asig – input signal

kincr – amount of foldover expressed in multiple of sampling rate. Must be >= 1

fold is an opcode which creates artificial foldover. For example, when *kincr* is equal to 1 with sr=44100, no foldover is added. When *kincr* is set to 2, the foldover is equivalent to a downsampling to 22050, when it is set to 4, to 11025 etc. Fractional values of *kincr* are possible, allowing a continuous variation of foldover amount. This can be used for a wide range of special effects.

EXAMPLE

```
          instr          1
kfreq   line      1,p3,200
a1      oscili    10000, 100, 1
k1      init      8.5
a1      fold      a1, kfreq
        out       a1
          endin
```

AUTHOR

Gabriel Maldonado
Italy
1999
New in Csound version 3.56

62 ZAK PATCH SYSTEM

The zak opcodes are used to create a system for i-rate, k-rate or a-rate patching. The zak system can be thought of as a global array of variables. These opcodes are useful for performing flexible patching or routing from one instrument to another. The system is similar to a patching matrix on a mixing console or to a modulation matrix on a synthesizer. It is also useful whenever an array of variables is required.

The zak system is initialized by the **zakinit** opcode, which is usually placed just after the other global initializations: **sr**, **kr**, **ksmps**, **nchnls**. The **zakinit** opcode defines two areas of memory, one area for i- and k-rate patching, and the other area for a-rate patching. The **zakinit** opcode may only be called once. Once the zak space is initialized, other zak opcodes can be used to read from, and write to the zak memory space, as well as perform various other tasks.

62.1 zakinit

zakinit isizea, isizek

DESCRIPTION

Establishes zak space. Must be called only once.

INITIALIZATION

isizea – the number of audio rate locations for a-rate patching. Each location is actually an array which is ksmps long.

isizek – the number of locations to reserve for floats in the zk space. These can be written and read at i- and k-rates.

PERFORMANCE

At least one location each is always allocated for both za and zk spaces. There can be thousands or tens of thousands za and zk ranges, but most pieces probably only need a few dozen for patching signals. These patching locations are referred to by number in the other zak opcodes.

To run **zakinit** only once, put it outside any instrument definition, in the orchestra file header, after **sr**, **kr**, **ksmps**, and **nchnls.**

EXAMPLE

```
zakinit 10 30
```

reserves memory for locations 0 to 30 of zk space and for locations 0 to 10 of a-rate za space. With ksmps = 8, this would take 31 floats for zk and 80 floats for za space.

AUTHOR

Robin Whittle
Australia
May 1997

62.2 ziw, zkw, zaw, ziwm, zkwm, zawm

```
ziw         isig, indx
zkw         ksig, kndx
zaw         asig, kndx
ziwm        isig, indx [, imix]
zkwm        ksig, kndx [, imix]
zawm        asig, kndx [, imix]
```

DESCRIPTION

Write to a location in zk space at either i-rate or k-rate, or a location in za space at a-rate. Writing can be with, or without, mixing.

INITIALIZATION

indx – points to the zk location to which to write.

isig – initializes the value of the zk location.

PERFORMANCE

kndx – points to the zk or za location to which to write.

ksig – value to be written to the zk location.

asig – value to be written to the za location.

ziw writes *isig* into the zk variable specified by *indx*.

zkw writes *ksig* into the zk variable specified by *kndx*.

zaw writes *asig* into the za variable specified by *kndx*.

These opcodes are fast, and always check that the index is within the range of zk or za space. If not, an error is reported, 0 is returned, and no writing takes place.

ziwm, **zkwm**, and **zawm** are mixing opcodes, i.e. they add the signal to the current value of the variable. If no *imix* is specified, mixing always occurs, but if *imix* is specified, *imix* = 0 will cause overwriting, like **ziw**, **zkw**, and **zaw**, and any other value will cause mixing.

Caution: When using the mixing opcodes **ziwm**, **zkwm**, and **zawm**, care must be taken that the variables mixed to, are zeroed at the end (or start) of each k or a cycle. Continuing to add signals to them, can cause their values can drift to astronomical figures.

One approach would be to establish certain ranges of zk or za variables to be used for mixing, then use **zkcl** or **zacl** to clear those ranges.

EXAMPLES

```
instr 1
zkw             kzoom, p8     ; p8 in the score line determines where
                              ; in zk space kzoom iswritten
endin

instr 2
zkw             kzoom, 7      ; always writes kzoom to zk location 7

endin
```

```
          instr 3
kxxx      phasor        1
kdest     =             40+kxxx*16    ; This will write azoom to
                                      ; locations 40 to 55
          zaw           azoom,kdest   ; on a one second scan cycle
          endin
```

AUTHOR

Robin Whittle
Australia
May 1997

62.3 zir, zkr, zar, zarg

```
ir      zir         indx
kr      zkr         kndx
ar      zar         kndx
ar      zarg        kndx, kgain
```

DESCRIPTION

Read from a location in zk space at i-rate or k-rate, or a location in za space at a-rate.

INITIALIZATION

kndx – points to the zk or za location to be read.

kgain – multiplier for the a-rate signal.

PERFORMANCE

zir reads the signal at *indx* location in zk space.

zkr reads the array of floats at *kndx* in zk space.

zar reads the array of floats at *kndx* in za space, which are ksmps number of a-rate floats to be processed in a k cycle.

zarg is similar to **zar**, but multiplies the a-rate signal by a k-rate value *kgain*.

AUTHOR

Robin Whittle
Australia
May 1997

62.4 zkmod, zamod, zkcl, zacl

```
kr      zkmod       ksig, kzkmod
ar      zamod       asig, kzamod
        zkcl        kfirst, klast
        zacl        kfirst, klast
```

DESCRIPTION

Clear and modulate the za and zk spaces.

PERFORMANCE

ksig – the input signal

kzkmod – controls which zk variable is used for modulation. A positive value means additive modulation, a negative value means multiplicative modulation. A value of 0 means no change to *ksig*. *kzkmod* can be i-rate or k-rate

kfirst – first zk or za location in the range to clear.

klast – last zk or za location in the range to clear.

zkmod facilitates the modulation of one signal by another, where the modulating signal comes from a zk variable. Either additive or multiplicative modulation can be specified.

zamod modulates one a-rate signal by a second one, which comes from a za variable. The location of the modulating variable is controlled by the i-rate or k-rate variable *kzamod*. This is the a-rate version of **zkmod**

zkcl clears one or more variables in the zk space. This is useful for those variables which are used as accumulators for mixing k-rate signals at each cycle, but which must be cleared before the next set of calculations.

zacl clears one or more variables in the za space. This is useful for those variables which are used as accumulators for mixing a-rate signals at each cycle, but which must be cleared before the next set of calculations.

EXAMPLES

```
k1      zkmod   ksig, 23            ; adds value at location 23 to ksig

a1      zamod   asig, -402   ; multiplies asig by value at location 402
```

AUTHOR

Robin Whittle
Australia
May 1997

63 Operations Using Spectral Data Types

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

63.1 specaddm, specdiff, specscal, spechist, specfilt

```
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
```

63.2 specptrk

```
koct,    specptrk    wsig, kvar, ilo, ihi, istr, idbthresh, inptls,//
kamp                 irolloff[, iodd, iconfs, interp, ifprd, iwtflg]
```

DESCRIPTION

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

INITIALIZATION

ilo, ihi, istr – 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, kamp*) 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).

iwtftg (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. (See **spectrum**.)

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 bibs/oct
            specdisp wsig, .2                  ; display this and now estimate
    koct,ka spectrk  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 \ 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
```

63.3 specsum, specdisp

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

INITIALIZATION

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

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

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

EXAMPLE

```
  ksum specsum    wsig,  1        ; sum the spec bins, and ksmooth
       if         ksum < 2000   kgoto  zero ; if sufficient amplitude
  koct specptrk   wsig            ;    pitch-track the signal
       kgoto      contin
zero:  koct        =     0        ; else output zero
contin:
.
```

63.4 spectrum

```
wsig    spectrum    xsig, iprd, iocts, ifrqs, iq[,ihann, idbout, idsprd,
                    idsinrs]
```

DESCRIPTION

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 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
```

64 SIGNAL INPUT AND OUTPUT: INPUT

64.1 in, ins, inq, inh, ino, soundin, diskin

```
ar1       in
ar1, ar2  ins
ar1, ar2, inq
ar3, ar4
ar1, ar2, inh
ar3, ar4,
ar5, ar6
ar1, ar2, ino
ar3, ar4,
ar5, ar6,
ar7, ar8
ar1       soundin     ifilcod[, iskptim[, iformat]]
ar1, ar2  soundin     ifilcod[, iskptim[, iformat]]
ar1, ar2, soundin     ifilcod[, iskptim[, iformat]]
ar3, ar4
ar1[,ar2  diskin      ifilcod, kpitch[, iskiptim [,iwraparound[, iformat]]]
[,a3,ar4]]
```

DESCRIPTION

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

INITIALIZATION

ifilcod – integer or character-string denoting the source soundfile name. An integer denotes the file soundin.filcod ; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named 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.

iwraparound – 1=on, 0=off (wraps around to end of file either direction) *kpitch* – can be any real number. a negative number signifies backwards playback. The given number is a pitch ratio, where:

- 1 = norm pitch,
- 2 = oct higher,
- 3 = 12th higher, etc;
- .5 = oct lower,
- .25 = 2oct lower, etc;
- -1 = norm pitch backwards,
- -2 = oct higher backwards, etc..

PERFORMANCE

in, **ins**, **inq, inh, ino** – 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. *stain* 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.

diskin is identical to **soundin**, except that it can alter the pitch of the sound that is being read.

AUTHORS

Barry Vercoe, Matt Ingols/Mike Berry
MIT, Mills College
1993-1997

64.2 inx, in32, inch, inz

```
ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8,          inx
ar9, ar10, ar11, ar12, ar13, ar14, ar15, ar16

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8,          in32
ar9, ar10, ar11, ar12, ar13, ar14, ar15, ar16,
ar17, ar18, ar19, ar20, ar21, ar22, ar23, ar24,
ar25, ar26, ar27, ar28, ar29, ar30, ar31, ar32

ar1                                              inch  ksig1

                                                 inz   ksig1
```

DESCRIPTION

These units read multi-channel audio data from an external device or stream.

PERFORMANCE

inx and **in32** read 16 and 32 channel inputs, respectively.

inch reads from a numbered channel determined by *ksig1* into *a1*.

inz reads audio samples in **nchnls** into a ZAK array starting at *ksig1*.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
May, 2000 (New in Csound version 4.06)

This page intentionally left blank.

65 SIGNAL INPUT AND OUTPUT: OUTPUT

65.1 soundout, soundouts, out, outs1, outs2, outs, outq1, outq2, outq3, outq4, outq, outh, outo

```
soundout    asig, ifilcod[, iskptim]
soundouts   asig, ifilcod[, iskptim]
out         asig
outs1       asig
outs2       asig
outs        asig1, asig2
outq1       asig
outq2       asig
outq3       asig
outq4       asig
outq        asig1, asig2, asig3, asig4
outh        asig1, asig2, asig3, asig4, asig5, asig6
outo        asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8
```

DESCRIPTION

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

INITIALIZATION

ifilcod – integer or character-string denoting the destination soundfile name. An integer denotes the file soundin.filcod; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named 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.

PERFORMANCE

out, **outs**, **outq, outh, outo –** 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, hex, or oct) should agree with **nchnls**, but as of version 3.50, will attempt to change and incorrect opcode, to arguer with **nchnls** statement. Units can be chosen to direct sound to any particular channel: **outs1** sends to stereo channel 1, **outq3** to quad channel 3, etc.

soundout and **soundouts** write audio output to a disk file. **soundouts** is currently not implemented.

AUTHORS

Barry Vercoe, Matt Ingols/Mike Berry
MIT, Mills College
1993-1997

65.2 outx, out32, outc, outch, outz

```
outx        asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8,
            asig9, asig10, asig11, asig12, asig13, asig14, asig15,
            asig16

out32       asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8,
            asig10, asig11, asig12, asig13, asig14, asig15, asig16,
            asig17, asig18, asig19, asig20, asig21, asig22, asig23,
            asig24, asig25, asig26, asig27, asig28, asig29, asig30,
            asig31, asig32

outc        asig1[, asig2,...]
outch       ksig1, asig1, ksig2, asig2, ...
outz        ksig1
```

DESCRIPTION

These units write multi-channel audio data to an external device or stream.

PERFORMANCE

outx and **out32** output 16 and 32 channels of audio.

outc outputs as many channels as provided. Any channels greater than **nchnls** are ignored, and zeros are added as necessary

outch outputs *asig1* on the channel determined by *ksig1*, *asig2* on the channel determined by *ksig2*, etc.

outz outputs from a ZAK array, for **nchnls** of audio.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
May, 2000 (New in Csound version 4.06)

This page intentionally left blank.

66 SIGNAL INPUT AND OUTPUT: FILE I/O

66.1 dumpk, dumpk2, dumpk3, dumpk4, readk, readk2, readk3, readk4

```
          dumpk       ksig, ifilname, iformat, iprd
          dumpk2      ksig1, ksig2, ifilname, iformat, iprd
          dumpk3      ksig1, ksig2, ksig3, ifilname, iformat, iprd
          dumpk4      ksig1, ksig2, ksig3, ksig4, ifilname, iformat, iprd
kr1       readk       ifilname, iformat, iprd[, ipol]
kr1,kr2   readk2      ifilname, iformat, iprd[, ipol]
kr1,kr2,  readk3      ifilname, iformat, iprd[, ipol]
kr3
kr1,kr2,  readk4      ifilname, iformat, iprd[, ipol]
kr3,kr4
```

DESCRIPTION

Periodically 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 either be a full path name with target directory specified or a simple filename to be created within the current directory

iformat – specifies the output data format:

- 1 = 8-bit signed char(high order 8 bits of a 16-bit integer
- 4 = 16-bit short integers
- 5 = 32bit 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. The output file contains no header information.

iprd – the period of *ksig* output i seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will create an output file sampled at the orchestra control rate.

ipol (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). Currently not supported.

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 **readk** units in an instrument or orchestra, and they may read from the same or different files. There may be any number of **dumpk** units in an instrument or orchestra, but each must write to a different file.

OPCODE HISTORY

dumpk opcodes were originally called **kdump.** As of Csound version 3.493 that name is deprecated. **dumpk** should be used instead of **kdump**. The **readk** opcodes were originally called **kread**, but were not implemented until Csound version 3.52. However, the optional **readk** argument, *ipol* is ignored. This situation is expected to be corrected in a later release.

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  specptrk wsig, 6, .9, 0 ;and the pitch
      dumpk3   knum, ktemp, cpsoct(koct), “what happened when”, 8 0
                               ;& save them
```

66.2 fout, foutk, fouti, foutir, fiopen

```
          fout        "ifilename", iformat, aout1[, aout2, aout3,...,aoutN]
          foutk       "ifilename", iformat, kout1[, kout2, kout3,.....,koutN]
          fouti       ihandle, iformat, iflag, iout1[, iout2,
                      iout3,....,ioutN]
          foutir      ihandle, iformat, iflag, iout1[, iout2,\\
                      iout3,....,ioutN]
ihandle   fiopen      "ifilename",imode
```

DESCRIPTION

fout, **foutk**, **fouti** and **foutir** output *N* a-, k-, or i-rate signals to a specified file of *N* channels. **fiopen** can be used to open a file in one of the specified modes.

INITIALIZATION

ifilename – a double-quote delimited string file name

iformat – a flag to choose output file format:

- for **fout** and **foutk** only:
- 0 – 32-bit floating point samples without header (binary PCM multichannel file)
- 1 – 16-bit integers without header (binary PCM multichannel file)
- 2 – 16-bit integers with .wav type header (Microsoft WAV mono or stereo file)
- for **fouti** and **foutir** only:
- 0 – floating point in text format
- 1 – 32-bit floating point in binary format

iflag – choose the mode of writing to the ASCII file (valid only in ASCII mode; in binary mode *iflag* has no meaning, but it must be present anyway). *iflag* can be a value chosen among the following:

- 0 – line of text without instrument prefix
- 1 – line of text with instrument prefix (see below)
- 2 – reset the time of instrument prefixes to zero (to be used only in some particular cases. See below)

iout,..., ioutN – values to be written to the file

imode – choose the mode of opening the file. *imode* can be a value chosen among the following:

- 0 – open a text file for writing
- 1 – open a text file for reading
- 2 – open a binary file for writing
- 3 – open a binary file for reading

PERFORMANCE

aout1,... aoutN – signals to be written to the file

kout1,...koutN – signals to be written to the file

fout (file output) writes samples of audio signals to a file with any number of channels. Channel number depends by the number of *aoutN* variables (i.e. a mono signal with only an a-rate argument, a stereo signal with two a-rate arguments etc.) Maximum number of channels is fixed to 64. Multiple **fout** opcodes can be present in the same instrument, referring to different files.

Notice that, unlike **out**, **outs** and **outq**, **fout** does not zero the audio variable, so you must zero it after calling **fout**, if polyphony is to be used. You can use **incr** and **clear** opcodes for this task.

foutk operates in the same way as **fout**, but with k-rate signals. *iformat* can be set only to 0 or 1.

fouti and **foutir** write i-rate values to a file. The main use of these opcodes is to generate a score file during a real-time session. For this purpose, the user should set *iformat* to 0 (text file output) and *iflag* to 1, which enable the output of a prefix consisting of the strings *inum*, *actiontime*, and *duration*, before the values of *iout1...ioutN* arguments. The arguments in the prefix refer to instrument number, action time and duration of current note.

The difference between **fouti** and **foutir** is that, in the case of **fouti**, when *iflag* is set to 1, the duration of the first opcode is undefined (so it is replaced by a dot). Whereas, **foutir** is defined at the end of note, so the corresponding text line is written only at the end of the current note (in order to recognize its duration). The corresponding file is linked by the *ihandle* value generated by the **fiopen** opcode (see below). So **fouti** and **foutir** can be used to generate a Csound score while playing a real-time session.

fiopen opens a file to be used by the **fout** family of opcodes. It must be defined in the header section, external to any instruments. It returns a number, *ihandle*, which is unequivocally referring to the opened file.

Notice that **fout** and **foutk** can use either a string containing a file pathname, or a handle-number generated by **fiopen**. Whereas, with **fouti** and **foutir**, the target file can be only specified by means of a handle-number.

AUTHOR

Gabriel Maldonado
Italy
1999
New in Csound version 3.56

66.3 fin, fink, fini

```
fin     "ifilename", iskipframes, iformat, ain1[, ain2,\\
        ain3,...,ainN]
fink    "ifilename", iskipframes, iformat, kin1[, kin2,\\
        kin3,...,kinN]
fini    "ifilename", iskipframes, iformat, in1[, in2,\\
        in3,...,inN]
```

DESCRIPTION

Read signals from a file (at a-, k-, and i-rate)

INITIALIZATION

ifilename – input file name (can be a string or a handle number generated by **fiopen**)

iskipframes – number of frames to skip at the start (every frame contains a sample of each channel)

iformat – a number specifying the input file format

for **fin** and **fink**:

- 0 – 32 bit floating points without header
- 1 – 16 bit integers without header

and for **fini**:

- 0 – floating points in text format (loop; see below)
- 1 – floating points in text format (no loop; see below)
- 2 – 32 bit floating points in binary format (no loop)

PERFORMANCE

fin (file input) is the complement of **fout**: it reads a multichannel file to generate audio rate signals. At the present time no header is supported for the file format. The user must be sure that the number of channels of the input file is the same as the number of *ainX* arguments. **fink** is the same as **fin**, but operates at k-rate.

fini is the complement of **fouti** and **foutir**, it reads the values each time the corresponding instrument note is activated. When *iformat* is set to 0, if the end of file is reached, the file pointer is zeroed, restarting the scan from the beginning. When *iformat* is set to 1 or 2, no looping is enabled, so at the end of file, the corresponding variables will be filled with zeroes.

AUTHOR

Gabriel Maldonado
Italy
1999
New in Csound version 3.56

66.4 vincr, clear

```
vincr       asig, aincr
clear       avar1[,avar2, avar3,...,avarN]
```

DESCRIPTION

vincr increments an audio variable of another signal, i.e. accumulates output. **clear** zeroes a list of audio signals.

PERFORMANCE

asig – audio variable to be incremented

aincr – incrementing signal

avar1 [,*avar2*, *avar3*,...,*avarN*] – signals to be zeroed

vincr (variable increment) and **clear** are intended to be used together. **vincr** stores the result of the sum of two audio variables into the first variable itself (which is intended to be used as an accumulator in polyphony). The accumulator variable can be used for output signal by means of **fout** opcode. After the disk writing operation, the accumulator variable should be set to zero by means of **clear** opcode (or it will explode).

AUTHOR

Gabriel Maldonado
Italy
1999
New in Csound version 3.56

67 Signal Input and Output: Sound File Queries

67.1 filelen, filesr, filenchnls, filepeak

```
ir      filelen     "ifilcod"
ir      filesr      "ifilcod"
ir      filenchnls  "ifilcod"
ir      filepeak    "ifilcod"[, ichnl]
```

Description

Obtains information about a sound file.

Initialization

ifilecod – sound file to be queried

ichnl – channel to be used in calculating the peak value. Default is 0.

- *ichnl* = 0 returns peak value of all channels
- *ichnl* > 0 returns peak value of *ichnl*

Performance

filelen returns the length of the sound file *ifilcod* in seconds. **filesr** returns the sample rate of the sound file *ifilcod*. **filenchnls** returns the number of channels in the sound file *ifilcod*. **filepeak** returns the peak absolute value of the sound file *ifilcod*. Currently, **filepeak** supports only AIFF-C float files.

Author

Matt Ingalls
July, 1999
New in Csound version 3.57

This page intentionally left blank.

68 SIGNAL INPUT AND OUTPUT: PRINTING AND DISPLAY

68.1 print, display, dispfft

```
print       iarg[, iarg,...]
display     xsig, iprd[, inprds[, iwtflg]]
dispfft     xsig, iprd, iwsiz[, iwtyp[, idbouti[, iwtflg]]]
```

DESCRIPTION

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

INITIALIZATION

iprd – the period of display in seconds.

iwsiz – size of the input window in samples. A window of *iwsiz* points will produce a Fourier transform of *iwsiz*/2 points, spread linearly in frequency from 0 to sr/2. *iwsiz* must be a power of 2, 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, .01 ; generate a note envelope
   display  k1, p3 ; and display entire shape
```

68.2 printk, printks

```
printk      itime, kval, [ispace]
printks     "txtstring", itime, kval1, kval2, kval3, kval4
```

DESCRIPTION

These opcodes are intended to facilitate the debugging of orchestra code.

INITIALIZATION

itime – time in seconds between printings. (Default 1 second.)

ispace (optional) – number of spaces to insert before printing. (Max 130.)

"txtstring" – text to be printed. Can be up to 130 characters and must be in double quotes.

PERFORMANCE

*kval*x – The k-rate values to be printed. These are specified in *"txtstring"* with the standard C value specifier %f, in the order given. Use 0 for those which are not used.

printk prints one k-rate value on every k cycle, every second or at intervals specified. First the instrument number is printed, then the absolute time in seconds, then a specified number of spaces, then the *kval* value. The variable number of spaces enables different values to be spaced out across the screen – so they are easier to view.

printks prints numbers and text, with up to four printable numbers – which can be i- or k-rate values. **printks** is highly flexible, and if used together with cursor positioning codes, could be used to write specific values to locations in the screen as the Csound processing proceeds.

A special mode of operation allows this **printks** to convert *kval1* input parameter into a 0 to 255 value and to use it as the first character to be printed. This enables a Csound program to send arbitrary characters to the console. To achieve this, make the first character of the string a # and then, if desired continue with normal text and format specifiers. Three more format specifies may be used – they access *kval2, kval3* and *kval4*.

Both these opcodes can be run on every k cycle they are run in the instrument. To every accomplish this, set *itime* to 0.

When *itime* is not 0, the opcode print on the first k cycle it is called, and subsequently when every *itime* period has elapsed. The time cycles start from the time the opcode is initialized – typically the initialization of the instrument.

PRINT OUTPUT FORMATTING

Standard C language printf() control characters may be used, but must be prefaced with an additional backslash:

```
\\n or \\N  Newline
\\t or \\T  Tab
```

The standard C language %f format is used to print *kval1, kval2, kval3*, and *kval4*. For example:

```
 %f          prints with full precision: 123.456789
 %6.2f       prints 1234.56
 %5.0p       prints 12345
```

EXAMPLES

The following:

```
  printks \"Volume = %6.2f Freq = %8.3f\n\", 0.1, kvol, kfreq, 0, 0
would print:
  Volume = 1234.56 Freq = 12345.678
```

The following:

```
  printks \"#x\\y = %6.2\n\", 0.1, kxy, 0, 0, 0
would print a tab character followed by:
  x\y = 1234.56
```

AUTHOR

Robin Whittle
Australia
May 1997

68.3 printk2

printk2 kvar [, numspaces]

INITIALIZATION

numspaces – number of space characters printed before the value of kvar

PERFORMANCE

kvar - signal to be printed

Derived from Robin Whittle's **printk**, prints a new value of *kvar* each time *kvar* changes. Useful for monitoring MIDI control changes when using sliders.

WARNING! don't use this opcode with normal, continuously variant k-signals, because it can hang the computer, as the rate of printing is too fast.

AUTHOR

Gabriel Maldonado
Italy
1998 (New in Csound version 3.48)

---

69 THE STANDARD NUMERIC SCORE

69.1 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**.

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.

Three additional features are available for p2 alone: +, ^ + ***x***, and ^ – ***x***. 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 symbols ^ + ***x*** and ^ – ***x*** determine the current p2 by adding or subtracting, respectively, the value of ***x*** from the preceding p2. These may be used in p2 only.

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

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 ....

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.

NOTE

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. User-written routines can be used to modify score files before or after the above processes, provided the final orchestra-readable statement format is not violated. Sections of different formats can be sequentially batched; and sections of like format can be merged for automatic sorting.

69.2 Next-P and Previous-P Symbols

At the close of any of the operations **Carry**, **Tempo**, and **Sort**, 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 **np**7 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**. 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.

69.3 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.

Starting with Csound version 3.52, using the symbols (or) will result in an exponential interpolation ramp, similar to **expon**. The symbols { and } to define an exponential ramp have been deprecated. Using the symbol ~ will result in uniform, random distribution between the first and last values of the ramp. Use of these functions must follow the same rules as the linear ramp function.

69.4 Score Macros

```
#define     NAME # replacement text #
#define     NAME(a' b' c') # replacement text #
$NAME.
#undef      NAME
```

DESCRIPTION

Macros are textual replacements which are made in the score as it is being presented to the system. The macro system in Csound is a very simple one, and uses the characters # and $ to define and call macros. This can allow for simpler score writing, and provide an elementary alternative to full score generation systems. The score macro system is similar to, but independent of, the macro system in the orchestra language.

#define NAME – defines a simple macro. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Case is significant. This form is limiting, in that the variable names are fixed. More flexibility can be obtained by using a macro with arguments, described below.

#define NAME(*a' b' c'*) – defines a macro with arguments. This can be used in more complex situations. The name of the macro must begin with a letter and can consist of any combination of letters and numbers. Within the replacement text, the arguments can be substituted by the form: $A. In fact, the implementation defines the arguments as simple macros. There may be up to 5 arguments, and the names may be any choice of letters. Remember that case is significant in macro names.

$NAME. – calls a defined macro. To use a macro, the name is used following a $ character. The name is terminated by the first character which is neither a letter nor a number. If it is necessary for the name not to terminate with a space, a period, which will be ignored, can be used to terminate the name. The string, **$NAME.**, is replaced by the replacement text from the definition. The replacement text can also include macro calls.

#undef NAME – undefines a macro name. If a macro is no longer required, it can be undefined with **#undef NAME**.

INITIALIZATION

 # replacement text # – The replacement text is any character string (not containing a #) and can extend over multiple lines. The replacement text is enclosed within the # characters, which ensure that additional characters are not inadvertently captured.

PERFORMANCE

Some care is needed with textual replacement macros, as they can sometimes do strange things. They take no notice of any meaning, so spaces are significant. This is why, unlike the C programming language, the definition has the replacement text surrounded by # characters. Used carefully, this simple macro system is a powerful concept, but it can be abused.

ANOTHER USE FOR MACROS

When writing a complex score it is sometimes all too easy to forget to what the various instrument numbers refer. One can use macros to give names to the numbers. For example:

```
#define Flute             #i1#
#define Whoop             #i2#

$Flute.    0              10    4000    440
$Whoop.    5              1
```

EXAMPLES

Simple Macro

a note-event has a set of p-fields which are repeated:

```
#define ARGS # 1.01 2.33 138#
i1 0 1 8.00               1000 $ARGS
i1 0 1 8.01               1500 $ARGS
i1 0 1 8.02               1200 $ARGS
i1 0 1 8.03               1000 $ARGS
This will get expanded before sorting into:
i1 0 1 8.00               1000 1.01 2.33 138
i1 0 1 8.01               1500 1.01 2.33 138
i1 0 1 8.02               1200 1.01 2.33 138
i1 0 1 8.03               1000 1.01 2.33 138
```

This can save typing, and is makes revisions easier. If there were two sets of p-fields one could have a second macro (there is no real limit on the number of macros one can define).

```
#define ARGS1 # 1.01 2.33 138#
#define ARGS2 # 1.41 10.33 1.00#
i1 0 1 8.00               1000 $ARGS1
i1 0 1 8.01               1500 $ARGS2
i1 0 1 8.02               1200 $ARGS1
i1 0 1 8.03               1000 $ARGS2
```

Macros with arguments

```
           #define ARG(A) # 2.345   1.03   $A   234.9#
           i1 0 1 8.00 1000 $ARG(2.0)
           i1  + 1 8.01 1200 $ARG(3.0)
which expands to
           i1 0 1 8.00 1000 2.345   1.03   2.0   234.9
           i1  + 1 8.01 1200 2.345   1.03   3.0   234.9
```

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

69.5 Multiple File Score

It is sometimes convenient to have the score in more than one file. This use is supported by the #include facility which is part of the macro system. A line containing the text

```
#include "filename"
```

where the character " can be replaced by any suitable character. For most uses the double quote symbol will probably be the most convenient. The file name can include a full path.

This takes input from the named file until it ends, when input reverts to the previous input. There is currently a limit of 20 on the depth of included files and macros.

A suggested use of #include would be to define a set of macros which are part of the composer's style. It could also be used to provide repeated sections.

```
s
#include "section1"
;; Repeat that
s
#include "section1"
```

Alternative methods of doing repeats, use the **r**, **m**, and **n statements**.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

69.6 Evaluation of Expressions

In earlier versions of Csound the numbers presented in a score were used as given. There are occasions when some simple evaluation would be easier. This need is increased when there are macros. To assist in this area the syntax of an arithmetic expressions within square brackets [ ] has been introduced. Expressions built from the operations +, -, *, /, %, and ^ are allowed, together with grouping with ( ). The expressions can include numbers, and naturally macros whose values are numeric or arithmetic strings. All calculations are made in floating point numbers. Note that unary minus is not yet supported.

New in Csound version 3.56 are @*x* (next power-of-two greater than or equal to *x*) and @@*x* (next power-of-two-plus-one greater than or equal to *x*).

EXAMPLE

```
r3          CNT

i1          0               [0.3*$CNT.]
i1          +               [($CNT./3)+0.2]

e
```

As the three copies of the section have the macro $CNT. with the different values of 1, 2 and 3, this expands to

```
s
i1          0               0.3
i1          0.3             0.533333
s
i1          0               0.6
i1          0.6             0.866667
s
i1          0               0.9
i1          0.9             1.2
e
```

This is an extreme form, but the evaluation system can be used to ensure that repeated sections are subtly different.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

69.7 f Statement (or Function Table Statement)

```
f          p1  p2  p3  p4 ...
```

DESCRIPTION

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

P FIELDS

```
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 **envplx, 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**)

69.8 i Statement (Instrument or Note Statement)

i p1 p2 p3 p4 ...

DESCRIPTION

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.

P FIELDS

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**.

It is possible to have multiple instances (usually, but not necessarily, notes of different pitches) of the same instrument, held simultaneously, via negative p3 values. The instrument can then be fed new parameters from the score. This is useful for avoiding long hard-coded **linseg**s, and can be accomplished by adding a decimal part to the instrument number.

For example, to hold three copies of instrument 10 in a simple chord:

```
i10.1    0    -1    7.00
i10.2    0    -1    7.04
i10.3    0    -1    7.07
```

Subsequent **i** statements can refer to the same sounding note instances, and if the instrument definition is done properly, the new p-fields can be used to alter the character of the notes in progress. For example, to bend the previous chord up an octave and release it:

```
i10.1    1    1    8.00
i10.2    1    1    8.04
i10.3    1    1    8.07
```

The instrument definition has to take this into account, however, especially if clicks are to be avoided (see the example below).

Note that the decimal instrument number notation cannot be used in conjunction with real-time MIDI. In this case, the instrument would be monophonic while a note was held.

Notes being tied to previous instances of the same instrument, should skip most initialization by means of **tigoto**, except for the values entered in score. For example, all table reading opcodes in the instrument, should usually be skipped, as they store their phase internally. If this is suddenly changed, there will be audible clicks in the output.

Note that many opcodes (such as **delay** and **reverb**) are prepared for optional initialization. To use this feature, the **tival** flag is suitable. Therefore, they need not be hidden by a **tigoto** jump.

Beginning with Csound version 3.53, strings are recognized in p- fields for opcodes that accept them (**convolve**, **adsyn**, **diskin**, etc.). There may be only one string per score line.

EXAMPLE

Here is an instrument which can find out whether it is tied to a previous note (tival returns 1), and whether it is held (negative p3). Attack and release are handled accordingly:

```
            instr 10

icps        init        cpspch(p4)    ;Get target pitch from score event
iportime    init        abs(p3)/7     ; Portamento time dep on note length
iamp0       init        p5            ; Set default amps
iamp1       init        p5
iamp2       init        p5
```

```
  itie       tival                        ; Check if this note is tied,
  if itie    ==  1 igoto    nofadein      ; if not fade in
  iamp0      init           0

nofadein:
  if p3      <   0 igoto nofadeout        ; Check if this note is held,
                                          ; if not fade out
  iamp2      init           0

nofadeout:
        ; Now do amp from the set values:
  kamp       linseg         iamp0, .03, iamp1, abs(p3)-.03, iamp2

        ; Skip rest of initialization on tied note:
             tigoto         tieskip

  kcps       init           icps         ; Init pitch for untied note
  kcps       port           icps, iportime, icps  ; Drift towards target pitch


  kpw        oscil          .4, rnd(1), 1, rnd(.7)       ; A simple triangle-saw oscil
  ar         vco            kamp, kcps, 3, kpw+.5, 1, 1/icps

  ; (Used in testing - one may set ipch to cpspch(p4+2)
  ;       and view output spectrum)
  ;       ar oscil kamp, kcps, 1

             out            ar

tieskip:                                  ; Skip some initialization on tied note

endin
```

A simple score using three instances of the above instrument:

```
  f1 0 8192 10 1                    ; Sine

  i10.1    0    -1    7.00    10000
  i10.2    0    -1    7.04
  i10.3    0    -1    7.07
  i10.1    1    -1    8.00
  i10.2    1    -1    8.04
  i10.3    1    -1    8.07
  i10.1    2     1    7.11
  i10.2    2     1    8.04
  i10.3    2     1    8.07
  e
```

Additional text (Csound version 4.0) explaining tied notes, edited by Rasmus Ekman from a note by David Kirsh, posted to the Csound mailing list. Example instrument by Rasmus Ekman.

69.9 a Statement (or Advance Statement)

a p1 p2 p3

DESCRIPTION

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

P FIELDS

```
p1   Carries no meaning. Usually zero.
p2   Action time, in beats, at which advance is to begin.
p3   Number of beats to advance without producing sound.
p4   |
p5   |   These carry no meaning.
p6   |
.
.
```

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, number of beats, 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.

69.10 t Statement (Tempo Statement)

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

DESCRIPTION

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

P FIELDS

```
p1       Must be zero.
p2       Initial tempo on beats per minute.
p3, p5, p7,... Times in beats per minute (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 accelerando 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.

69.11 b Statement

b p1

DESCRIPTION

This statement resets the clock for subsequent **i statements**.

P FIELDS

p1 Specifies how the clock is to be set.

SPECIAL CONSIDERATIONS

p1 is the number of beats by which p2 values of subsequent **i statements** are modified. If p1 is positive, the clock is reset forward, and subsequent notes appear later, the number of beats specified by p1 being added to the note's p2. If p1 is negative, the clock is reset backward, and subsequent notes appear earlier, the number of beats specified by p1 being subtracted from the note's p2. There is no cumulative affect. The clock is reset with each **b statement**. If p1 = 0, the clock is returned to its original position, and subsequent notes appear at their specified p2.

EXAMPLE

```
i1   0   2
i1   10  888

b 5                              ; set the clock "forward"
i2   1   1   440                 ; start time = 6
i2   2   1   480                 ; start time = 7

b -1                             ; set the clock back
i3   3   2   3.1415              ; start time = 2
i3   5.5 1   1.1111              ; start time = 4.5

b 0                              ; reset clock to normal
i4   10  200 7                   ; start time = 10
```

Explanation suggested and example provided by Paul Winkler. (Csound version 4.0)

69.12 v Statement

v p1

DESCRIPTION

The **v statement** provides for locally variable time warping of score events.

P FIELDS

p1 Time warp factor (must be positive).

SPECIAL CONSIDERATIONS

The **v statement** takes effect with the following **i statement**, and remains in effect until the next **v**, **s**, or **e statement**.

EXAMPLES

The value of p1 is used as a multiplier for the start times (p2) of subsequent **i statements**.

```
i1        0 1           ;note1
v2
i1        1 1           ;note2
```

In this example, the second note occurs two beats after the first note, and is twice as long.

Although the **v statement** is similar to the **t statement**, the **v statement** is local in operation. That is, **v** affects only the following notes, and its effect may be cancelled or changed by another **v statement**.

Carried values (see Section 14.1.1) are unaffected by the **v statement**.

```
i1        0 1           ;note1
v2
i1        1 .           ;note2
i1        2 .           ;note3
v1
i1        3 .           ;note4
i1        4 .           ;note5
e
```

In this example, note2 and note4 occur simultaneously, while note3 actually occurs before note2, that is, at its original place. Durations are unaffected.

```
i1        0 1
v2
i.        + .
i.        . .
```

In this example, the **v statement** has no effect.

69.13 s Statement

s anything

DESCRIPTION

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

P FIELDS

All p-fields 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 **f0** statement.

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

Note: 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.

69.14 e Statement

e anything

DESCRIPTION

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

P FIELDS

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.

69.15 r Statement (Repeat Statement)

r p1 p2

DESCRIPTION

Starts a repeated section, which lasts until the next **s**, **r** or **e statement**.

P FIELDS

p1 Number of times to repeat the section.
p2 Macro(name) to advance with each repetition (optional).

SPECIAL CONSIDERATIONS

In order that the sections may be more flexible than simple editing, the macro named p2 is given the value of 1 for the first time through the section, 2 for the second, and 3 for the third. This can be used to change p-field parameters, or ignored.

WARNING: Because of serious problems of interaction with macro expansion, sections must start and end in the same file, and not in a macro.

EXAMPLE

In the following example, the section is repeated 3 times. The macro NN is used and advanced with each repetition.

```
r3        NN            ;start of repeated section – use macro NN
          some code
                        .
                        .
                        .
s                       ;end repeat - go back to previous r if repetitions < 3
```

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

69.16 m Statement (Mark Statement)

m p1

DESCRIPTION

Sets a named mark in the score, which can be referenced by an **n statement**.

P FIELDS

p1 Name of mark.

SPECIAL CONSIDERATIONS

This can be helpful in setting a up verse and chorus structure in the score. Names may contain letters and numerals.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

69.17 n Statement

n p1

DESCRIPTION

Repeats a section from the referenced **m statement**.

P FIELDS

p1 Name of mark to repeat.

SPECIAL CONSIDERATIONS

This can be helpful in setting a up verse and chorus structure in the score. Names may contain letters and numerals.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
April, 1998 (New in Csound version 3.48)

70 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.

70.1 GEN01

f # time size 1 filcod skiptime format channel

DESCRIPTION

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

INITIALIZATION

size – number of points in the table. Must be a power of 2 or a power-of-2 plus 1 (see **f statement**), with the one exception: allocation of table size can be deferred by setting this parameter to 0. See Notes, below. The maximum table size is 16777216 (2^{24}) points. Reading stops at end-of-file or when the table is full. Table locations not filled will contain zeros.

filcod – integer or character-string denoting the source soundfile name. An integer denotes the file **soundin**.*filcod* ; a character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a 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.

channel – channel number to read in. 0 denotes read all channels. An AIFF source can be mono or stereo.

format – specifies the audio data-file format:

- 1 - 8-bit signed character
- 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 *format* = 0 the sample format is taken from the soundfile header, or by default from the Csound **-o** command flag.

NOTES

If the source soundfile is of type AIFF, allocation of table size can be deferred by setting *size* to 0. The size allocated is then the number of points (or samples) in the file, which is probably not a power-of-2. In this case, the table generated is usable only by **loscil** Using the form "@*N*" for *size*, where *N* = the number of samples in the sound file, will give the lowest power of 2 greater than or equal to *N*. Using the form "@@*N*", adds one to that number, giving a power-of-2 plus 1 sized table.

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
f   2  0  0  -1  “trumpet  A#5”  0  4
```

The tables are filled from 2 files, "soundin.23" and "trumpet A#5", expected in SSDIR or SFDIR. The first table is pre-allocated; the second is allocated dynamically, and its rescaling is inhibited. .

70.2 GEN02

f # time size 2 v1 v2 v3 . . .

DESCRIPTION

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

INITIALIZATION

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.

70.3 GEN03

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

DESCRIPTION

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

INITIALIZATION

size – number of points in the table. Must be a power of 2 or a power-of-2 plus 1 ( ).

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

c0 + *c1x* + *c2x2* + . . . + *cnxn*

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.

70.4 GEN04

f # time size 4 source# sourcemode

DESCRIPTION

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

INITIALIZATION

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.

70.5 GEN05, GEN07

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

DESCRIPTION

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

INITIALIZATION

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.

70.6 GEN06

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

DESCRIPTION

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

INITIALIZATION

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.

70.7 GEN08

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

DESCRIPTION

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

INITIALIZATION

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.

70.8 GEN09, GEN10, 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  .   .   .
```

DESCRIPTION

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**.

INITIALIZATION

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.

70.9 GEN11

```
f  #    time    size    11    nh    [lh    [r]]
```

DESCRIPTION

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

INITIALIZATION

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.

70.10 GEN12

```
f  #    time    size    -12    xint
```

DESCRIPTION

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

INITIALIZATION

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 oscils, 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.

70.11 GEN13, GEN14

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

DESCRIPTION

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 pre-definable spectrum.

INITIALIZATION

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.

70.12 GEN15

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

DESCRIPTION

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

INITIALIZATION

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, labeled **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.

70.13 GEN16

```
f  #    time    size    15    beg    dur    type    end
```

DESCRIPTION

Creates a table from *beg* value to *end* value of *dur* steps.

INITIALIZATION

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.

beg – starting value

dur – number of segments

type – if 0, a straight line is produced. If non-zero, then **GEN16** creates the following curve, for *dur* steps:

```
beg+(end-beg)*(1-exp(i*type/(dur-1)))/(1-exp(type))
```

end – value after *dur* segments

NOTES

If *type* > 0, there is a slowly rising, fast decaying (convex) curve, while if *type* < 0, the curve is fast rising, slowly decaying (concave). See also **transeg**.

AUTHOR

John ffitch
University of Bath, Codemist. Ltd.
Bath, UK
October, 2000
New in Csound version 4.09

70.14 GEN17

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

DESCRIPTION

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

INITIALIZATION

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.

NOTE

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.

70.15 GEN20

```
f  #     time     size     20     window     max     [opt]
```

DESCRIPTION

This subroutine generates functions of different windows. These windows are usually used for spectrum analysis or for grain envelopes.

INITIALIZATION

size – number of points in the table. Must be a power of 2 (+ 1).

window – Type of window to generate.

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

max – For negative p4 this will be the absolute value at window peak point. If p4 is positive or p4 is negative and p6 is missing the table will be post-rescaled to a maximum value of 1.

opt – Optional argument required by the Kaiser window.

EXAMPLES

```
f       1       0       1024    20      5
```

This creates a function which contains a 4 – term Blackman – Harris window with maximum value of 1.

```
f       1       0       1024    -20     2       456
```

This creates a function that contains a Hanning window with a maximum value of 456.

```
f       1       0       1024    -20     1
```

This creates a function that contains a Hamming window with a maximum value of 1.

```
f       1       0       1024    20      7       1       2
```

This creates a function that contains a Kaiser window with a maximum value of 1. The extra argument specifies how "open" the window is, for example a value of 0 results in a rectangular window and a value of 10 in a Hamming like window.

For diagrams, see Appendix. Section 76.4.

AUTHORS

Paris Smaragdis
MIT, Cambridge
1995

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
New in Csound version 3.2

70.16 GEN21

```
f  #     time    size     21      type      level     [arg1 [arg2]]
```

DESCRIPTION

This generates tables of different random distributions. (See also **x-class noise generators**.)

time and *size* are the usual GEN function arguments. *level* defines the amplitude. Note that **GEN21** is not self-normalizing as are most other GEN functions. *type* defines the distribution to be used as follows:

- 1 = Uniform (positive numbers only)
- 2 = Linear (positive numbers only)
- 3 = Triangle (positive and negative numbers)
- 4 = Exponential (positive numbers only)
- 5 = Biexponential (positive and negative numbers)
- 6 = Gaussian (positive and negative numbers)
- 7 = Cauchy (positive and negative numbers)
- 8 = Positive Cauchy (positive numbers only)
- 9 = Beta (positive numbers only)
- 10 = Weibull (positive numbers only)
- 11 = Poisson (positive numbers only)

Of all these cases only 9 (Beta) and 10 (Weibull) need extra arguments. Beta needs two arguments and Weibull one.

EXAMPLES

```
f1 0 1024 21 1            ; Uniform (white noise)
f1 0 1024 21 6            ; Gaussian
f1 0 1024 21 9 1 1 2      ; Beta (note that level precedes arguments)
f1 0 1024 21 10 1 2       ; Weibull
```

All of the above additions were designed by the author between May and December 1994, under the supervision of Dr. Richard Boulanger.

AUTHORS

Paris Smaragdis
MIT, Cambridge
1995

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
New in Csound version 3.2

70.17 GEN23

```
f  #     time    size     -23       "filename.txt"
```

DESCRIPTION

This subroutine reads numeric values from an external ASCII file

INITIALIZATION

"filename.txt" – numeric values contained in "filename.txt" (which indicates the complete pathname of the character file to be read), can be separated by spaces, tabs, newline characters or commas. Also, words that contains non-numeric characters can be used as comments since they are ignored.

size – number of points in the table. Must be a power of 2 , power of 2 + 1, or zero. If *size* = 0, table size is determined by the number of numeric values in *filename.txt*. (New in Csound version 3.57)

NOTE

All characters following ';' (comment) are ignored until next line (numbers too).

AUTHOR

Gabriel Maldonado
Italy
February, 1998
New in Csound version 3.47

70.18 GEN25, GEN27

```
f  #    time    size   25   x1  y1  x2  y2  x3  .  .  .
f  #    time    size   27   x1  y1  x2  y2  x3  .  .  .
```

DESCRIPTION

These subroutines are used to construct functions from segments of exponential curves (**GEN25)** or straight lines (**GEN27**) in breakpoint fashion.

INITIALIZATION

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

x1, x2, x3, etc. – locations in table at which to attain the following y value. Must be in increasing order. If the last value is less than size, then the rest will be set to zero. Should not be negative but can be zero.

y1, y2, y3,, etc. – Breakpoint values attained at the location specified by the preceding x value. For **GEN25** these must be non-zero and must be alike in sign. No such restrictions exist for **GEN27**.

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.

EXAMPLE

```
f 1 0 257 27 0 0 100 1 200 -1 256 0
```

This describes a function which begins at 0, rises to 1 at the 100th table location, falls to -1, by the 200th location, and returns to 0 by the end of the table. The interpolation is linear.

AUTHOR

John ffitch
University of Bath/Codemist Ltd.
Bath, UK
New in Csound version 3.49

70.19 GEN28

```
f  #    time    size    28    ifilcod
```

DESCRIPTION

This function generator reads a text file which contains sets of three values representing the xy coordinates and a time-tag for when the signal should be placed at that location, allowing the user to define a time-tagged trajectory. The file format is in the form:

```
time1    X1      Y1
time2    X2      Y2
time3    X3      Y3
```

The configuration of the XY coordinates in space places the signal in the following way:

- a1 is -1, 1
- a2 is 1, 1
- a3 is -1, -1
- a4 is 1, -1.

This assumes a loudspeaker set up as a1 is left front, a2 is right front, a3 is left back, a4 is right back. Values greater than 1 will result in sounds being attenuated as if in the distance. **GEN28** creates values to 10 milliseconds of resolution.

INITIALIZATION

size – number of points in the table. Must be 0. **GEN28** takes 0 as the size and automatically allocates memory.

ifilcod – character-string denoting the source soundfile name. A character-string (in double quotes, spaces permitted) gives the filename itself, optionally a full pathname. If not a full path, the named file is sought in the current directory.

EXAMPLE

```
f1 0 0 28 “move”
```

The file "move” should look like:

```
0       -1       1
1        1       1
2        4       4
2.1     -4      -4
3        10     -10
5       -40      0
```

Since **GEN28** creates values to 10 milliseconds of resolution, there will be 500 values created by interpolating X1 to X2 to X3 and so on, and Y1 to Y2 to Y3 and so on, over the appropriate number of values that are stored in the function table. The sound will begin in the left front, over 1 second it will move to the right front, over another second it move further into the distance but still in the left front, then in just 1/10th of a second it moves to the left rear, a bit distant. Finally over the last .9 seconds the sound will move to the right rear, moderately distant, and it comes to rest between the two left channels (due west!), quite distant.

AUTHOR

Richard Karpen
Seattle, Wash
1998 (New in Csound version 3.48)

---

71 THE CSOUND COMMAND

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

71.1 Order of Precedence

With some recent additions to Csound, there are now three places (and in some cases four) where options for Csound performance may be set. They are processed in the following order:

1. Csound's own defaults
2. `.csoundrc` file
3. Csound command line
4. **<CsOptions>** tag in a `.csd` file
5. Orchestra header (for **sr**, **kr**, **ksmps**, **nchnls)**

The last assignment of an option will override any earlier ones.

71.2 Generic Flags

These are generic Csound command flags. Various platform implementations may not react the same way to different flags!

The format of a command is:

```
csound [-flags] orchname scorename
```

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

```
-U unam run utility program unam
-C      use Cscore processing of scorefile
-I      i-time only orch run
-n      no sound onto disk
-i fnam sound input filename
-o fnam sound output filename
-b N    sample frames (or -kprds) per software sound I/O buffer
-B N    samples per hardware sound I/O buffer
-A      create an AIFF format output soundfile
-W      create a WAV format output soundfile
-J      create an IRCAM format output soundfile
-h      no header on output soundfile
```

```
-c      8-bit signed_char sound samples
-a      alaw sound samples
-8      8-bit unsigned_char sound samples
-u      ulaw sound samples
-s      short_int sound samples
-l      long_int sound samples
-f      float sound samples
-r N    orchestra srate override
-k N    orchestra krate override
-v      verbose orch translation
-m N    TTY message level. Sum of: 1=note amps, 2=out-of-range msg,
            4=warnings
-d      suppress all displays
-g      suppress graphics, use ASCII displays
-G      suppress graphics, use Postscript displays
-S      score is in Scot format
-x fnam extract from score.srt using extract file 'fnam'
-t N    use uninterpreted beats of the score, initially at tempo N
-L dnam read Line-oriented real-time score events from device
            'dnam'
-M dnam read MIDI real-time events from device 'dnam'
-F fnam read MIDI file event stream from file 'fnam'
-P N    MIDI sustain pedal threshold (N = 0-128)
-R      continually rewrite header while writing soundfile
            (WAV/AIFF)
-H/H1   generates a rotating line progress report
-H2     generates a . every time a buffer is written
-H3     reports the size of the output in seconds. In Windows,
            writes the information to the window title bar.
-H4     sounds a bell for every buffer of the output written
-N      notify (ring the bell) when score or MIDI track is done
-T      terminate the performance when MIDI track is done
-D      defer GEN01 soundfile loads until performance time
-z      List opcodes in this version
-z1     List opcodes with arguments in this version
-- fnam Log all text output to file fnam
-j fnam derive console messages from database fnam
-Z      Switch on dithering of audio conversion from internal
            floating point to 32, 16 and 8 bit formats. (New in
            Csound version 4.05)
-K num  Switch off peak chunks.
```

71.3 PC Windows Specific flags

```
-j num  set the number of console text rows (default 25)
-J num  set the number of console text columns (default 80)
-q num  WAVE OUT device id number (use only if more than one WAVE
            device is  installed)
-p num  number of WAVE OUT buffers (default 4; max. 40)
-O      suppresses all console text output for better real-time
            performance
-e      allows any sample rate (use only with WAVE cards supporting
            this   feature)
```

```
-y      doesn't wait for keypress on exit
-E      allows graphic display for WCSHELL by Riccardo Bianchini
-Q num  enable MIDI OUT. num (optional) = MIDI OUT port device id
            number
-Y      suppresses real-time WAVE OUT for better MIDI OUT timing
            performance
-*      yields control to the system until audio output buffer is
            full
```

71.4 Macintosh Specific Flags

```
-q sampdir  set the directory for finding samples
-Q analdir  set the directory for finding analyses
-X snddir   set the directory for saving sound files
-V num      set screen buffer size
-E num      set number of graphs saved
-p          play on finishing
-e num      set rescaling factor
-w          set recording of MIDI data
-y num      set rate for progress display
-Y num      set rate for profile display
```

71.5 Description

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

```
csound –nm3 orchname –Sxxfilename scorename
csound –n –m 3 orchname –x xfilename –S scorename
```

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

```
csound –s –otest –b1024 –B1024 –m7 –P128 orchname scorename
```

where orchname is a file containing Csound orchestra code, and scorename is a file of score data in standard numeric score format, optionally presorted and time-warped. If scorename is *omitted*, there are two default options:

- if real-time input is expected (-L, -M or -F), a dummy score file is substituted consisting of the single statement 'f 0 3600' (i.e. listen for RT input for one hour)
- else Csound uses the previously processed *score.srt* in the current directory.

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

Csound -U

Invoke Utility Preprocessing programs: sndinfo, hetro, lpanal, pvanal, cvanal, and pvlook.

Csound -I

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

Csound -n

No sound. Do all processing, but bypass writing of sound to disk. This flag does not change the execution in any other way.

Csound -i isfname

Input soundfile name. If not a full pathname, the file will be sought first in the current directory, then in that given by the environment variable SSDIR (if defined), then by SFDIR. The name *stdin* will cause audio to be read from standard input. If RTAUDIO is enabled, the name *devaudio* will request sound from the host audio input device.

Csound -o osfname

Output soundfile name. If not a full pathname, the soundfile will be placed in the directory given by the environment variable SFDIR (if defined), else in the current directory. The name *stdout* will cause audio to be written to standard output. If no name is given, the default name will be *test*. If RTAUDIO is enabled, the name *devaudio* will send to the host audio output device.

Csound -b Numb

Number of audio sample-frames per sound i/o *software* buffer. Large is efficient, but small will reduce audio I/O delay. The default is 1024. In real-time performance, Csound waits on audio I/O on *Numb* boundaries. It also processes audio (and polls for other input like MIDI) on orchestra *ksmps* boundaries. The two can be made synchronous. For convenience, if Numb = -N (is negative) the effective value is *ksmps * N* (audio synchronous with k-period boundaries). With N small (e.g. 1) polling is then frequent and also locked to fixed DAC sample boundaries.

Csound -B Numb

Number of audio sample-frames held in the DAC *hardware* buffer. This is a threshold on which *software* audio I/O (above) will wait before returning. A small number reduces audio I/O delay; but the value is often hardware limited, and small values will risk data lattes. The default is 1024.

Csound -h

No header on output soundfile. Don't write a file header, just binary samples.
Csound {-c, -a, -u, -s, -l, -f}
Audio sample format of the output soundfile. One of:

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

Csound -A

Write an AIFF output soundfile. Restricts the above formats to c, s, l, or f (AIFC).

Csound –W

Write a .wav output soundfile.

Csound –J

Write an IRCAM output soundfile.

Csound -v

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

Csound -m Numb

Message level for standard (terminal) output. Takes the *sum* of 3 print control flags, turned on by the following values:

- 1 = note amplitude messages
- 2 = samples out of range message
- 4 = warning messages. The default value is *m7* (all messages on).

Csound -d

Suppress all displays.

Csound -g

Recast graphic displays into ASCII characters, suitable for any terminal.

Csound -S

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

Csound -x xfile

Extract a portion of the sorted score score.srt, according to xfile (see Extract).

Csound -t Numb

Use the uninterpreted beats of *score.srt* for this performance, and set the initial tempo at *Numb* beats per minute. When this flag is set, the tempo of score performance is also controllable from within the orchestra. The flag `-t0` will prevent Csound from deleting the sorted score file, `score.srt,` upon exit.

Csound -L devname

Read Line-oriented real-time score events from device *devname*. The name *stdin* will permit score events to be typed at your terminal, or piped from another process. Each line-event is terminated by a carriage-return. Events are coded just like those in a *standard numeric score*, except that an event with p2=0 will be performed immediately, and an event with p2=T will be performed T seconds after arrival. Events can arrive at any time, and in any order. The score **carry** feature is legal here, as are held notes (p3 negative) and string arguments, but ramps and **pp** or **np** references are not.

Csound -M devname

Read MIDI events from device *devname*.

Csound -F mfname

Read MIDI events from MIDI file *mfname*.

Csound -P Numb

Set MIDI sustain pedal threshold (0 – 128). The official switch value of 64 is normally too low, and is more realistic above 100. The default value of 128 will block all pedal info.

Csound -N

Notify (ring the bell) when score or MIDI track is done.

Csound -T

Terminate the performance when MIDI track is done.

Csound –j fnam

Use database *fnam* for messages to print to console during performance. (New in version 3.55)

Csound -K num

Switch off peak chunks. New in Csound version 4.09.

PC/WINDOWS-SPECIFIC FLAGS

Csound -q num

WAVE OUT device id number (optional, use only if WAVE OUT devices are more than one)

Csound -p num

number of WAVE OUT buffers (optional; default=4, maximum=40). -b (buffer length) and -p flags are related each other. Finding the optimum values for "-b" and "-p" flags requires some experimentation: more buffer length means more latency delay but also more safety from dropouts and sound interruptions (flag "-B" is now obsolete, don't use it). You now can drastically reduce buffer length and delay by using -e flag and 'rounded' sr and kr. Note that sometimes a smaller buffer length can handle sound flow better than a larger. Only experiments can lead you toward optimal '-b' values. -b and -p flags value can now be reduced considerably by using "rounded" ar and kr values (for example ar=32000 and kr=320; ar=40000 and kr=400 and so on) together with -e flag. This feature has been tested only with a SB16 ASP and with an AWE32 card. Support by other cards is unknown. Reducing "-b" and "-p" flag values means reducing latency delay and so a more interactive real-time playing.

Csound -j num

console virtual text rows number.

Csound -J num

console virtual text columns number.

Csound -O (uppercase letter)

suppresses all printf for better real-time performance. This switch is better than '-m0' because '-m0' still leaves some message output to the console. Use both switches together for maximum performance speed.

Csound -e

allows arbitrary output sample rate (for cards that support this feature).

Csound -y

doesn't wait for keypress on exit.

Csound -E

graphic display for WCSHELL by Riccardo Bianchini.

Csound -Q num

enables MIDI OUT operations and optionally chooses device id num (if num argument is present). This flag allows parallel MIDI OUT and DAC performance. Unfortunately the real-time timing implemented in Csound is completely managed by DAC buffer sample flow. So MIDI OUT operations can present some time irregularities. These irregularities can be fully eliminated when suppressing DAC operations themselves (see -Y flag).

Csound -Y

disables WAVE OUT (for better MIDI OUT timing performances). This enhances timing of MIDI out operations when used in conjunction with "-Q" flag. Low k-rates (max. kr=1000) are recommended for use with the "-Y" flag. As in Win95 maximum timer resolution is 1/1000 of second, unpredictable results can occur when using it at k-rates greater than 1000. Also it is very important to set only kr values in which the following division: 1000/kr produces integer results (some example: kr = 10; 20; 50; 100; 125; 200; 250 etc.) because Win95 timer only handles integer periods in milliseconds.
If you use a kr value that produces a non-integer result in the above formula, Csound seems to run normally but times will be not reliable. A value of kr=200 works well on most computers. Maybe with slower computers a lower value works better. Experiment! Values greater than 200 increase the overhead affecting the entire system, and do not give a notable precision improvement. A time resolution of 1/200 of sec is precise enough for almost all MIDI applications. The sr/kr/ksmps ratio must be respected, or an error message will stop the performance, even if sr value is meaningless when using "-Y" flag.

Csound -*

compels Csound to yield control to system until audio output buffer content passes a certain threshold. Below this threshold Csound continues processing, while over this threshold Csound yields control to Windows. This gives a big enhancement in multitasking processes. Enabling this option reduces polyphony a bit when using short buffer space. So the user should increase the number ('-p' flag) and the length ('-b' flag) of buffers when setting '-*' flag. Experiment to find best values. Do not use this flag when time response to gestural actions is critical.

This page intentionally left blank.

72 UNIFIED FILE FORMAT FOR ORCHESTRAS AND SCORES

72.1 Description

The Unified File Format, introduced in Csound version 3.50, enables the orchestra and score files, as well as command line flags, to be combined in one file. The file has the extension `.csd`. This format was originally introduced by Michael Gogins in AXCsound.

The file is a structured data file which uses markup language, similar to any SGML such as HTML. Start tags (<*tag*>) and end tags (</*tag*>) are used to delimit the various elements. The file is saved as a text file.

72.2 Structured Data File Format

MANDATORY ELEMENTS

The Csound Element is used to alert the csound compiler to the **.csd** format. The file must begin with the start tag **<CsoundSynthesizer>.** The last line of the file must be the end tag **</CsoundSynthesizer>.** The remaining elements are defined below.

Options

Csound command line flags are put in the Options Element. This section is delimited by the start tag **<CsOptions>** and the end tag **</CsOptions>** Lines beginning with **#** or **;** are treated as comments. For precedence of flags, options, and header statements, see Section 67.1.

Instruments (Orchestra)

The instrument definitions (orchestra) are put into the Instruments Element. The statements and syntax in this section are identical to the Csound orchestra file, and have the same requirements, including the header statements (**sr**, **kr**, etc.) This Instruments Element is delimited with the start tag **<CsInstruments>** and the end tag **</CsInstruments>.**

Score

Csound score statements are put in the Score Element. The statements and syntax in this section are identical to the Csound score file, and have the same requirements. The Score Element is delimited by the start tag **<CsScore>** and the end tag **</CsScore>.**

OPTIONAL ELEMENTS

Included Base64 Files

Base64 encoded MIDI files may be included with the tag **<CsMidifileB filename**=*filename***>**, where *filename* is the name of the file containing the MIDI information. There is no matching end tag. New in Csound version 4.07.

Base64 encoded sample files may be included with the tag **<CsSampleB filename=***filename***>**, where *filename* is the name of the file containing the sample. There is no matching end tag. New in Csound version 4.07.

Version Blocking

Versions of Csound may blocked by placing one of the following statements between the start tag **<CsVersion>** and the end tag **</CsVersion>**:

```
Before #.#
```

or

```
After #.#
```

where #.# is the requested Csound version number. The second statement may be written simply as:

```
#.#
```

See example below. New in Csound version 4.09.

72.3 Example

Below is a sample file, `test.csd`, which renders a .wav file at 44.1 kHz sample rate containing one second of a 1 kHz sine wave. Displays are suppressed. test.csd was created from two files, `tone.orc` and `tone.sco`, with the addition of command line flags.

```
<CsoundSynthesizer>
  ; test.csd - a Csound structured data file

<CsOptions>
  -W -d -o tone.wav
</CsOptions>

<CsVersion>   ;optional section
  Before 4.10  ;these two statements check for
  After 4.08   ;   Csound version 4.09
</CsVersion>

<CsInstruments>
  ; originally tone.orc
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
            instr 1
a1          oscil          p4, p5, 1 ; simple oscillator
            out            a1
            endin
</CsInstruments>

<CsScore>
  ; originally tone.sco
  f1 0 8192 10 1
  i1 0 1 20000 1000 ;play one second of one kHz tone
  e
</CsScore>
</CsoundSynthesizer>
```

72.4 Command Line Parameter File

If the file **.csoundrc** exists, it will be used to set the command line parameters. These can be overridden. It uses the same form as a **.csd** file. Lines beginning with **#** or **;** are treated as comments.

This page intentionally left blank.

73 Score File Preprocessing

73.1 The Extract Feature

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

```
instruments 1  2  from  1:27.5  to  2:2
```

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

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

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

```
all instruments, beginning of score, end of score.
```

73.2 Independent Pre-Processing with Scsort

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

```
scot filename
```

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

The command

```
scscort < infile > outfile
```

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

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

```
extract xfile < score.sort > score.extract
```

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

```
scsort < scorefile | extract xfile > score.extract
```

74 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 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 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 files, which is thus a portable format. 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 , only a single channel is read. For stereo or quad files, the default is channel one; alternate channels may be obtained on request.

Author

Dan Ellis
MIT Media Lab
Cambrige, Massachussetts
1990

74.1 sndinfo

DESCRIPTION

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, 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), 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
```

74.2 hetro

DESCRIPTION

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 is a function of memory available.

-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 Hz), 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 (Hz). 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 ..........
```

If the filename passed to hetro has the extension *.sdif*, data will be written in SDIF format as 1TRC frames of additive synthesis data. The utility program **sdif2ads** can be used to convert any SDIF file containing a stream of 1TRC data to the Csound **adsyn** format. New in Csound version 4.08.

74.3 lpanal

DESCRIPTION

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.

-a – [alternate storage] asks lpanal to write a file with filter poles values rather than the usual filter coefficient files. When lpread / lpreson are used with pole files, automatic stabilization is performed and the filter should not get wild. (This is the default in the Windows GUI) – Changed by Marc Resibois.

-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 – hop size (in samples) between frames of analysis. This determines the number of frames per second (srate / hopsize) in the output control file. The analysis framesize is hopsize * 2 samples. 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 Hz) of pitch tracking. -P0 means no pitch tracking.

-Q maxcps – highest frequency (in Hz) 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 -a -p26 -d2.5 -P100 -Q400 audiofile.test lpfil22
```

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. Stabilized (*-a*) output will be placed in "lpfil22" 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.

74.4 pvanal

DESCRIPTION

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.

DIAGNOSTICS

Prints total number of frames, and frames completed on every 20th.

AUTHOR

Dan Ellis
MIT Media Lab
Cambrige, Massachussetts
1990

74.5 cvanal

DESCRIPTION

Impulse Response Fourier Analysis for **convolve** operator

```
Csound -U cvanal [flags] infilename outfilename
```

cvanal converts a soundfile into a single Fourier transform frame. The output file can be used by the **convolve** operator to perform Fast Convolution between an input signal and the original impulse response. Analysis is conditioned by the flags below. A space is optional between the flag and its argument.

-s rate – 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. If omitted, the default is to process all channels. If a value is given, only the selected channel will be processed.

-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.

EXAMPLE

```
cvanal asound cvfile
```

will analyze the soundfile "asound" to produce the file "cvfile" for the use with **convolve**.

To use data that is not already contained in a soundfile, a soundfile converter that accepts text files may be used to create a standard audio file, e.g., the .DAT format for SOX. This is useful for implementing FIR filters.

FILES

The output file has a special **convolve** header, containing details of the source audio file. The analysis data is stored as 'float', in rectangular (real/imaginary) form.

Note: The analysis file is *not* system independent! Ensure that the original impulse recording/data is retained. If/when required, the analysis file can be recreated.

AUTHOR

Greg Sullivan
(Based on algorithm given in 'Elements Of Computer Music', by F. Richard Moore.

74.6 pvlook

DESCRIPTION

View formatted text output of STFT analysis files created with **pvanal**.

```
csound -U pvlook [flags] infilename
```

or

```
pvlook [flags] infilename
```

pvlook reads a file, and frequency and amplitude trajectories for each of the analysis bins, in readable text form. The file is assumed to be an STFT analysis file created by **pvanal**. By default, the entire file is processed.

-*bb* N – begin at analysis bin number N, numbered from 1. Default is 1.

-*eb* N – end at analysis bin number N. Defaults to the highest.

-*bf* N – begin at analysis frame number N, numbered from 1. Defaults to 1.

-*ef* N – end at analysis frame number N. Defaults to the highest.

-*i* – prints values as integers. Defaults to floating point.

EXAMPLE

```
enakis 259% ../csound -U pvlook test.pv
Using csound.txt
Csound Version 3.57 (Aug  3 1999)
util PVLOOK:
; Bins in Analysis: 513
; First Bin Shown: 1
; Number of Bins Shown: 513
; Frames in Analysis: 1184
; First Frame Shown: 1
; Number of Data Frames Shown: 1184

Bin 1 Freqs.0.000 87.891 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

```
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 -87.891 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 87.891 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

```
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000

Bin 1 Amps. 0.180 0.066 0.252 0.248 0.245 0.246 0.246 0.249
0.252 0.251 0.250 0.248 0.244 0.245 0.248 0.250 0.254 0.251
0.248 0.247 0.244 0.246 0.249 0.250 0.253 0.251 0.247 0.246
0.245 0.246 0.250 0.251 0.252 0.250 0.247 0.245 0.246 0.247
0.251 0.252 0.250 0.249 0.246 0.245 0.248 0.249 0.252 0.253
0.249 0.248 0.245 0.245 0.249 0.251 0.252 0.252 0.249 0.246
0.246 0.245 0.249 0.252 0.252 0.251 0.249 0.245 0.246 0.248
0.250 0.253 0.251 0.249 0.247 0.244 0.247 0.249 0.250 0.253
0.251 0.248 0.247 0.245 0.247 0.250 0.252 0.252 0.251 0.247
0.246 0.246 0.247 0.251 0.252 0.251 0.249 0.246 0.245 0.248
0.249 0.252 0.252 0.249 0.248 0.246 0.245 0.249 0.250 0.252
0.252 0.249 0.247 0.246 0.246 0.249 0.252 0.252 0.251 0.248
0.245 0.246 0.247 0.249 0.253 0.251 0.249 0.247 0.245 0.246
0.248 0.250 0.253 0.251 0.248 0.247 0.244 0.246 0.250 0.251
0.252 0.250 0.247 0.246 0.246 0.248 0.251 0.252 0.251 0.250
0.246 0.245 0.247 0.248 0.251 0.252 0.250 0.248 0.246 0.245
0.248 0.249 0.252 0.252 0.248 0.247 0.245 0.245 0.249 0.251
0.251 0.251 0.248 0.246 0.246 0.247 0.250 0.252 0.251 0.250
0.248 0.244 0.246 0.248 0.250 0.253 0.251 0.248 0.247 0.245
0.247 0.249 0.250 0.252 0.250 0.247 0.246 0.245 0.247 0.251
0.252 0.251 0.250 0.246 0.245 0.247 0.248 0.252 0.252 0.249
0.248 0.245 0.245 0.248 0.249 0.251 0.252 0.248 0.247 0.245
0.245 0.249 0.250 0.251 0.251 0.248 0.246 0.245 0.246 0.249
0.252 0.251 0.250 0.247 0.244 0.246 0.247 0.249 0.252 0.251
0.249 0.247 0.244 0.247 0.249 0.250 0.252 0.250 0.247 0.246
0.245 0.247 0.250 0.251 0.251 0.250 0.246 0.245 0.246 0.248
0.251 0.252 0.250 0.249 0.245 0.245 0.247 0.248 0.251 0.252
0.249 0.247 0.245 0.245 0.248 0.250 0.251 0.251 0.247 0.246
0.245 0.245 0.249 0.251 0.251 0.250 0.247 0.245 0.246 0.246
0.249 0.252 0.251 0.249 0.247 0.244 0.247 0.248 0.250 0.252
0.250 0.247 0.246 0.245 0.247 0.250 0.251 0.252 0.249 0.246
0.245 0.245 0.247 0.251 0.251 0.250 0.249 0.246 0.245 0.247
0.248 0.251 0.251 0.249 0.248 0.245 0.245 0.248 0.249 0.251
0.251 0.248 0.246 0.245 0.245 0.249 0.251 0.251 0.251 0.247
0.245 0.245 0.246 0.249 0.251 0.250 0.249 0.247 0.244 0.246
0.248 0.250 0.252 0.250 0.247 0.246 0.245 0.247 0.249 0.250
0.251 0.249 0.246 0.246 0.245 0.247 0.250 0.250 0.250 0.249
0.245 0.245 0.246 0.248 0.251 0.251 0.249 0.248 0.245 0.245
0.247 0.249 0.251 0.251 0.248 0.246 0.245 0.245 0.248 0.250
0.251 0.250 0.247 0.245 0.245 0.246 0.249 0.251 0.250 0.249
0.246 0.244 0.246 0.247 0.250 0.251 0.250 0.248 0.246 0.245
0.247 0.249 0.250 0.251 0.249 0.247 0.246 0.245 0.247 0.250
0.250 0.251 0.248 0.245 0.245 0.246 0.248 0.251 0.251 0.249
0.248 0.245 0.245 0.247 0.249 0.251 0.251 0.248 0.247 0.245
0.245 0.248 0.249 0.250 0.250 0.247 0.246 0.246 0.246 0.249
0.251 0.250 0.250 0.246 0.245 0.246 0.247 0.250 0.251 0.249
0.248 0.246 0.244 0.246 0.248 0.250 0.251 0.249 0.247 0.246
0.245 0.247 0.250 0.250 0.251 0.249 0.245 0.245 0.246 0.248
0.251 0.250 0.250 0.248 0.245 0.245 0.247 0.248 0.251 0.250
0.248 0.247 0.245 0.246 0.248 0.250 0.251 0.250 0.247 0.246
0.245 0.246 0.249 0.251 0.250 0.249 0.246 0.245 0.246 0.247
0.250 0.251 0.250 0.249 0.246 0.244 0.246 0.248 0.250 0.251
0.249 0.247 0.246 0.245 0.247 0.249 0.250 0.251 0.287 0.331
0.178 0.008 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

```
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
0.140 1.265 2.766 3.289 3.296 3.293 3.296 3.296 3.290 3.293
3.292 3.291 3.297 3.295 3.294 3.296 3.291 3.292 3.294 3.291
3.296 3.297 3.292 3.295 3.292 3.290 3.295 3.293 3.294 3.297
3.292 3.293 3.294 3.290 3.295 3.295 3.292 3.296 3.293 3.291
3.294 3.291 3.293 3.297 3.292 3.295 3.294 3.288 3.293 3.293
3.292 3.297 3.294 3.292 3.295 3.290 3.292 3.295 3.292 3.295
3.295 3.290 3.294 3.292 3.292 3.297 3.293 3.293 3.295 3.290
3.292 3.293 3.290 3.296 3.296 3.292 3.295 3.291 3.290 3.294
3.291 3.294 3.296 3.291 3.293 3.293 3.290 3.295 3.294 3.293
3.296 3.291 3.291 3.293 3.290 3.294 3.296 3.292 3.295 3.293
3.288 3.293 3.292 3.292 3.297 3.292 3.293 3.294 3.289 3.292
3.294 3.291 3.296 3.293 3.291 3.294 3.291 3.292 3.296 3.292
3.294 3.295 3.289 3.292 3.292 3.291 3.296 3.294 3.292 3.295
3.290 3.290 3.293 3.291 3.295 3.296 3.291 3.294 3.291 3.289
3.294 3.292 3.293 3.295 3.291 3.292 3.293 3.290 3.294 3.295
3.292 3.294 3.291 3.289 3.293 3.291 3.293 3.296 3.292 3.293
3.293 3.288 3.292 3.293 3.292 3.296 3.293 3.291 3.294 3.289
3.292 3.295 3.291 3.294 3.293 3.289 3.292 3.291 3.290 3.295
3.293 3.292 3.294 3.289 3.291 3.293 3.290 3.295 3.294 3.290
3.293 3.290 3.289 3.294 3.291 3.293 3.295 3.290 3.292 3.292
3.289 3.293 3.293 3.292 3.295 3.291 3.289 3.292 3.290 3.292
3.295 3.291 3.293 3.292 3.288 3.292 3.291 3.291 3.295 3.291
3.291 3.292 3.289 3.291 3.294 3.291 3.294 3.292 3.289 3.292
3.290 3.290 3.295 3.292 3.293 3.294 3.289 3.291 3.292 3.290
3.294 3.293 3.291 3.293 3.289 3.290 3.293 3.291 3.294 3.295
3.290 3.292 3.291 3.289 3.294 3.293 3.292 3.294 3.290 3.290
3.292 3.289 3.293 3.294 3.291 3.293 3.291 3.289 3.292 3.291
3.291 3.295 3.291 3.291 3.292 3.288 3.292 3.293 3.291 3.295
3.292 3.290 3.292 3.289 3.291 3.294 3.291 3.293 3.292 3.288
3.291 3.291 3.290 3.295 3.292 3.291 3.293 3.289 3.290 3.292
3.290 3.294 3.293 3.290 3.292 3.290 3.289 3.293 3.291 3.292
3.294 3.290 3.290 3.291 3.289 3.293 3.293 3.291 3.293 3.290
3.288 3.291 3.290 3.292 3.294 3.290 3.292 3.291 3.288 3.291
3.291 3.291 3.294 3.291 3.290 3.291 3.288 3.291 3.293 3.291
3.293 3.292 3.288 3.291 3.290 3.290 3.294 3.291 3.291 3.292
3.288 3.290 3.291 3.290 3.294 3.293 3.290 3.292 3.289 3.289
3.293 3.290 3.292 3.293 3.289 3.291 3.290 3.289 3.293 3.292
3.291 3.293 3.289 3.289 3.291 3.289 3.292 3.293 3.290 3.292
3.290 3.288 3.292 3.291 3.291 3.294 3.290 3.290 3.291 3.288
3.291 3.292 3.291 3.293 3.291 3.288 3.291 3.289 3.290 3.293
3.290 3.292 3.292 3.288 3.291 3.291 3.290 3.293 3.291 3.290
3.292 3.288 3.289 3.292 3.290 3.292 3.293 3.289 3.291 3.289
3.288 3.293 3.291 3.291 3.292 3.288 3.289 3.290 3.288 3.292
3.293 3.290 3.292 3.289 3.288 3.291 3.290 3.291 3.293 3.289
3.290 3.290 3.287 3.291 3.291 3.290 3.293 3.290 3.288 3.290
3.288 3.290 3.293 3.291 3.292 3.291 3.288 3.290 3.289 3.289
3.293 3.290 3.290 3.291 3.287 3.289 3.291 3.289 3.292 3.291
3.288 3.290 3.288 3.288 3.292 3.290 3.291 3.292 3.288 3.289
3.290 3.288 3.292 3.292 3.290 3.292 3.289 3.288 3.291 3.289
3.291 3.293 3.289 3.291 3.290 3.287 3.291 3.290 3.290 3.293
3.289 3.289 3.290 3.287 3.290 3.292 3.290 3.292 3.290 3.287
3.290 3.289 3.289 3.292 3.290 3.290 3.291 3.287 3.289 3.290
3.289 3.292 3.291 3.289 3.291 3.288
```

etc…

AUTHOR

Richard Karpen
Seattle, Wash
1993 (New in Csound version 3.57)

74.7 sdif2ads

DESCRIPTION

Convert files Sound Description Interchange Format (SDIF) to the format usable by Csound's **adsyn** opcode. As of Csound version 4.10, **sdif2ads** was available only as a standalone program for Windows console and DOS.

```
Csound -U sdif2ads [flags] infilename outfilename
```

-*s*N – apply amplitude scale factor N

-*p*N – keep only the first N partials. Limited to 1024 partials. The source partial track indices are used directly to select internal storage. As these can be arbitrary values, the maximum of 1024 partials may not be realized in all cases.

-*r* – byte-reverse output file data. The byte-reverse option facilitates transfer across platforms, as Csound's **adsyn** file format is not portable.

If the filename passed to **hetro** has the extension .sdif, data will be written in SDIF format as 1TRC frames of additive synthesis data. The utility program **sdif2ads** can be used to convert any SDIF file containing a stream of 1TRC data to the Csound **adsyn** format. **sdif2ads** allows the user to limit the number of partials retained, and to apply an amplitude scaling factor. This is often necessary, as the SDIF specification does not, as of the release of **sdif2ads**, require amplitudes to be within a particular range. **sdif2ads** reports information about the file to the console, including the frequency range.

The main advantages of SDIF over the **adsyn** format, for Csound users, is that SDIF files are fully portable across platforms (data is "big-endian"), and do not have the duration limit of 32.76 seconds imposed by the 16 bit **adsyn** format. This limit is necessarily imposed by **sdif2ads**. Eventually, SDIF reading will be incorporated directly into **adsyn**, thus enabling files of any length (subject to system memory limits) to be analysed and processed.

Users should remember that the SDIF formats are still under development. While the 1TRC format is now fairly well established, it can still change.

For detailed information on the Sound Description Interchange Format, refer to the CNMAT website:

- http://cnmat.CNMAT.Berkeley.EDU/SDIF

Some other SDIF resources (including a viewer) are available via the NC_DREAM website:

- http://www.bath.ac.uk/~masjpf/NCD/dreamhome.html

AUTHOR

Richard Dobson
Somerset, England
August, 2000
New in Csound version 4.08

This page intentionally left blank.

75 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.

75.1 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);          create a blank event with n pfields
     int  n;
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);           create an empty event list with n slots
     int  n;
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);    read next nbeats beats from score input
     float  nbeats;
a = lgetuntil(beatno);   read all events from score input up to beat beatno
     float  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);  extract notes of time-span, creating new events
     float  from, to;
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
```

75.2 Writing a Main 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 Hz.

```
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;                 /* initialize 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;
```

75.3 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.

```
./.htmlinclude "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 Clines, 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.

```
./.htmlinclude "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-modify 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.

75.4 Compiling a Cscore Program

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

```
cscore –U pvanal 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.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/rdscore.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.o –lcsounc –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*'.

This page intentionally left blank.

76 ADDING YOUR OWN CMODULES TO CSOUND

If the existing Csound generators do not suit your needs, you can write your own modules in C and add them to the run-time system. When you invoke Csound on an orchestra and score file, the orchestra is first read by a table-driven translator 'otran' and the instrument blocks converted to coded templates ready for loading into memory by 'oload' on request by the score reader. To use your own C-modules within a standard orchestra you need only add an entry in otran's table and relink Csound with your own code.

The translator, loader, and run-time monitor will treat your module just like any other provided you follow some conventions. You need a *structure* defining the inputs, outputs and workspace, plus some *initialization code* and some *perf-time code*. Let's put an example of these in two new files, **newgen.h** and **newgen.c**:

```
  typedef struct {         /*  newgen.h  -  define a structure     */
        OPDS
        h;                 /* required header  */
        float   *result, *istrt, *incr, *itime, *icontin;  /* addr outarg, inargs
*/
        float   curval, vincr;   /* private dataspace  */
        long    countdown;       /* ditto              */
    }  RMP;

#include "cs.h"          /*  newgen.c -  init and perf code     */
#include "newgen.h"

void rampset(RMP *p)    /* at note initialization:  */
{
        if  (*p-icontin == 0.)
                p-curval = *p-istrt;  /* optionally get new start value */
        p-vincr = *p-incr / esr;      /* set s-rate increment per sec.  */
        p-countdown = *p-itime * esr; /* counter for itime seconds      */
}

void ramp(RMP *p)       /* during note performance:     */
 {
        float *rsltp = p-result;  /* init an output array pointer  */
        int nn = ksmps;            /* array size from orchestra    */
               do {
                *rsltp++ = p-curval;      /* copy current value to output */
                if (--p-countdown = 0)    /* for the first itime seconds, */
                        p-curval += p-vincr;    /* ramp the value  */
        } while (--nn);
         }
```

Now we add this module to the translator table **entry.c**, under the opcode name **rampt**:

```
     #include "newgen.h"
     void rampset(), ramp();

 /*   opcode    dspace    thread    outarg    inargs      isub       ksub     asub
*/
     { "rampt",  S(RMP),  5,        "a",      "iiio",     rampset,   NULL,    ramp  },
```

Finally we relink Csound to include the new module. If your Csound installation has created a libcsound.a, you can do this by typing

```
cc  -o mycsound  newgen.c  entry.c  -lcsound –lX11 –lm
(X11 if included at installation)
```

Else copy **.c, *.h* and Makefile from the Csound sources, add **newgen.o** to the Makefile list OBJS, add **newgen.h** as a dependency for entry.o, and a new dependency **'newgen.o: newgen.h'**, then run '*make Csound*'. If your host is a Macintosh, simply add **newgen.h and newgen.c** to one of the segments in the Csound Project, and invoke the **C** compiler.

The above actions have added a new generator to the Csound language. It is an audio-rate linear ramp function which modifies an input value at a user-defined slope for some period. A ramp can optionally continue from the previous note's last value. The Csound manual entry would look like:

```
ar   rampt     istart,  islope, itime [,  icontin]
```

istart – beginning value of an audio-rate linear ramp. Optionally overridden by a continue flag.

islope – slope of ramp, expressed as the y-interval change per second.

itime – ramp time in seconds, after which the value is held for the remainder of the note.

icontin (optional) – continue flag. If zero, ramping will proceed from input *istart* . If non-zero, ramping will proceed from the last value of the previous note. The default value is zero.

The file *newgen.h* includes a one-line list of output and input parameters. These are the ports through which the new generator will communicate with the other generators in an instrument. Communication is by *address*, not *value*, and this is a list of pointers to floats. There are no restrictions on names, but the input-output argument types are further defined by character strings in entry.c (inargs, outargs). Inarg types are commonly **x**, **a**, **k**, and **I**, in the normal Csound manual conventions; also available are o (optional, defaulting to 0), p (optional, defaulting to 1). Outarg types include **a**, **k**, **I** and **s** (asig or ksig). It is important that all listed argument names be assigned a corresponding argument type in entry.c. Also, I-type args are valid only at initialization time, and other-type args are available only at perf time. Subsequent lines in the RMP structure declare the work space needed to keep the code re-entrant. These enable the module to be used multiple times in multiple instrument copies while preserving all data.

The file **newgen.c** contains two subroutines, each called with a pointer to the uniquely allocated RMP structure and its data. The subroutines can be of three types: note initialization, k-rate signal generation, a-rate signal generation. A module normally requires two of these initialization, and either k-rate or a-rate subroutines which become inserted in various threaded lists of runnable tasks when an instrument is activated. The thread-types appear in entry.c in two forms: *isub*, *ksub* and *asub* names; and a threading index which is the sum of isub=1, ksub=2, asub=4. The code itself may reference global variables defined in **cs.h** and **oload.c**, the most useful of which are:

```
extern  OPARMS  O ;          float   esr
     user-defined sampling rate   float   ekr
     user-defined control rate    float   ensmps
     user-defined ksmps           int     ksmps
     user-defined ksmps           int     nchnls
     user-defined nchnls          int     O.odebug
     command-line –v flag         int     O.msglevel
     command-line –m level        float   pi, twopi    obvious
     constants                    float   tpidsr    twopi / esr float
     sstrcod                      special code for string arguments
```

FUNCTION TABLES

To access stored function tables, special help is available. The newly defined structure should include a pointer

```
FUNC        *ftp;
```

initialized by the statement

```
ftp = ftpfind(p-ifuncno);
```

where float *ifuncno is an I-type input argument containing the ftable number. The stored table is then at ftp-ftable, and other data such as length, phase masks, cps-to-incr converters, are also accessed from this pointer. See the FUNC structure in cs.h, the ftfind() code in fgens.c, and the code for oscset() and koscil() in opcodes2.c.

ADDITIONAL SPACE

Sometimes the space requirement of a module is too large to be part of a structure (upper limit 65535 bytes), or it is dependent on an I-arg value which is not known until initialization. Additional space can be dynamically allocated and properly managed by including the line

```
AUXCH      auxch;
```

in the defined structure (*p), then using the following style of code in the init module:

```
if (p-auxch.auxp == NULL)
     auxalloc(npoints * sizeof(float), &p-auxch);
```

The address of this auxiliary space is kept in a chain of such spaces belonging to this instrument, and is automatically managed while the instrument is being duplicated or garbage-collected during performance. The assignment

```
char *auxp = p-auxch.auxp;
```

will find the allocated space for init-time and perf-time use. See the LINSEG structure in opcodes1.h and the code for lsgset() and klnseg() in opcodes1.c.

FILE SHARING

When accessing an external file often, or doing it from multiple places, it is often efficient to read the entire file into memory. This is accomplished by including the line

```
MEMFIL    *mfp;
```

in the defined structure (*p), then using the following style of code in the init module:

```
if (p-mfp == NULL)
     p-mfp = ldmemfile(filname);
```

where char *filname is a string name of the file requested. The data read will be found between

```
(char *)  p-mfp-beginp; and (char *) p-mfp-endp;
```

Loaded files do not belong to a particular instrument, but are automatically shared for multiple access. See the ADSYN structure in opcodes3.h and the code for adset() and adsyn() in opcodes3.c.

STRING ARGUMENTS

To permit a quoted string input argument (float *ifilnam, say) in our defined structure (*p), assign it the argtype **S** in entry.c, include another member char *strarg in the structure, insert a line

```
TSTRARG( “rampt”,  RMP)  \
```

in the file **oload.h**, and include the following code in the init module:

```
if (*p-ifilnam == sstrcod)
     strcpy(filename, unquote(p-strarg));
```

See the code for adset() in opcodes3.c, lprdset() in opcodes5.c, and pvset() in opcodes8.c.

---

77 Appendix A: Miscellaneous Information

77.1 Pitch Conversion

Note	Hz	cpspch	MIDI	Note	Hz	cpspch	MIDI
C-1	8.176	3.00	0	E4	329.628	8.04	64
C#-1	8.662	3.01	1	F4	349.228	8.05	65
D-1	9.177	3.02	2	F#4	369.994	8.06	66
D#-1	9.723	3.03	3	G4	391.995	8.07	67
E-1	10.301	3.04	4	G#4	415.305	8.08	68
F-1	10.913	3.05	5	A4	440.000	8.09	69
F#-1	11.562	3.06	6	A#4	466.164	8.10	70
G-1	12.250	3.07	7	B4	493.883	8.11	71
G#-1	12.978	3.08	8	C5	523.251	9.00	72
A-1	13.750	3.09	9	C#5	554.365	9.01	73
A#-1	14.568	3.10	10	D5	587.330	9.02	74
B-1	15.434	3.11	11	D#5	622.254	9.03	75
C0	16.352	4.00	12	E5	659.255	9.04	76
C#0	17.324	4.01	13	F5	698.456	9.05	77
D0	18.354	4.02	14	F#5	739.989	9.06	78
D#0	19.445	4.03	15	G5	783.991	9.07	79
E0	20.602	4.04	16	G#5	830.609	9.08	80
F0	21.827	4.05	17	A5	880.000	9.09	81
F#0	23.125	4.06	18	A#5	932.328	9.10	82
G0	24.500	4.07	19	B5	987.767	9.11	83
G#0	25.957	4.08	20	C6	1046.502	10.00	84
A0	27.500	4.09	21	C#6	1108.731	10.01	85
A#0	29.135	4.10	22	D6	1174.659	10.02	86
B0	30.868	4.11	23	D#6	1244.508	10.03	87
C1	32.703	5.00	24	E6	1318.510	10.04	88
C#1	34.648	5.01	25	F6	1396.913	10.05	89
D1	36.708	5.02	26	F#6	1479.978	10.06	90
D#1	38.891	5.03	27	G6	1567.982	10.07	91
E1	41.203	5.04	28	G#6	1661.219	10.08	92
F1	43.654	5.05	29	A6	1760.000	10.09	93
F#1	46.249	5.06	30	A#6	1864.655	10.10	94
G1	48.999	5.07	31	B6	1975.533	10.11	95
G#1	51.913	5.08	32	C7	2093.005	11.00	96
A1	55.000	5.09	33	C#7	2217.461	11.01	97
A#1	58.270	5.10	34	D7	2349.318	11.02	98
B1	61.735	5.11	35	D#7	2489.016	11.03	99
C2	65.406	6.00	36	E7	2637.020	11.04	100
C#2	69.296	6.01	37	F7	2793.826	11.05	101
D2	73.416	6.02	38	F#7	2959.955	11.06	102
D#2	77.782	6.03	39	G7	3135.963	11.07	103
E2	82.407	6.04	40	G#7	3322.438	11.08	104
F2	87.307	6.05	41	A7	3520.000	11.09	105
F#2	92.499	6.06	42	A#7	3729.310	11.10	106
G2	97.999	6.07	43	B7	3951.066	11.11	107
G#2	103.826	6.08	44	C8	4186.009	12.00	108
A2	110.000	6.09	45	C#8	4434.922	12.01	109
A#2	116.541	6.10	46	D8	4698.636	12.02	110
B2	123.471	6.11	47	D#8	4978.032	12.03	111

Note	Hz	cpspch	MIDI	Note	Hz	cpspch	MIDI
C3	130.813	7.00	48	E8	5274.041	12.04	112
C#3	138.591	7.01	49	F8	5587.652	12.05	113
D3	146.832	7.02	50	F#8	5919.911	12.06	114
D#3	155.563	7.03	51	G8	6271.927	12.07	115
E3	164.814	7.04	52	G#8	6644.875	12.08	116
F3	174.614	7.05	53	A8	7040.000	12.09	117
F#3	184.997	7.06	54	A#8	7458.620	12.10	118
G3	195.998	7.07	55	B8	7902.133	12.11	119
G#3	207.652	7.08	56	C9	8372.018	13.00	120
A3	220.000	7.09	57	C#9	8869.844	13.01	121
A#3	233.082	7.10	58	D9	9397.273	13.02	122
B3	246.942	7.11	59	D#9	9956.063	13.03	123
C4	261.626	8.00	60	E9	10548.08	13.04	124
C#4	277.183	8.01	61	F9	11175.30	13.05	125
D4	293.665	8.02	62	F#9	11839.82	13.06	126
D#4	311.127	8.03	63	G9	12543.85	13.07	127

77.2 Sound Intensity Values (for a 1000 Hz tone)

Dynamics	Intensity (W/m^2)	Level (dB)
pain	1	120
fff	10^-2	100
f	10^-4	80
p	10^-6	60
ppp	10^-8	40
threshold	10^-12	0

77.3 Formant Values

	f1	f2	f3	f4	f5		f1	f2	f3	f4	f5
soprano "a"						**tenor "a"**					
freq (Hz)	800	1150	2900	3900	4950	freq (Hz)	650	1080	2650	2900	3250
amp (dB)	0	-6	-32	-20	-50	amp (dB)	0	-6	-7	-8	-22
bw (Hz)	80	90	120	130	140	bw (Hz)	80	90	120	130	140
soprano "e"						**tenor "e"**					
freq (Hz)	350	2000	2800	3600	4950	freq (Hz)	400	1700	2600	3200	3580
amp (dB)	0	-20	-15	-40	-56	amp (dB)	0	-14	-12	-14	-20
bw (Hz)	60	100	120	150	200	bw (Hz)	70	80	100	120	120
soprano "i"						**tenor "i"**					
freq (Hz)	270	2140	2950	3900	4950	freq (Hz)	290	1870	2800	3250	3540
amp (dB)	0	-12	-26	-26	-44	amp (dB)	0	-15	-18	-20	-30
bw (Hz)	60	90	100	120	120	bw (Hz)	40	90	100	120	120
soprano "o"						**tenor "o"**					
freq (Hz)	450	800	2830	3800	4950	freq (Hz)	400	800	2600	2800	3000
amp (dB)	0	-11	-22	-22	-50	amp (dB)	0	-10	-12	-12	-26
bw (Hz)	70	80	100	130	135	bw (Hz)	40	80	100	120	120
soprano "u"						**tenor "u"**					
freq (Hz)	325	700	2700	3800	4950	freq (Hz)	350	600	2700	2900	3300
amp (dB)	0	-16	-35	-40	-60	amp (dB)	0	-20	-17	-14	-26
bw (Hz)	50	60	170	180	200	bw (Hz)	40	60	100	120	120
alto "a"						**bass "a"**					
freq (Hz)	800	1150	2800	3500	4950	freq (Hz)	600	1040	2250	2450	2750
amp (dB)	0	-4	-20	-36	-60	amp (dB)	0	-7	-9	-9	-20
bw (Hz)	80	90	120	130	140	bw (Hz)	60	70	110	120	130
alto "e"						**bass "e"**					
freq (Hz)	400	1600	2700	3300	4950	freq (Hz)	400	1620	2400	2800	3100
amp (dB)	0	-24	-30	-35	-60	amp (dB)	0	-12	-9	-12	-18
bw (Hz)	60	80	120	150	200	bw (Hz)	40	80	100	120	120
alto "i"						**bass "i"**					
freq (Hz)	350	1700	2700	3700	4950	freq (Hz)	250	1750	2600	3050	3340
amp (dB)	0	-20	-30	-36	-60	amp (dB)	0	-30	-16	-22	-28
bw (Hz)	50	100	120	150	200	bw (Hz)	60	90	100	120	120
alto "o"						**bass "o"**					
freq (Hz)	450	800	2830	3500	4950	freq (Hz)	400	750	2400	2600	2900
amp (dB)	0	-9	-16	-28	-55	amp (dB)	0	-11	-21	-20	-40
bw (Hz)	70	80	100	130	135	bw (Hz)	40	80	100	120	120
alto "u"						**bass "u"**					
freq (Hz)	325	700	2530	3500	4950	freq (Hz)	350	600	2400	2675	2950
amp (dB)	0	-12	-30	-40	-64	amp (dB)	0	-20	-32	-28	-36
bw (Hz)	50	60	170	180	200	bw (Hz)	40	80	100	120	120

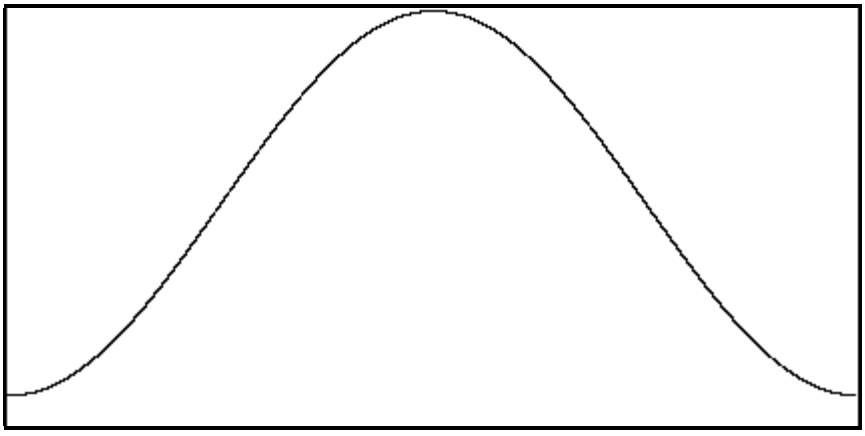
	f1	f2	f3	f4	f5
countertenor "a"					
freq (Hz)	660	1120	2750	3000	3350
amp (dB)	0	-6	-23	-24	-38
bw (Hz)	80	90	120	130	140
countertenor "e"					
freq (Hz)	440	1800	2700	3000	3300
amp (dB)	0	-14	-18	-20	-20
bw (Hz)	70	80	100	120	120
countertenor "i"					
freq (Hz)	270	1850	2900	3350	3590
amp (dB)	0	-24	-24	-36	-36
bw (Hz)	40	90	100	120	120
countertenor "o"					
freq (Hz)	430	820	2700	3000	3300
amp (dB)	0	-10	-26	-22	-34
bw (Hz)	40	80	100	120	120
countertenor "u"					
freq (Hz)	370	630	2750	3000	3400
amp (dB)	0	-20	-23	-30	-34
bw (Hz)	40	60	100	120	120

77.4 Window Functions

Windowing functions are used for analysis, and as waveform envelopes, particularly in granular synthesis. Window functions are built in to some opcodes, but others require a function table to generate the window. **GEN20** is used for this purpose. The diagram of each window below, is accompanied by the **f statement** used to generate the it.

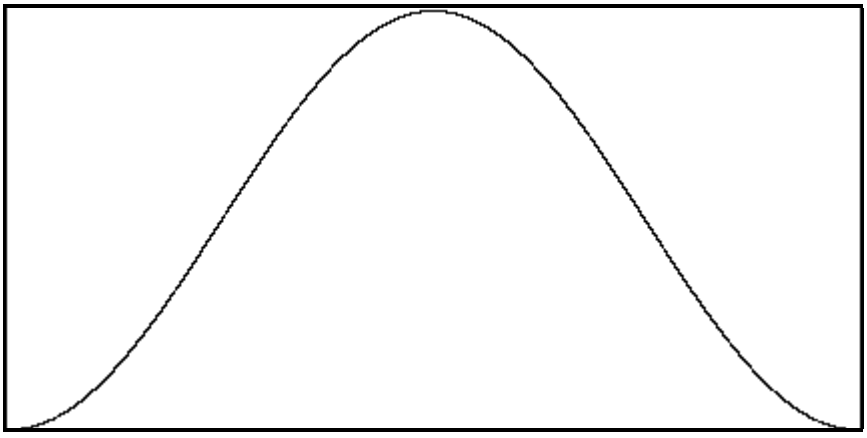
HAMMING

```
f81   0   8192   20   1   1
```



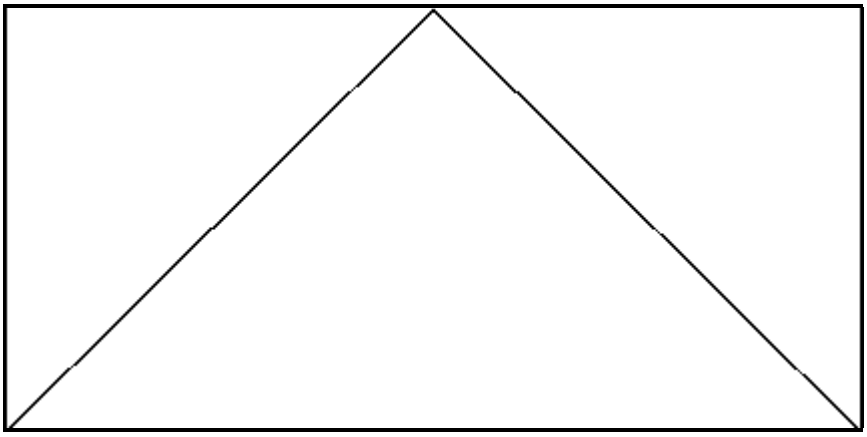
HANNING

```
f82   0   8192   20   2   1
```



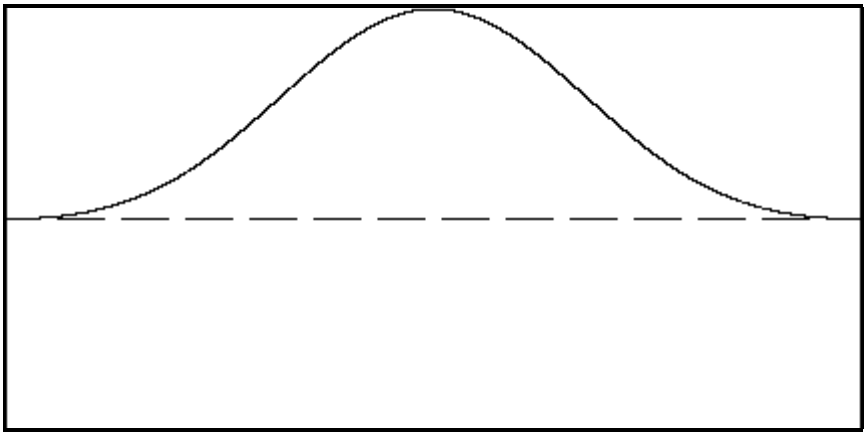
BARTLETT

```
f83   0   8192   20   3   1
```



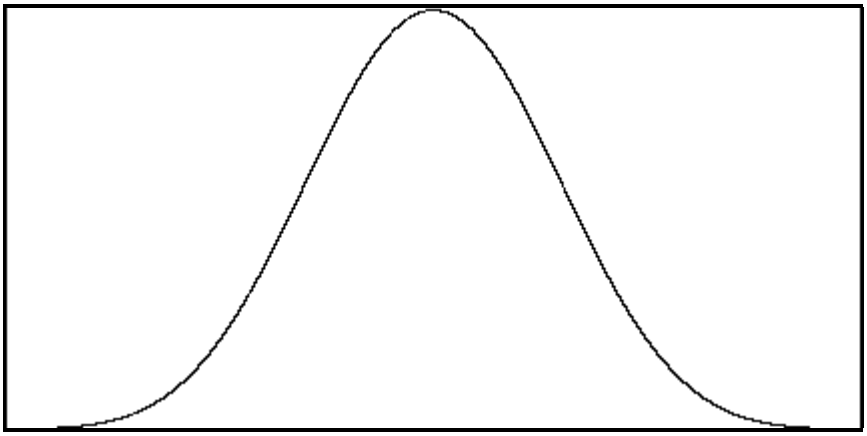
BLACKMAN

```
f84   0   8192   20   4   1
```



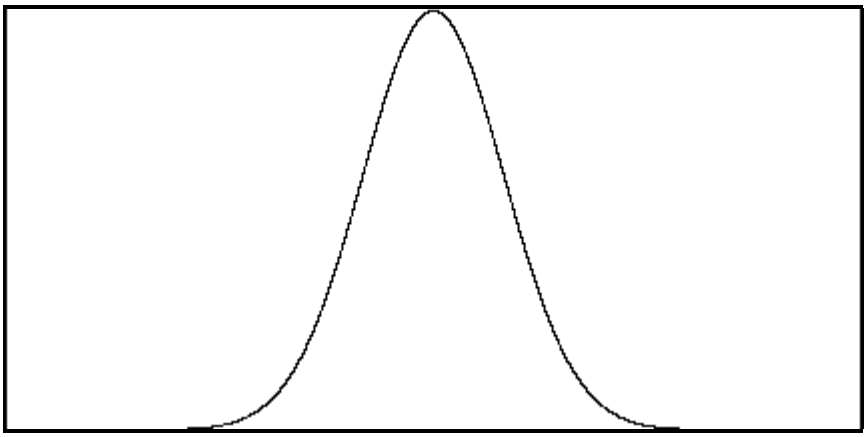
BLACKMAN-HARRIS

```
f85   0   8192   20   5   1
```



GAUSSIAN

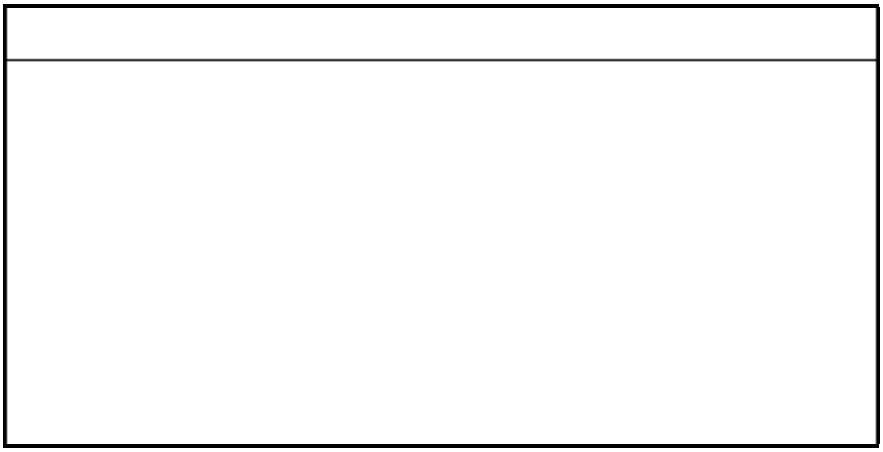
```
f86   0   8192   20   6   1
```



RECTANGLE

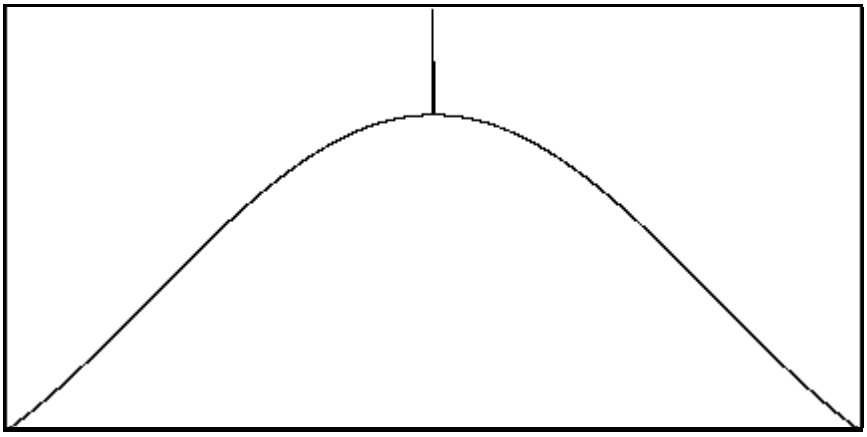
```
f88   0   8192   -20   8   .1
```

Note: Vertical scale is exaggerated in this diagram.



SYNC

```
f89   0   4096   -20   9   .75
```



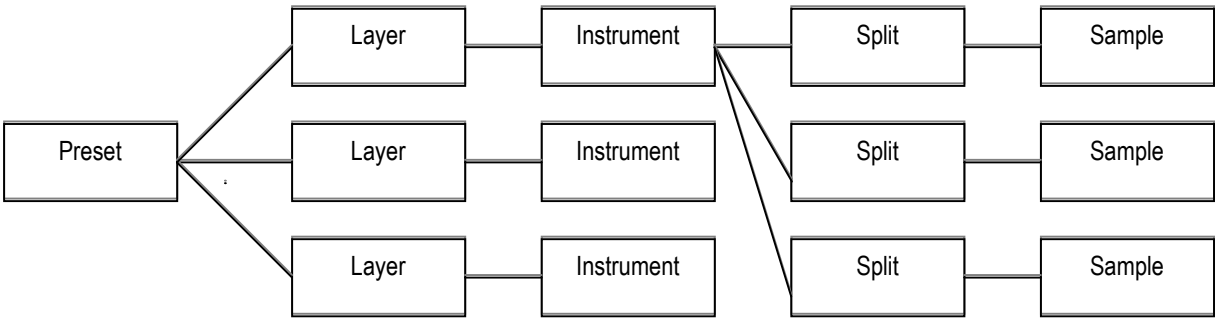
77.5 SoundFont2 File Format

Beginning with Csound Version 4.06, Csound supports SoundFont2 sample file format. SoundFont2 (or SF2) is a widespread standard which allows encoding banks of wavetable-based sounds into a binary file. In order to understand the usage of these opcodes, the user must have some knowledge of the SF2 format, so a brief description of this format follows.

The SF2 format is made by generator and modulator objects. All current Csound opcodes regarding SF2 support the generator function only.

There are several levels of generators having a hierarchical structure. The most basic kind of generator object is a sample. Samples may or may not be be looped, and are associated with a MIDI note number, called the base-key. When a sample is associated with a range of MIDI note numbers, a range of velocities, a transposition (coarse and fine tuning), a scale tuning, and a level scaling factor, the sample and its associations make up a "split." A set of splits, together with a name, make up an "instrument." When an instrument is associated with a key range, a velocity range, a level scaling factor, and a transposition, the instrument and its associations make up a "layer." A set of layers, together with a name, makes up a "preset." Presets are normally the final sound-generating structures ready for the user. They generate sound according to the settings of their lower-level components.

Both sample data and structure data is embedded in the same SF2 binary file. A single SF2 file can contain up to a maximum of 128 banks of 128 preset programs, for a total of 16384 presets in one SF2 file. The maximum number of layers, instruments, splits, and samples is not defined, and probably is only limited by the computer's memory.



SoundFont2 File Structure

77.6 Print Edition Update Procedure

Updated pages only, are available, in Adobe Acrobat® (*.pdf*) format, separately from the complete manual. There are separate sets of files for single- and double-sided printing. The name of the update file for single-sided printing, will be with the Csound version number to which they correspond, ending with *up.pdf*. For example, the update file for version 3.52 will be called *3_52up.pdf*.

The files containing the update pages only, for double-sided printing, will follow the same convention as for single-sided printing, except that a *1* will be appended for the odd numbered pages and a *2* for even numbered pages. Example: *3_52up1.pdf* and *3_52up2.pdf* for Csound version 3.52.

There will be as many sets of updates on the server as space permits, in the event the user misses an update before the next one is released. The version of the manual is stated on the title page, in the footer of each page, and in Section 22.6 (Manual Update History).

To update an existing manual, print the update file(s) for either single- or double-sided printing, as required. Insert the new pages, and replace the changed pages, as needed, discarding the old pages that have been replaced.

Where to Get the Manual

The manual files are available from browser download from the editor's website:

```
http://www.lakewoodsound.com/csound
```

or via anonymous ftp:

```
ftp://ftp.csounds.com/manual
```

All the files are zipped for easy downloading, but the Acrobat files are not compressed. Also available at this site are an HTML Edition, and ASCII text edition, and a Spanish Edition, also in Acrobat format, translated by Servando Valero.

Bug Reports

We have worked to make these manuals as accurate as possible. Errors, however, will happen. If you find a bug, an error, or omission, please report it to the editor (csound@lakewoodsound.com).

77.7 Manual Update History

Note: Beginning with Version 3.55, page numbers in **bold** indicate new or added pages. Other page numbers are revised, existing pages.

DATE	VERSION	NEW OR CHANGED PAGES
September, 1998	Version 3.48	All
3 November, 1998	Version 3.49	iii, v-xiv, 1:3, 2:1-2:2, 2:13, 5:4, 7:12-7:14, 8:4, 8:16-8:65, 9:3-9:4, 9:31-9:40, 11:5-11:6, 13:1-13:2, 13:9, 14:4-14:18, 15:18-15:20, 16:4-16:8, 22:6
8 November, 1998	Version 3.491	I, iii, v-xiv, 2:2, 2:16, 8:66, 12:5-12:6, 14:7, 22:6
Included in 3.493 release	Version 3.492	7:3, 7:14 – 7:18, 8:59 – 8:60, 8:66, 9:8 – 9:9, 9:35, 13:5 – 13:6
24 November, 1998	Version 3.493	iii, vi – xiv, 2:5, 5:3 8:18, 8:20, 22:6
5 January, 1999	Version 3.494	iii, vii-xi, xiii-xiv, 3:1, 4:1, 8:14, 8:15, 8:35, 8:59, 9:32, 9:41, 9:42, 22:6
21 January, 1999	Version 3.50	iii-xvi, 2:9, 5:3, 7:3, 7:19-22, 8:4-6, 8:18, 8:61 8:67-70, 9:8-9, 9:21, 9:23 9:36-37, 9:42-44 13:1-2, 13:5-6, 17:1-2, 18:1-2, 22:6
27 January, 1999	Version 3.51	iii, v, vii, xiv, xv, 8:65, 9:14, 17:1, 17:2, 22:6
24 February, 1999	Version 3.52	iii; v-xi; xiii-xv; 5:4; 7:4-7; 7:11; 7:19-22; 8:8-12; 8:49; 8:54; 8:71-72; 9:28; 9:38; 9:42; 12:5; 13:5; 13:6; 13:7; 14:4; 22:6
23 March, 1999	Version 3.53	All
20 May, 1999	Version 3.54	I; iii; v-xv; xix; xxiii-xxv; xxviii; xxx; xxxii-xxxiii; 2:2; 5:4-6; 7:6-7; 7:9; 7:12; 7:20; 8:4-5; 8:8; 8:10-11; 8:16; 8:18, 8:28; 8:30; 8:34; 8:34; 8:44-46; 8:65; 8:67-68; 9:1-2; 9:6; 9:10; 9:12; 9:39-40; 9:46; 9:49-50; Sec13; 19:1-2; 19:9
21 June, 1999	Version 3.55	I; iii; v-xvi; xix; xxxi; xxxv-xxxvi; 1:5; 2:1-3; 2:10-11; 4:1; 5:4-6; 5:7-8; 7:18; 7:20; 8:5-15; 8:20; 8:24; 8:32; 8:34; 8:37; 8:39; 8:42; 8:60; 8:64; 8:68; 8:70; 8:72; 9:13; 9:23; 9:26; 9:32; 9:34; 9:36-38; 9:39; 9:41; 9:44; 9:48-49; 9:50-58; 10:6; 13:2; 13:6; 13:8; 13:11-12; 13:14-19; 14:1-2; 14:4-6; 14:9-10; 14:14-17; 14:19-21; 15:1-7; 15:10-14; 15:16-20; 15:21-22; 16:2-8; 17:2; 18:2; 19:4-6; 19:10; 20:1; 20:5-7; 20:10-13; 21:2; 22:6; 22:10-11
22 July, 1999	Version 3.56	i-xxxviii; xxxix-xl; 7:6-9; 7:11-12; 7:14-16; 7:18-21; 8:1; 8:4-5; 8:8-9; 8:11; 8:13-15; 8:22-23; 8:63-74; 8:75-76; 9:59-62; 12:7-8; 13:20; 13:21-22; 14:8; 14:11; 22:11-12; 23:1-24
9 August, 1999	Version 3.57	I; v-xl; xli-xlvi; 2:9; 2:16; 5:2; 5:9-10; 6:1; 7:2; 7:18; 7:20; 8:1-3; 8:6-7; 8:18; 8:20; 8:22-23; 8:25; 8:32; 8:40; 8:60; 8:63; 9:1-62; 9:63-64; 12:1; 12:7-8; 12:9-10; 13:20; 13:22; 14:7; 15:16-22; 19:4-10; 19:11-16; 20:04; 22:11; 23:1-22; (delete 23:23-24)

Manual Update History (Continued)

18 August, 1999	Version 3.58	I; v-xv; iii; xxxi; xxxii; xxxiii; xxxiv; 1:3; 5:4; 7:1; 9:3; 9:13; 9:16; 9:20; 9:25-26; 9:36; 9:38; 9:40; 9:46-64; 14:11-22; **14:23-24**; 15:19; 19:11; 22:10; 22:12
16 November, 1999	Version 4.0	All
23 November, 1999	Version 4.01	I; ix-xvii; 42-3; 49-7 – 49-22; **49-23 – 49-24**; 70-5; 76-12; QR10; QR13
25 February, 2000	Version 4.03	I – xvii; 2-3 – 2-5; 3-1; 3-3; **11-11**; **11-12**; 42-7; 45-1; 50-3; 50-4; **50-5**; **50-6**; 52-1; 52-2; 68-15; 76-12; QR2; QR11; QR13
1 August, 2000	Version 4.06	All
15 September, 2000	Version 4.07	I–xviii; 29-2; 34-1; 35-3; 39-3; 40-11; **42-11–42-12**; 43-3; 43-12; **43-13–43-16**; 56-1; 56-3; **58-3**; **58-4**; 60-6; 60-11–60-13; 70-2–70-24; 77-13; QR10; QR14; QR16
20 March, 2001	Version 4.10	All

Csound Quick Reference

Orchestra Syntax: Orchestra Header Statements

sr	=	iarg
kr	=	iarg
ksmps	=	iarg
nchnls	=	iarg
	strset	iarg, “stringtext”
	pset	con1, con2, ...
	seed	ival
gir	**ftgen**	ifn, itime, isize, igen, iarga[, iargb, …iargz]
	massign	ichnl, insnum
	ctrlinit	ichnkm, ictlno1, ival1[, ictlno2, ival2[, ictlno3, ival3[, …ival32]]

Orchestra Syntax: Variable Data Types

iname	(init variable - initialization only)
kname	(control signal - performance time, control rate)
aname	(audio signal - performance time, audio rate)
giname	(global init variable - initialization only)
gkname	(global control signal - performance time, control rate)
ganame	(global audio signal - performance time, audio rate)
wname	(spectral data – performance time, control rate)

Orchestra Syntax: Instrument Block Statements

instr	*NN*
endin	

Orchestra Syntax: Variable Initialization

i/k/ar	**=**	iarg
i/k/ar	**init**	iarg
ir	**tival**	
i/k/ar	**divz**	ia, ib, isubst

Instrument Control: Instrument Invocation

schedule	insnum, iwhen, idur[, p4, p5, …]
schedwhen	ktrigger, kinsnum, kwhen, kdur[, p4, p5, …]
schedkwhen	ktrigger, kmintim, kmaxnum, kinsnum, kwhen, kdur[, kp4, kp5, …]
turnon	insnum[, itime]

Instrument Control: Duration Control

ihold
turnoff

Csound Quick Reference

Instrument Control: Realtime Performance Control

ir	**active**	insnum
	cpuprc	insnum, ipercent
	maxalloc	insnum, icount
	prealloc	insnum, icount

Instrument Control: Time Reading

i/kr	**timek**	
i/kr	**times**	
kr	**timeinstk**	
kr	**timeinsts**	

Instrument Control: Clock Control

	clockon	inum
	clockoff	inum
ir	**readclock**	inum

Instrument Control: Sensing and Control

kpitch, kamp	**pitch**	asig, iupdte, ilo, ihi, idbthresh[, ifrqs, iconf, istrt, iocts, iq, inptls, irolloff, iskip]
kcps, krms	**pitchamdf**	asig, imincps, imaxcps[, icps[, imedi[, idowns [, iexcps]]]]
ktemp	**tempest**	kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, istartempo, ifn[, idisprd, itweek]
kr	**follow**	asig, idt
kout	**trigger**	ksig, kthreshold, kmode
k/ar	**peak**	k/asig
	tempo	ktempo, istartempo
kx, ky	**xyin**	iprd, ixmin, ixmax, iymin, iymax[, ixinit, iyinit]
ar	**follow2**	asig, katt, krel
	setctrl	inum, kval, itype
kr	**control**	inum
kr	**button**	inum
kr	**checkbox**	inum
kr	**sensekey**	

Instrument Control: 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)

Csound Quick Reference

Instrument Control: Macros

```
        #define     NAME # replacement text #
        #define     NAME(a'b'c) # replacement text #
        $NAME.
        #undef      NAME
        #include    "filename"
```

Instrument Control: Program Flow 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
label   :
```

Instrument Control: Reinitialization

```
        reinit      label
        rigoto      label
        rireturn
```

Mathematical Operations: Arithmetic and Logic Operations

```
        - a         (no rate restriction)
        + a         (no rate restriction)
        a && b      (logical AND; not audio-rate)
        a || b      (logical OR; not audio-rate)
        a + b       (no rate restriction)
        a - b       (no rate restriction)
        a * b       (no rate restriction)
        a / b       (no rate restriction)
        a ^ b       (b not audio-rate)
        a % b       (no rate restriction)
```

Csound Quick Reference

Mathematical Operations: Mathematical Functions

int(x)	(init-rate or control-rate args only)
frac(x)	(init-rate 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)
log10(x)	(no rate restriction)
sqrt(x)	(no rate restriction)
powoftwo(x)	(init-rate or control-rate args only)
logbtwo(x)	(init-rate or control-rate args only)

Mathematical Operations: Trigonometric Functions

sin(x)	(no rate restriction)
cos(x)	(no rate restriction)
tan(x)	(no rate restriction)
sininv(x)	(no rate restriction)
cosinv(x)	(no rate restriction)
taninv(x)	(no rate restriction)
sinh(x)	(no rate restriction)
cosh(x)	(no rate restriction)
tanh(x)	(no rate restriction)

Mathematical Operations: Amplitude Functions

dbamp(x)	(init-rate or control-rate args only)
ampdb(x)	(no rate restriction)
dbfsamp(x)	(init-rate or control-rate args only)
ampdbfs(x)	(no rate restriction)

Mathematical Operations: Random Functions

rnd(x)	(init- or control-rate only)
birnd(x)	(init- or control-rate only)

Mathematical Operations: Opcode Equivalents of Functions

ar	**sum**	asig1, asig2[,asig3…asigN]
ar	**product**	asig1, asig2[,asig3…asigN]
i/k/ar	**pow**	i/k/aarg, i/k/pow
i/k/ar	**taninv2**	i/k/ax, i/k/ay
ar	**mac**	asig1, ksig1, asig2, ksig2, asig3, …
ar	**maca**	asig1, ksig1, asig2, ksig2, asig3, …

Csound Quick Reference

Pitch Converters: Functions

```
          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)
```

Pitch Convertors: Tuning Opcodes

```
icps      cps2pch      ipch, iequal
icps      cpsxpch      ipch, iequal, irepeat, ibase
```

MIDI Support: Converters

```
ival      notnum
ival      veloc        [ilow, ihigh]
icps      cpsmidi
i/kcps    cpsmidib     [irange]
icps      cpstmid      ifn
ioct      octmidi
i/koct    octmidib     [irange]
ipch      pchmidi
i/kpch    pchmidib     [irange]
iamp      ampmidi      iscal[, ifn]
kaft      aftouch      [imin[, imax]]
i/kbend   pchbend      [imin[, imax]]
i/kval    midictrl     inum[, imin[, imax]]
```

MIDI Support: Controller Input

```
          initc7       ichan, ictlno, ivalue
          initc14      ichan, ictlno1, ictlno2, ivalue
          initc21      ichan, ictlno1, ictlno2, ictlno3, ivalue
i/kdest   midic7       ictlno, i/kmin, i/kmax[, ifn]
i/kdest   midic14      ictlno1, ictlno2, i/kmin, i/kmax[, ifn]
i/kdest   midic21      ictlno1, ictlno2, ictlno3, i/kmin, i/kmax[, ifn]
i/kdest   ctrl7        ichan, ictlno, i/kmin, i/kmax[, ifn]
i/kdest   ctrl14       ichan, ictlno1, ictlno2, i/kmin, i/kmax[, ifn]
i/kdest   ctrl21       ichan, ictlno1, ictlno2, ictlno3, i/kmin, i/kmax[, ifn]
i/kval    chanctrl     ichnl, ictlno[, ilow, ihigh]
```

MIDI Support: Slider Banks

i/k1, …, i/k8	**slider8**	ichan, ictlnum1, imin1, imax1, init1, ifn1, …, ictlnum8, imin8, imax8, init8, ifn8
i/k1, …, i/k16	**slider16**	ichan, ictlnum1, imin1, imax1, init1, ifn1, …, ictlnum16, imin16, imax16, init16, ifn16
i/k1, …, i/k32	**slider32**	ichan, ictlnum1, imin1, imax1, init1, ifn1, …, ictlnum32, imin32, imax32, init32, ifn32
i/k1, …, i/k64	**slider64**	ichan, ictlnum1, imin1, imax1, init1, ifn1, …, ictlnum64, imin64, imax64, init64, ifn64
k1, …, k8	**slider8f**	ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, …, ictlnum8, imin8, imax8, init8, ifn8, icutoff8
k1, …, k16	**slider16f**	ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, …, ictlnum16, imin16, imax16, init16, ifn16, icutoff16
k1, …, k32	**slider32f**	ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, …, ictlnum32, imin32, imax32, init32, ifn32, icutoff32
k1, …, k64	**slider64f**	ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1, …, ictlnum64, imin64, imax64, init64, ifn64, icutoff64
i/k1, …, i/k16	**s16b14**	ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, initvalue1, ifn1, …, ictlno_msb16, ictlno_lsb16, imin16, imax16, initvalue16, ifn16
i/k1, …, i/k32	**s32b14**	ichan, ictlno_msb1, ictlno_lsb1, imin1, imax1, initvalue1, ifn1, …, ictlno_msb32, ictlno_lsb32, imin32, imax32, initvalue32, ifn32

MIDI Support: Generic I/O

kstatus, kchan, kdata1, kdata2	**midiin**	
	midiout	kstatus, kchan, kdata1, kdata2

MIDI Support: Note-on/Note-off

noteon	ichn, inum, ivel
noteoff	ichn, inum, ivel
noteondur	ichn, inum, ivel, idur
noteondur2	ichn, inum, ivel, idur
moscil	kchn, knum, kvel, kdur, kpause
midion	kchn, knum, kvel
midion2	kchn, knum, kvel, ktrig

MIDI Support: MIDI Message Output

	outic	ichn, inum, ivalue, imin, imax
	outkc	kchn, knum, kvalue, kmin, kmax
	outic14	ichn, imsb, ilsb, ivalue, imin, imax
	outkc14	kchn, kmsb, klsb, kvalue, kmin, kmax
	outipb	ichn, ivalue, imin, imax
	outkpb	kchn, kvalue, kmin, kmax
	outiat	ichn, ivalue, imin, imax
	outkat	kchn, kvalue, kmin, kmax
	outipc	ichn, iprog, imin, imax
	outkpc	kchn, kprog, kmin, kmax
	outipat	ichn, inotenum, ivalue, imin, imax
	outkpat	kchn, knotenum, kvalue, kmin, kmax
	nrpn	kchan, kparmnum, kparmvalue
	mdelay	kstatus, kchan, kd1, kd2, kdelay

MIDI Support: Realtime Messages

	mclock	ifreq
	mrtmsg	imsgtype

MIDI Support: MIDI Event Extenders

	xtratim	iextradur
kflag	**release**	

Signal Generators: Linear and Exponential Generators

k/ar	**line**	ia, idur1, ib
k/ar	**expon**	ia, idur1, ib
k/ar	**linseg**	ia, idur1, ib[, idur2, ic[…]]
k/ar	**linsegr**	ia, idur1, ib[, idur2, ic[…]], irel, iz
k/ar	**expseg**	ia, idur1, ib[, idur2, ic[…]]
k/ar	**expsegr**	ia, idur1, ib[, idur2, ic[…]], irel, iz
ar	**expsega**	ia, idur1, ib[, idur2, ic[…]]
k/ar	**adsr**	iatt, idec, islev, irel[, idel]
k/ar	**madsr**	iatt, idec, islev, irel[, idel]
k/ar	**xadsr**	iatt, idec, islev, irel[, idel]
k/ar	**mxadsr**	iatt, idec, islev, irel[, idel]
k/ar	**transeg**	ibeg, idur, itype, ival

Csound Quick Reference

Signal Generators: Table Access

i/k/ar	**table**	i/k/andx, ifn[, ixmode[, ixoff[, iwrap]]]
i/k/ar	**tablei**	i/k/andx, ifn[, ixmode[, ixoff[, iwrap]]]
i/k/ar	**table3**	i/k/andx, ifn[, ixmode[, ixoff[, iwrap]]]
kr	**oscil1**	idel, kamp, idur, ifn
kr	**oscil1i**	idel, kamp, idur, ifn
ar	**osciln**	kamp, ifrq, ifn, itimes

Signal Generators: Phasors

k/ar	**phasor**	k/xcps[, iphs]
k/ar	**phasorbnk**	k/xcps, kindx, icnt [, iphs]

Signal Generators: Basic Oscillators

k/ar	**oscil**	k/xamp, k/xcps, ifn[, iphs]
k/ar	**oscili**	k/xamp, k/xcps, ifn[, iphs]
k/ar	**oscil3**	k/xamp, k/xcps, ifn[, iphs]
k/ar	**poscil**	kamp, kcps, ifn[, iphs]
k/ar	**poscil3**	kamp, kcps, ifn[, iphs]
k/ar	**lfo**	kamp, kcps[, itype]

Signal Generators: Dynamic Spectrum Oscillators

ar	**buzz**	xamp, xcps, knh, ifn[, iphs]
ar	**gbuzz**	xamp, xcps, knh, klh, kr, ifn[, iphs]
ar	**vco**	kamp, kfqc[, iwave][, ipw][, ifn][, imaxd]

Signal Generators: Additive Synthesis/Resynthesis

ar	**adsyn**	kamod, kfmod, ksmod, ifilcod
ar	**adsynt**	kamp, kcps, iwfn, ifreqfn, iampfn, icnt[, iphs]
ar	**hsboscil**	kamp, ktone, kbrite, ibasfreq, iwfn, ioctfn[, ioctcnt [, iphs]]

Csound Quick Reference

Signal Generators: FM Synthesis

ar	**foscil**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar	**foscili**	xamp, kcps, kcar, kmod, kndx, ifn[, iphs]
ar	**fmvoice**	kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate, ifn1, ifn2, ifn3, ifn4, ivibfn
ar	**fmbell**	kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar	**fmrhode**	kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar	**fmwurlie**	kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar	**fmmetal**	kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar	**fmb3**	kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
ar	**fmpercfl**	kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn

Signal Generators: Sample Playback

ar[, ar2]	**loscil**	xamp, kcps, ifn[, ibas[, imod1, ibeg1, iend1[, imod2, ibeg2, iend2]]]
ar[, ar2]	**loscil3**	xamp, kcps, ifn[, ibas[, imod1, ibeg1, iend1[, imod2, ibeg2, iend2]]]
ar	**lposcil**	kamp, kfreqratio, kloop, kend, ifn[, iphs]
ar	**lposcil3**	kamp, kfreqratio, kloop, kend, ifn[, iphs]

Signal Generators: Granular Synthesis

ar	**fof**	xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur[, iphs[, ifmode]]
ar	**fof2**	xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur, kphs, kgliss
ar	**fog**	xamp, xdens, xtrans, xspd, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, itotdur[, iphs[, itmode]]
ar	**grain**	xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn, iwfn, imgdur[, igrnd]
ar	**granule**	xamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip, igskip_os, ilength, kgap, igap_os, kgsize, igsize_os, iatt, idec[, iseed[, ipitch1[, ipitch2[, ipitch3[, ipitch4[, ifnenv]]]]]]]
ar[, ac]	**sndwarp**	xamp, xtimewarp, xresample, ifn1, ibeg, iwsize, irandw, ioverlap, ifn2, itimemode
ar1, ar2 [,ac1, ac2]	**sndwarpst**	xamp, xtimewarp, xresample, ifn1, ibeg, iwsize, irandw, ioverlap, ifn2, itimemode

Csound Quick Reference

Signal Generators: Waveguide Physical Modeling

ar	**pluck**	kamp, kcps, icps, ifn, imeth[, iparm1, iparm2]
ar	**wgpluck**	icps, iamp, kpick, iplk, idamp, ifilt, axcite
ar	**repluck**	iplk, xam, icps, kpick, krefl, axcite
ar	**wgpluck2**	iplk, xam, icps, kpick, krefl
ar	**wgbow**	kamp, kfreq, kpres, krat, kvibf, kvamp, ifn[, iminfreq]
ar	**wgflute**	kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn[, iminfreq[, kjetrf[, kendrf]]]
ar	**wgbrass**	kamp, kfreq, iatt, kvibf, kvamp, ifn[, iminfreq]
ar	**wgclar**	kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, ifn[, iminfreq]

Signal Generators: Models and Emulations

ar	**moog**	kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn, iwfn, ivfn
ar	**shaker**	kamp, kfreq, kbeans, kdamp, ktimes[, idecay]
ar	**marimba**	kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec
ar	**vibes**	kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec
ar	**mandol**	kamp, kfreq, kpluck, kdetune, kgain, ksize, ifn[, iminfreq]
ar	**gogobel**	kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn
ar	**voice**	kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn
ax, ay, az	**lorenz**	ks, kr, kb, kh, ix, iy, iz, iskip
ax, ay, az	**planet**	kmass1, kmass2, ksep, ix, iy, iz, ivx, ivy, ivz, idelta, ifriction

Signal Generators: STFT Resynthesis (Vocoding)

ar	**pvoc**	ktimpnt, kfmod, ifilcod, ifn, ibins[, ibinoffset, ibinincr, iextractmode, ifreqlim, igatefn]
kfr, kap	**pvread**	ktimpnt, ifile, ibin
	pvbufread	ktimpnt, ifile
ar	**pvinterp**	ktimpnt, kfmod, ifile, kfreqscale1, kfreqscale2, kampscale1, kampscale2, kfreqinterp, kampinterp
ar	**pvcross**	ktimpnt, kfmod, ifile, kamp1, kamp2[, ispecwp]
	tableseg	ifn1, idur1, ifn2[, idur2, ifn3[…]]
	tablexseg	ifn1, idur1, ifn2[, idur2, ifn3[…]]
ar	**vpvoc**	ktimpnt, kfmod, ifile[, ispecwp]
ar	**pvadd**	ktimpnt, kfmod, ifilcod, ifn, ibins[, ibinoffset, ibinincr, iextractmode, ifreqlim, igatefn]

Csound Quick Reference

Signal Generators: LPC Resynthesis

krmsr, krmso, kerr, kcps	**lpread**	ktimpnt, ifilcod[, inpoles[, ifrmrate]]
ar	**lpreson**	asig
ar	**lpfreson**	asig, kfrqratio
	lpslot	islot
	lpinterp	islot1, islot2, kmix

Signal Generators: Random (Noise) Generators

k/ar	**rand**	k/xamp[, iseed[, isize]]
k/ar	**randh**	k/xamp, k/xcps[, iseed[, isize]]
k/ar	**randi**	k/xamp, k/xcps[, iseed[, isize]]
i/k/ar	**linrand**	krange
i/k/ar	**trirand**	krange
i/k/ar	**exprand**	krange
i/k/ar	**bexprnd**	krange
i/k/ar	**cauchy**	kalpha
i/k/ar	**pcauchy**	kalpha
i/k/ar	**poisson**	klambda
i/k/ar	**gauss**	krange
i/k/ar	**weibull**	ksigma, ktau
i/k/ar	**betarand**	krange, kalpha, kbeta
i/k/ar	**unirand**	krange
ar	**pinkish**	xin[, imethod, inumbands, iseed, iskip]
ar	**noise**	xamp, kbeta

Function Table Control: Table Queries

	nsamp(x)	(init-rate args only)
	ftlen(x)	(init-rate args only)
	ftlptim(x)	(init-rate args only)
	ftsr(x)	(init-rate args only)
i/kr	**tableng**	i/kfn

Function Table Control: Table Selection

k/ar	**tablekt**	k/xndx, i/kfn[, ixmode[, ixoff[, iwrap]]]
k/ar	**tableikt**	k/xndx, kfn[, ixmode[, ixoff[, iwrap]]]

Function Table Control: Read/Write Operations

	tablew	i/k/asig, i/k/andx, ifn[, ixmode[, ixoff[, iwgmode]]]
	tablewkt	k/asig, k/andx, kfn[, ixmode[, ixoff[, iwgmode]]]
	tableiw	isig, indx, ifn[, ixmode[, ixoff[, iwrap]]]
	tableigpw	ifn
	tablegpw	kfn
	tableimix	idft, idoff, ilen, is1ft, is1off, is1g, is2ft, is2off, is2g
	tablemix	kdft, kdoff, klen, ks1ft, ks1off, ks1g, ks2ft, ks2off, ks2g
	tableicopy	idft, isft
	tablecopy	kdft, ksft
ar	**tablera**	kfn, kstart, koff
kstart	**tablewa**	kfn, asig, koff

Signal Modifiers: Standard Filters

kr	**portk**	ksig, khtim[, isig]
kr	**port**	ksig, ihtim[, isig]
kr	**tonek**	ksig, khp[, iskip]
ar	**tone**	asig, khp[, iskip]
kr	**atonek**	ksig, khp[, iskip]
ar	**atone**	asig, khp[, iskip]
kr	**resonk**	ksig, kcf, kbw[, iscl, iskip]
ar	**reson**	asig, kcf, kbw[, iscl, iskip]
kr	**aresonk**	ksig, kcf, kbw[, iscl, iskip]
ar	**areson**	asig, kcf, kbw[, iscl, iskip]
ar	**tonex**	asig, khp[, inumlayer, iskip]
ar	**atonex**	asig, khp[, inumlayer, iskip]
ar	**resonx**	asig, kcf, kbw[, inumlayer, iscl, iskip]
ar	**resonr**	asig, kcf, kbw[,iscl, iskip]
ar	**resonz**	asig, kcf, kbw[,iscl, iskip]
ar	**resony**	asig, kbf, kbw, inum, ksep[,iscl, iskip]
ar	**lowres**	asig, kcutoff, kresonance[, iskip]
ar	**lowresx**	asig, kcutoff, kresonance[, inumlayer, iskip]
ar	**vlowres**	asig, kfco, kres, iord, ksep
ar	**lowpass2**	asig, kcf, kq[, iskip]
ar	**biquad**	asig, kb0, kb1, kb2, ka0, ka1, ka2[, iskip]
ar	**rezzy**	asig, xfco, xres[, imode]
ar	**moogvcf**	asig, xfco, xres[, iscale]
alow, ahigh, aband	**svfilt**	asig, kcf, kq[, iscl]
ar1, ar2	**hilbert**	asig
ar	**butterhp**	asig, kfreq[, iskip]
ar	**butterlp**	asig, kfreq[, iskip]
ar	**butterbp**	asig, kfreq, kband[, iskip]
ar	**butterbr**	asig, kfreq, kband[, iskip]
k/ar	**filter2**	k/asig, iM, iN, ib0, ib1, …, ibM, ia1, ia2, …, iaN
ar	**zfilter2**	asig, kdamp, kfreq, iM, iN, ib0, ib1, …, ibM, ia1, ia2, …, iaN
ar	**lpf18**	asig, kfco, kres, kdist
ar	**tbvcf**	asig, xfco, xres, kdist, kasym

Signal Modifiers: Specialized Filters

ar	**nlfilt**	ain, ka, kb, kd, kL, kC
ar	**pareq**	asig, kc, iv, iq, imode
ar	**dcblock**	asig[, ig]

Csound Quick Reference

Signal Modifiers: Envelope Modifiers

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

Signal Modifiers: Amplitude Modifiers

kr	**rms**	asig[, ihp, iskip]
ar	**gain**	asig, krms[, ihp, iskip]
ar	**balance**	asig, acomp[, ihp, iskip]
ar	**dam**	ain, kthreshold, icomp1, icomp2, rtime, ftime

Signal Modifiers: Signal Limiters

i/k/ar	**wrap**	i/k/asig, i/k/klow, i/k/khigh
i/k/ar	**mirror**	i/k/asig, i/k/klow, i/k/khigh
i/k/ar	**limit**	i/k/asig, i/k/klow, i/k/khigh

Signal Modifiers: Delay

ar	**delayr**	idlt[, iskip]
	delayw	asig
ar	**delay**	asig, idlt[, iskip]
ar	**delay1**	asig[, iskip]
ar	**deltap**	kdlt
ar	**deltapi**	xdlt
ar	**deltapn**	xnumsamps
ar	**deltap3**	xdlt
ar	**multitap**	asig, itime1, igain1, itime2, igain2…
ar	**vdelay**	asig, adel, imaxdel[, iskip]
ar	**vdelay3**	asig, adel, imaxdel[, iskip]

Signal Modifiers: Reverberation

ar	**reverb**	asig, krvt[, iskip]
ar	**reverb2**	asig, ktime, khdif[, iskip]
ar	**nreverb**	asig, ktime, khdif[, iskip][,inumCombs, ifnCombs][, inumAlpas, ifnAlpas]
ar	**comb**	asig, krvt, ilpt[, iskip][, insmps]
ar	**alpass**	asig, krvt, ilpt[, iskip][, insmps]
ar	**nestedap**	asig, imode, imaxdel, idel1, igain1[, idel2, igain2[, idel3, igain3]]
a1, a2	**babo**	asig, ksrcx, ksrcy, ksrcz, irx, iry, irz[, idiff[, ifno]]

Signal Modifiers: Waveguides

ar	**wguide1**	asig, kfreq, kcutoff, kfeedback
ar	**wguide2**	asig, kfreq1, kfreq2, kcutoff1, kcutoff2, kfeedback1, kfeedback2
ar	**streson**	asig, kfr, ifdbgain
ar	**nlalp**	asig, klcf, knlcf [, iskip[, iupdm]]

Signal Modfiers: Special Effects

ar	**harmon**	asig, kestfrq, kmaxvar, kgenfreq1, kgenfreq2, imode, iminfrq, iprd
ar	**flanger**	asig, adel, kfeedback[, imaxd]
ar	**distort1**	asig[, kpregain[, kpostgain[, kshape1[, kshape2]]]]
ar	**phaser1**	asig, kfreq, iord, kfeedback[, iskip]
ar	**phaser2**	asig, kfreq, iord, imode, ksep, kfeedback

Signal Modifiers: Convolution and Morphing

ar1[, ar2[, ar3[, ar4]]]	**convolve**	ain, ifilcod, ichannel
ar	**cross2**	ain1, ain2, isize, ioverlap, iwin, kbias

Signal Modifiers: Panning and Spatialization

a1, a2, a3, a4	**pan**	asig, kx, ky, ifn[, imode[, ioffset]]
a1, a2	**locsig**	asig, kdegree, kdistance, kreverbsend
a1, a2, a3, a4	**locsig**	asig, kdegree, kdistance, kreverbsend
a1, a2	**locsend**	
a1, a2, a3, a4	**locsend**	
a1, a2, a3, a4	**space**	asig, ifn, ktime, kreverbsend[, kx, ky]
a1, a2, a3, a4	**spsend**	
k1	**spdist**	ifn, ktime[, kx, ky]
aleft, aright	**hrtfer**	asig, kaz, kelev, “HRTFcompact”

Signal Modifiers: Sample Level Operators

kr	**downsamp**	asig[, iwlen]
ar	**upsamp**	ksig
ar	**interp**	ksig[, iskip]
k/ar	**integ**	k/asig[, iskip]
k/ar	**diff**	k/asig[, iskip]
k/ar	**samphold**	x/asig, k/xgate[, ival, ivstor]
i/k/ar	**ntrpol**	i/k/asig1, i/k/asig2, i/k/kpoint[, imin, imax]
ar	**fold**	asig, kincr

Zak Patch System

	zakinit	isizea, isizek
	ziw	isig, indx
	zkw	ksig, kndx
	zaw	asig, kndx
	ziwm	isig, indx[, imix]
	zkwm	ksig, kndx[, kmix]
	zawm	asig, kndx[, kmix]
ir	**zir**	indx
kr	**zkr**	kndx
ar	**zar**	kndx
ar	**zarg**	kndx, kgain
kr	**zkmod**	ksig, kzkmod
ar	**zamod**	asig, kzamod
	zkcl	kfirst, klast
	zacl	kfirst, klast

Operations Using Spectral Data-Types

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]
wsig	**spectrum**	xsig, iprd, iocts, ifrqa, iq[, ihann, idbout, idsprd, idsinrs]

Signal Input and Output: Input

```
a1          in
a1, a2      ins
a1, a2,     inq
a3, a4
a1, a2,     inh
a3, a4,
a5, a6
a1, a2,     ino
a3, a4,
a5, a6,
a7, a8
a1          soundin     ifilcod[, iskptim[, iformat]]
a1, a2      soundin     ifilcod[, iskptim[, iformat]]
a1, a2,     soundin     ifilcod[, iskptim[, iformat]]
a3, a4
a1[,a2      diskin      ifilcod, kpitch[, iskiptim[, iwraparound[, iformat]]]
[,a3,a4]]
```

Signal Input and Output: Output

```
          out           asig
          outs1         asig
          outs2         asig
          outs          asig1, asig2
          outq1         asig
          outq2         asig
          outq3         asig
          outq4         asig
          outq          asig1, asig2, asig3, asig4
          outh          asig1, asig2, asig3, asig4, asig5, asig6
          outo          asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8
          soundout      asig1, ifilcod[, iformat]
          soundouts     asig1, asig2, ifilcod[, iformat] (**Not implemented***)
```

Csound Quick Reference

Signal Input and Output: File I/O

	dumpk	ksig, ifilname, iformat, iprd
	dumpk2	ksig1, ksig2, ifilname, iformat, iprd
	dumpk3	ksig1, ksig2, ksig3, ifilname, iformat, iprd
	dumpk4	ksig1, ksig2, ksig3, ksig4, ifilname, iformat, iprd
ksig	**readk**	ifilname, iformat[, ipol]
k1, k2	**readk2**	ifilname, iformat[, ipol]
k1,k2,k3	**readk3**	ifilname, iformat[, ipol]
k1,k2, k3,k4	**readk4**	ifilname, iformat[, ipol]
	fout	“ifilename”, iformat, aout1[, aout2, aout3,...,aoutN]
	foutk	“ifilename”, iformat, aout1[, aout2, aout3,...,aoutN]
	fouti	ihandle, iformat, iflag, iout1[, iout2, iout3,....,ioutN]
	foutir	ihandle, iformat, iflag, iout1[, iout2, out3,....,ioutN]
ihandle	**fiopen**	“ifilename”,imode
	fin	“ifilename”, iskipframes, iformat, ain1[, ain2, ain3,...,ainN]
	fink	“ifilename”, iskipframes, iformat, kin1[, kin2, kin3,...,kinN]
	fini	“ifilename”, iskipframes, iformat, in1[, in2, in3,...,inN]
	vincr	asig, aincr
	clear	avar1[,avar2, avar3,…,avarN]

Signal Input and Output: Sound File Queries

ir	**filelen**	“ifilcod”
ir	**filesr**	“ifilcod”
ir	**filenchnls**	“ifilcod”
ir	**filepeak**	“ifilcod”[, ichnl]

Signal Input and Output: Printing and Display

print	iarg[, iarg, ...]
display	xsig, iprd[, inprds[, iwtflg]]
dispfft	xsig, iprd, iwsiz[, iwtyp[, idbouti[, iwtflg]]]
printk	kval, ispace[, itime]
printks	“txtstring”, itime, kval1, kval2, kval3, kval4
printk2	kvar[, numspaces]

Csound Quick Reference

Score Syntax: Statements

f “table number” “action time” “size” “GEN routine” arg1[arg2...arg...]

f0 “action time” (Dummy f-table for padding score sections with silence and reporting on progress of long running jobs).

b “base clock time” (Effective prior to score sorting. This time base is pre-warped.)

t 0 “initial tempo” “time in beats” “tempo2”[“time in beats” “tempo3” “time in...]

a 0 “begin time advance in beats” “duration of time advance in beats”

i “instrument number” “start” “duration” [p4 p5 p...]

s (marks end of section and restarts score counting from time 0)

m “score location name” (marks a score section with a name)

n “score location name” (named score section is re-read into the score file at this location)

r “integer repeat count” “a macro name” (begins a new repeating sections)

e (marks end of score - optional)

Score Syntax: P-Field Substitution

. (carries same p-field value from preceding “**i**” statement with like instrument #)

+ (determines current start from sum of preceeding durations by adding p2 + p3 from previous “**i**” statement. legal in p2 only.)

^+*x* (determines current start of instrument from sum of preceeding written event by adding last p2 to *x*. legal in p2 only.)

^-*x* (determines current start of instrument from sum of preceeding written event by subtracting *x* from last p2. legal in p2 only.)

np*x* (replace with p-field(x) value from next note statement illegal in p1 p2 p3.)

pp*x* (replace with p-field(x) value from previous note statement illegal in p1 p2 p3.)

< (p-field replaced by value derived from linear interpolation between previous and subsequent “anchor” values in same p-field. illegal in p1 p2 p3)

> (p-field replaced by value derived from linear interpolation between previous and subsequent “anchor” values in same p-field. illegal in p1 p2 p3)

) (p-field replaced by value derived from exponential interpolation between previous and subsequent “anchor” values in same p-field. illegal in p1 p2 p3)

((p-field replaced by value derived from exponential interpolation between previous and subsequent “anchor” values in same p-field. illegal in p1 p2 p3)

~ (p-field replaced by value derived from random value in the range between previous and subsequent “anchor” values in same p-field. illegal in p1 p2 p3)

Csound Quick Reference

Score Syntax: Expressions

[*x***+***y***]** (add value *x* to value *y* within a p-field. Note expressions must be in **[**brackets**]**)

[*x***-***y***]** (subtract value *y* from value *x* within a p-field. Note expressions must be in **[**brackets**]**)

[*x****** *y***]** (multiply value *x* by value *y* within a p-field. Note expressions must be in **[**brackets**]**)

[*x***/***y***]** (divide value *x* by value *y* within a p-field. Note expressions must be in **[**brackets**]**)

[*x***%***y***]** (value *x* remainder value *y* within a p-field. Note expressions must be in **[**brackets**]**)

[*x***^***y***]** (power of value *x* to value *y* within a p-field. Note expressions must be in **[**brackets**]**)

[@*x***]** (next power-of-two greater than or equal to *x*. Note expressions must be in **[**brackets**]**)

[@@*x***]** (next power-of-two-plus-one greater than or equal to *x*. Note expressions must be in **[**brackets**]**)

Score Syntax: Macros

#define NAME # replacement text #
#define NAME(a'b'c) # replacement text #
$NAME.
#undef NAME
#include "filename"

Csound Quick Reference

GEN Routines: Sine/Cosine Generators

```
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
```

GEN Routines: Line/Exponential Segment Generators

```
f  #  time  size  5   a        n1        b        n2     c     …
f  #  time  size  6   a        n1        b        n2     c     n3    d     …
f  #  time  size  7   a        n1        b        n2     c     …
f  #  time  size  8   a        n1        b        n2     c     n3    d     …
f  #  time  size  16  beg      dur       type     end
f  #  time  size  25  x1       y1        x2       y2     x3    ...
f  #  time  size  27  x1       y1        x2       y2     x3    ...
```

GEN Routines: File Access

```
f  #  time  size  1   filcod   skiptime  format   channel
f  #  time  size  23  “filename.txt”
f  #  time  0     28  filcod
```

GEN Routines: Numeric Value Access

```
f  #  time  size  2   v1       v2        v3     …
f  #  time  size  17  x1       a         x2     b     x3    c     …
```

GEN Routines: Window Functions

```
f  #  time  size  20  window   max       op
```

GEN Routines: Random Functions

```
f  #  time  size  21  type     lvl       arg1     arg2
```

GEN Routines: Waveshaping

```
f  #  time  size  3   xval1    xval2     c0       c1     c2    …     cn
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
```

GEN Routines: Amplitude Scaling

```
f  #  time  size  4   source#  sourcemode
f  #  time  size  12  xint
```

Command Line Flags: Generic

-I	i-time only orch run
-n	no sound onto disk
-i *fnam*	sound input filename *fnam*
-o *fnam*	sound output filename *fnam*
-b *N*	sample frames (or -kprds) per software sound I/O buffer
-B *N*	samples per hardware sound I/O buffer
-A	create an AIFF format output soundfile
-W	create a WAV format output soundfile
-J	create an IRCAM format output soundfile
-h	no header on output soundfile
-c	8-bit signed_char sound samples
-a	alaw sound samples
-8	8-bit unsigned_char sound samples
-u	ulaw sound samples
-s	short_int sound samples
-l	long_int sound samples
-f	float sound samples
-r *N*	orchestra srate override
-k *N*	orchestra krate override
-v	verbose orch translation
-m *N*	tty message level. *N* = Sum of: 1 = note amps, 2 = out-of-range msg, 4 = warnings
-d	suppress all displays
-g	suppress graphics, use ASCII displays
-G	create Postscript displays of any display
-S	score is in Scot format
-x *fnam*	extract from score.srt using extract file *fnam*
-t *N*	use uninterpreted beats of the score, initially at tempo *N*
-L *dnam*	read Line-oriented real-time score events from device *dnam*
-M *dnam*	read MIDI real-time events from device *dnam*
-F *fnam*	read MIDI file event stream from file *fnam*
-P *N*	MIDI sustain pedal threshold (*N* = 0 - 128)
-R	continually rewrite header while writing soundfile (WAV/AIFF)
-H/H1	print a heartbeat character at each soundfile write
-H2	generates a “.” every time a buffer is written.
-H3	reports the size in seconds of the output.
-H4	sounds a bell for every buffer of the output written.
-N	notify (ring the bell) when score or MIDI file is done
-T	terminate the performance when MIDI file is done
-D	defer GEN01 soundfile loads until performance time
-z	List opcodes in this version
-z1	List opcodes and arguments in this version
-- *lognam*	Log all text output to *lognam*
-j fnam	Derive console messages from database *fnam*
-K	Switch off peak chunks.

Csound Quick Reference

Command Line Flags: Utility Invocation

-U sndinfo run utility program **sndinfo**
-U hetro run utility program **hetro**
-U lpanal run utility program **lpanal**
-U pvanal run utility program **pvanal**
-U cvanal run utility program **cvanal**
-U pvlook run utility program **pvlook**
-C use Cscore processing of scorefile

Command Line Flags: PC and Windows-Specific

-j *num* set the number of console text rows (default 25)
-J *num* set the number of console text columns (default 80)
-K *num* enables MIDI IN. *num* (optional) = MIDI IN port device id number
-q *num* WAVE OUT device id number (use only if more WAVE devices are installed)
-p *num* number of WAVE OUT buffers (default 4; max. 40)
-O suppresses all console text output for better real-time performance
-e allows any sample rate (to use only with WAVE cards supporting this feature)
-y doesn't wait for keypress on exit
-E allows graphic display for WCSHELL by Riccardo Bianchini
-Q *num* enable MIDI OUT. *num* (optional) = MIDI OUT port device id number
-Y suppresses real-time WAVE OUT for better MIDI OUT timing performance
-* yields control to the system until audio output buffer is full

Command Line Flags: Macintosh-Specific

-q *sampdir* set the directory for finding samples
-Q *analdir* set the directory for finding analyses
-X *snddir* set the directory for saving sound files
-V *num* set screen buffer size
-E *num* set number of graphs saved
-p play on finishing
-e *num* set rescaling factor
-w set recording of MIDI data
-y *num* set rate for progress display
-Y *num* set rate for profile display

Utilities: Analysis File Generation

hetro	-sr *n*	infilename	outfilename	Hetrodyne analysis *sample rate*
	-c *n*	infilename	outfilename	Hetrodyne analysis *channel number*
	-b *n*	infilename	outfilename	Hetrodyne analysis *segment begin time*
	-d *n*	infilename	outfilename	Hetrodyne analysis *segment duration*
	-f *n*	infilename	outfilename	Hetrodyne analysis *beginning frequency*
	-h *n*	infilename	outfilename	Hetrodyne analysis *number of partials*
	-M *n*	infilename	outfilename	Hetrodyne analysis *maximum amplitude*
	-m *n*	infilename	outfilename	Hetrodyne analysis *minimum amplitude*
	-n *n*	infilename	outfilename	Hetrodyne analysis *number of breakpoints*
	-l *n*	infilename	outfilename	Hetrodyne analysis use third order low-pass filter with f_c of *n*
lpanal	-a	infilename	outfilename	LPC analysis write filter pole instead of coeffecients
	-s *n*	infilename	outfilename	LPC analysis *sample rate*
	-c *n*	infilename	outfilename	LPC analysis *channel number*
	-b *n*	infilename	outfilename	LPC analysis *segment begin time*
	-d *n*	infilename	outfilename	LPC analysis *segment duration*
	-p *n*	infilename	outfilename	LPC analysis *number of poles*
	-h *n*	infilename	outfilename	LPC analysis *hop size* in samples
	-C *s*	infilename	outfilename	LPC analysis *text string* for comments
	-P *n*	infilename	outfilename	LPC analysis *lowest frequency*
	-Q *n*	infilename	outfilename	LPC analysis *highest frequency*
	-v *n*	infilename	outfilename	LPC analysis *verbosity level* of terminal messages
pvanal	-s *n*	infilename	outfilename	STFT analysis *sample rate*
	-c *n*	infilename	outfilename	STFT analysis *channel number*
	-b *n*	infilename	outfilename	STFT analysis *segment begin time*
	-d *n*	infilename	outfilename	STFT analysis *segment duration*
	-n *n*	infilename	outfilename	STFT analysis *frame size*
	-w *n*	infilename	outfilename	STFT analysis *window overlap factor*
	-h *n*	infilename	outfilename	STFT analysis *hop size* in samples
cvanal	-s *n*	infilename	outfilename	FFT analysis *sample rate*
	-c *n*	infilename	outfilename	FFT analysis *channel number*
	-b *n*	infilename	outfilename	FFT analysis *segment begin time*
	-d *n*	infilename	outfilename	FFT analysis *segment duration*

Utilities: File Queries

sndinfo		soundfilename	get info about one or more sound files *soundfilename*
pvlook	-bb *n*	infilename	STFT analysis file formatted text output *beginning bin number*
	-eb *n*	infilename	STFT analysis file formatted text output *ending bin number*
	-bf *n*	infilename	STFT analysis file formatted text output *beginning frame number*
	-ef *n*	infilename	STFT analysis file formatted text output *ending frame number*
	-i	infilename	STFT analysis file formatted text output as integers