

# **The Canonical Csound Reference Manual**

**Barry Vercoe, MIT Media Lab  
Other Contributors**

**Edited by John ffitch, Jean Piché, Peter Nix, Richard Boulanger,  
Rasmus Ekman, David Boothe, Kevin Conder, Steven Yi, Michael Go-  
gins, and Andrés Cabrera**

---

# **The Canonical Csound Reference Manual**

by Barry Vercoe, Other Contributors , John ffitch, Jean Piché, Peter Nix, Richard Boulanger,  
Rasmus Ekman, David Boothe, Kevin Conder, Steven Yi, Michael Gogins, and Andrés Cabrera  
Copyright © 1986, 1992 Massachusetts Institute of Technology

---

---

---

# Table of Contents

Preface .....	xxiii
Preface to the Csound Manual .....	xxiii
Copyright Notice .....	xxiv
Acknowledgements .....	xxiv
History of the Canonical Csound Reference Manual .....	xxv
Licenses .....	xxv
I. Overview .....	27
Introduction .....	30
Recent Developments .....	31
New Features in Csound 5 .....	32
Features of CsoundVST .....	34
Csound Links .....	35
Installing .....	36
Downloading .....	36
Configuring .....	36
CsoundVST .....	36
Using .....	38
Real-Time Audio .....	38
Windows with ASIO .....	38
Using Csound5 with JACK .....	38
Csound .....	39
The <code>csound</code> Command .....	39
CsoundVST .....	39
Standalone .....	40
Python Scripting .....	40
VST Plugin .....	42
Building Csound .....	44
To Do .....	46
The Csound Command .....	47
Order of Precedence .....	47
Description .....	47
Command-line Flags .....	48
Csound Environment Variables .....	54
Unified File Format for Orchestras and Scores .....	57
Description .....	57
Structured Data File Format .....	57
Command Line Parameter File .....	59
Score File Preprocessing .....	59
The Extract Feature .....	59
Independent Pre-Processing with Scsort .....	60
Syntax of the Orchestra .....	61
Directories and Files .....	61
Nomenclature .....	61
Orchestra Statement Types .....	62
Constants and Variables .....	62
Expressions .....	63
Orchestra Header Statements .....	64
Instrument and Opcode Block Statements .....	64
Variable Initialization .....	64
Named Instruments .....	64
The Standard Numeric Score .....	68
Preprocessing of Standard Scores .....	68
Carry .....	68
Tempo .....	68
Sort .....	68
N.B. ....	69
Next-P and Previous-P Symbols .....	69

Ramping .....	69
Score Macros .....	70
Multiple File Score .....	72
Evaluation of Expressions .....	73
Score Statements .....	74
Sine/Cosine Generators .....	74
Line/Exponential Segment Generators .....	74
File Access GEN Routines .....	75
Numeric Value Access GEN Routines .....	75
Window Function GEN Routines .....	75
Random Function GEN Routines .....	75
Waveshaping GEN Routines .....	75
Amplitude Scaling GEN Routines .....	75
Mixing GEN Routines .....	75
TclCsound .....	76
The Tcl interpreter: cstclsh .....	76
Cswish: the windowing shell .....	76
A Csound server .....	77
A Scripting Environment .....	78
TclCsound as a language wrapper .....	79
TclCsound Command Reference .....	80
II. Opcodes Overview .....	82
Signal Generators .....	85
Additive Synthesis/Resynthesis .....	85
Basic Oscillators .....	85
Dynamic Spectrum Oscillators .....	85
FM Synthesis .....	86
Granular Synthesis .....	86
Linear and Exponential Generators .....	86
Envelope Generators .....	87
Models and Emulations .....	87
Phasors .....	88
Random (Noise) Generators .....	88
Sample Playback .....	89
Soundfonts .....	89
Scanned Synthesis .....	90
Table Access .....	92
Wave Terrain Synthesis .....	92
Waveguide Physical Modeling .....	92
Signal Input and Output .....	94
File Input and Output .....	94
Signal Input .....	94
Signal Output .....	94
Printing and Display .....	94
Sound File Queries .....	95
Signal Modifiers .....	96
Amplitude Modifiers and Dynamic processing .....	96
Convolution and Morphing .....	96
Delay .....	96
Panning and Spatialization .....	97
Reverberation .....	98
Sample Level Operators .....	98
Signal Limiters .....	99
Special Effects .....	99
Standard Filters .....	99
Specialized Filters .....	101
Waveguides .....	101
Instrument Control .....	102
Clock Control .....	102
Conditional Values .....	102
Duration Control Statements .....	102
Introduction to FLTK Widgets and GUI controllers .....	102
FLTK Containers .....	105

FLTK Valuator	105
Other FLTK Widgets	105
Instrument Invocation	106
Macros	106
Program Flow Control	106
Real-time Performance Control	107
Reinitialization	107
Sensing and Control	107
Sub-instrument Control	109
Time Reading	109
Function Table Control	110
Table Queries	110
Read/Write Operations	110
Table Reading with Dynamic Selection	110
Mathematical Operations	112
Amplitude Converters	112
Arithmetic and Logic Operations	112
Mathematical Functions	112
Opcode Equivalents of Functions	112
Random Functions	113
Trigonometric Functions	113
Pitch Converters	114
Functions	114
Tuning Opcodes	114
Real-time MIDI Support	115
MIDI input	115
MIDI Message Output	115
Generic Input and Output	115
Converters	116
Event Extenders	116
Note-on/Note-off Output	116
System Realtime Messages	116
Slider Banks	116
Spectral Processing	118
Short-time Fourier Transform (STFT) Resynthesis	118
Linear Predictive Coding (LPC) Resynthesis	118
Non-standard Spectral Processing	119
Tools for Real-time Spectral Processing (pvs opcodes)	119
ATS Spectral Processing	120
Loris Opcodes	121
Strings	125
String Conversion Opcodes	125
Vectorial Opcodes	127
Tables of vectors operators	127
Operations Between a Vectorial and a Scalar Signal	127
Operations Between two Vectorial Signals	127
Vectorial Envelope Generators	127
Limiting and wrapping of vectorial control signals	128
Vectorial Control-rate Delay Paths	128
Vectorial Random Signal Generators	128
Zak Patch System	130
DSSI and LADSPA for Csound	131
VST for Csound	132
OSC and Csound	133
	133
Mixer Opcodes	134
Python Opcodes	135
Introduction	135
Orchestra Syntax	135
III. Reference	137
Orchestra Opcodes and Operators	154
!=	155
#define	157

#include .....	160
#undef .....	162
#ifdef .....	163
\$NAME .....	165
% .....	168
&& .....	170
> .....	172
>= .....	174
< .....	176
<= .....	178
* .....	180
+ .....	182
- .....	184
/ .....	186
= .....	188
== .....	190
^ .....	192
round .....	194
.....	195
0dbfs .....	197
& .....	199
.....	200
⊃ .....	201
# .....	202
a .....	203
abetarand .....	205
abexprnd .....	206
abs .....	207
acauchy .....	209
active .....	210
adsr .....	213
adsyn .....	216
adsynt .....	218
adsynt2 .....	221
aexprand .....	222
aftouch .....	223
agauss .....	225
agogobel .....	226
alinrand .....	227
alpass .....	228
ampdb .....	230
ampdbfs .....	232
ampmidi .....	234
apcauchy .....	236
apoisson .....	237
apow .....	238
areson .....	239
aresonk .....	241
atone .....	242
atonek .....	244
atonex .....	245
atrirand .....	246
ATSadd .....	247
ATSaddnz .....	249
ATSbufread .....	251
ATScross .....	253
ATSinfo .....	255
ATSinterpread .....	257
ATSread .....	258
ATSreadnz .....	260
ATSpartialtap .....	262
ATSinnoi .....	263
aunirand .....	265

aweibull .....	266
babo .....	267
balance .....	271
bamboo .....	273
barmodel .....	275
bbcutm .....	277
bbcuts .....	282
betarand .....	284
bexprnd .....	286
bformenc .....	288
bformdec .....	290
binit .....	292
biquad .....	293
biquada .....	295
birnd .....	296
bqrez .....	298
butbp .....	300
butbr .....	301
buthp .....	302
butlp .....	303
butterbp .....	304
butterbr .....	306
butterhp .....	308
butterlp .....	310
button .....	312
buzz .....	313
cabasa .....	315
cauchy .....	317
ceil .....	319
cent .....	320
cggoto .....	322
chanctrl .....	324
changed .....	325
chani .....	327
chano .....	328
checkbox .....	329
chn .....	331
chnclear .....	333
chnexport .....	334
chnget .....	336
chnmix .....	338
chnparams .....	339
chnset .....	340
cigoto .....	342
ckgoto .....	344
clear .....	346
clfilt .....	347
clip .....	350
clock .....	352
clockoff .....	353
clockon .....	354
cngoto .....	355
comb .....	357
compress .....	359
control .....	361
convle .....	362
convolve .....	363
cos .....	366
cosh .....	368
cosinv .....	370
cps2pch .....	372
cpsmidi .....	376
cpsmidib .....	378



cpsoct .....	380
cpspch .....	382
cpstmid .....	384
cpstun .....	386
cpstuni .....	389
cpsxpcch .....	392
cpuprc .....	396
cross2 .....	398
crunch .....	400
ctrl14 .....	402
ctrl21 .....	404
ctrl7 .....	406
ctrlinit .....	408
cusernd .....	409
dam .....	411
db .....	414
dbamp .....	416
dbfsamp .....	418
dcblock .....	420
dconv .....	422
delay .....	424
delay1 .....	426
delayk .....	427
delayr .....	429
delayw .....	430
deltap .....	432
deltap3 .....	434
deltapi .....	436
deltapn .....	438
deltapx .....	440
deltapxw .....	442
denorm .....	444
diff .....	445
diskin .....	447
diskin2 .....	450
dispfft .....	453
display .....	455
distort .....	457
distort1 .....	459
divz .....	461
downsamp .....	463
dripwater .....	465
dssiactivate .....	467
dssiaudio .....	468
dssictls .....	469
dssiinit .....	470
dssilist .....	473
dumpk .....	474
dumpk2 .....	476
dumpk3 .....	478
dumpk4 .....	480
dusernd .....	482
else .....	484
elseif .....	485
endif .....	486
endin .....	487
endop .....	489
envlpx .....	490
envlpxr .....	493
event .....	495
event_i .....	498
exitnow .....	499
exp .....	500

expon .....	502
exprand .....	504
expseg .....	506
expsega .....	508
expsegr .....	510
ficlose .....	512
filelen .....	513
filenchnls .....	515
filepeak .....	517
filesr .....	519
filter2 .....	521
fin .....	523
fini .....	524
fink .....	525
fiopen .....	526
flanger .....	527
flashtxt .....	529
FLbox .....	531
FLbutBank .....	535
FLbutton .....	537
FLcolor .....	541
FLcolor2 .....	542
FLcount .....	543
FLgetsnap .....	546
FLgroup .....	547
FLgroupEnd .....	549
FLgroupEnd .....	550
FLhide .....	551
FLjoy .....	552
FLkeyb .....	555
FLknob .....	556
FLlabel .....	559
FLloadsnap .....	561
flooper .....	562
floor .....	564
FLpack .....	565
FLpackEnd .....	568
FLpack_end .....	569
FLpanel .....	570
FLpanelEnd .....	573
FLpanel_end .....	574
FLprintk .....	575
FLprintk2 .....	576
FLroller .....	577
FLrun .....	580
FLsavesnap .....	581
FLscroll .....	583
FLscrollEnd .....	586
FLscroll_end .....	587
FLsetAlign .....	588
FLsetBox .....	589
FLsetColor .....	591
FLsetColor2 .....	593
FLsetFont .....	594
FLsetPosition .....	596
FLsetSize .....	597
FLsetsnap .....	598
FLsetText .....	600
FLsetTextColor .....	601
FLsetTextSize .....	602
FLsetTextType .....	603
FLsetVal_i .....	606
FLsetVal .....	607

FLshow .....	608
FLslidBnk .....	609
FLslider .....	612
FLtabs .....	616
FLtabsEnd .....	621
FLtabs_end .....	622
FLtext .....	623
FLupdate .....	626
fluidAllOut .....	627
fluidCCi .....	630
fluidCCk .....	631
fluidControl .....	632
fluidEngine .....	634
fluidLoad .....	637
fluidNote .....	641
fluidOut .....	644
fluidProgramSelect .....	647
FLvalue .....	650
fmb3 .....	652
fmbell .....	654
fmmetal .....	656
fmpercfl .....	659
fmrhode .....	661
fmvoice .....	664
fmwurlie .....	666
fof .....	669
fof2 .....	672
fofilter .....	674
fog .....	676
fold .....	678
follow .....	680
follow2 .....	682
foscil .....	684
foscili .....	686
fout .....	688
fouti .....	692
foutir .....	694
foutk .....	696
fprintks .....	698
fprints .....	700
frac .....	702
freeverb .....	704
ftchnls .....	706
ftconv .....	708
ftfree .....	711
ftgen .....	712
ftgentmp .....	714
ftlen .....	715
ftload .....	717
ftloadk .....	718
ftlptim .....	719
ftmorf .....	721
ftsav .....	723
ftsavk .....	725
ftsr .....	726
gain .....	728
gauss .....	729
gbuzz .....	731
getcfcg .....	733
gogobel .....	734
goto .....	736
grain .....	738
grain2 .....	740

grain3 .....	744
granule .....	749
guiro .....	752
harmon .....	754
hilbert .....	757
hrtfer .....	761
hsboscil .....	763
i .....	766
ibetarand .....	767
ibexprnd .....	768
icauchy .....	769
ictrl14 .....	770
ictrl21 .....	771
ictrl7 .....	772
iexprand .....	773
if .....	774
igauss .....	778
igoto .....	779
ihold .....	781
ilinrand .....	783
imidic14 .....	784
imidic21 .....	785
imidic7 .....	786
in .....	787
in32 .....	788
inch .....	789
inh .....	790
init .....	791
initc14 .....	792
initc21 .....	793
initc7 .....	794
ino .....	795
inq .....	796
ins .....	797
instimek .....	798
instimes .....	799
instr .....	800
int .....	802
integ .....	804
interp .....	806
invalue .....	809
inx .....	810
inz .....	811
ioff .....	812
ion .....	813
iondur .....	814
iondur2 .....	815
ioutat .....	816
ioutc .....	817
ioutc14 .....	818
ioutpat .....	819
ioutpb .....	820
ioutpc .....	821
ipcauchy .....	822
ipoisson .....	823
ipow .....	824
is16b14 .....	825
is32b14 .....	826
islider16 .....	827
islider32 .....	828
islider64 .....	829
islider8 .....	830
itablecopy .....	831

itablegpw .....	832
itablemix .....	833
itablew .....	834
itrirand .....	835
iunirand .....	836
iweibull .....	837
jitter .....	838
jitter2 .....	840
jspline .....	842
k .....	843
kbetarand .....	844
kbexprnd .....	845
kcauchy .....	846
kdump .....	847
kdump2 .....	848
kdump3 .....	849
kdump4 .....	850
kexprand .....	851
kfilter2 .....	852
kgauss .....	853
kgoto .....	854
klinrand .....	856
kon .....	857
koutat .....	858
koutc .....	859
koutc14 .....	860
koutpat .....	861
koutpb .....	862
koutpc .....	863
kpcauchy .....	864
kpoisson .....	865
kpow .....	866
kr .....	867
kread .....	868
kread2 .....	869
kread3 .....	870
kread4 .....	871
ksmps .....	872
ktableseg .....	873
ktrirand .....	874
kunirand .....	875
kweibull .....	876
lfo .....	877
limit .....	879
line .....	880
linen .....	882
linenr .....	883
lineto .....	884
linrand .....	885
linseg .....	887
linsegr .....	889
locsend .....	891
locsig .....	893
log .....	896
log10 .....	898
logbtwo .....	900
loop .....	902
loopseg .....	904
loopsegp .....	906
lorenz .....	907
lorisread .....	910
lorismorph .....	912
lorisplay .....	913

loscil .....	914
loscil3 .....	917
lowpass2 .....	920
lowres .....	922
lowresx .....	924
lpf18 .....	926
lpfreson .....	928
lphasor .....	929
lpinterp .....	930
lposcil .....	931
lposcil3 .....	932
lpread .....	933
lpreson .....	935
lpshold .....	936
lpsholdp .....	938
lpslot .....	939
mac .....	940
maca .....	941
madsr .....	942
mandel .....	945
mandol .....	946
marimba .....	948
massign .....	950
max .....	952
maxabs .....	953
maxabsaccum .....	954
maxaccum .....	955
maxalloc .....	956
max_k .....	958
mclock .....	959
mdelay .....	960
metro .....	961
midic14 .....	963
midic21 .....	965
midic7 .....	967
midichannelaftertouch .....	969
midichn .....	972
midicontrolchange .....	975
midictrl .....	977
mididefault .....	978
midiin .....	980
midinoteoff .....	981
midinoteoncps .....	984
midinoteonkey .....	987
midinoteonoct .....	990
midinoteonpch .....	993
midion .....	996
midion2 .....	997
midiout .....	998
midipitchbend .....	999
midipolyaftertouch .....	1002
midiprogramchange .....	1004
miditempo .....	1006
min .....	1007
minabs .....	1008
minabsaccum .....	1009
minaccum .....	1010
mirror .....	1011
MixerSetLevel .....	1012
MixerGetLevel .....	1014
MixerSend .....	1015
MixerReceive .....	1017
MixerClear .....	1019

monitor .....	1020
moog .....	1021
moogladder .....	1023
moogvcf .....	1025
moscil .....	1027
mpulse .....	1028
mrtmsg .....	1030
multitap .....	1031
mute .....	1032
mxadsr .....	1034
nchnls .....	1036
nestedap .....	1037
nlfilt .....	1040
noise .....	1042
noteoff .....	1044
noteon .....	1045
noteondur .....	1046
noteondur2 .....	1047
notnum .....	1048
nreverb .....	1050
nrpn .....	1053
nsamp .....	1054
nstrnum .....	1056
ntrpol .....	1057
octave .....	1058
octcps .....	1060
octmidi .....	1062
octmidib .....	1064
octpch .....	1066
opcode .....	1068
OSCsend .....	1073
OSCinit .....	1075
OSClisten .....	1076
oscbnk .....	1078
oscil .....	1083
oscil1 .....	1085
oscil1i .....	1086
oscil3 .....	1087
oscili .....	1089
oscilikt .....	1091
osciliktp .....	1093
oscilikts .....	1095
osciln .....	1097
oscils .....	1098
oscilx .....	1100
out .....	1101
out32 .....	1102
outc .....	1103
outch .....	1104
outh .....	1105
outiat .....	1106
outic .....	1107
outic14 .....	1108
outipat .....	1109
outipb .....	1110
outipc .....	1111
outkat .....	1112
outkc .....	1113
outkc14 .....	1114
outkpat .....	1115
outkpb .....	1116
outkpc .....	1117
outo .....	1118

outq .....	1119
outq1 .....	1120
outq2 .....	1121
outq3 .....	1122
outq4 .....	1123
outs .....	1124
outs1 .....	1125
outs2 .....	1126
outvalue .....	1127
outx .....	1128
outz .....	1129
p .....	1130
pan .....	1132
pareq .....	1134
partials .....	1137
pcauchy .....	1139
pchbend .....	1141
pchmidi .....	1143
pchmidib .....	1145
pchoct .....	1147
pconvolve .....	1149
peak .....	1152
peakk .....	1154
pgmassign .....	1155
phaser1 .....	1159
phaser2 .....	1162
phasor .....	1165
phasorbnk .....	1167
pinkish .....	1169
pitch .....	1172
pitchamdf .....	1175
planet .....	1178
pluck .....	1180
poisson .....	1182
polyaft .....	1184
port .....	1186
portk .....	1187
poscil .....	1188
poscil3 .....	1190
pow .....	1192
powoftwo .....	1194
prealloc .....	1196
print .....	1198
printf .....	1200
printk .....	1201
printk2 .....	1203
printks .....	1205
prints .....	1208
product .....	1210
pset .....	1211
puts .....	1212
pvadd .....	1213
pvbufread .....	1216
pvcross .....	1218
pvinterp .....	1220
pvoc .....	1222
pvread .....	1224
pvsadsyn .....	1226
pvsanal .....	1228
pvsarp .....	1231
pvscross .....	1233
pvscent .....	1234
pvsdemix .....	1235



pvsfread .....	1237
pvsfreeze .....	1238
pvsftr .....	1239
pvsftw .....	1241
pvsifd .....	1243
pvsinfo .....	1245
pvsinit .....	1246
pvsmaska .....	1247
pvsynth .....	1249
pvscale .....	1251
pvshift .....	1253
pvmix .....	1255
pvsMOOTH .....	1256
pvsfilter .....	1258
pvsblur .....	1260
pvsTencil .....	1261
pvsvoc .....	1263
pyassign Opcodes .....	1265
pycall Opcodes .....	1266
pyeval Opcodes .....	1269
pyexec Opcodes .....	1270
pyinit Opcodes .....	1273
pyrun Opcodes .....	1274
rand .....	1276
randh .....	1278
randi .....	1280
random .....	1282
randomh .....	1284
randomi .....	1286
rbjeq .....	1288
readclock .....	1291
readk .....	1293
readk2 .....	1295
readk3 .....	1297
readk4 .....	1299
reinit .....	1301
release .....	1303
repluck .....	1305
reson .....	1307
resonk .....	1309
resonr .....	1310
resonx .....	1313
resonxk .....	1314
resony .....	1315
resonz .....	1317
resyn .....	1319
reverb .....	1321
reverb2 .....	1323
reverbSc .....	1324
reZzy .....	1326
rigoto .....	1328
rireturn .....	1329
rms .....	1331
rnd .....	1332
rnd31 .....	1334
rspline .....	1339
rtclock .....	1340
s16b14 .....	1342
s32b14 .....	1344
samphold .....	1346
sandpaper .....	1347
scanhammer .....	1349
scans .....	1350

scantable .....	1352
scanu .....	1354
schedkwhen .....	1356
schedkwhennamed .....	1358
schedule .....	1359
schedwhen .....	1361
seed .....	1363
sekere .....	1364
semitone .....	1366
sense .....	1368
sensekey .....	1369
seqtime .....	1371
seqtime2 .....	1373
setctrl .....	1375
setksmps .....	1377
sfilist .....	1379
sfinstr .....	1380
sfinstr3 .....	1382
sfinstr3m .....	1384
sfinstrm .....	1386
sfload .....	1388
sfpassign .....	1389
sfplay .....	1390
sfplay3 .....	1392
sfplay3m .....	1394
sfplaym .....	1396
sfplist .....	1398
sfpreset .....	1399
shaker .....	1400
sin .....	1402
sinh .....	1404
sininv .....	1406
sinsyn .....	1408
sleighbells .....	1410
slider16 .....	1412
slider16f .....	1414
slider32 .....	1416
slider32f .....	1418
slider64 .....	1420
slider64f .....	1422
slider8 .....	1424
slider8f .....	1426
sndloop .....	1428
sndwarp .....	1430
sndwarpst .....	1434
soundin .....	1437
soundout .....	1440
soundouts .....	1441
space .....	1442
spat3d .....	1446
spat3di .....	1454
spat3dt .....	1458
spdist .....	1462
specaddm .....	1466
specdiff .....	1467
specdisp .....	1468
specfilt .....	1469
spechist .....	1470
specptrk .....	1471
specscal .....	1473
specsum .....	1474
spectrum .....	1475
splitrig .....	1477

spsend .....	1479
sprintf .....	1482
sqrt .....	1483
sr .....	1485
statevar .....	1486
stix .....	1488
strchar .....	1490
strchark .....	1491
strcpy .....	1492
strcpyk .....	1493
strcat .....	1494
strcatk .....	1495
strcmp .....	1496
strcmpk .....	1497
streson .....	1498
strget .....	1500
strindex .....	1501
strindexk .....	1502
strlen .....	1503
strlenk .....	1504
strlower .....	1505
strlowerk .....	1506
strrindex .....	1507
strrindexk .....	1508
strset .....	1509
strsub .....	1510
strsubk .....	1511
strtod .....	1512
strtodk .....	1513
strtol .....	1514
strtolk .....	1515
strupper .....	1516
strupperk .....	1517
subinstr .....	1518
subinstrinit .....	1521
sum .....	1522
svfilter .....	1523
syncgrain .....	1525
timedseq .....	1527
tb .....	1529
tab .....	1532
tabrec .....	1533
table .....	1534
table3 .....	1536
tablecopy .....	1537
tablegpw .....	1538
tablei .....	1539
tableicopy .....	1540
tableigpw .....	1541
tableikt .....	1542
tableimix .....	1544
tableiw .....	1546
tablekt .....	1548
tablemix .....	1550
tableng .....	1552
tablera .....	1554
tableseg .....	1557
tablew .....	1558
tablewa .....	1561
tablewkt .....	1564
tablexkt .....	1566
tablexseg .....	1568
tambourine .....	1569

tan .....	1571
tanh .....	1573
taninv .....	1575
taninv2 .....	1577
tbvcf .....	1579
tempest .....	1582
tempo .....	1584
tempoval .....	1586
tigoto .....	1588
timeinstk .....	1589
timeinsts .....	1591
timek .....	1593
times .....	1595
timeout .....	1597
tival .....	1598
tlineto .....	1599
tone .....	1600
tonek .....	1601
tonex .....	1602
tradsyn .....	1603
transeg .....	1605
trcross .....	1606
trfilter .....	1608
trhighest .....	1610
trigger .....	1611
trigseq .....	1613
trirand .....	1615
trlowest .....	1617
trmix .....	1618
trscale .....	1619
trshift .....	1620
trsplit .....	1621
turnoff .....	1623
turnoff2 .....	1625
turnon .....	1626
unirand .....	1627
upsamp .....	1629
urd .....	1630
vadd .....	1631
vaddv .....	1632
valpass .....	1633
vbap16 .....	1634
vbap16move .....	1636
vbap4 .....	1638
vbap4move .....	1640
vbap8 .....	1642
vbap8move .....	1644
vbaplsinit .....	1646
vbapz .....	1648
vbapzmove .....	1650
vcella .....	1652
vco .....	1653
vco2 .....	1656
vco2ft .....	1659
vco2ift .....	1661
vco2init .....	1662
vcomb .....	1664
vcopy .....	1665
vcopy_i .....	1667
vdelay .....	1669
vdelay3 .....	1671
vdelayx .....	1673
vdelayxq .....	1675

vdelayxs .....	1677
vdelayxw .....	1679
vdelayxwq .....	1681
vdelayxws .....	1683
vdivv .....	1685
vdelayk .....	1686
vecdelay .....	1687
veloc .....	1688
vexp .....	1690
vexpseg .....	1691
vexpv .....	1693
vibes .....	1694
vibr .....	1696
vibrato .....	1698
vincr .....	1700
vlimit .....	1701
vlinseg .....	1702
vlowres .....	1704
vmap .....	1706
vmirror .....	1707
vmult .....	1708
vmultv .....	1710
voice .....	1711
vport .....	1713
vpow .....	1714
vpowv .....	1715
vpvoc .....	1716
vrandh .....	1718
vrandi .....	1719
vstaudio, vstaudiog .....	1720
vstbankload .....	1721
vstedit .....	1722
vstinit .....	1723
vstinfo .....	1724
vstmidiout .....	1725
vstnote .....	1727
vstparamset, vstparamget .....	1729
vstprogset .....	1731
vsubv .....	1732
vtablei .....	1733
vtablek .....	1735
vtablea .....	1737
vtablewi .....	1738
vtablewk .....	1739
vtablewa .....	1741
vtabi .....	1743
vtabk .....	1744
vtaba .....	1745
vtabwi .....	1746
vtabwk .....	1747
vtabwa .....	1748
vwrap .....	1749
waveset .....	1750
weibull .....	1752
wgbow .....	1754
wgbowedbar .....	1756
wgbrass .....	1758
wgclar .....	1760
wgflute .....	1762
wgpluck .....	1764
wgpluck2 .....	1767
wguide1 .....	1769
wguide2 .....	1771

wrap .....	1773
wterrain .....	1774
xadsr .....	1776
xin .....	1778
xout .....	1780
xscanmap .....	1782
xscansmap .....	1783
xscans .....	1784
xscanu .....	1786
xtratim .....	1788
xyin .....	1790
zaci .....	1792
zakinit .....	1794
zamod .....	1796
zar .....	1798
zarg .....	1800
zaw .....	1802
zawm .....	1804
zfilter2 .....	1806
zir .....	1808
ziw .....	1810
ziwm .....	1812
zkci .....	1814
zkmod .....	1816
zkr .....	1818
zkw .....	1820
zkwm .....	1822
Score Statements and GEN Routines .....	1825
Score Statements .....	1825
GEN Routines .....	1843
The Utility Programs .....	1908
Directories .....	1908
Soundfile Formats .....	1908
Analysis File Generation (HETRO,LPANAL, PVANAL, CVANAL, ATSA) .....	1908
File Queries (SNDINFO) .....	1920
File Conversion (DNOISE, PVLOOK, SDIF2AD, SRCONV) .....	1922
Other Csound Utilities (MAKECSD, CS) .....	1934
Cscore .....	1945
Events, Lists, and Operations .....	1945
Writing a Main Program .....	1946
More Advanced Examples .....	1952
Compiling a Cscore Program .....	1953
Extending Csound .....	1955
Adding Unit Generators .....	1955
Creating a Builtin Unit Generator .....	1955
Adding a Plugin Unit Generator .....	1959
OENTRY Reference .....	1959
A. Pitch Conversion .....	1962
B. Sound Intensity Values .....	1966
C. Formant Values .....	1967
D. Window Functions .....	1972
E. SoundFont2 File Format .....	1977
F. Csound64 .....	1978
Glossary .....	1979
G. Quick Reference .....	1980

---

# Preface

## Table of Contents

Preface to the Csound Manual .....	xxiii
Copyright Notice .....	xxiv
Acknowledgements .....	xxiv
History of the Canonical Csound Reference Manual .....	xxv
Licenses .....	xxv

## Preface to the Csound Manual

Barry Vercoe, MIT Media Lab

by Barry L. 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 (see *Syntax of the Orchestra*) are defined in a simple syntax that invokes complex audio processing routines. A score (see *The Standard Numeric Score*) passed to this orchestra contains numerically coded pitch and control information, in standard numeric score format. Although many users are content with this format, higher level score processing languages are often convenient.

The 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.

## Copyright Notice

Copyright (c) 1986, 1992 by the Massachusetts Institute of Technology. All rights reserved.

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

Copyright (c) 2003 by Kevin Conder for modifications made to the Public Csound Reference Manual.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of this license is available in the examples sub-directory [examples/fdl.txt].

A legal notice from the Public Csound Reference Manual... “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. A Print Edition, in Adobe Acrobat format, was then maintained by *David M. Boothe*. 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.”

The Public Csound Reference Manual's last known network location was <http://www.lakewoodsound.com/csound/hypertext/manual.htm>.

The Alternative Csound Reference Manual's network location, for both the Transparent and Opaque copies, is <http://kevindumpscore.com/download.html#csound-manual>.

## Acknowledgements

In addition to the core code developed by Barry L. Vercoe at M.I.T., 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:

**Table 1. Contributors**

Mike Berry
Eli Breder
Andres Cabrera
Michael Casey
Michael Clark
Perry Cook
Sean Costello
Rasmus Ekman
Richard Dobson
Mark Dolson
Dan Ellis
Tom Erbe
John ffitch
Bill Gardner



Michael Gogins

Matt Ingalls

Richard Karpen

Victor Lazzarini

Allan Lee

David Macintyre

Gabriel Maldonado

Max Mathews

Hans Mikelson

Peter Neubäcker

Peter Nix

Jean Piché

Ville Pulkki

John Ramsdell

Marc Resibois

Rob Shaw

Paris Smaragdis

Greg Sullivan

Istvan Varga

Bill Verplank

Robin Whittle

Steven Yi

The official manual was compiled from the canonical Csound Manual sources maintained by John ffitich, Richard Boulanger, Jean Piché, Peter Nix, and David M. Boothe. The Alternative Csound Reference Manual was maintained by Kevin Conder. The Canonical Csound Reference Manual is maintained by the Csound community.

## History of the Canonical Csound Reference Manual

This manual is a product of the Csound community. The current version of the manual is based on the Alternative Csound Reference Manual, developed by Kevin Conder using *DocBook/SGML* [<http://www.docbook.org/>]. This was in itself based on the Official Csound Reference Manual (last known address: <http://www.lakewoodsound.com/csound/hypertext/manual.htm>) [<http://www.lakewoodsound.com/csound/hypertext/manual.htm>]), which was maintained by David M. Boothe.

In the winter of 2004, the manual was converted to DocBook/XML by Steven Yi to allow for more people to be able to compile and maintain the manual. The manual continues to be a community run project that depends on the contributions of developers and users to help refine the coverage and accuracy of its contents. All contributions are welcome and appreciated.

Written by Steven Yi, January 2005.

## Licenses

### Csound and CsoundVST

Csound is copyright 1991-2005 by Barry Vercoe and John ffitich.

CsoundVST is copyright 2001-2005 by Michael Gogins.

Csound and CsoundVST are free software; you can redistribute them and/or modify them under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

Csound and CsoundVST are distributed in the hope that they will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with Csound and CsoundVST; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

## Manual

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of this license is available in the doc/manual/copying.txt file.

This Csound language documentation in this manual is derived from Kevin Conder's *Alternative Csound Reference Manual*, which in turn is derived from the *Public Csound Reference Manual*.

Copyright 2003 by Kevin Conder for modifications made to the *Public Csound Reference Manual*.

Copyright 2004-2005 by Michael Gogins for modifications made to the *Alternative Csound Reference Manual*.

This legal notice is from the *Public Csound Reference Manual*: “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. A Print Edition, in Adobe Acrobat format, was then maintained by *David M. Boothe*. 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.”

The Public Csound Reference Manual's last known network location was <http://www.lakewoodsound.com/csound/hypertext/manual.htm>.

The Alternative Csound Reference Manual's network location, for both the Transparent and Opaque copies, is <http://kevindumpscore.com/download.html#csound-manual>.

The Csound and CsoundVST Manual's network location is <http://sourceforge.net/projects/csound>.

## Virtual Synthesis Technology

Virtual Synthesis Technology (VST) PlugIn interface technology by Steinberg Soft- und Hardware GmbH.

CsoundVST source code contains modified versions of source code files from the VST SDK distributed by Steinberg. *These files are to be used only for building CsoundVST.* You are *not* licensed to use these files for any other purpose. If you make a derived product based on CsoundVST or the modified VST source files herein, you *must* apply to Steinberg for your own license to use the VST SDK.

---

# Part I. Overview

---

---

## Table of Contents

Introduction .....	30
Recent Developments .....	31
New Features in Csound 5 .....	32
Features of CsoundVST .....	34
Csound Links .....	35
Installing .....	36
Downloading .....	36
Configuring .....	36
CsoundVST .....	36
Using .....	38
Real-Time Audio .....	38
Windows with ASIO .....	38
Using Csound5 with JACK .....	38
Csound .....	39
The <code>csound</code> Command .....	39
CsoundVST .....	39
Standalone .....	40
Python Scripting .....	40
VST Plugin .....	42
Building Csound .....	44
To Do .....	46
The Csound Command .....	47
Order of Precedence .....	47
Description .....	47
Command-line Flags .....	48
Csound Environment Variables .....	54
Unified File Format for Orchestras and Scores .....	57
Description .....	57
Structured Data File Format .....	57
Command Line Parameter File .....	59
Score File Preprocessing .....	59
The Extract Feature .....	59
Independent Pre-Processing with Scsort .....	60
Syntax of the Orchestra .....	61
Directories and Files .....	61
Nomenclature .....	61
Orchestra Statement Types .....	62
Constants and Variables .....	62
Expressions .....	63
Orchestra Header Statements .....	64
Instrument and Opcode Block Statements .....	64
Variable Initialization .....	64
Named Instruments .....	64
The Standard Numeric Score .....	68
Preprocessing of Standard Scores .....	68
Carry .....	68
Tempo .....	68
Sort .....	68
N.B. ....	69
Next-P and Previous-P Symbols .....	69
Ramping .....	69
Score Macros .....	70
Multiple File Score .....	72
Evaluation of Expressions .....	73
Score Statements .....	74
Sine/Cosine Generators .....	74
Line/Exponential Segment Generators .....	74

File Access GEN Routines .....	75
Numeric Value Access GEN Routines .....	75
Window Function GEN Routines .....	75
Random Function GEN Routines .....	75
Waveshaping GEN Routines .....	75
Amplitude Scaling GEN Routines .....	75
Mixing GEN Routines .....	75
TclCsound .....	76
The Tcl interpreter: cstclsh .....	76
Cswish: the windowing shell .....	76
A Csound server .....	77
A Scripting Environment .....	78
TclCsound as a language wrapper .....	79
TclCsound Command Reference .....	80

---

# Introduction

Csound is a unit generator-based, user-programmable computer music system. It was originally written by Barry Vercoe at the Massachusetts Institute of Technology in 1984 as the first C language version of this type of software. Since then Csound has received numerous contributions from researchers, programmers, and musicians from around the world.

Around 1991, John ffitch ported Csound to Microsoft DOS. Csound currently runs on many varieties of UNIX and Linux, Microsoft DOS and Windows, all versions of the Macintosh operating system including Mac OS X, and others.

There are newer computer music systems that have graphical patch editors (e.g. Max/MSP, PD, jMax, or Open Sound World), or that use more advanced techniques of software engineering (e.g. Nyquist or SuperCollider). Yet Csound still has the largest and most varied set of unit generators, is the best documented, runs on the most platforms, and is the easiest to extend. It is possible to compile Csound using double-precision arithmetic throughout for superior sound quality. In short, Csound must be considered one of the most powerful musical instruments ever created.

To make music with Csound:

1. Write an orchestra (`.orc` file) that creates instruments and signal processors by connecting unit generators (also called opcodes, in Csound-speak) using Csound's simple programming language.
2. Write a score (`.sco` file) that specifies a list of notes and other events to be rendered by the orchestra.
3. Run Csound to compile the orchestra and score, run the sorted and preprocessed score through the orchestra, and write digital audio out to a soundfile or sound card.

CsoundVST is an extended version of Csound that adds a graphical user interface, C++ and Python APIs, Python scripting, a library of Python extension modules for algorithmic composition, a VST plugin interface, and a Mathematica interface.

In addition to this "canonical" version of Csound and CsoundVST, there are other versions of Csound and other front ends for Csound, many of which can be found at <http://csounds.com>.

---

# Recent Developments

In the time since Barry Vercoe wrote the original Preface to this manual, printed above, many further contributions have been made to Csound. The current stable version of Csound is 5.02. CsoundVST is an extended version of Csound 5.

---

# New Features in Csound 5

Csound 5 begins a new major version of Csound that includes the following new features:

- Now licensed under the GNU Lesser General Public License, an open source license.
  - A new, easier to manage build system using SCons.
  - The use of widely--accepted open source libraries:
    - libsndfile for soundfile input and output.
    - PortAudio with ASIO drivers for low-latency, real-time audio input and output.
    - FLTK for graphical widgets that can be programmed in orchestra code.
    - PortMidi for real-time MIDI input and output.
- In addition, Istvan Varga has contributed native MIDI and audio drivers for Windows and Linux.
- Simplified audio buffering system.
  - Status returns from all internal functions, including opcode functions.
  - MIDI interop opcodes, that enable the same instrument definitions to be used interchangeably for either live MIDI performance or off-line, score-driven performance.
  - Plugin opcodes are working and becoming more widely accepted. Many opcodes have been moved to plugins. Most new opcodes are plugins, including:
    - The FluidSynth-based SoundFont opcodes.
    - Python opcodes allowing Python code to execute in the orchestra header or in instrument code, at *i*-rate or *k*-rate.
    - Loris opcodes for time/frequency analysis and resynthesis.
    - Control bus opcodes.
    - Audio mixer opcodes.
    - String conversion opcodes.
    - Improved Open Sound Control (OSC) opcodes.
    - The STK opcodes, consisting of Perry Cook's original Synthesis Toolkit in C++ instruments, in C++, adapted as opcodes.
    - vst4csVST plugin adapter opcodes.
  - The `OpcodeBase.hpp` header file for writing plugin opcodes in C++. This is based on the technique of static polymorphism via template inheritance.
  - Victor Lazzarini's Tcl/Tk frontends for Csound, `csclsh` and `cswish`.
  - The Csound API is becoming more standardized and more widely used. There are interfaces or wrappers to the API in the following languages:
    - C (`include csound.h`).
    - C++ (`include csound.hpp`). This API includes Csound score and orchestra file container functions.



- Python (`import csnd`).
- Java (`import csnd.*;`).
- LISP (use the CFFI file `csound5.lisp`).
- Csound is now truly re-entrant, meaning that multiple instances of Csound can run at the same time, in the same process.

John ffitch plans to replace the handwritten parser with one written using a parser generator, which should make it more bug-free and perhaps more efficient.

---

# Features of CsoundVST

CsoundVST is an extended version of Csound that runs both as a shared library (as a VST plugin or as an embedded synthesizer) and as a standalone program. Its main purposes are (a) to make it easier to extend Csound (e.g. the Loris plugin opcodes with their Python scripting), and (b) to streamline the actual use of Csound in composing, particularly for algorithmic composition, by integrating more tightly with other languages and other software.

- C++ library for algorithmic composition, based on my concept of music graphs.
- Python wrappers for the Csound API and for music graphs.
- Built-in Python interpreter. This enables one to embed orchestras and scores into Python code, and to write Csound pieces in Python, including both composition (with music graphs) and synthesis.
- Runs as a VST effect or VST plugin:
  - Loads and saves `.csd` and `.py` files in presets and banks.
  - Starts, stops, and restarts.
  - Allows one to write Csound pieces in music notation and hear the results immediately.
  - Synchronizes with other tracks in the same host, including looping.
- Runs as a standalone application.
- Runs as a Python extension module. This enables one to write Csound pieces in any Python interpreter.

---

# Csound Links

Csound's "home page" is maintained by Richard Boulanger at <http://csounds.com>.

The Csound source code is maintained by John ffitch and others at <http://www.sourceforge.net/projects/csound>. The most recent versions and precompiled packages for most platforms also can be downloaded here [[http://sourceforge.net/project/showfiles.php?group\\_id=81968](http://sourceforge.net/project/showfiles.php?group_id=81968)].

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](mailto:csound-subscribe@lists.bath.ac.uk) [<mailto:csound-subscribe@lists.bath.ac.uk>]. You can also subscribe to the digest (1 message per day) by sending an empty email to: [csound-digest-subscribe@lists.bath.ac.uk](mailto:csound-digest-subscribe@lists.bath.ac.uk) [<mailto:csound-digest-subscribe@lists.bath.ac.uk>]. Posts sent to [csound@lists.bath.ac.uk](mailto:csound@lists.bath.ac.uk) [<mailto:csound@lists.bath.ac.uk>] go to all subscribed members of the list. You can browse the csound mailing list archives here [[http://agencities.cs.bath.ac.uk/%7ebwillkie/list\\_arch.php](http://agencities.cs.bath.ac.uk/%7ebwillkie/list_arch.php)]

Similarly, the Csound-devel mailing list exists to discuss Csound development. For more information on this list, go to <http://lists.sourceforge.net/lists/listinfo/csound-devel>. Posts sent to [csound-devel@lists.sourceforge.net](mailto:csound-devel@lists.sourceforge.net) [<mailto:csound-devel@lists.sourceforge.net>] go to all subscribed members of the list.

Suspected bugs in the code may be entered using the bug tracking system at the Sourceforge bug tracker [[http://sourceforge.net/tracker/?group\\_id=81968&atid=564599](http://sourceforge.net/tracker/?group_id=81968&atid=564599)].

---

# Installing

Csound can either be built from source code, or installed from a packaged archive or installer program. This section describes how to install from an archive or installer program.

## Downloading

Archives and installers containing Csound binaries can be found at <http://sourceforge.net/projects/csound/>, on the *Files* page, in various packages.

The most complete distribution can be found in the *csoundvst* package, which contains pre-built binaries for Csound 5 and CsoundVST on Windows, made with the MinGW compiler, as well as complete sources and SCons build system for all platforms. To install from the archive, unzip it into a *csound5* directory on your computer, and configure it as explained below.

The *csoundvst* package also contains a Windows installer with the same Windows binaries, examples, and documentation. To install using the installer, simply execute it and follow the instructions provided by the installer.

## Configuring

Once you have either unpacked a binary distribution, or built Csound from sources, you will need to configure Csound so that it will run properly on your system.

On all platforms, make sure the directory or directories containing Csound's plugin libraries are in an `OPCODEDIR` or `OPCODEDIR64` environment variable depending on the precision of the compiled binary.

The Python opcodes, currently require Python 2.4 which can be downloaded from [www.python.org](http://www.python.org) [<http://www.python.org>] if it is not already on your system. You can check if it is available by typing 'python' on a command prompt or DOS window.

## Windows

On Windows, make sure the directory or directories (normally the *csound5* directory) containing the Csound executables directory are in your `PATH` variable, or else copy all the executable files to your Windows `system32` directory. Depending on your installation method, you might also need to set the `OPCODEDIR` and `OPCODEDIR64` environment variables. Assuming that the binaries archive is unpacked in `C:\` you can use (otherwise set the paths accordingly):

```
set OPCODEDIR=C:\csound5\plugins
set OPCODEDIR64=C:\csound5\plugins64
set PATH=%PATH%;C:\csound5\bin
```

If you get a pop-up about the missing Python library (`python24.dll`) and don't need the python opcodes, just delete `csound5\plugins\py.dll` and `csound5\plugins64\py.dll`, and the pop-up about the missing Python library should be gone.

## Unix and Linux

On Unix and Linux, either install the Csound program in one of the system `bin` directories, typically `/usr/local/bin`, and the Csound and plugin shared libraries in places like `/usr/local/lib/csound/plugins` or `/usr/local/lib/csound/plugins64` and make sure that `OPCODEDIR` and `OPCODEDIR64` environment variable are set correctly.

## CsoundVST

CsoundVST requires some additional configuration. On all platforms, CsoundVST requires that you have Python installed on your computer. The directory containing the `_CsoundVST` shared library and the `CsoundVST.py` file must be in your `PYTHONPATH` environment variable, so that the Python runtime knows how to load these files.

---

# Using

Assuming that you have installed and configured the software, Csound and CsoundVST can be operated in a variety of modes and configurations. The `.csd` and `.py` files in the `examples` directory demonstrate a few of these modes of operation. Some of these scores are simple, others are moderately complex.

You may need to edit the SoundFont file paths in instrument definitions that use the fluid SoundFont 2 player opcode to match your environment.

## Real-Time Audio

For real-time audio output, with or without MIDI control, you will probably want to tune the `kr` and `ksmps` orchestra statements, and the `-b` and `-B` command-line options, to give you the shortest possible latency that does not cause clicks or stutters in Csound's audio output.

In general, `-b` (Csound's audio buffer) should be set to a small power of 2 (such as 64 or 128), and `-B` (the audio driver's buffer) should be set to 2 or 4 times that.

Currently, with `-B` set to 512, audio output latency is about 12 milliseconds, fast enough for reasonably responsive keyboard playing.

Even shorter latencies, as low as 3 milliseconds on some systems, are feasible.

## Windows with ASIO

If your sound card does not have an ASIO driver, you can still use Csound with ASIO by downloading and installing the asio4all adapter from <http://www.asio4all.com>.

## Using Csound5 with JACK

### Command line options

To enable the JACK plugin, use this command line option:

```
-+rtaudio=jack
```

Additionally, there are some command line options specific to JACK:

### JACK Command-line Flags

`--jack_client=[client_name]`

The client name used by Csound, defaults to 'csound5'. If multiple instances of Csound connect to the JACK server, different client names need to be used to avoid name conflicts.

`--jack_inportname=[input port name prefix], -  
--jack_outportname=[output port name prefix]`

Name prefix of Csound JACK input/output ports; the default is 'input' and 'output'. The actual port name is the channel number appended to the name prefix. Example: with the above default settings, a stereo orchestra will create these ports in full duplex operation:

```
csound5:input1  
csound5:input2  
csound5:output1  
csound5:output2
```

```
(rec  
(rec  
(pla  
(pla
```

`-+jack_sleep_time=[sleep time in microseconds]`

As of Csound version 5.01, this option is deprecated and ignored.

## Connecting Csound to other JACK clients

By default, no connections are made (you need to use `jack_connect`, `qjackctl`, or a similar utility); however, the plugin can connect to ports specified as `'-iadc:portname_prefix'` or `'-odac:portname_prefix'`, where `portname_prefix` is the full name of a port without a channel number, such as `'alsa_pcm:capture_'` (for `-i adc`), or `'alsa_pcm:playback_'` (for `-o dac`).

## Notes on buffer sizes

Audio data is received from and sent to the JACK server by Csound using a ring buffer that is controlled by the `-b` and `-B` flags. `-B` is the total size of the buffer, while `-b` is the size of a single period. These values are rounded so that the total size is an integer multiple of, and greater than the period size. The difference of the Csound buffer and period size must be greater than or equal to the JACK period size.

If both `-iadc` and `-odac` are used at the same time, the `-b` option should be set to an integer multiple of `ksmps`.

An example of buffer settings for low latency on a fast system:

```
jackd -d alsa -P -r 48000 -p 64 -n 4 -zt &  
csound -+rtaudio=jack -b 64 -B 256 [...]
```

with real time scheduling (as root):

```
jackd -R -P 90 -d alsa -P -r 48000 -p 64 -n 2 -zt &  
csound --sched=80,90,10 -d -+rtaudio=jack -b 64 -B 192 [...]
```

To improve performance, use `ksmps` values like 32 and 64.

The sample rate of the orchestra must be the same as that of the JACK server.

# Csound

## The `csound` Command

The original method for running Csound was as a console program. This, of course, still works. Running `csound` without any arguments prints out a list of command-line options, which are more fully explained below. Normally, the user executes something like `csound -W -omysoundfile myorchestra.orc myscore.sco` or, to use the single-file Csound structured data (`.csd`) format, `csound myscore.csd`.

Csound can read and write soundfiles (off-line rendering), read and write digital audio using a sound card (real-time rendering), read and write MIDI files, and read and write MIDI using a MIDI interface and controller (real-time control).

## CsoundVST

CsoundVST is a multi-function front end for Csound, based on the Csound API. CsoundVST runs

as a stand-alone graphical user interface to Csound, or as a VST plugin in hosts such as the Cubase audio sequencer. CsoundVST provides both a C++ and a Python API to Csound, and to a set of classes for algorithmic composition.

CsoundVST contains a built-in Python interpreter. With Python, the user can generate a score, import a MIDI file, process notes, load and run a Csound orchestra, and in general do anything that can be done either with Csound or in Python.

## Standalone

To run CsoundVST as a stand-alone front end to Csound, execute CsoundVST. When the program has loaded, you will see a graphical user interface with a row of buttons along the top. Click on the *Open...* button to load a *.csd* file. You can also click on the *Open...* button and load a *.orc* file, then click on the *Import...* button to add a *.sco* file. You can edit the Csound command, the orchestra file, or the score file in the respective tabs of the user interface. When all is satisfactory, click on the *Perform* button to run Csound. You can stop a performance at any time by clicking on the *Stop* button.

## Python Scripting

You can use CsoundVST as a Python extension module. In fact, you can do this either in a standard Python interpreter, such as Python command line or the Idle Python GUI, or in CsoundVST itself in Python mode.

To use CsoundVST in a standard Python interpreter, import CsoundVST.

```
import CsoundVST
```

The CsoundVST module automatically creates an instance of CppSound named *csound*, which provides an object-oriented interface to the Csound API. In a standard Python interpreter, you can load a Csound *.csd* file and perform it like this:

```
C:\Documents and Settings\mkg>python
Python 2.3.3 (#51, Dec 18 2003, 20:22:39) [MSC v.1200 32 bit (Intel)]
Type "help", "copyright", "credits" or "license" for more information
>>> import CsoundVST
>>> csound.load("c:/projects/csound5/examples/trapped.csd")
1
>>> csound.exportForPerformance()
1
>>> csound.perform()
BEGAN CppSound::perform(5, 988ee0)...
BEGAN CppSound::compile(5, 988ee0)...
Using default language
0dBFS level = 32767.0
Csound version 5.00 beta (float samples) Jun 7 2004
libsndfile-1.0.10pre6
orchname: temp.orc
scorename: temp.sco
orch compiler:
398 lines read
instr 1
instr 2
instr 3
instr 4
instr 5
instr 6
instr 7
instr 8
instr 9
instr 10
```



```
instr 11
instr 12
instr 13
instr 98
instr 99
sorting score ...
... done
Csound version 5.00 beta (float samples) Jun 6 2004
displays suppressed
0dBFS level = 32767.0
orch now loaded
audio buffered in 16384 sample-frame blocks
SFDIR undefined. using current directory
writing 131072-byte blks of shorts to test.wav
WAV
SECTION 1:
ENDED CppSound::compile.
ftable 1:
ftable 2:
ftable 3:
ftable 4:
ftable 5:
ftable 6:
ftable 7:
ftable 8:
ftable 9:
ftable 10:
ftable 11:
ftable 12:
ftable 13:
ftable 14:
ftable 15:
ftable 16:
ftable 17:
ftable 18:
ftable 19:
ftable 20:
ftable 21:
ftable 22:
new alloc for instr 1:
B 0.000 .. 1.000 T 1.000 TT 1.000 M: 32.7 0.0
new alloc for instr 1:
B 1.000 .. 3.600 T 3.600 TT 3.600 M: 207.6 0.1

...

B 93.940 .. 94.418 T 98.799 TT281.799 M: 477.6 85.0
B 94.418 ..100.000 T107.172 TT290.172 M: 118.9 11.5
end of section 4 sect peak amps: 25950.8 26877.4
inactive allocs returned to freespace
end of score. overall amps: 32204.8 31469.6
overall samples out of range: 0 0
0 errors in performance
782 131072-byte soundblks of shorts written to test.wav WAV
Elapsed time = 13.469000 seconds.
ENDED CppSound::perform.
1
>>>
```

To use CsoundVST itself as your Python interpreter, click on the CsoundVST Settings tab, and select the Python check box in the Csound performance mode box. Do not create a new CppSound object; you must use the builtin `csound` object in the CsoundVST module.

The `koch.py` script shows how to use Python to do algorithmic composition for Csound. You can use Python triple-quoted string literals to hold your Csound files right in your script, and assign them to Csound:

```
csound.setOrchestra(''sr = 44100
kr = 441
ksmps = 100
nchnls = 2
0dbfs = .1
instr 1,2,3,4,5 ; FluidSynth General MID
I; INITIALIZATION
; Channel, bank, and program determine the preset, that is, the actual
ichannel      =          p1
iprogram      =          p6
ikey          =          p4
ivelocity     =          p5 + 12
ijunk6        =          p6
ijunk7        =          p7
; AUDIO
istatus       =          144;
print         iprogram, istatus, ichannel, ikey, ivelocityale
fluid         "c:/projects/csound5/samples/VintageDreamsWaves
iprogram, istatus, ichannel, ikey, ivelocity, 1
outs          aleft, arightendin'')
csound.setCommand("csound --opcode-lib=c:/projects/csound5/fluid.dll
-RWdfo ./koch.wav ./temp.orc ./temp.sco")
csound.exportForPerformance()
csound.perform()
```

To run your script in Csound VST, click on the *Perform* button.

## VST Plugin

The following instructions are for Cubase SX. You would follow roughly similar procedures in other hosts.

Use the *Devices* menu, *Plug-In Information* dialog, *VST Plug-Ins* tab, *Shared VST Plug-ins Folder* text field to add your csound5 directory to Cubase's plugin path. You can have multiple directories separated by semicolons.

Quit Cubase, and start it again.

Use the *File* menu, *New Project* dialog to create a new song.

Use the *Project* menu, *Add Track* submenu, to add a new MIDI track.

Use the pencil tool to draw a *Part* on the track a few measures long. Write some music in the *Part* using the *Event* editor or the *Score* editor.

Use the *Devices* menu (or the F11 key) to open the *VST Instruments* dialog.

Click on one of the *No VST Instrument* labels, and select `\_CsoundVST` from the list that pops up.

Click on the *e* (for edit) button to open the `\_CsoundVST` dialog.

On the *Settings* page, check the *Instrument* box in the *VST Plugin* group, and the *Classic* box in the *Csound performance mode* group. Then click on the *Apply* button.

Click on the *Open* button to bring up the file selector dialog. Navigate to a directory containing a Csound `csd` file suitable for MIDI performance, such as `csound/CsoundVST/examples/CsoundVST.csd`. Click on the *OK* button to load the file. You can also open and import a suitable `.orc` and `.sco` file as described above.

In any event, the command line in the *Classic Csound command line* text box must specify `-+rtmidi=null -M0`, and should read something like this:

```
csound -f -h --rtmidi=null -M0 -d -n -m7 temp.orc temp.sco
```

Click on the *VST Instruments* dialog's on/off button to turn it on. This should compile the Csound orchestra. *Note: If you don't compile the orchestra, you won't be able to assign the plugin to a track.*

In the *Cubase Track Inspector*, click on the *out: Not Assigned* label and select *\_CsoundVST* from the list that pops up.

On the ruler at the top of the *Arrangement* window, select the loop end point and drag it to the end of your part, then click on the loop button to enable looping.

Click on the *play* button on the *Transport* bar. You should hear your music played by CsoundVST.

Try assigning your track to different channels; a different Csound instrument will perform each channel.

When you save your song, your Csound orchestra will be saved as part of the song and re-loaded when you re-load the song.

You can click on the *Orchestra* tab and edit your Csound instruments while CsoundVST is playing. To hear your changes, just click on the CsoundVST *Perform* button to recompile the orchestra.

You can assign up to 16 channels to a single CsoundVST plugin. However, you can't have more than one CsoundVST plugin in the same song!

---

# Building Csound

Csound has become a complex project and can involve many dependencies. Unless you are a Csound developer or need to develop Csound plugins, you should try to use one of the precompiled distributions from <http://sourceforge.net/projects/csound>.

The latest Csound source code is available through the Concurrent Versions System (CVS)(<http://www.cvshome.org>). To download Csound sources using CVS, run the following commands:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/csound login
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/csound co csound
```

Information about accessing the CVS repository may be found in the SourceForge document *Basic Introduction to CVS and SourceForge.net (SF.net) Project CVS Services*.

If you wish to become a Csound developer, first obtain a SourceForge login, and then apply to John ffitch at the <http://www.sourceforge.net/projects/csound> site.

The procedure for building Csound 5 is briefly and incompletely outlined here.

The manual is built using make. Scripts are used for a few other tasks. However, this section focuses on the main Csound build system, which uses SCons, a Python program that replaces make for cross-platform configuration and building.

(Alternatively, for building a minimal version of Csound 5 (API library compiled as DLL, plugin libraries, and command line frontend) on Windows with MinGW/MSYS, you may alternatively edit and use Makefile-win32, eliminating the dependencies on Python and SCons.)

All Csound 5 SCons builds require the following:

- On Linux, install gcc; on Windows, install all of MinGW 3.4.2 (3.4.4 does not work) from <http://www.mingw.org> then MSYS then update it including the DTK toolkit, autoconf, automake, and libtool, in that order, or install MSVC; on OS X, install the latest XCode development system.
- Install Python from <http://www.python.org>. Note that on Windows, if you have installed both MinGW and MSVC, it is best to use batch files to set up a separate environment for each compiler that does not refer to any header, library, DLL, or executable of the other compiler. On Windows, with Python 2.4 and later, Csound will link directly with the Python DLL. Earlier versions of the Python DLL will require a MinGW import library that the Csound build system should create.
- Install the Software Interface and Wrapper Generator (SWIG) for generating Python and Java interfaces, from <http://www.swig.org>.
- Install SCons from <http://www.scons.org>. On Windows, the MSys shell does not allow the user to execute the scons script directly. Therefore, you need to make sure that Python is in your Windows executable path, and run the build like this: `$$ python c:/tools/python23/scripts/scons .`
- Install libsndfile version 1.0.13 or later from <http://www.mega-nerd.com/libsndfile>.

Optional configurations can include the following. In most cases it is best to install the most recent stable versions.

- For GUI widgets, install FLTK 1.1 from <http://www.fltk.org>. You must configure and build FLTK with `--enable-shared --enable-threads`.
- Real-time audio can use ALSA, JACK, CoreAudio, the Windows multimedia library, or PortAudio (v19-devel branch) from <http://www.portaudio.com/usingcvs.html>.
- Real-time MIDI can use the ALSA raw MIDI interface, Windows multimedia library, or PortMidi from <http://www.cs.cmu.edu/~music/portmusic>.
- CsoundVST, which is both a standalone GUI, a Python extension module, and a VST plugin form of Csound with extensive facilities for algorithmic composition, requires FLTK and the boost C++ template libraries for random numbers and linear algebra, from <http://www.boost.org>. The CsoundVST Random class requires that boost must be later than version 1.32.1.
- The fluid opcodes require the Fluidsynth library from <http://savannah.nongnu.org/download/fluid>. For Windows, use the prebuilt binaries.
- The STK opcodes require STK source code from <http://ccrma.stanford.edu/software/stk>, copied into `csound5/Opcodes/stk`.
- The Loris opcodes require Loris source code from <http://sourceforge.net/projects/loris>, copied into `csound5/Opcodes/Loris`.
- The OSC opcodes require the latest version of the liblo library from <http://plugin.org.uk/liblo>. The library does not yet build properly on Windows.

Execute `scons -h` to discover the current configuration options.

Modify `custom.py` as required for your installation (usually required on Windows, may not be required on Linux).

Execute `scons` with the options you desire.

Set the environment variable `OPCODEDIR` to the directory where plugin libraries are installed; in the case of a double precision build, `OPCODEDIR64` should be set instead. The NSIS installer performs this step.

To install on Linux, execute `./install.py` or `scons install`.

To create a Windows installer, build Csound for double precision samples and including the Loris, STK, py, vst4cs, and Fluidsynth opcodes, build the manual, install the NSIS installer from <http://nsis.sourceforge.net>, and run `csound5/installer/windows/csound.nsi`.

---

# To Do

This is a “to do” list, not necessarily complete, and in no particular order of priority or time, for Csound and CsoundVST:

- See also the *To-fix-and-do* in the csound5 directory.
- Create better examples, especially to demonstrate the use of Python and of VST plugins. One example should be a live performance instrument with a Python GUI that controls instrument parameters or algorithmic composition parameters in real time.
- `intseg` opcode that would act like a Buchla sequencer: for duration `x0` hold level `y0`, for duration `x1` hold level `y1`, etc.
- Make release envelopes work identically for MIDI, VST, and score-driven performance.

---

# 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.

## Order of Precedence

With some recent additions to Csound, there are now five places where options for Csound performance may be set. They are processed in the following order:

1. Csound's own defaults
2. File defined by CSOUNDRC environment variable, or .csoundrc file in the HOME directory
3. .csoundrc file in the current directory
4. <CsOptions> tag in a .csd file
5. Csound command line

The last assignment of an option will override any earlier ones. As of version 5.01, sample and control rate overrides (*-r* and *-k* flags) specified anywhere override *sr*, *kr*, and *ksmps* in the orchestra header.

## 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:

1. 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)

2. 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 flag arguments.

## Command-line Flags

Many flags are generic Csound command-line flags. Various platform implementations may not react the same way to different flags!

Listed first are the traditional flags present in Csound 4. Look further down for Csound 5 specific flags.

The format of a command is either:

```
csound [-flags] [orchname] [scorename]
or
csound [-flags] [csdfilename]
```

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

### Command-line Flags

-@ FILE	Provide an extended command-line in file “FILE”
-3, --format=24bit	Use 24-bit audio samples.
-8, --format=uchar	Use 8-bit unsigned character audio samples.
--format=type	Set the audio file output format to one of the formats available in libsndfile. At present the list is aiff, au, avr, caf, flac, htk, ircam, mat4, mat5, nis, paf, pvf, raw, sd2, sds, svx, voc, w64, wav, wavex and xi. Can also be used as --format=type:format or -format=format:type to set both the file type (wav, aiff, etc.) and sample format (short, long, float, etc.) at the same time.
-A, --aiff, --format=aiff	Write an AIFF format soundfile. Use with the -c, -s, -l, or -f flags.
-a, --format=alaw	Use a-law audio samples.
-B NUM, -hardwarebufsamps=NUM	Number of audio sample-frames held in the DAC <i>hardware</i> buffer. This is a threshold on which <i>software</i> 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 lates. In the case of portaudio output (the default real-time output), the -B parameter (more precisely, -B / sr) is passed as the "suggested latency" value. Other than that, Csound has no control over how PortAudio interprets the parameter. The default is 1024 on Linux, 4096 on Mac OS X and 16384 on Windows.
-b NUM, --iobufsamps=NUM	Number of audio sample-frames per sound i/o <i>software</i> buffer. Large is efficient, but small will reduce audio I/O delay and improve the accuracy of the timing of real time events. The default is 256 on Linux, 1024 on MacOS X, and 4096 on Windows. In real-time performance, Csound waits on audio I/O on <i>NUM</i> 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 NUM is negative, the effective value is *ksmps* \* -NUM (audio synchronous with k-period boundaries). With NUM small (e.g. 1) polling is then frequent and also locked to fixed DAC sample boundaries.

Note: if both -iadc and -odac are used at the same time (full duplex real time audio), the -b option should be set to an integer multiple of *ksmps*.

-C, --cscore	Use Cscore processing of the scorefile.
-c, --format=schar	Use 8-bit signed character audio samples.
-D, --defer-gen1	Defer GEN01 soundfile loads until performance time.
-d, --nodisplays	Suppress all displays.
--expression-opt	<i>Since Csound 5.</i> Turns on some optimizations in expressions: <ul style="list-style-type: none"><li>• Redundant assignment operations are eliminated whenever possible. This means that for example this line <code>a1 = a2 + a3</code> will compile as <code>a1 Add a2, a3</code> instead of <code>#a0 Add a2, a3 a1 = #a0</code> saving a temporary variable and an opcode call. Less opcode calls result in reduced CPU usage (an average orchestra may compile about 10% faster with --expression-opt, but it depends largely on how many expressions are used, what the control rate is (see also below), etc.; thus, the difference may be less, but also much more).</li><li>• number of a- and k-rate temporary variables is significantly reduced. This expression</li></ul>

(a1 + a2 + a3 + a4)

will compile as

```
#a0 Add a1, a2
#a0 Add #a0, a3
#a0 Add #a0, a4           ; (the result is in #a
```

instead of

```
#a0 Add a1, a2
#a1 Add #a0, a3
#a2 Add #a1, a4           ; (the result is in #a
```

The advantages of less temporary variables are:

- less cache memory is used, which may improve performance of orchestras with many a-rate expressions and a low control rate (e.g. *ksmps* = 100)
- large orchestras may load faster due to less different identifier names

- index overflow errors (i.e. when messages like this Case2: indx=-56004 (ffff253c); (short)indx = 9532 (253c) are printed and odd behavior or a Csound crash occurs) may be fixed, because such errors are triggered by too many different (especially a-rate) variable names in a single instrument.

Note that this optimization (due to technical reasons) is not performed on i-rate temporary variables.



## Warning

When `--expression-opt` is turned on, it is not allowed to use the `i()` function with an expression argument, and relying on the value of k-rate expressions at i-time is unsafe.

<code>-F FILE, --midifile=FILE</code>	Read MIDI events from MIDI file <i>FILE</i> . The file should have only one track in Csound versions 4.xx and earlier; this limitation is removed in Csound 5.00.
<code>-f, --format=float</code>	Use single-format float audio samples (not playable on some systems, but can be read by <i>-i, soundin</i> and <i>GEN01</i>
<code>-G, --postscriptdisplay</code>	Suppress graphics, use PostScript displays instead.
<code>-g, --asciidisplay</code>	Suppress graphics, use ASCII displays instead.
<code>-H#, --heartbeat=NUM</code>	Print a heartbeat after each soundfile buffer write: <ul style="list-style-type: none"> <li>• no NUM, a rotating bar.</li> <li>• NUM = 1, a rotating bar.</li> <li>• NUM = 2, a dot (.)</li> <li>• NUM = 3, filesize in seconds.</li> <li>• NUM = 4, sound a bell.</li> </ul>
<code>-h, --noheader</code>	No header on output soundfile. Don't write a file header, just binary samples.
<code>--help</code>	Display on-line help message.
<code>-I, --i-only</code>	<i>i-time only</i> . 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.
<code>-i FILE, --input=FILE</code>	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 <i>stdin</i> will cause audio to be read from standard input. <p>The name <i>devaudio</i> or <i>adc</i> will request sound from the host audio input device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a <code>:</code> character. It depends on the host audio interface whether a device number or a name should be used. In</p>

	the first case, an out of range number usually results in an error and listing the valid device numbers.
-J, --ircam, --format=ircam	Write an IRCAM format soundfile.
-j FILE	<i>Currently disabled.</i> Use database <i>FILE</i> for messages to print to console during performance. In Csound 5.00 and later versions, the localization of messages is controlled by two environment variables, both of which are optional. <i>CSSTRNGS</i> points to a directory containing .xmg files, and <i>CS_LANG</i> selects a language.
-K, --nopeaks	Do not generate any PEAK chunks.
-k NUM, --control-rate=NUM	Override the control rate ( <i>KR</i> ) supplied by the orchestra.
-L DEVICE, - -score-in=DEVICE	Read line-oriented real-time score events from device <i>DEVICE</i> . The name <i>stdin</i> 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 <i>standard numeric score</i> , except that an event with <i>p2=0</i> will be performed immediately, and an event with <i>p2=T</i> will be performed <i>T</i> seconds after arrival. Events can arrive at any time, and in any order. The score <i>carry</i> feature is legal here, as are held notes ( <i>p3</i> negative) and string arguments, but ramps and <i>pp</i> or <i>np</i> references are not.
-l, --format=long	Use long integer audio samples.
-M DEVICE, - -midi-device=DEVICE	Read MIDI events from device <i>DEVICE</i> . If using ALSA MIDI ( <i>--rtmidi=alsa</i> ), devices are selected by name and not number. So, you need to use an option like <i>-M hw:CARD,DEVICE</i> where <i>CARD</i> and <i>DEVICE</i> are the card and device numbers (e.g. <i>-M hw:1,0</i> ). In the case of PortMidi and MME, <i>DEVICE</i> should be a number, and if it is out of range, an error occurs and the valid device numbers are printed.
-m NUM, --messagelevel=NUM	<p>Message level for standard (terminal) output. Takes the <i>sum</i> of any of the following values:</p> <ul style="list-style-type: none"> <li>• 1 = note amplitude messages</li> <li>• 2 = samples out of range message</li> <li>• 4 = warning messages</li> <li>• 128 = print benchmark information</li> </ul> <p>And exactly one of these to select note amplitude format:</p> <ul style="list-style-type: none"> <li>• 0 = raw amplitudes</li> <li>• 32 = dB, no colors</li> <li>• 64 = dB, out of range highlighted with red</li> <li>• 96 = dB, all colors</li> </ul> <p>The default is 135 (128+4+2+1), which means all messages, raw amplitude values, and printing elapsed time at the end of performance.</p>
--midioutfile=FILENAME	Save MIDI output to a file (Csound 5.00 and later only).
-N, --notify	Notify (ring the bell) when score or MIDI track is done.

---

-n, --nosound	No sound. Do all processing, but bypass writing of sound to disk. This flag does not change the execution in any other way.
-O FILE, --logfile=FILE	Log output to file <i>FILE</i> .
-o FILE, --output=FILE	Output soundfile name. If not a full pathname, the soundfile will be placed in the directory given by the environment variable SF- <i>DIR</i> (if defined), else in the current directory. The name <i>stdout</i> will cause audio to be written to standard output, while <i>null</i> results in no sound output similarly to the -n flag. If no name is given, the default name will be <i>test</i> .  The name <i>devaudio</i> or <i>dac</i> will request writing sound to the host audio output device. It is possible to select a device number by appending an integer value in the range 0 to 1023, or a device name separated by a : character. It depends on the host audio interface whether a device number or a name should be used. In the first case, an out of range number usually results in an error and listing the valid device numbers.
-Q DEVICE	Enables MIDI OUT operations to device id <i>DEVICE</i> . 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 reduced by using a lower value for the -b flag.  If using ALSA MIDI (--rtmidi=alsa), devices are selected by name and not number. So, you need to use an option like -Q hw:CARD,DEVICE where CARD and DEVICE are the card and device numbers (e.g. -Q hw:1,0). In the case of PortMidi and MME, DEVICE should be a number, and if it is out of range, an error occurs and the valid device numbers are printed.
-R, --rewrite	Continually rewrite the header while writing the soundfile (WAV/AIFF).
-r NUM, --sample-rate=NUM	Override the sampling rate (SR) supplied by the orchestra.
-s, --format=short	Use short integer audio samples.
--sched	<i>Linux only</i> . Use real-time scheduling and lock memory. (Also requires -d and either -o <i>dac</i> or -o <i>devaudio</i> ). See also --sched=N below for Csound 5.00 and later.
--strset	<i>Csound 5</i> . The --strset option allows setting strset string values from the command line, in the format '--strsetN=VALUE'. It is useful for passing parameters to the orchestra (e.g. file names).
-T, --terminate-on-midi	Terminate the performance when the end of MIDI file is reached.
-t0, --keep-sorted-score	Prevents Csound from deleting the sorted score file, score.srt, upon exit.
-t NUM, --tempo=NUM	Use the uninterpreted beats of <i>score.srt</i> for this performance, and set the initial tempo at <i>NUM</i> beats per minute. When this flag is set, the tempo of score performance is also controllable from within the orchestra. WARNING: this mode of operation is experimental and may be unreliable.
-U UTILITY, - -utility=UTILITY	Invoke the utility program <i>UTILITY</i> . Use any invalid name to list the available utilities.

---

-u, --format=ulaw	Use u-law audio samples.
-v, --verbose	Verbose translate and run. Prints details of orch translation and performance, enabling errors to be more clearly located.
-W, --wave, --format=wave	Write a WAV format soundfile.
-x FILE, --extract-score=FILE	Extract a portion of the sorted score, score.srt, using the extract file <i>FILE</i> (see <i>Extract</i> ).
-Z, --dither	Switch on dithering of audio conversion from internal floating point to 32, 16 and 8-bit formats.
-z NUM, --list-opcodesNUM	List opcodes in this version: <ul style="list-style-type: none"> <li>• no NUM, just show names</li> <li>• NUM = 0, just show names</li> <li>• NUM = 1, show arguments to each opcode using the format &lt;opname&gt; &lt;outargs&gt; &lt;inargs&gt;</li> </ul>

## Csound5 Command-line Flags

--sched=N	<i>Linux only.</i> Same as --sched, but allows specifying a priority value: if N is positive (in the range 1 to 99) the scheduling policy SCHED_RR will be used with a priority of N; otherwise, SCHED_OTHER is used with the nice level set to N. Can also be used in the format --sched=N,MAXCPU,TIME to enable the use of a "watchdog" thread that terminates Csound if the average CPU usage exceeds MAXCPU percents over a period of TIME seconds (new in Csound 5.00).
--displays	Enables displays, reverting the effect of any previous -d flag.
--no-expression-opt	Disables expression optimization.
++id_artist=string	(max. length = 200 characters) Artist tag in output soundfile (no spaces)
++id_comment=string	(max. length = 200 characters) Comment tag in output soundfile (no spaces)
++id_copyright=string	(max. length = 200 characters) Copyright tag in output soundfile (no spaces)
++id_date=string	(max. length = 200 characters) Date tag in output soundfile (no spaces)
++id_software=string	(max. length = 200 characters) Software tag in output soundfile (no spaces)
++id_title=string	(max. length = 200 characters) Title tag in output soundfile (no spaces)
++max_str_len=integer	(min: 10, max: 10000) Maximum length of string variables + 1; defaults to 256 allowing a length of 255 characters. The length of string constants is not limited by this parameter.
++msg_color=boolean	Enable message attributes (colors etc.); might need to be disabled on some terminals which print strange characters in-

	stead of modifying text attributes. default: true.
<code>--mute_tracks=string</code>	(max. length = 255 characters) Ignore events (other than tempo changes) in MIDI file tracks defined by pattern (for example, <code>--mute_tracks=00101</code> will mute the third and fifth tracks).
<code>--raw_controller_mode=boolean</code>	Disable special handling of MIDI controllers like sustain pedal, all notes off etc., allowing the use of all the 128 controllers for any purpose. This will also set the initial value of all controllers to zero. Default: no.
<code>--rtaudio=string</code>	(max. length = 20 characters) Real time audio module name. The default is PortAudio. Also available, depending on platform and build options: Linux: alsa, jack; Windows: mme; Mac OS X: CoreAudio. In addition, null can be used on all platforms, to disable the use of any real time audio plugin.
<code>--rtmidi=string</code>	(max. length = 20 characters) Real time MIDI module name. Defaults to PortMidi, other options (depending on build options): Linux: alsa; Windows: mme, winmm. In addition, null can be used on all platforms, to disable the use of any real time MIDI plugin.  ALSA MIDI devices are selected by name and not number. So, you need to use an option like <code>-M hw:CARD,DEVICE</code> where CARD and DEVICE are the card and device numbers (e.g. <code>-M hw:1,0</code> ).
<code>--skip_seconds=float</code>	(min: 0) Start playback at the specified time (in seconds), skipping earlier events in the score and MIDI file.
<code>--omacro:XXX=YYY</code>	Set orchestra macro XXX to value YYY
<code>--smacro:XXX=YYY</code>	Set score macro XXX to value YYY
<code>--env:NAME=VALUE</code>	Set environment variable NAME to VALUE; note: not all environment variables can be set this way, because some are read before parsing the command line. INCDIR, SADIR, SFDIR, and SSDIR are known to work.
<code>--env:NAME+=VALUE</code>	Append VALUE to ';' separated list of search paths in environment variable NAME (should be INCDIR, SADIR, SFDIR, or SSDIR). If a file is found in multiple directories, the last will be used.

## Csound Environment Variables

The following environment variables can be used by Csound:

- **SFDIR**: Default directory for sound files. Used if no full path is given for sound files.
- **SSDIR**: Default directory for input (source) sound files. Used if no full path is given for sound files. May be used in conjunction with SFDIR to set separate input and output directories.
- **SADIR**: Default directory for analysis files. Used if no full path is given for analysis files.
- **SFOUTYP**: Sets the default output file type. Currently only 'WAV', 'AIFF' and 'IRCAM' are valid. This flag is checked by the csound executable and the utilities and is used if no file output type is specified.

- **INCDIR**: Include directory. Specifies the location of files used by *#include* statements.
- **OPCODEDIR**: Defines the location of csound opcode plugins for the single precision float (32-bit) version.
- **OPCODEDIR64**: Defines the location of csound opcode plugins for the double precision float (64-bit) version.
- **SNAPDIR**: Is used by the FLTK widget opcodes when loading and saving snapshots.
- **CSOUNDRC**: Defines the csound resource (or configuration) file. A full path and filename containing csound flags must be specified. This variable defaults to `.csoundrc` if not present.
- **CSSTRNGS**: In Csound 5.00 and later versions, the localisation of messages is controlled by two environment variables `CSSTRNGS` and `CS_LANG`, both of which are optional. `CSSTRNGS` points to a directory containing `.xmg` files.
- **CS\_LANG**: Selects a language for csound messages.
- **RAWWAVE\_PATH**: Is used by the STK opcodes to find the raw wave files. Only relevant if you are using STK wrapper opcodes like `STKBowed` or `STKBrass`.
- **CSNOSTOP**: If this environment variable is set to "yes", then any graph displays are closed automatically at the end of performance (meaning that you possibly will not see much of them in the case of a short non-realtime render). Otherwise, you need to click "Quit" in the FLTK display window to exit, allowing for viewing the graphs even after the end of score is reached.

For more information about `SFDIR`, `SSDIR`, `SADIR` and `INCDIR` see *Directories and files*.

The only mandatory environment variables are `OPCODEDIR` and `OPCODEDIR64`. It is very important to set them correctly, otherwise most of the opcodes will not be available. Make sure you set the path correctly depending on the precision of your binary. If you run csound on a command line without any arguments you should see some text like : Csound version 5.01.0 beta (float samples) Mar 23 2006. This text refers to the single precision version.

`CSSTRNGS` and `CS_LANG` currently have very limited use since Csound has not yet been completely translated into other languages.

Other environment variables which are not exclusive to Csound but which might be of importance are:

- **PATH**: The directory containing csound executables should be listed in this variable.
- **PYTHONPATH**: If you intend to use `CsoundVST` and python, the directory containing the `_CsoundVST` shared library and the `CsoundVST.py` file must be in your `PYTHONPATH` environment variable (or the default path python searches in), so that the Python runtime knows how to load these files.
- **LADSPA\_PATH** and **DSSI\_PATH**: These environment variables are required if you are using the *dssi4cs* (LADSPA and DSSI host) plug-in opcodes.

## Setting environment variables

### On the command line

You can set environment variables on the command line or the configuration file `.csoundrc` by using the command line flag `--env:NAME=VALUE` or `--env:NAME+=VALUE`, where `NAME` is the environment variable name, and `VALUE` is its value. See *Command-line Flags*



## Note

Please note that this method of setting environment variables will not work for variables which are parsed before the command line arguments. SADIR, SSDIR, SFDIR, INCDIR, SNAPDIR, RAWWAVE\_PATH, CSNOSTOP, SFOUTYP should work, but the following environment variables must be set on the system prior to running csound: OPCODEDIR, OPCODEDIR64, CSSTRINGS, and CS\_LANG. CSOUNDRC can currently (v. 5.02) be set using --env, but this behavior is not guaranteed for future versions.

## Windows

To set a csound environment on Windows XP and 2000 go to Control Panel->System->Advanced and click on the button 'Environment Variables'. On other Windows earlier than XP you set environment variables in the autoexec.bat file. Go to 'My Computer', select C: drive, right click on autoexec.bat, and select 'Edit'. The statement format is: SET NAME=VALUE .

## Linux

You can set environment variables on Linux in many ways. You can set them using the *export* shell command, by setting them on .bashrc or similar files or by adding them to the /etc/profile file.

## Mac

If the user has a Mac that shipped with an OS X version prior to 10.3 (includes 10.2 and 10.1) then it is possible that the default shell is the Tenex C-shell (tcsh). If this is the case, then you either have to type:

```
~% setenv OPCODEDIR "/Users/you/your/Csound5/build"
```

or change your /etc/profile and or edit your .tcshrc file.

If the user has a Mac that shipped with OS X 10.3 or 10.4 then it likely has the "Bourne-again" C-shell (bash) as the default shell. If this is the case, then the user must type something like:

```
~$ export OPCODEDIR=/Users/you/your/Csound5/build
```

in addition if the bash shell is the default, then it is usually easier to edit your .bashrc or /etc/profile.

Note that if users choose one of the above methods, ie editing the .bashrc file then the environment variables are executed when a new shell is created. This can be problematic if your application implements a Quartz or Aqua interface and does not use the commandline.

If this is the case, then the standard solution (up to OS 10.3.9 and unless the application uses the csoundAPI and sets the environ variables directly) is to create an XML property list file (called a .plist file by the OS). This file should nominally be located at ~/.MacOSX/Environment.plist. This has been a solution specifically for the [csoundapi~] object for Pd on OS X. Since Pd uses an OS X native .app style packaging, and runs off of the Aqua interface, the standard means of supplying environment variables to Csound do not work. The solution is to set Csound's environment variables for the Aqua environment.

Likely, most users will not have the hidden folder .MacOSX located in their \$HOME directory (aka ~/) This folder must first be created and the Environment.plist added to this folder. The contents of the Environment.plist file should be something like:

```
<?xml version="1.0" encoding='UTF-8'?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN" "http://www.apple
```



```
<plist version="1.0">
<dict>
<key>OPCODEDIR</key>
<string>/Library/Frameworks/CsoundLib.framework/Versions/5.1/Resources/Opcodes<
<key>OPCODEDIR64</key>
<string>/Volumes/ExternalHD/devel/csound5/lib64</string>
<key>INCDIR</key>
<string>/Volumes/ExternalHD/CSOUND/include</string>
<key>SFDIR</key>
<string>/Volumes/ExternalHD/iTunes/csoundaudio</string>
</dict>
</plist>
```

and so on, using the XML `<key>` tag for each environment variable required by the API and the `<string>` tag for it's corresponding path on the system.

Please note that you must login out and login in for these changes to take effect.

# Unified File Format for Orchestras and Scores

## 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.

## 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.

### 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 files may be included with the tag `<CsFileB filename=filename>`, where *filename* is the name of the file to be included. The Base64 encoded data should be terminated with a `</CsFileB>` tag. For encoding files, the `csb64enc` and `makecsd` utilities (included with Csound 5.00 and newer) can be used. The file will be extracted to the current directory, and deleted at end of performance. If there is an already existing file with the same name, it is not overwritten, but an error will occur instead.

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. Using this tag is not recommended; use `<CsFileB>` instead.

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. Using this tag is not recommended; use `<CsFileB>` instead.

## Version Blocking

Versions of Csound may be 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.

## 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>
```

## Command Line Parameter File

If the file *.csoundrc* exists, it will be used to set the command line parameters. These can be overridden. Csound 5.00 and newer versions read this file from the HOME directory first (or the full path file name defined by the CSOUNDRC environment variable), and then the current directory. If both exist, options in the *.csoundrc* in the current directory will have higher precedence. It uses the same form as a *.csd* file, but no tags are needed. Lines beginning with # or ; are treated as comments.

## Score File Preprocessing

### 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.
```

## 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

```
scsort < 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
```

---

# 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 Flow Control*). 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.

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:

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

## 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.

In Csound version 5.00 and later, these environment variables can specify multiple directories as a ; separated list. If a file is found in more than one location, the last one has the highest precedence.

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.

## 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.

Throughout 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."

## Orchestra Statement Types

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

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

## 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 con-

trolling 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 broad-casting 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:

**Table 1. Types of Variables**

Type	When Renewable	Local	Global
reserved symbols	permanent	--	rsymbol
score pfields	i-time	p number	--
init variables	i-time	i name	gi name
control signals	p-time, k-rate	k name	gk name
audio signals	p-time, k-rate	a name	ga name
spectral data types	k-rate	w name	--
streaming spectral data types	k-rate	f name	gf name
string variables	i-time and optionally k-rate	S name	gS 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, the underscore character, 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.

## 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.

## Orchestra Header Statements

Statements that are normally placed in an orchestra header are *ctrlinit*, *figen*, *kr*, *ksmps*, *massign*, *nchnls*, *pgmassign*, *pset*, *seed*, *sr*, and *strset*.

## Instrument and Opcode Block Statements

Statements that define an instrument block are *endin* and *instr*.

Statements that define a user defined opcode block are *endop* and *opcode*.

## Variable Initialization

Opcodes that let one initialize variables are *assign*, *divz*, *init*, and *tival*.

## Named Instruments

As a recent addition to the orchestra syntax, instruments can be defined with string names. Such named instruments are callable from the score, and are supported by a number of opcodes.

## Syntax

A named instrument is declared as shown below:

```
instr Name[ , Name2[ , Name3[ , ...]]]
[... ]
endin
```

A single instrument can have any number of names, and any of these names can be used to call the instrument. Additionally, it is possible to use numbers as name, denoting a standard numbered instrument, so the following declaration is also valid:

```
instr 100, Name1, 99, Name2, 1, 2, 3
```

An instrument name may consist of any number of letters, digits, and the underscore (`_`) character, however, the first character must not be a digit. Optionally, the instrument name may be prefixed with the '+' character (see below), for example:

```
instr 100, Name1, 99, Name2, 1, 2, 3
```

An instrument name may consist of any number of letters, digits, and the underscore (`_`) character, however, the first character must not be a digit. Optionally, the instrument name may be prefixed with the '+' character (see below), for example:

```
instr +Reverb
```



For all instrument names, a number is automatically assigned (note: if the message level (-m) is not zero, these numbers are printed to the console during orchestra compilation), following these rules:

- any unused instrument numbers are taken up in ascending order, starting from 1
- the numbers are assigned in the order of instrument name definition, so named instruments that are defined later will always have a higher number (except if the '+' modifier is used)
- if the instrument name was prefixed with '+', the assigned number will be higher than that of any of the (both numbered and named) other instruments without '+'. If there are multiple '+' instruments, the numbering of these will follow the order of definition, according to the above rule.

Using '+' is mainly useful for global output or effect instruments, that must be performed after the other instruments.

An example for instrument numbers:

```
instr 1, 2
endin

instr Instr1
endin

instr +Effect1, Instr2
endin

instr 100, Instr3, +Effect2, Instr4, 5
endin
```

In this example, the instrument numbers are assigned as follows:

```
Instr1: 3
Effect1: 101
Instr2: 4
Instr3: 6
Effect2: 102
Instr4: 7
```

## Using Named Instruments

Named instruments can be called by using the name in double quotes as the instrument number (note: the '+' character should be omitted). Currently (as of Csound 4.22.4), named instruments are supported by:

- \* 'i' and 'q' score events



### Notes

1. in score files, unmatched quotes, and spaces or other invalid characters in the strings should be avoided, otherwise (at least with current version) unpredictable behavior may occur (this problem does not exist for -L line events). However, there is checking for undefined instruments, and in such cases, the event is simply ignored with a warning.

2. Stand-alone utilities (score sort and extract) do not support named instruments. It is still possible to sort such scores by using the -t0 option of the main Csound executable)

- real-time line events (-L)
- event, schedkwhen, subinstr, and subinstrinit opcodes
- massign, pgmassign, prealloc, and mute opcodes

Additionally, there is a new opcode (nstrnum) that returns the number of a named instrument:

```
insno nstrnum "name"
```

With the above example, nstrnum "Effect1" would return 101. If an instrument with the specified name does not exist, an init error occurs, and -1 is returned.

## Example

```
; ---- orchestra ----

sr      = 44100
ksmps   = 10
nchnls  = 1

prealloc "SineWave", 20
prealloc "MIDISineWave", 20

massign 1, "MIDISineWave"

gaOutSend      init 0

instr +OutputInstr

out gaOutSend
clear gaOutSend

endin

instr SineWave

a1      oscils p4, p5, 0
vincr gaOutSend, a1

endin

instr MIDISineWave

iamp     veloc
inote    notnum
icps     = cpsoct(inote / 12 + 3)
a1       oscils iamp * 100, icps, 0
vincr gaOutSend, a1

endin

; ---- score ----

i "SineWave" 0 2 12000 440
i "OutputInstr" 0 3
```

e

## Author

Istvan Varga

2002

---

# The Standard Numeric Score

## 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.

## N.B.

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

## Next-P and Previous-P Symbols

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

E.g.: the statements

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

will result in

```
i1  0    1    10    20    0
i1  1    1    20    30    20
i1  2    1    30    0    30
```

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

## Ramping

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

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

will result in

```
i1  0    1    100
i1  1    1    200
i1  2    1    300
i1  3    1    400
i1  4    1    200
i1  5    1    0
```

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

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.

## Score Macros

### 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.

## Syntax

```
#define NAME # replacement text #
```

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

```
$NAME.
```

```
#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

### Example 1. 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
```

## Example 2. 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
```

## Credits

Author: John ffitch

University of Bath/Codemist Ltd.

Bath, UK

April, 1998 (New in Csound version 3.48)

## Multiple File Score

## Description



Using the score in more than one file.

## Syntax

```
#include "filename"
```

## Performance

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 statement*, *m statement*, and *n statement*.

## Credits

Author: John ffitch

University of Bath/Codemist Ltd.

Bath, UK

April, 1998 (New in Csound version 3.48)

Thanks to Luis Jure for pointing out the incorrect syntax in multiple file include statement.

## 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.

## Credits

Author: John ffitch

University of Bath/Codemist Ltd.

Bath, UK

April, 1998 (New in Csound version 3.48)

## Score Statements

The statements used in scores are  $a$ ,  $b$ ,  $e$ ,  $f$ ,  $i$ ,  $m$ ,  $n$ ,  $q$ ,  $r$ ,  $s$ ,  $t$ ,  $v$ , and  $x$ .

## Sine/Cosine Generators

The GEN routines that generate sine or cosine values are *GEN09*, *GEN10*, *GEN11*, *GEN19*, *GEN30*, *GEN33*, and *GEN34*.

## Line/Exponential Segment Generators

GEN routines that generate tables with linear or exponential segments are *GEN05*, *GEN06*, *GEN07*, *GEN08*, *GEN16*, *GEN25*, and *GEN27*.

## File Access GEN Routines

The GEN routines that access files are *GEN01*, *GEN23*, and *GEN28*.

## Numeric Value Access GEN Routines

The GEN routines that generate tables from numeric values are *GEN02* and *GEN17*.

## Window Function GEN Routines

The GEN routine for window functions is *GEN20*.

## Random Function GEN Routines

GEN routines that generate random distributions are *GEN21*, *GEN40*, *GEN41*, and *GEN42*.

## Waveshaping GEN Routines

The GEN routines that have waveshaping functionality are *GEN03*, *GEN13*, *GEN14*, and *GEN15*.

## Amplitude Scaling GEN Routines

GEN routines that perform amplitude scaling are *GEN04*, *GEN12*, and *GEN24*.

## Mixing GEN Routines

GEN routines that mix together waveforms are *GEN18*, *GEN31*, and *GEN32*.

---

# TclCsound

TclCsound was introduced to provide a simple scripting interface to Csound. Tcl is a simple language that is easy to extend and provide nice facilities such as easy file access and TCP networking. With its Tk component, it can also handle a graphic and event interface. TclCsound provides three 'points of contact' with Tcl:

1. a csound-aware tcl interpreter (cstclsh)
2. a csound-aware windowing shell (cswish)
3. a csound commands module for Tcl/Tk (tclcsound dynamic lib)

## The Tcl interpreter: cstclsh

With cstclsh, it is possible to have interactive control over a csound performance. The command starts an interactive shell, which holds an instance of Csound. A number of commands can then be used to control it. For instance, the following command can compile csound code and load it in memory ready for performance:

```
csCompile -odac orchestra score -m0
```

Once this is done, performance can be started in two ways: using csPlay or csPerform . The command

```
csPlay
```

will start the Csound performance in a separate thread and return to the cstclsh prompt. A number of commands can then be used to control Csound. For instance,

```
csPause
```

will pause performance; and

```
csRewind
```

will rewind to the beginning of the note-list. The csNote, csTable and csEvent commands can be used to add Csound score events to the performance, on-the-fly. The csPerform command, as opposed to csPlay , will not launch a separate thread, but will run Csound in the same thread, returning only when the performance is finished. A variety of other commands exist, providing full control of Csound.

## Cswish: the windowing shell

With Cswish, Tk widgets and commands can be used to provide graphical interface and event handling. As with cstclsh, running the cswish command also opens an interactive shell. For instance, the following commands can be used to create a transport control panel for Csound:

```
frame .fr
button .fr.play -text play -command csPlay
button .fr.pause -text pause -command csPause
button .fr.rew -text rew -command csRewind
pack .fr .fr.play .fr.pause .fr.rew
```

Similarly, it is possible to bind keys to commands so that the computer keyboard can be used to play Csound.

Particularly useful are the control channel commands that TclCsound provides. For instance, named IO channels can be registered with TclCsound and these can be used with the `invalue`, `outvalue` opcodes. In addition, the Csound API also provides a complete software bus for audio, control and string channels. It is possible in TclCsound to access control and string bus channels (the audio bus is not implemented, as Tcl is not able to handle such data). With these TclCsound commands, Tk widgets can be easily connected to synthesis parameters.

## A Csound server

In Tcl, setting up TCP network connections is very simple. With a few lines of code a csound server can be built. This can accept connections from the local machine or from remote clients. Not only Tcl/Tk clients can send commands to it, but TCP connections can be made from other software, such as, for instance, Pure Data (PD). A Tcl script that can be run under the standard `tcsh` interpreter is shown below. It uses the `Tclcsound` module, a dynamic library that adds the Csound API commands to Tcl.

```
# load tclcsound.so
#(OSX: tclcsound.dylib, Windows: tclcsound.dll)
load tclcsound.so Tclcsound
set forever 0

# This arranges for commands to be evaluated
proc ChanEval { chan client } {
    if { [catch { set rtn [eval [gets $chan]] } err] } {
        puts "Error: $err"
    } else {
        puts $client $rtn
        flush $client
    }
}

# this arranges for connections to be made

proc NewChan { chan host port } {
    puts "Csound server: connected to $host on port $port ($chan)"
    fileevent $chan readable [list ChanEval $chan $host]
}

# this sets up a server to listen for
# connections

set server [socket -server NewChan 40001]
set sinfo [fconfigure $server -sockname]
puts "Csound server: ready for connections on port [lindex $sinfo 2]"
vwait forever
```

With the server running, it is then possible to set up clients to control the Csound server. Such clients can be run from standard Tcl/Tk interpreters, as they do not evaluate the Csound commands themselves. Here is an example of client connections to a Csound server, using Tcl:

```
# connect to server
set sock [socket localhost 40001]

# compile Csound code
puts $sock "csCompile -odac orchestra score"
```

```
flush $sock
```

```
# start performance
puts $sock "csPlay"
flush $sock
```

```
# stop performance
puts $sock "csStop"
flush $sock
```

As mentioned before, it is possible to set up clients using other software systems, such as PD. Such clients need only to connect to the server (using a netsend object) and send messages to it. The first item of each message is taken to be a command. Further items can optionally be added to it as arguments to that command.

## A Scripting Environment

With TclCsound, it is possible to transform the popular text editor e-macs into a Csound scripting/performing environment. When in Tcl mode, the editor allows for Tcl expressions to be evaluated by selection and use of a simple escape sequence (Ctrl-C Ctrl-X). This facility allows the integrated editing and performance of Csound and Tcl/Tk code.

In Tcl it is possible to write score and orchestra files that can be saved, compiled and run by the same script, under the e-macs environment. The following example shows a Tcl script that builds a csound instrument and then proceeds to run a csound performance. It creates 10 slightly detuned parallel oscillators, generating sounds similar to those found in Risset's *Inharmonique*.

```
load tclcsound.so Tclcsound

# set up some intermediary files

set orcfile "tcl.orc"
set scofile "tcl.sco"
set orc [open $orcfile w]
set sco [open $scofile w]

# This Tcl procedure builds an instrument
proc MakeIns { no code } {
    global orc sco
    puts $orc "instr $no"
    puts $orc $code
    puts $orc "endin"
}

# Here is the instrument code
append ins "asum init 0 \n"
append ins "ifreq = p5 \n"
append ins "iamp = p4 \n"

for { set i 0 } { $i < 10 } { incr i } {
    append ins "a$i oscili iamp,
ifreq+ifreq*[expr $i * 0.002], 1\n"
}
```

```
for { set i 0 } { $i < 10 } { incr i } {
if { $i } {
append ins " + a$i"
} else {
append ins "asum = a$i "
}
}

append ins "\nk1 linen 1, 0.01, p3, 0.1 \n"
append ins "out asum*k1"

# build the instrument and a dummy score

MakeIns 1 $ins
puts $sco "f0 10"
close $orc
close $sco

# compile
csCompile $orcfile $scofile -odac -d -m0

# set a wavetable
csTable 1 0 16384 10 1 .5 .25 .2 .17 .15 .12 .1

# send in a sequence of events and perform it
for {set i 0} { $i < 60 } { incr i } {
csNote 1 [expr $i * 0.1] .5 \
[expr ($i * 10) + 500] [expr 100 + $i * 10]
}
csPerform

# it is possible to run it interactively as
# well
csNote 1 0 10 1000 200
csPlay
```

The use of such facilities as provided by e-macs can emulate an environment not unlike the one found under the so-called 'modern synthesis systems', such as SuperCollider (SC). In fact, it is possible to run Csound in a client-server set-up, which is one of the features of SC3. A major advantage is that Csound provides about three or four times the number of unit generators found in that language (as well as providing a lower-level approach to signal processing, in fact these are but a few advantages of Csound).

## TclCsound as a language wrapper

It is possible to use TclCsound at a slightly lower level, as many of the C API functions have been wrapped as Tcl commands. For instance it is possible to create a 'classic' Csound command-line frontend completely written in Tcl. The following script demonstrates this:

```
#!/usr/local/bin/cstclsh
```

```
set result 1
csCompileList $argv
while { $result != 0 } {
set result csPerformKsmpts
}
```

## TclCsound Command Reference

Performance control commands:

**csCompile [csound command-line]** : compiles an orc/sco/csd + any options

**csCompileList arglist** : compiles an orc/sco/csd + options given as a Tcl list 'arglist'

**csPerform** : plays the score, returning when finished

**csPerformKsmpts** : performs one ksmpts block of audio samples, returning when finished

**csPerformBuffer** : performs one buffersize block of audio samples, returning when finished

**csPlay** : starts asynchronous performance in a separate thread, returning immediately

**csPause** : pauses playback

**csStop** : stops performance and resets csound

**csRewind** : rewinds the score

**csOffset secs** : offsets score playback by secs

**csGetoffset** : returns the score offset in secs

**csGetScoreTime** : returns the score time in secs

Event commands:

**csNote [p-fields]** : sends in a i-statement event

**csTable [p-fields]** : sends in a f-statement event

**csEvent opcode [p-fields]** : sends in a score event defined by 'opcode' plus p-fields

**csNoteList arglist** : sends in a i-statement event with p-fields as a Tcl list 'arglist'

**csTableList arglist** : sends in a f-statement event with p-fields as a Tcl list 'arglist'

**csEventList arglist** : sends in a score event defined by 'opcode' plus p-fields as a Tcl list 'arglist'

Invalue, outvalue, control and string channel commands:

**csInChannel name** : registers a csound invalue channel

**csOutChannel name** : registers a csound outvalue channel and creates tcl global variable 'name'

**csInValue channel value** : sets the value of a csound invalue channel

**csOutValue channel** : returns the value of a csound outvalue channel

**csSetControlChannel channel value** : sets the value of control channel 'channel', creating it if it does not exist

**csGetControlChannel channel** : returns the value of control channel 'channel'; creates the channel



it if it does not exist

**csSetStringChannel channel string** : sets the string channel 'channel', creating it if it does not exist

**csGetStringChannel channel** : returns the string in channel 'channel'; creates the channel if it does not exist

Message commands:

**csMessageOutput var**: appends all csound messages to the tcl variable var.

Table commands:

**csGetTableSize ftn** : returns the size of function table ftn (-1 if non-existent)

**csSetTable ftn index value** : sets the value of position 'index' to 'value' in function table 'ftn'

**csGetTable ftn index** : returns the value of position 'index' in function table 'ftn'

Environment variable commands:

**csOpcodedir opcodedir** : sets the opcode directory

**csSetenv envvar value** : sets any environment variable (eg. SFDIR, SADIR)

---

## Part II. Opcodes Overview

---

---

## Table of Contents

Signal Generators .....	85
Additive Synthesis/Resynthesis .....	85
Basic Oscillators .....	85
Dynamic Spectrum Oscillators .....	85
FM Synthesis .....	86
Granular Synthesis .....	86
Linear and Exponential Generators .....	86
Envelope Generators .....	87
Models and Emulations .....	87
Phasors .....	88
Random (Noise) Generators .....	88
Sample Playback .....	89
Soundfonts .....	89
Scanned Synthesis .....	90
Table Access .....	92
Wave Terrain Synthesis .....	92
Waveguide Physical Modeling .....	92
Signal Input and Output .....	94
File Input and Output .....	94
Signal Input .....	94
Signal Output .....	94
Printing and Display .....	94
Sound File Queries .....	95
Signal Modifiers .....	96
Amplitude Modifiers and Dynamic processing .....	96
Convolution and Morphing .....	96
Delay .....	96
Panning and Spatialization .....	97
Reverberation .....	98
Sample Level Operators .....	98
Signal Limiters .....	99
Special Effects .....	99
Standard Filters .....	99
Specialized Filters .....	101
Waveguides .....	101
Instrument Control .....	102
Clock Control .....	102
Conditional Values .....	102
Duration Control Statements .....	102
Introduction to FLTK Widgets and GUI controllers .....	102
FLTK Containers .....	105
FLTK Valuators .....	105
Other FLTK Widgets .....	105
Instrument Invocation .....	106
Macros .....	106
Program Flow Control .....	106
Real-time Performance Control .....	107
Reinitialization .....	107
Sensing and Control .....	107
Sub-instrument Control .....	109
Time Reading .....	109
Function Table Control .....	110
Table Queries .....	110
Read/Write Operations .....	110
Table Reading with Dynamic Selection .....	110
Mathematical Operations .....	112
Amplitude Converters .....	112

Arithmetic and Logic Operations .....	112
Mathematical Functions .....	112
Opcode Equivalents of Functions .....	112
Random Functions .....	113
Trigonometric Functions .....	113
Pitch Converters .....	114
Functions .....	114
Tuning Opcodes .....	114
Real-time MIDI Support .....	115
MIDI input .....	115
MIDI Message Output .....	115
Generic Input and Output .....	115
Converters .....	116
Event Extenders .....	116
Note-on/Note-off Output .....	116
System Realtime Messages .....	116
Slider Banks .....	116
Spectral Processing .....	118
Short-time Fourier Transform (STFT) Resynthesis .....	118
Linear Predictive Coding (LPC) Resynthesis .....	118
Non-standard Spectral Processing .....	119
Tools for Real-time Spectral Processing (pvs opcodes) .....	119
ATS Spectral Processing .....	120
Loris Opcodes .....	121
Strings .....	125
String Conversion Opcodes .....	125
Vectorial Opcodes .....	127
Tables of vectors operators .....	127
Operations Between a Vectorial and a Scalar Signal .....	127
Operations Between two Vectorial Signals .....	127
Vectorial Envelope Generators .....	127
Limiting and wrapping of vectorial control signals .....	128
Vectorial Control-rate Delay Paths .....	128
Vectorial Random Signal Generators .....	128
Zak Patch System .....	130
DSSI and LADSPA for Csound .....	131
VST for Csound .....	132
OSC and Csound .....	133
.....	133
Mixer Opcodes .....	134
Python Opcodes .....	135
Introduction .....	135
Orchestra Syntax .....	135

---

# Signal Generators

## Additive Synthesis/Resynthesis

The opcodes for additive synthesis and resynthesis are:

- *adsyn*
- *adsynt*
- *adsynt2*
- *hsboscil*

See the section *Spectral processing* for more information and further additive/resynthesis opcodes.

## Basic Oscillators

The basic oscillator opcodes are: (note that opcodes that end with 'i' implement linear interpolation and those that end with '3' implement cubic interpolation)

- Oscillator Banks: *oscbnk*
- Simple table oscillators: *oscil*, *oscil3* and *oscili*.
- Simple, fast sine oscillator: *oscils*
- Precision oscillators: *poscil* and *poscil3*.
- More flexible oscillators: *oscilikt*, *osciliktp*, *oscilikts* and *osciln* (also called *oscilx*).

## LFOs

- *lfo*
- *vibr*
- *vibrato*

See the section *Table access* for other table reading opcodes that can be used as oscillators. Also see the section *Dynamic spectrum Oscillators*.

## Dynamic Spectrum Oscillators

The opcodes that generate dynamic spectra are:

- Harmonic spectra: *buzz* and *gbuzz*
- Impulse generator: *mpulse*
- Band limited oscillators (analog modelled): *vco* and *vco2*

## FM Synthesis

The FM synthesis opcodes are:

- *foscil*
- *foscili*

## FM instrument models

- *fmb3*
- *fmbell*
- *fmmetal*
- *fmpercfl*
- *fmrhode*
- *fmvoice*
- *fmwurlie*

## Granular Synthesis

The granular synthesis opcodes are:

- *fof*
- *fof2*
- *fog*
- *grain*
- *grain2*
- *grain3*
- *granule*
- *sndwarp*
- *sndwarpst*
- *syncgrain*

## Linear and Exponential Generators

The opcodes that generate linear or exponential curves or segments are:

- *expon*
- *expseg*

- *expsega*
- *expsegr*
- *jspline*
- *line*
- *linseg*
- *linsegr*
- *loopseg*
- *lpshold*
- *lpsholdp*
- *rspline*
- *transeg*

## Envelope Generators

The following envelope generators are available:

- *adsr*
- *madsr*
- *mxadsr*
- *xadsr*

Consult the *Linear and exponential generators* section for additional methods to create envelopes.

## Models and Emulations

The following opcodes model or emulate the sounds of other instruments (some based on the STK toolkit by Perry Cook):

- *bamboo*
- *cabasa*
- *crunch*
- *dripwater*
- *gogobel*
- *guiro*
- *lorenz*
- *mandol*
- *marimba*
- *moog*

- *planet*
- *sandpaper*
- *sekere*
- *shaker*
- *sleighbells*
- *stix*
- *tambourine*
- *vibes*
- *voice*

## Phasors

The opcodes that generate a moving phase value:

- *phasor*
- *phasorbnk*

These opcodes are useful for usage with the *Table access* opcodes.

## Random (Noise) Generators

Opcodes that generate random numbers are:

- *betarnd*
- *bexprnd*
- *cauchy*
- *cusernd*
- *dusernd*
- *exprand*
- *gauss*
- *linrand*
- *noise*
- *pcauchy*
- *pinkish*
- *poisson*
- *rand*
- *randh*



- *randi*
- *rnd31*
- *random*
- *randomh*
- *randomi*
- *trirand*
- *unirand*
- *urd*
- *weibull*
- *jitter*
- *jitter2*

See *seed* which 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*, *randh*, *randi*, *rnd(x)* and *birnd(x)* are not affected by *seed*.

See also functions which generate random numbers in the section *Random Functions*.

## Sample Playback

Opcodes that implement sample playback and looping are:

- *bbcutm*
- *bbcuts*
- *flooper*
- *loscil*
- *loscil3*
- *lphasor*
- *lposcil*
- *lposcil3*
- *sndloop*
- *waveset*

See also the *Signal Input* section for other ways to input sound.

## Soundfonts

### Fluid Opcodes

The fluid family of opcodes wraps Peter Hannape's SoundFont 2 player, FluidSynth: *fluidEngine* for instantiating a FluidSynth engine, *fluidLoad* for loading SoundFonts, *fluidProgramSelect* for assign-

ing presets from a SoundFont to a FluidSynth engine's MIDI channel, *fluidNote* for playing a note on a FluidSynth engine's MIDI channel, *fluidCCi* for sending a controller message at i-time to a FluidSynth engine's MIDI channel, *fluidCCk* for sending a controller message at k-rate to a FluidSynth engine's MIDI channel. *fluidControl* for playing and controlling loaded Soundfonts (using 'raw' MIDI messages), *fluidOut* for receiving audio from a single FluidSynth engine, and *fluidAllOut* for receiving audio from all FluidSynth engines.

- *fluidAllOut*
- *fluidCCi*
- *fluidCCk*
- *fluidControl*
- *fluidEngine*
- *fluidLoad*
- *fluidNote*
- *fluidOut*
- *fluidProgramSelect*

## Old Soundfont opcodes

These opcodes can also use soundfonts to generate sound. The usage of the fluid Opcodes (above) is highly recommended instead of these opcodes.

- *sflist*
- *sfinstr*
- *sfinstr3*
- *sfinstr3m*
- *sfinstrm*
- *sfload*
- *sfpassign*
- *sfplay*
- *sfplay3*
- *sfplay3m*
- *sfplaym*
- *sfplist*
- *sfpreset*

## 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:

	1	2	3	4
1				
2				
3				
4				

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:

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0

The above table format of the connection matrix is for conceptual convenience only. The actual values shown in the 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 matrix 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://cSounds.com).

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

Opcodes that implement scanned synthesis are:

- *scanhammer*
- *scans*
- *scantable*
- *scanu*
- *xscanmap*
- *xscans*
- *xscansmap*
- *xscanu*

## Table Access

The opcodes that access tables are:

- *oscill*
- *oscilli*
- *osciln*
- *oscilx*
- *table*
- *table3*
- *tablei*

Opcodes ending in 'i' implement linear interpolation and opcodes ending in '3' implement cubic interpolation.

The following opcodes implement fast table reading/writing without boundary checks:

- *tab*
- *tab\_i*
- *tabw*
- *tabw\_i*

## Wave Terrain Synthesis

The opcode that uses wave terrain synthesis is *wterrain*.

## Waveguide Physical Modeling

The opcodes that implement waveguide physical modeling are:

- *pluck*
- *repluck*
- *wgbow*
- *wgbowedbar*
- *wgbrass*
- *wgclar*
- *wgflute*
- *wgpluck*
- *wgpluck2*

---

# Signal Input and Output

## File Input and Output

The opcodes for file input and output are:

- File open/close: *fiopen* and *ficlose*.
- File output: *dumpk*, *dumpk2*, *dumpk3*, *dumpk4*, *fout*, *fouti*, *foutir* and *foutk*
- File input: *readk*, *readk2*, *readk3*, *readk4*, *fin*, *fini* and *fink*
- Utilities for use with the *fout* opcodes: *clear*, *vincr*

## Signal Input

The opcodes that receive audio signals are:

- Synchronous input: *in*, *in32*, *inch*, *inh*, *ino*, *inq*, *ins* and *inx*
- File streaming: *diskin*, *diskin2* and *soundin*
- Software bus input: *chani* and *chnget*
- User defined channel input: *invalue*
- Direct to zak input: *inz*

## Signal Output

The opcodes that write audio signals are:

- Synchronous output: *out*, *out32*, *outc*, *outch*, *outh*, *outo*, *outq*, *outq1*, *outq2*, *outq3*, *outq4*, *outs*, *outs1*, *outs2* and *outx*
- Streaming output: *soundout* and *soundouts*
- Software bus input: *chano* and *chnset*
- User defined channel output: *outvalue*
- Direct from zak output: *outz*

The opcode *monitor* can be used for monitoring the complete output of *csound* (the output spout frame).

## Printing and Display

Opcodes for printing and displaying values are:

- *dispfft*

- *display*
- *flashtxt*
- *print*
- *printf*
- *printf\_i*
- *printk*
- *printk2*
- *printks*
- *prints*

## Sound File Queries

The opcodes that query information about files are:

- *filelen*
- *filenchnls*
- *filepeak*
- *filesr*

---

# Signal Modifiers

## Amplitude Modifiers and Dynamic processing

The opcodes that modify amplitude are:

- *balance*
- *compress*
- *clip*
- *dam*
- *gain*

## Convolution and Morphing

The opcodes that convolve and morph signals are:

- *convolve* also called *convle*
- *cross2*
- *dconv*
- *ftconv*
- *ftmorf*
- *pconvolve*

## Delay

### Fixed delays

- *delay*
- *delay1*
- *delayk*

### Delay Lines

- *delayr*
- *delayw*
- *deltap*
- *deltap3*



- *deltapi*
- *deltapn*
- *deltapx*
- *deltapxw*

## Variable delays

- *vdelay*
- *vdelay3*
- *vdelayx*
- *vdelayxs*
- *vdelayxq*
- *vdelayxw*
- *vdelayxwq*
- *vdelayxws*

## Multitap delays

- *multitap*

# Panning and Spatialization

## Amplitude spatialization

- *locsend*
- *locsig*
- *pan*
- *space*
- *spdist*
- *spsend*

## 3D spatialization with simulation of room acoustics

- *spat3d*
- *spat3di*
- *spat3dt*

## Vector Base Amplitude Panning

- *vbap16*
- *vbap16move*
- *vbap4*
- *vbap4move*
- *vbap8*
- *vbap8move*
- *vbaplsinit*
- *vbapz*
- *vbapzmove*

## Binaural spatialization

- *hrtfer*

## Reverberation

The opcodes one can use for reverberation are:

- *alpass*
- *babo*
- *comb*
- *freeverb*
- *nestedap*
- *nreverb* (also called *reverb2*)
- *reverb*
- *reverbsc*
- *valpass*
- *vcomb*

## Sample Level Operators

The opcodes one may use to modify signals are:

- *a(k)*
- *denorm*

- *diff*
- *downsamp*
- *fold*
- *i(k)*
- *integ*
- *interp*
- *ntrpol*
- *samphold*
- *upsamp*

## Signal Limiters

Opcodes that can be used to limit signals are:

- *limit*
- *mirror*
- *wrap*

## Special Effects

Opcodes that generate special effects are:

- *distort*
- *distort1*
- *flanger*
- *harmon*
- *phaser1*
- *phaser2*

## Standard Filters

### Resonant Low-pass filters

- *areson*
- *lowpass2*
- *lowres*
- *lowresx*

- *lpf18*
- *moogvcf*
- *moogladder*
- *reson*
- *resonr*
- *resonx*
- *resony*
- *resonz*
- *rezzy*
- *statevar*
- *svfilter*
- *tbvcf*
- *vlowres*

## Standard filters

- Hi-pass filters: *atone*, *atonex*
- Low-pass filters: *tone*, *tonex*
- Biquad filters: *biquad* and *biquada*.
- Butterworth filters: *butterbp*, *butterbr*, *butterhp*, *butterlp* (which are also called *butbp*, *butbr*, *buthp*, *butlp*)
- General filters: *clfilt*, *bqrez*

## Control signal filters

- *aresonk*
- *atonek*
- *lineto*
- *port*
- *portk*
- *resonk*
- *resonxk*
- *tlineto*
- *tonek*

## Specialized Filters

### High pass filters

- *dcblock*

### Parametric EQ

- *pareq*
- *rbjeq*

### Other filters

- *nlfilt*
- *filter2*
- *fofilter*
- *hilbert*
- *zfilter2*

## Waveguides

The opcodes that use waveguides to modify a signal are:

- *streson*
- *wguide1*
- *wguide2*

---

# Instrument Control

## Clock Control

The opcodes to start and stop internal clocks are:

- *clockoff*
- *clockon*

These clocks count CPU time. There are 32 independent clocks available. You can use the opcode *readclock* to read current values of a clock. See *Time Reading* for other timing opcodes.

## Conditional Values

The opcodes for conditional values are `==`, `>=`, `>`, `<`, `<=`, and `!=`.

## Duration Control Statements

The opcodes one can use to manipulate a note's duration are:

- *ihold*
- *turnoff*
- *turnoff2*
- *turnon*

For other realtime instrument control see *Real-time Performance Control* and *Instrument Invocation*.

## Introduction to FLTK Widgets and GUI controllers

Written by Gabriel Maldonado (<http://csounds.com/maldonado>)

Widgets allow the design of a custom Graphical User Interface to control an orchestra in real-time. They are derived from the open-source library FLTK (Fast Light Tool Kit). Such library is one of the fastest graphic libraries available, supports OpenGL and should be source compatible with different platforms (Windows, Linux, Unix and Mac OS). The subset of FLTK implemented in Csound provides the following types of objects:

- Containers
- Valuator
- Other widgets

Containers are widgets that contain other widgets such as panels, windows, etc. Csound provides the following container objects:

- Panels
- Scroll areas
- Pack
- Tabs
- Groups

The most useful objects are named valuator. These objects allow the user to vary synthesis parameter values in real-time. Csound provides the following valuator objects:

- Sliders
- Knobs
- Rollers
- Text fields
- Joysticks
- Counters

There are other widgets that are not valuator nor containers:

- Buttons
- Button banks
- Labels

Also there are some other opcodes useful to modify the widget appearance:

- Updating widget value.
- Setting primary and selection colors of a widget.
- Setting font type, size and color of widgets.
- Resizing a widget.
- Hiding and showing a widget.

At last, there are three important opcodes that allow the following actions:

- Running the widget thread.
- Loading snapshots containing the status of all valuator of an orchestra.
- Saving snapshots containing the status of all valuator of an orchestra.

Here is an example preview of Csound code for a window containing a valuator. Notice that all opcodes are `init-rate` and must be called only once per session. The best way to use them is to place them in the header section of an orchestra, externally to any instrument. Even though placing them

inside an instrument is not prohibited, unpredictable results can occur if that instrument is called more than once.

Each container is made up of a couple of opcodes: the first indicating the start of the container block and the last indicating the end of that container block. Some container blocks can be nested but they must not be crossed. After defining all containers, a widget thread must be run by using the special FLrun opcode that takes no arguments.

Here is an example of creating a window:

```
;*****
sr=48000
kr=480
ksmps=100
nchnls=1

;*** It is recommended to put almost all GUI code in the
;*** header section of an orchestra

        FLpanel          "Panel1",450,550 ;***** start of container
; some widgets should be contained here
        FLpanelEnd       ;***** end of container

        FLrun            ;***** runs the widget thread, it is always required!
instr 1
;put some synthesis code here
endin
;*****
```

The previous code simply creates a panel (an empty window because no widgets are defined inside the container).

The following example creates two panels and inserts a slider inside each of them:

```
;*****
sr=48000
kr=480
ksmps=100
nchnls=1

        FLpanel          "Panel1",450,550,100,100 ;***** start of container
gk1,iha FLslider          "FLslider 1", 500, 1000, 0 ,1, -1, 300,15, 20,50
        FLpanelEnd       ;***** end of container

        FLpanel          "Panel2",450,550,100,100 ;***** start of container
gk2,ihb FLslider          "FLslider 2", 100, 200, 0 ,1, -1, 300,15, 20,50
        FLpanelEnd       ;***** end of container

        FLrun            ;***** runs the widget thread, it is always required!

instr 1
;put some synthesis code here
; gk1 and gk2 variables that contain the output of valuator
; widgets previously defined, can be used inside any instrument
endin
;*****
```

All widget opcodes are init-rate opcodes, even if valuator's output k-rate variables. This happens because an independent thread is run based on a callback mechanism. It consumes very few processing



resources since there is no need of polling. (This differs from other MIDI based controller opcodes.) So you can use any number of windows and valuator without degrading the real-time performance.

Since FLTK toolkit is still in evolution process, opcode syntax provided in Csound could be modified in future version. This could cause some incompatibilities between orchestras of a determinate version. However it should not be hard to modify early orchestras in order to make them compatible with later versions.

For more information, see the following sections.

## FLTK Containers

The opcodes for FLTK containers are *FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*, and *FLtabsEnd*.

## FLTK Valuator

The opcodes for FLTK valuator are *FLcount*, *FLjoy*, *FLkeyb*, *FLknob*, *FLroller*, *FLslider*, and *FLtext*.

## Other FLTK Widgets

Other FLTK widget opcodes are *FLbox*, *FLbutBank*, *FLbutton*, *FLprintk*, *FLprintk2*, and *FLvalue*,

## Modifying FLTK Widget Appearance

The following opcodes modify FLTK widget appearance:

- *FLcolor2*
- *FLhide*
- *FLlabel*
- *FLsetAlign*
- *FLsetBox*
- *FLsetColor*
- *FLsetColor2*
- *FLsetFont*
- *FLsetPosition*
- *FLsetSize*
- *FLsetText*
- *FLsetTextColor*
- *FLsetTextSize*
- *FLsetTextType*
- *FLsetVal\_i*
- *FLsetVal*
- *FLshow*

## General FLTK Widget-related Opcodes

The general FLTK widget-related opcodes are *FLgetsnap*, *FLloadsnap*, *FLrun*, *FLsavesnap*, *FLsetsnap*, and *FLupdate*.

## FLTK Slider Bank

The opcode for the FLTK slider bank is *FLslidBnk*.

# Instrument Invocation

The opcodes one can use to create score events from within an orchestra are:

- *event*
- *event\_i*
- *schedule*
- *schedwhen*
- *schedkwhen*
- *schedkwhennamed*

## Macros

The opcodes one can use to create, call, or undefine macros are:

- *#define*
- *\$NAME*
- *#ifdef*
- *#end*
- *#include*
- *#undef*

These opcodes refer to orchestra macros, for score macros refer to *Score Macros*.

# Program Flow Control

The opcodes to manipulate which orchestra statements are executed are:

- *cgoto*
- *cigoto*
- *ckgoto*
- *cngoto*
- *elseif*

- *else*
- *endif*
- *goto*
- *if*
- *igoto*
- *kgoto*
- *loop\_ge*
- *loop\_gt*
- *loop\_le*
- *loop\_lt*
- *tigoto*
- *timeout*



### Warning

Some of these opcodes work at i-rate even if they contain k- or a- rate comparisons. See the *Reinitialization* section.

## Real-time Performance Control

Opcodes that monitor and control real-time performance are:

- *active*
- *cpuprc*
- *maxalloc*
- *prealloc*

The running csound process can be terminated using *exitnow*.

## Reinitialization

The opcodes that can generate another initialization phase are:

- *reinit*
- *rigoto*
- *rireturn*

## Sensing and Control

## TCL/TK widgets

- *button*
- *checkbox*
- *control*
- *setctrl*

## Keyboard and mouse sensing

- *sensekey* (also called *sense*)
- *xyin*

## Envelope followers

- *follow*
- *follow2*
- *peak*
- *rms*

## Tempo and Pitch estimation

- *pitch*
- *pitchamdf*
- *tempest*

## Tempo and Sequencing

- *tempo*
- *tempoval*
- *seqtime*
- *trigger*
- *trigseq*

## Sub-instrument Control

These opcodes let one define and use a sub-instrument:

- *subinstr*
- *subinstrinit*

See also the UDO and *Orchestra Macros* Macros section for similar functionality.

## Time Reading

Opcodes one can use to read time values are:

- *readclock*
- *rtclock*
- *timeinstk*
- *timeinsts*
- *times*
- *timek*

You can also set up counters using *clockoff* and *clockon*.

---

# Function Table Control

Refer to the *f score statement*, the *fgen* and the *GEN Routines* section for information on creating tables.

## Table Queries

Opcodes the query tables for information are:

- For tables loaded from a sound file (using *GENOI*): *ftchnls*, *ftlen*, *ftlptim* and *ftsr*
- For any table: *nsamp*, *ftlen*, *tbleng*

## Read/Write Operations

Opcodes that read and write to a table are:

- *ftloadk*
- *ftload*
- *ftsavek*
- *ftsave*
- *tablecopy*
- *tablegpw*
- *tableicopy*
- *tableigpw*
- *tableimix*
- *tableiw*
- *tablemix*
- *tablera*
- *tablew*
- *tablewa*
- *tablewkt*

## Table Reading with Dynamic Selection

Opcodes that let one dynamically (at k-rate) select tables are:

- *tableikt*
- *tablekt*

- *tablext*

---

# Mathematical Operations

## Amplitude Converters

Opcodes to convert between different amplitude measurements are:

- *ampdb*
- *ampdbfs*
- *dbamp*
- *dbfsamp*

Use *rms* to find the rms value of a signal. See also *Odbfs* for another way to handle amplitudes in csound.

## Arithmetic and Logic Operations

Opcodes that perform arithmetic and logic operations are -, +, &&, //, \*, /, ^, and %.

See the *Conditional Values* section and the *if* family of opcodes for usage of logical operators.

## Mathematical Functions

Opcodes that perform mathematical functions are:

- *abs*
- *ceil*
- *divz*
- *exp*
- *floor*
- *frac*
- *int*
- *log*
- *log10*
- *logbtwo*
- *powoftwo*
- *round*
- *sqrt*

## Opcode Equivalents of Functions

Opcodes that perform the equivalent of mathematical functions are:



- *mac*
- *maca*
- *pow*
- *product*
- *sum*

## Random Functions

Opcodes that perform random functions are:

- *birnd*
- *rnd*

See the section *Random (Noise) Generators* for opcodes that generate random signals.

## Trigonometric Functions

Opcodes that perform trigonometric functions are:

- *cos*, *cosh* and *cosinv*
- *sin*, *sinh* and *sininv*
- *tan*, *tanh*, *taninv*, and *taninv2*.

---

# Pitch Converters

## Functions

Opcodes that provide common pitch functions are *cent*, *cpsoct*, *cpspch*, *db*, *octave*, *octcps*, *octpch*, *pchoct*, and *semitone*.

## Tuning Opcodes

Opcodes that provide tuning functions are *cps2pch*, *cpsxpch*, *cpstun*, and *cpstuni*.

---

# Real-time MIDI Support

## MIDI input

The following opcodes can receive MIDI information:

- MIDI information for any instruments: *aftouch*, *chanctrl* and *polyaft*, *pchbend*.
- MIDI information for MIDI-triggered instruments: *veloc*, *midictrl* and *notnum*. See also *Converters*.
- MIDI Controller input for any instrument: *midic7*, *midic14* and *midic21*.
- MIDI Controller input for MIDI-triggered instruments: *ctrl7*, *ctrl14* and *ctrl21*.
- MIDI controller value initialization: *initc7*, *initc14* and *initc21*.
- For designing instruments that can be driven by MIDI or score events: *midichannelaftertouch*, *midichn*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch* and *midiprogramchange*.

## MIDI Message Output

Opcodes that send MIDI output are:

- *mdelay*
- *nrpn*
- *outiat*
- *outic*
- *outic14*
- *outipat*
- *outipb*
- *outipc*
- *outkat*
- *outkc*
- *outkc14*
- *outkpat*
- *outkpb*
- *outkpc*

## Generic Input and Output

Opcodes for generic MIDI input and output are *midiin* and *midiout*.

## Converters

The following opcodes can convert MIDI information from a MIDI-triggered instrument instance:

- MIDI note number to frequency converters: *cpsmidi*, *cpsmidib*, *cpstmid*, *octmidi*, *pchmidi*, *pchmidib* and *octmidib*.
- MIDI velocity to amplitude converters: *ampmidi*.

## Event Extenders

Opcodes that let one extend the duration of an event are:

- *release*
- *xtratin*

## Note-on/Note-off Output

Opcodes to output MIDI note on or off messages are:

- *midion*
- *midion2*
- *moscil*
- *noteoff*
- *noteon*
- *noteondur*
- *noteondur2*

## System Realtime Messages

Opcodes for System Realtime MIDI messages are: *mclock* and *mrtmsg*.

## Slider Banks

Opcodes for slider banks of MIDI controls are:

- *slider8*
- *slider8f*
- *slider16*
- *slider16f*
- *slider32*

- *slider32f*
- *slider64*
- *slider64f*
- *s16b14*
- *s32b14*

---

# Spectral Processing

See the section *Additive Synthesis/Resynthesis* for the basic resynthesis opcodes.

## Short-time Fourier Transform (STFT) Resynthesis



### Use of PVOC-EX files with the old Csound pvoc opcodes

All the original pvoc opcodes can now read a PVOC-EX file, as well as the native non-portable file format. As the PVOC-EX file uses a double-size analysis window, users may find that this gives a useful improvement in quality, for some sounds and processes, despite the fact that the resynthesis does not use the same window size.

Apart from the window size parameter, the main difference between the original .pv format and PVOC-EX is in the amplitude range of analysis frames. While rescaling is applied, so that no significant difference in output level is experienced, whichever file format is used, some slight loss of amplitude can still arise, as the double window usage itself modifies frame amplitudes, of which the resynthesis code is unaware. Note that all the original pvoc opcodes expect a mono analysis file, and multi-channel PVOC-EX files will accordingly be rejected.

Opcodes that implement STFT resynthesis are:

- *ktableseg*
- *pvadd*
- *pvbufread*
- *pvcross*
- *pvinterp*
- *pvoc*
- *pvread*
- *tableseg*
- *tablexseg*
- *vpvoc*

Use the utility *PVANAL* to generate pv analysis files.

## Linear Predictive Coding (LPC) Resynthesis

The linear predictive coding resynthesis opcodes are:

- *lpfreson*
- *lpinterp*
- *lpread*

- *lpreson*
- *lpslot*

LPC analysis files can be created using the *LPANAL* utility.

## Non-standard Spectral Processing

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 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.

The opcodes for non-standard spectral processing are *specaddm*, *specdiff*, *specdisp*, *specfilt*, *spechist*, *specptrk*, *specscal*, *specsum*, and *spectrum*.

## Tools for Real-time Spectral Processing (pvs opcodes)

With these opcodes, two new core facilities are added to Csound. They offer improved audio quality, and fast performance, enabling high-quality analysis and resynthesis (together with transformations) to be applied in real-time to live signals. The original Csound phase vocoder remains unaltered; the new opcodes use an entirely separate set of functions based on “pvoc.c” in the CARL distribution, written by Mark Dolson.

The Csound *dnoise* and *srconv* utilities (also by Dolson, from CARL) also use this pvoc engine. CARL pvoc is also the basis for the phase vocoder included in the Composer's Desktop Project. A few small but important modifications have been made to the original CARL code to support real-time streaming.

1. Support for the new PVOC-EX analysis file format. This is a fully portable (cross-platform) open file format, supporting three analysis formats, and multi-channel signals. Currently only the standard amplitude+frequency format has been implemented in the opcodes, but the file format itself supports amplitude+phase and complex (real-imaginary) formats. In addition to the new opcodes, the original Csound pvoc opcodes have been extended (and thereby with enhanced audio quality in some cases) to read PVOC-EX files as well as the original (non-portable) format.

Full details of the structure of a PVOC-EX file are available via the website: <http://www.cs.bath.ac.uk/~jpff/NOS-DREAM/researchdev/pvocex/pvocex.html>. This site also gives details of the freely available console programs pvocex and pvocex2 which can be used to create PVOC-EX files in all supported formats.

2. A new frequency-domain signal type, fully streamable, with *f* as the leading character. In this document it is conveniently referred to as an *fsig*. Primary support for fsigs is provided by the opcodes pvsanal and pvsynth, which perform conventional phase vocoder overlap-add analysis and resynthesis, independently of the orchestra control-rate. The only requirement is that the control-rate *kr* be higher than or equal to the analysis rate, which can be expressed by the requirement that  $ksmps \leq overlap$ , where *overlap* is the distance in samples between analysis frames, as specified for pvsanal. As *overlap* is typically at least 128, and more usually 256, this is not an onerous restriction in practice. The opcode pvsinfo can be used at init time to acquire the properties of an fsig.

The fsig enables the nominal separation between the analysis and resynthesis stages of the phase vocoder to be exposed to the Csound programmer, so that not only can alternatives be employed for either or both of these stages (not only oscillator-bank resynthesis, but also the

generation of synthetic fsig streams), but opcodes, operating on the fsig stream, can themselves become more elemental. Thus the fsig enables the creation of a true streaming plugin framework for frequency domain signals. With the old pvoc opcodes, each opcode is required to act as a resynthesiser, so that facilities such as pitch scaling are duplicated in each opcode; and in many cases the opcodes are parameter-rich. The separation of analysis and synthesis stages by means of the fsig encourages the development of a wide range of simple building-block opcodes implementing one or two functions, with which more elaborate processes can be constructed.

This is very much a preliminary and experimental release, and it is possible that the precise definition of the opcodes may change, in response to user feedback. Also, clearly, many new possibilities for opcodes are opened up; these factors may also have a retrospective influence on the opcodes presented here.

Note that some opcode parameters currently have restricted or missing implementation. This is at least in part in order to keep the opcodes simple at this stage, and also because they highlight important design issues on which no decision has yet been made, and on which opinions from users are sought.

One important point about the new signal type is that because the analysis rate is typically much lower than kr, new analysis frames are not available on each k-cycle. Internally, the opcodes track ksmps, and also maintain a frame counter, so that frames are read and written at the correct times; this process is generally transparent to the user. However, it means that k-rate signals only act on an fsig at the analysis rate, not at each k-cycle. The opcode pvsftw returns a k-rate flag that is set when new fsig data is valid.

Because of the nature of the overlap-add system, the use of these opcodes incurs a small but significant delay, or latency, determined by the window size ( $\max(\text{ifftsize}, \text{iwinsize})$ ). This is typically around 23msecs. In this first release, the delay is slightly in excess of the theoretical minimum, and it is hoped that it can be reduced, as the opcodes are further optimized for real-time streaming.

The opcodes for real-time spectral processing are *pvsadsyn*, *pvsanal*, *pvsacross*, *pvsfread*, *pvsftr*, *pvsftw*, *pvsinfo*, *pvsmaska*, and *pvsynth*.

In addition there are a number of opcodes available as plugins in Csound5. These are *pvscent*, *pvsdemix*, *pvsfreeze*, *pvscale*, *pvshift*, *pvsifd*, *pvsinit*, *pvmix*, *pvmooth*, *pvsfilter*, *pvsblur*, *pvsstencil*, *pvsarp*, *pvsvoc*

A number of opcodes are designed to generate and process streaming partials tracks data. these are *partials*, *trcross*, *trfilter*, *trsplit*, *trmix*, *trscale*, *trshift*, *trlowest*, *trhighest* *tradsyn*, *sinsyn*, *resyn*, *binit*

## ATS Spectral Processing

These opcodes can read, transform and resynthesize ATS analysis files. Please note that you need the ATS application to produce analysis files. From the ATS Reference Manual:

*"ATS is a software library of functions for spectral Analysis, Transformation, and Synthesis of sound based on a sinusoidal plus critical-band noise model. A sound in ATS is a symbolic object representing a spectral model that can be sculpted using a variety of transformation functions."*

For more information on ATS visit: <http://www-ccrma.stanford.edu/~juan/ATS.html>.

ATS analysis files can be produced using the ATS software or the csound utility *ATSA*. The opcodes for ATS processing are:

- *ATSinfo*: reads data out of the header of an ATS file.
- *ATSread*, *ATSreadnz*, *ATSbufread*, *ATSinterpread*, *ATSpartialtap*: read data from an ATS file or buffer.



- *ATSadd*, *ATSaddnz*, *ATScross*, *ATSinnoi*: Resynthesize sound.

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

## Loris Opcodes



### Note

These opcodes are an optional component of Csound5. You can check if they are installed by using the command 'csound -z' which lists all available opcodes.

The Loris family of opcodes wraps: *lorisread* which imports a set of bandwidth-enhanced partials from a SDIF-format data file, applying control-rate frequency, amplitude, and bandwidth scaling envelopes, and stores the modified partials in memory; *lorismorph*, which morphs two stored sets of bandwidth-enhanced partials and stores a new set of partials representing the morphed sound. The morph is performed by linearly interpolating the parameter envelopes (frequency, amplitude, and bandwidth, or noisiness) of the bandwidth-enhanced partials according to control-rate frequency, amplitude, and bandwidth morphing functions, and *lorisout*, which renders a stored set of bandwidth-enhanced partials using the method of Bandwidth-Enhanced Additive Synthesis implemented in the Loris software, applying control-rate frequency, amplitude, and bandwidth scaling envelopes.

Note that a version of Loris with a Python interface is packaged as part of the CsoundVST distribution, so it is possible to perform both analysis and synthesis with Loris in Csound 5.

For more information about sound morphing and manipulation using Loris and the Reassigned Bandwidth-Enhanced Additive Model, visit the Loris web site at [www.cerlsoundgroup.org/Loris](http://www.cerlsoundgroup.org/Loris) [<http://www.cerlsoundgroup.org/Loris>].

## Examples

### Example 1.

```
;
; Play the partials in clarinet.sdif
; from 0 to 3 sec with 1 ms fadetime
; and no frequency , amplitude, or
; bandwidth modification.
;
instr 1
    ktime      linseg      0, p3, 3.0      ; linear time function from 0 to 3 seconds
               lorisread   ktime, "clarinet.sdif", 1, 1, 1, 1, .001
    asig       lorisplay   1, 1, 1, 1
               out         asig
endin
```

### Example 2.

```
; Play the partials in clarinet.sdif
; from 0 to 3 sec with 1 ms fadetime
; adding tuning and vibrato, increasing the
; "breathiness" (noisiness) and overall
; amplitude, and adding a highpass filter.
;
instr 2
  ktime      linseg      0, p3, 3.0      ; linear time function from 0 to 3 seconds
  ; compute frequency scale for tuning
  ; (original pitch was G#4)
  ifscale =      cpspch(p4)/cpspch(8.08)

  ; make a vibrato envelope
  kvenv      linseg      0, p3/6, 0, p3/6, .02, p3/3, .02, p3/6, 0, p3/6, 0
  kvib       oscil       kvenv, 4, 1      ; table 1, sinusoid

  kbwenv     linseg      1, p3/6, 1, p3/6, 2, 2*p3/3, 2
  lorisread  ktime, "clarinet.sdif", 1, 1, 1, 1, .001
  a1         lorisplay   1, ifscale+kvib, 2, kbwenv
  a2         atone       a1, 1000         ; highpass filter, cutoff 1000 Hz
  out        a2
endin
```

The instrument in the first example synthesizes a clarinet tone from beginning to end using partials derived from reassigned bandwidth-enhanced analysis of a three-second clarinet tone, stored in a file, `clarinet.sdif`. The instrument in Example 2 adds tuning and vibrato to the clarinet tone synthesized by instr 1, boosts its amplitude and noisiness, and applies a highpass filter to the result. The following score can be used to test both of the instruments described above.

```
      ; make sinusoid in table 1
f 1 0 4096 10 1

      ; play instr 1
      ;   strt  dur
i 1   0       3
i 1   +       1
i 1   +       6
s

      ; play instr 2
      ;   strt  dur  ptch
i 2   1       3    8.08
i 2   3.5     1    8.04
i 2   4       6    8.00
i 2   4       6    8.07

e
```

### Example 3.

```
; Morph the partials in clarinet.sdif into the
; partials in flute.sdif over the duration of
; the sustained portion of the two tones (from
; .2 to 2.0 seconds in the clarinet, and from
; .5 to 2.1 seconds in the flute). The onset
; and decay portions in the morphed sound are
; specified by parameters p4 and p5, respectively.
; The morphing time is the time between the
; onset and the decay. The clarinet partials are
; shifted in pitch to match the pitch of the flute
```

```
; tone (D above middle C).
;
instr 1
  ionset    =          p4
  idecay    =          p5
  itmorph   =          p3 - (ionset + idecay)
  ipshift   =          cpspch(8.02)/cpspch(8.08)

  ktcl      linseg      0, ionset, .2, itmorph, 2.0, idecay, 2.1    ; clarinet
  ktfl      linseg      0, ionset, .5, itmorph, 2.1, idecay, 2.3    ; flute ti
  kmurph     linseg      0, ionset, 0, itmorph, 1, idecay, 1
             lorisread   ktcl, "clarinet.sdif", 1, ipshift, 2, 1, .001
             lorisread   ktfl, "flute.sdif", 2, 1, 1, 1, .001
             lorismorph   1, 2, 3, kmurph, kmurph, kmurph
  asig      lorisplay    3, 1, 1, 1
             out          asig
endin
```

#### Example 4.

```
; Morph the partials in trombone.sdif into the
; partials in meow.sdif. The start and end times
; for the morph are specified by parameters p4
; and p5, respectively. The morph occurs over the
; second of four pitches in each of the sounds,
; from .75 to 1.2 seconds in the flutter-tongued
; trombone tone, and from 1.7 to 2.2 seconds in
; the cat's meow. Different morphing functions are
; used for the frequency and amplitude envelopes,
; so that the partial amplitudes make a faster
; transition from trombone to cat than the frequencies.
; (The bandwidth envelopes use the same morphing
; function as the amplitudes.)
;
instr 2
  ionset    =          p4
  imorph    =          p5 - p4
  irelease  =          p3 - p5

  kttbn     linseg      0, ionset, .75, imorph, 1.2, irelease, 2.4
  ktmeow    linseg      0, ionset, 1.7, imorph, 2.2, irelease, 3.4

  kmfreq    linseg      0, ionset, 0, .75*imorph, .25, .25*imorph, 1, irelease
  kmamp      linseg      0, ionset, 0, .75*imorph, .9, .25*imorph, 1, irelease,

             lorisread   kttbn, "trombone.sdif", 1, 1, 1, 1, .001
             lorisread   ktmeow, "meow.sdif", 2, 1, 1, 1, .001
             lorismorph   1, 2, 3, kmfreq, kmamp, kmamp
  asig      lorisplay    3, 1, 1, 1
             out          asig
endin
```

The instrument in the first morphing example performs a sound morph between a clarinet tone and a flute tone using reassigned bandwidth-enhanced partials stored in `clarinet.sdif` and `flute.sdif`.

The morph is performed over the sustain portions of the tones, 2. seconds to 2.0 seconds in the case of the clarinet tone and .5 seconds to 2.1 seconds in the case of the flute tone. The time index functions, `ktcl` and `ktfl`, align the onset and decay portions of the tones with the specified onset and decay times for the morphed sound, specified by parameters `p4` and `p5`, respectively. The onset in the morphed sounds is purely clarinet partial data, and the decay is purely flute data. The clarinet par-

tials are shifted in pitch to match the pitch of the flute tone (D above middle C).

The instrument in the second morphing example performs a sound morph between a flutter-tongued trombone tone and a cat's meow using reassigned bandwidth-enhanced partials stored in `trombone.sdif` and `meow.sdif`. The data in these SDIF files have been channelized and distilled to establish correspondences between partials.

The two sets of partials are imported and stored in memory locations labeled 1 and 2, respectively. Both of the original sounds have four notes, and the morph is performed over the second note in each sound (from .75 to 1.2 seconds in the flutter-tongued trombone tone, and from 1.7 to 2.2 seconds in the cat's meow). The different time index functions, `ktbn` and `ktmeow`, align those segments of the source and target partial sets with the specified morph start, morph end, and overall duration parameters. Two different morphing functions are used, so that the partial amplitudes and bandwidth coefficients morph quickly from the trombone values to the cat's-meow values, and the frequencies make a more gradual transition. The morphed partials are stored in a memory location labeled 3 and rendered by the subsequent `lorisplay` instruction. They could also have been used as a source for another morph in a three-way morphing instrument. The following score can be used to test both of the instruments described above.

```
; play instr 1
;   strt  dur  onset  decay
i 1    0    3    .25   .15
i 1    +    1    .10   .10
i 1    +    6    1.    1.
s

; play instr 2
;   strt  dur  morph_start  morph_end
i 2    0    4    .75        2.75

e
```

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz ([loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org) [mailto:[loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org)]).

It is patterned after a prototype implementation of the *lorisplay* unit generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael Gogins.

---

# Strings

String variables are variables with a name starting with S or gS (for a local or global string variable, respectively), and can store any string with a maximum length defined by the `--max_str_len` command line flag (255 characters by default). These variables can be used as input argument to any opcode that expects a quoted string constant, and can be manipulated at initialization or performance time with the opcodes listed below.

It is also possible to use string p-fields. The string p-field can be used by many orchestra opcodes directly, or it can be copied to a string variable first:

```
a1      diskln2 p5, 1
```

```
Sname strget p5  
a1      diskln2 Sname, 1
```



## Note

String variables and related opcodes are not available in Csound versions older than 5.00.

Strings can also be linked to a number using *strset* and *strget*.

## String Conversion Opcodes

Csound 5 also has improvements in parsing string constants. It is possible to specify a multi-line string by enclosing it within `{{` and `}}` instead of the usual double quote characters (note that the length of string constants is not limited, and is not affected by the `--max_str_len` option), and the following escape sequences are automatically converted:

- `\a` alert bell
- `\b` backspace
- `\n` new line
- `\r` carriage return
- `\t` tab
- `\\` a single `'` character
- `\nnn` the character of which the ASCII code (in octal) is `nnn`

These opcodes perform operations on string variables (note: most of the opcodes run at init time only, and have a version with a "k" suffix that runs at both init and performance time; exceptions to this rule include *puts* and *strget*):

- *strcpy* - Assigns to a string variable.
- *strcat* - Concatenates strings, and stores the result in a variable.
- *strcmp* - Compares strings.

- *strget* - Assigns to a string variable, from strset table at the specified index, or string score p-field.
- *strlen* - Returns the length of a string.
- *sprintf* - printf-style formatted output conversion, storing the result in a string variable.
- *puts* - Prints a string constant or variable.
- *strtod* - Converts string value to a floating point value at i-rate.
- *strtol* - Converts string value to signed integer at i-rate.
- *strchar* - Returns the ASCII code of a character in a string.
- *strsub* - Returns a substring of the input string.
- *strlower* - Converts a string to lower case.
- *strupper* - Converts a string to upper case.
- *strindex* - Returns the first occurrence of a string in another string.
- *strrindex* - Returns the last occurrence of a string in another string.

---

# Vectorial Opcodes

## Tables of vectors operators

Vectorial opcodes support read/write access to arrays of vectors (or arrays of arrays).

*vtablei*, *vtablek*, *vtablea*, *vtablewi*, *vtablewk*, *vtablewa*, *vtabi*, *vtabk*, *vtaba*, *vtabwi*, *vtabwk* and *vtabwa*.

Author: Gabriel Maldonado

Originally available on CsoundAV.

Added to csound5.

## Operations Between a Vectorial and a Scalar Signal

These opcodes perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

Operations Between a Vectorial and a Scalar Signal: *vadd*, *vmult*, *vpow* and *vexp*.

Author: Gabriel Maldonado

Originally available on CsoundAV.

Added to csound5.

## Operations Between two Vectorial Signals

These opcodes perform operations between two vectorial control signals, that is, each element of the first vector is processed (only) with the corresponding element of the other vector. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vector that overrides the old values of *ifn1*.

All these opcodes work at k-rate.

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

Operations Between two Vectorial Signals: *vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy*, *vcopy\_i* and *vmap*.

Author: Gabriel Maldonado

Originally available on CsoundAV.

Added to csound5.

## Vectorial Envelope Generators

These opcodes are similar to *linseg* and *expseg*, but operate with vectorial signals instead of with scalar signals.

Output is a vectorial control signal hosted by *ifnout* (that must be previously allocated), while each break-point of the envelope is actually a vector of values. All break-points must contain the same number of elements (*ielements*).

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

Operations Between a Vectorial and a Scalar Signal: *vlinseg* and *vexpseg*.

Author: Gabriel Maldonado Originally available on CsoundAV. Added to csound5.

## Limiting and wrapping of vectorial control signals

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate with a vectorial signal instead of with a scalar signal.

Result overrides old values of *ifn1*, if these are out of min/max interval. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

All these opcodes work at k-rate.

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

Operations Between two Vectorial Signals: *vlimit*, *vwrap* and *vmirror*.

Author: Gabriel Maldonado

Originally available on CsoundAV.

Added to csound5.

## Vectorial Control-rate Delay Paths

Vectorial Control-rate Delay Paths: *vdelayk*, *vport* and *vecdelay*.

Author: Gabriel Maldonado

Originally available on CsoundAV.

Added to csound5.

## Vectorial Random Signal Generators

These opcodes generate vectors of random numbers to be stored in tables. They generate a sort of 'vectorial band-limited noise'. All these opcodes work at k-rate.

Vectorial random signal generators: *vrandh* and *vrandi*.

Author: Gabriel Maldonado

Originally available on CsoundAV.

Added to csound5.

Cellular automata vectors:



*vcella.*

---

# 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.

Opcodes for the zak patch system are:

- Audio Rate: *zaci*, *zakinit*, *zamid*, *zar*, *zarg*, *zaw* and *zawm*.
- Control Rate: *zkci*, *zkmid*, *zkr*, *zkw*, and *zkwm*.
- At initialization: *zir*, *ziw* and *ziwm*

---

# DSSI and LADSPA for Csound

*dssi4cs* enables the use of DSSI and LADSPA plugin effects and synthesizers within Csound on Linux. The following opcodes are available:

- *dssiinit* - Loads a plugin.
- *dssiactivate* - Activates or deactivates a plugin if it has this facility
- *dssilist* - Lists all available plugins found in the LADSPA\_PATH and DSSI\_PATH global variables.
- *dssiaudio* - Process audio using a Plugin.
- *dssictls* - Send control information to a plugin's control port.

See the entry for *dssiinit* for a usage example.

## **Credits.**

2005

By: Andres Cabrera

Uses code from Richard Furse's LADSPA sdk.

---

# VST for Csound

*vst4cs* enables the use of VST plugin effects and synthesizers within Csound. The following op-codes are available:

- *vstinit* - Loads a plugin.
- *vstaudio*, *vstaudiog* - Returns a plugin's output.
- *vstmidiout* - Sends MIDI data to a plugin.
- *vstparamset*, *vstparamget* - Sends and receives automation data to and from the plugin.
- *vstnote* - Sends a MIDI note with definite duration.
- *vstinfo* - Outputs the Parameter and Program names for a plugin.
- *vstbankload* - Loads an .fxb Bank.
- *vstprogset* - Sets a Program in an .fxb Bank.
- *vstedit* - Opens the GUI editor for the plugin, when available.

*vst4cs* currently works for Windows only.

## Credits.

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

---

# OSC and Csound

*OSC* enables interaction between different audio processes, and in particular between Csound and other synthesis engines. The following opcodes are available:

- *OSCinit* - Start an OSC listener thread/
- *OSClisten* - Receive OSC messages.
- *OSCsend* - Send an OSC message.

## **Credits.**

By: John ffitich with the liblo library as inspiration and support.

---

# Mixer Opcodes

The Mixer family of opcodes provides a global mixer for Csound. The Mixer opcodes include *MixerSend* for sending (that is, mixing in) an arate signal from any instrument to a channel of a mixer buss, *MixerReceive* for receiving an arate signal from a channel of any mixer buss in any instrument, *MixerSetLevel* for controlling (at krte) the level of the signal sent from a particular send to a particular buss, *MixerGetLevel* for reading (at krte) the level for sending a signal from a particular send to a particular buss, and *MixerClear* for resetting the busses to zero before the next kperiod of a performance.

---

# Python Opcodes

## Introduction

Using the Python opcode family, you can interact with a Python interpreter embedded in Csound in five ways:

1. Initialize the Python interpreter (the *pyinit* opcodes),
2. Run a statement (the *pyrun* opcodes),
3. Execute a script (the *pyexec* opcodes),
4. Invoke a callable and pass arguments (the *pycall* opcodes),
5. Evaluate an expression (the *pyeval* opcodes), or
6. Change the value of a Python object, possibly creating a new Python object (the *pyassign* opcodes);

and you can do any of these things:

1. At i-time or at k-time,
2. In the global Python namespace, or in a namespace specific to an individual instance of a Csound instrument (local or "l" context),
3. And can you can retrieve from 0 to 8 return values from callables that accept N parameters...

...this means that there are many Python-related opcodes. But all of these opcodes share the same *py* prefix, and have a regular naming scheme:

"py" + [optional context prefix] + [action name] + [optional x-time prefix]

## Orchestra Syntax

Blocks of Python code, and indeed entire scripts, can be embedded in Csound orchestras using the `{{` and `}}` directives to enclose the script, as follows:

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

giSinusoid ftgen 0, 0, 8192, 10, 1

pyruni {{
import random

pool = [(1 + i/10.0) ** 1.2 for i in range(100)]

def get_number_from_pool(n, p):
    if random.random() < p:
        i = int(random.random() * len(pool))
```

```
        pool[i] = n
    return random.choice(pool)
}}

instr 1
    k1 oscil 1, 3, giSinusoid
    k2 pycall1 "get_number_from_pool", k1 + 2, p4
    printk 0.01, k2
endin
```

**Credits.**

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved.

Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.



---

## **Part III. Reference**

---

---

## Table of Contents

Orchestra Opcodes and Operators .....	154
!= .....	155
#define .....	157
#include .....	160
#undef .....	162
#ifdef .....	163
\$NAME .....	165
% .....	168
&& .....	170
> .....	172
>= .....	174
< .....	176
<= .....	178
* .....	180
+ .....	182
- .....	184
/ .....	186
= .....	188
== .....	190
^ .....	192
round .....	194
.....	195
Odbfs .....	197
& .....	199
.....	200
⊃ .....	201
# .....	202
a .....	203
abetarand .....	205
abexprnd .....	206
abs .....	207
acauchy .....	209
active .....	210
adsr .....	213
adsyn .....	216
adsynt .....	218
adsynt2 .....	221
aexprand .....	222
aftouch .....	223
agauss .....	225
agogobel .....	226
alinrand .....	227
alpass .....	228
ampdb .....	230
ampdbfs .....	232
ampmidi .....	234
apcauchy .....	236
apoisson .....	237
apow .....	238
areson .....	239
aresonk .....	241
atone .....	242
atonek .....	244
atonex .....	245
atrirand .....	246
ATSadd .....	247
ATSaddnz .....	249

---

ATSBufread .....	251
ATScross .....	253
ATSinfo .....	255
ATSinterpread .....	257
ATSread .....	258
ATSreadnz .....	260
ATSpartialtap .....	262
ATSinnoi .....	263
aunirand .....	265
aweibull .....	266
babo .....	267
balance .....	271
bamboo .....	273
barmodel .....	275
bbcutm .....	277
bbcuts .....	282
betarand .....	284
bexprnd .....	286
bformenc .....	288
bformdec .....	290
binit .....	292
biquad .....	293
biquada .....	295
birnd .....	296
bqrez .....	298
butbp .....	300
butbr .....	301
buthp .....	302
butlp .....	303
butterbp .....	304
butterbr .....	306
butterhp .....	308
butterlp .....	310
button .....	312
buzz .....	313
cabasa .....	315
cauchy .....	317
ceil .....	319
cent .....	320
cggoto .....	322
chanctrl .....	324
changed .....	325
chani .....	327
chano .....	328
checkbox .....	329
chn .....	331
chnclear .....	333
chnexport .....	334
chnget .....	336
chnmix .....	338
chnparams .....	339
chnset .....	340
cigoto .....	342
ckgoto .....	344
clear .....	346
clfilt .....	347
clip .....	350
clock .....	352
clockoff .....	353
clockon .....	354
cngoto .....	355
comb .....	357
compress .....	359

---

---

control .....	361
convle .....	362
convolve .....	363
cos .....	366
cosh .....	368
cosinv .....	370
cps2pch .....	372
cpsmidi .....	376
cpsmidib .....	378
cpsoct .....	380
cpspch .....	382
cpstmid .....	384
cpstun .....	386
cpstuni .....	389
cpsxpch .....	392
cpuprc .....	396
cross2 .....	398
crunch .....	400
ctrl14 .....	402
ctrl21 .....	404
ctrl7 .....	406
ctrlinit .....	408
cuserrnd .....	409
dam .....	411
db .....	414
dbamp .....	416
dbfsamp .....	418
dcblock .....	420
dconv .....	422
delay .....	424
delay1 .....	426
delayk .....	427
delayr .....	429
delayw .....	430
deltap .....	432
deltap3 .....	434
deltapi .....	436
deltapn .....	438
deltapx .....	440
deltapxw .....	442
denorm .....	444
diff .....	445
diskin .....	447
diskin2 .....	450
dispfft .....	453
display .....	455
distort .....	457
distort1 .....	459
divz .....	461
downsamp .....	463
dripwater .....	465
dssiactivate .....	467
dssiaudio .....	468
dssictls .....	469
dssiinit .....	470
dssilist .....	473
dumpk .....	474
dumpk2 .....	476
dumpk3 .....	478
dumpk4 .....	480
dusernd .....	482
else .....	484
elseif .....	485

---

---

endif .....	486
endin .....	487
endop .....	489
envlpx .....	490
envlpxr .....	493
event .....	495
event_i .....	498
exitnow .....	499
exp .....	500
expon .....	502
exprand .....	504
expseg .....	506
expsega .....	508
expsegr .....	510
ficlose .....	512
filelen .....	513
filenchnls .....	515
filepeak .....	517
filesr .....	519
filter2 .....	521
fin .....	523
fini .....	524
fink .....	525
fiopen .....	526
flanger .....	527
flashtxt .....	529
FLbox .....	531
FLbutBank .....	535
FLbutton .....	537
FLcolor .....	541
FLcolor2 .....	542
FLcount .....	543
FLgetsnap .....	546
FLgroup .....	547
FLgroupEnd .....	549
FLgroupEnd .....	550
FLhide .....	551
FLjoy .....	552
FLkeyb .....	555
FLknob .....	556
FLlabel .....	559
FLloadsnap .....	561
flooper .....	562
floor .....	564
FLpack .....	565
FLpackEnd .....	568
FLpack_end .....	569
FLpanel .....	570
FLpanelEnd .....	573
FLpanel_end .....	574
FLprintk .....	575
FLprintk2 .....	576
FLroller .....	577
FLrun .....	580
FLsavesnap .....	581
FLscroll .....	583
FLscrollEnd .....	586
FLscroll_end .....	587
FLsetAlign .....	588
FLsetBox .....	589
FLsetColor .....	591
FLsetColor2 .....	593
FLsetFont .....	594

---

---

FLsetPosition .....	596
FLsetSize .....	597
FLsetsnap .....	598
FLsetText .....	600
FLsetTextColor .....	601
FLsetTextSize .....	602
FLsetTextType .....	603
FLsetVal_i .....	606
FLsetVal .....	607
FLshow .....	608
FLslidBnk .....	609
FLslider .....	612
FLtabs .....	616
FLtabsEnd .....	621
FLtabs_end .....	622
FLtext .....	623
FLupdate .....	626
fluidAllOut .....	627
fluidCCi .....	630
fluidCCk .....	631
fluidControl .....	632
fluidEngine .....	634
fluidLoad .....	637
fluidNote .....	641
fluidOut .....	644
fluidProgramSelect .....	647
FLvalue .....	650
fmb3 .....	652
fmbell .....	654
fmmetal .....	656
fmpercfl .....	659
fmrhode .....	661
fmvoice .....	664
fmwurlie .....	666
fof .....	669
fof2 .....	672
fofilter .....	674
fog .....	676
fold .....	678
follow .....	680
follow2 .....	682
foscil .....	684
foscili .....	686
fout .....	688
fouti .....	692
foutir .....	694
foutk .....	696
fprintks .....	698
fprints .....	700
frac .....	702
freeverb .....	704
ftchnls .....	706
ftconv .....	708
ftfree .....	711
ftgen .....	712
ftgentmp .....	714
ftlen .....	715
ftload .....	717
ftloadk .....	718
ftlptim .....	719
ftmorf .....	721
ftsav .....	723
ftsavk .....	725

---

ftsr .....	726
gain .....	728
gauss .....	729
gbuzz .....	731
getcfig .....	733
gogobel .....	734
goto .....	736
grain .....	738
grain2 .....	740
grain3 .....	744
granule .....	749
guiro .....	752
harmon .....	754
hilbert .....	757
hrtfer .....	761
hsboscil .....	763
i .....	766
ibetarand .....	767
ibexprnd .....	768
icauchy .....	769
ictrl14 .....	770
ictrl21 .....	771
ictrl7 .....	772
iexprand .....	773
if .....	774
igauss .....	778
igoto .....	779
ihold .....	781
ilinrand .....	783
imidic14 .....	784
imidic21 .....	785
imidic7 .....	786
in .....	787
in32 .....	788
inch .....	789
inh .....	790
init .....	791
initc14 .....	792
initc21 .....	793
initc7 .....	794
ino .....	795
inq .....	796
ins .....	797
instimek .....	798
instimes .....	799
instr .....	800
int .....	802
integ .....	804
interp .....	806
invalue .....	809
inx .....	810
inz .....	811
ioff .....	812
ion .....	813
iondur .....	814
iondur2 .....	815
ioutat .....	816
ioutc .....	817
ioutc14 .....	818
ioutpat .....	819
ioutpb .....	820
ioutpc .....	821
ipcauchy .....	822

---

ipoisson .....	823
ipow .....	824
is16b14 .....	825
is32b14 .....	826
islider16 .....	827
islider32 .....	828
islider64 .....	829
islider8 .....	830
itablecopy .....	831
itablegpw .....	832
itablemix .....	833
itablew .....	834
itrirand .....	835
iunirand .....	836
iweibull .....	837
jitter .....	838
jitter2 .....	840
jspline .....	842
k .....	843
kbetarand .....	844
kbexprnd .....	845
kcauchy .....	846
kdump .....	847
kdump2 .....	848
kdump3 .....	849
kdump4 .....	850
kexprand .....	851
kfilter2 .....	852
kgauss .....	853
kgoto .....	854
klinrand .....	856
kon .....	857
koutat .....	858
koutc .....	859
koutc14 .....	860
koutpat .....	861
koutpb .....	862
koutpc .....	863
kpcauchy .....	864
kpoisson .....	865
kpow .....	866
kr .....	867
kread .....	868
kread2 .....	869
kread3 .....	870
kread4 .....	871
ksmps .....	872
ktableseg .....	873
ktrirand .....	874
kunirand .....	875
kweibull .....	876
lfo .....	877
limit .....	879
line .....	880
linen .....	882
linenr .....	883
lineto .....	884
linrand .....	885
linseg .....	887
linsegr .....	889
locsend .....	891
locsig .....	893
log .....	896

---



---

log10 .....	898
logbtwo .....	900
loop .....	902
loopseg .....	904
loopsegg .....	906
lorenz .....	907
lorisread .....	910
lorismorph .....	912
lorisplay .....	913
loscil .....	914
loscil3 .....	917
lowpass2 .....	920
lowres .....	922
lowresx .....	924
lpf18 .....	926
lpfreson .....	928
lphasor .....	929
lpinterp .....	930
lposcil .....	931
lposcil3 .....	932
lpread .....	933
lpreson .....	935
lpshold .....	936
lpsholdp .....	938
lpslot .....	939
mac .....	940
maca .....	941
madsr .....	942
mandel .....	945
mandol .....	946
marimba .....	948
massign .....	950
max .....	952
maxabs .....	953
maxabsaccum .....	954
maxaccum .....	955
maxalloc .....	956
max_k .....	958
mclock .....	959
mdelay .....	960
metro .....	961
midic14 .....	963
midic21 .....	965
midic7 .....	967
midichannelaftertouch .....	969
midichn .....	972
midicontrolchange .....	975
midictrl .....	977
mididefault .....	978
midiin .....	980
midinoteoff .....	981
midinoteoncps .....	984
midinoteonkey .....	987
midinoteonoct .....	990
midinoteonpch .....	993
midion .....	996
midion2 .....	997
midiout .....	998
midipitchbend .....	999
midipolyaftertouch .....	1002
midiprogramchange .....	1004
miditempo .....	1006
min .....	1007

---

---

minabs .....	1008
minabsaccum .....	1009
minaccum .....	1010
mirror .....	1011
MixerSetLevel .....	1012
MixerGetLevel .....	1014
MixerSend .....	1015
MixerReceive .....	1017
MixerClear .....	1019
monitor .....	1020
moog .....	1021
moogladder .....	1023
moogvcf .....	1025
moscil .....	1027
mpulse .....	1028
mrtmsg .....	1030
multitap .....	1031
mute .....	1032
mxadsr .....	1034
nchnls .....	1036
nestedap .....	1037
nlfilt .....	1040
noise .....	1042
noteoff .....	1044
noteon .....	1045
noteondur .....	1046
noteondur2 .....	1047
notnum .....	1048
nreverb .....	1050
nrpn .....	1053
nsamp .....	1054
nstrnum .....	1056
ntrpol .....	1057
octave .....	1058
octcps .....	1060
octmidi .....	1062
octmidib .....	1064
octpch .....	1066
opcode .....	1068
OSCsend .....	1073
OSCinit .....	1075
OSClisten .....	1076
oscbnk .....	1078
oscil .....	1083
oscil1 .....	1085
oscilli .....	1086
oscil3 .....	1087
oscili .....	1089
oscilikt .....	1091
osciliktp .....	1093
oscilikts .....	1095
osciln .....	1097
oscils .....	1098
oscilx .....	1100
out .....	1101
out32 .....	1102
outc .....	1103
outch .....	1104
outh .....	1105
outiat .....	1106
outic .....	1107
outic14 .....	1108
outipat .....	1109

---

outipb .....	1110
outipc .....	1111
outkat .....	1112
outkc .....	1113
outkc14 .....	1114
outkpat .....	1115
outkpb .....	1116
outkpc .....	1117
outo .....	1118
outq .....	1119
outq1 .....	1120
outq2 .....	1121
outq3 .....	1122
outq4 .....	1123
outs .....	1124
outs1 .....	1125
outs2 .....	1126
outvalue .....	1127
outx .....	1128
outz .....	1129
p .....	1130
pan .....	1132
pareq .....	1134
partials .....	1137
pcauchy .....	1139
pchbend .....	1141
pchmidi .....	1143
pchmidib .....	1145
pchoct .....	1147
pconvolve .....	1149
peak .....	1152
peakk .....	1154
pgmassign .....	1155
phaser1 .....	1159
phaser2 .....	1162
phasor .....	1165
phasorbnk .....	1167
pinkish .....	1169
pitch .....	1172
pitchamdf .....	1175
planet .....	1178
pluck .....	1180
poisson .....	1182
polyaft .....	1184
port .....	1186
portk .....	1187
poscil .....	1188
poscil3 .....	1190
pow .....	1192
powoftwo .....	1194
prealloc .....	1196
print .....	1198
printf .....	1200
printk .....	1201
printk2 .....	1203
printks .....	1205
prints .....	1208
product .....	1210
pset .....	1211
puts .....	1212
pvadd .....	1213
pvbufread .....	1216
pvcross .....	1218

---

pvinterp .....	1220
pvoc .....	1222
pvread .....	1224
pvsadsyn .....	1226
pvsanal .....	1228
pvsarp .....	1231
pvsccross .....	1233
pvscent .....	1234
pvsdemix .....	1235
pvsfread .....	1237
pvsfreeze .....	1238
pvsftr .....	1239
pvsftw .....	1241
pvsifd .....	1243
pvsinfo .....	1245
pvsinit .....	1246
pvsmaska .....	1247
pvsynth .....	1249
pvscale .....	1251
pvshift .....	1253
pvmix .....	1255
pvsMOOTH .....	1256
pvsfilter .....	1258
pvsblur .....	1260
pvstencil .....	1261
pvsvoc .....	1263
pyassign Opcodes .....	1265
pycall Opcodes .....	1266
pyeval Opcodes .....	1269
pyexec Opcodes .....	1270
pyinit Opcodes .....	1273
pyrun Opcodes .....	1274
rand .....	1276
randh .....	1278
randi .....	1280
random .....	1282
randomh .....	1284
randomi .....	1286
rbjeq .....	1288
readclock .....	1291
readk .....	1293
readk2 .....	1295
readk3 .....	1297
readk4 .....	1299
reinit .....	1301
release .....	1303
repluck .....	1305
reson .....	1307
resonk .....	1309
resonr .....	1310
resonx .....	1313
resonxk .....	1314
resony .....	1315
resonz .....	1317
resyn .....	1319
reverb .....	1321
reverb2 .....	1323
reverb3 .....	1324
rezzy .....	1326
rigoto .....	1328
rireturn .....	1329
rms .....	1331
rnd .....	1332

---

rnd31 .....	1334
rspline .....	1339
rtclock .....	1340
s16b14 .....	1342
s32b14 .....	1344
samphold .....	1346
sandpaper .....	1347
scanhammer .....	1349
scans .....	1350
scantable .....	1352
scanu .....	1354
schedkwhen .....	1356
schedkwhennamed .....	1358
schedule .....	1359
schedwhen .....	1361
seed .....	1363
sekere .....	1364
semitone .....	1366
sense .....	1368
sensekey .....	1369
seqtime .....	1371
seqtime2 .....	1373
setctrl .....	1375
setksmps .....	1377
sfelist .....	1379
sfinstr .....	1380
sfinstr3 .....	1382
sfinstr3m .....	1384
sfinstrm .....	1386
sfload .....	1388
sfpassign .....	1389
sfplay .....	1390
sfplay3 .....	1392
sfplay3m .....	1394
sfplaym .....	1396
sfplist .....	1398
sfpreset .....	1399
shaker .....	1400
sin .....	1402
sinh .....	1404
sininv .....	1406
sinsyn .....	1408
sleighbells .....	1410
slider16 .....	1412
slider16f .....	1414
slider32 .....	1416
slider32f .....	1418
slider64 .....	1420
slider64f .....	1422
slider8 .....	1424
slider8f .....	1426
sndloop .....	1428
sndwarp .....	1430
sndwarpst .....	1434
soundin .....	1437
soundout .....	1440
soundouts .....	1441
space .....	1442
spat3d .....	1446
spat3di .....	1454
spat3dt .....	1458
spdist .....	1462
specaddm .....	1466

specdiff .....	1467
specdisp .....	1468
specfilt .....	1469
spechist .....	1470
specptrk .....	1471
specscal .....	1473
specsum .....	1474
spectrum .....	1475
splitrig .....	1477
spsend .....	1479
sprintf .....	1482
sqrt .....	1483
sr .....	1485
statevar .....	1486
stix .....	1488
strchar .....	1490
strchark .....	1491
strcpy .....	1492
strcpyk .....	1493
strcat .....	1494
strcatk .....	1495
strcmp .....	1496
strcmpk .....	1497
streson .....	1498
strget .....	1500
strindex .....	1501
strindexk .....	1502
strlen .....	1503
strlenk .....	1504
strlower .....	1505
strlowerk .....	1506
strrindex .....	1507
strrindexk .....	1508
strset .....	1509
strsub .....	1510
strsubk .....	1511
strtod .....	1512
strtodk .....	1513
strtol .....	1514
strtolk .....	1515
strupper .....	1516
strupperk .....	1517
subinstr .....	1518
subinstrinit .....	1521
sum .....	1522
svfilter .....	1523
syncgrain .....	1525
timedseq .....	1527
tb .....	1529
tab .....	1532
tabrec .....	1533
table .....	1534
table3 .....	1536
tablecopy .....	1537
tablegpw .....	1538
tablei .....	1539
tableicopy .....	1540
tableigpw .....	1541
tableikt .....	1542
tableimix .....	1544
tableiw .....	1546
tablekt .....	1548
tablemix .....	1550

tableng .....	1552
tablera .....	1554
tableseg .....	1557
tablew .....	1558
tablewa .....	1561
tablewkt .....	1564
tablexkt .....	1566
tablexseg .....	1568
tambourine .....	1569
tan .....	1571
tanh .....	1573
taninv .....	1575
taninv2 .....	1577
tbvcf .....	1579
tempest .....	1582
tempo .....	1584
tempoval .....	1586
tigoto .....	1588
timeinstk .....	1589
timeinsts .....	1591
timek .....	1593
times .....	1595
timout .....	1597
tival .....	1598
tlineto .....	1599
tone .....	1600
tonek .....	1601
tonex .....	1602
tradsyn .....	1603
transeg .....	1605
trcross .....	1606
trfilter .....	1608
trhighest .....	1610
trigger .....	1611
trigseq .....	1613
trirand .....	1615
trlowest .....	1617
trmix .....	1618
trscale .....	1619
trshift .....	1620
trsplt .....	1621
turnoff .....	1623
turnoff2 .....	1625
turnon .....	1626
unirand .....	1627
upsamp .....	1629
urd .....	1630
vadd .....	1631
vaddv .....	1632
valpass .....	1633
vbap16 .....	1634
vbap16move .....	1636
vbap4 .....	1638
vbap4move .....	1640
vbap8 .....	1642
vbap8move .....	1644
vbaplsinit .....	1646
vbapz .....	1648
vbapzmove .....	1650
vcella .....	1652
vco .....	1653
vco2 .....	1656
vco2ft .....	1659

---

vco2ift .....	1661
vco2init .....	1662
vcomb .....	1664
vcopy .....	1665
vcopy_i .....	1667
vdelay .....	1669
vdelay3 .....	1671
vdelayx .....	1673
vdelayxq .....	1675
vdelayxs .....	1677
vdelayxw .....	1679
vdelayxwq .....	1681
vdelayxws .....	1683
vdivv .....	1685
vdelayk .....	1686
vecdelay .....	1687
veloc .....	1688
vexp .....	1690
vexpseg .....	1691
vexpv .....	1693
vibes .....	1694
vibr .....	1696
vibrato .....	1698
vincr .....	1700
vlimit .....	1701
vlinseg .....	1702
vlowres .....	1704
vmap .....	1706
vmirror .....	1707
vmult .....	1708
vmultv .....	1710
voice .....	1711
vport .....	1713
vpow .....	1714
vpowv .....	1715
vpvoc .....	1716
vrandh .....	1718
vrandi .....	1719
vstaudio, vstaudiog .....	1720
vstbankload .....	1721
vstedit .....	1722
vstinit .....	1723
vstinfo .....	1724
vstmidiout .....	1725
vstnote .....	1727
vstparamset, vstparamget .....	1729
vstprogset .....	1731
vsubv .....	1732
vtablei .....	1733
vtablek .....	1735
vtablea .....	1737
vtablewi .....	1738
vtablewk .....	1739
vtablewa .....	1741
vtabi .....	1743
vtabk .....	1744
vtaba .....	1745
vtabwi .....	1746
vtabwk .....	1747
vtabwa .....	1748
vwrap .....	1749
waveset .....	1750
weibull .....	1752

---



wgbow .....	1754
wgbowedbar .....	1756
wgbrass .....	1758
wgclar .....	1760
wgflute .....	1762
wgpluck .....	1764
wgpluck2 .....	1767
wguide1 .....	1769
wguide2 .....	1771
wrap .....	1773
wterrain .....	1774
xadsr .....	1776
xin .....	1778
xout .....	1780
xscanmap .....	1782
xscansmap .....	1783
xscans .....	1784
xscanu .....	1786
xtratim .....	1788
xyin .....	1790
zaci .....	1792
zakinit .....	1794
zamod .....	1796
zar .....	1798
zarg .....	1800
zaw .....	1802
zawm .....	1804
zfilter2 .....	1806
zir .....	1808
ziw .....	1810
ziwm .....	1812
zkci .....	1814
zkmod .....	1816
zkr .....	1818
zkw .....	1820
zkwm .....	1822
Score Statements and GEN Routines .....	1825
Score Statements .....	1825
GEN Routines .....	1843
The Utility Programs .....	1908
Directories .....	1908
Soundfile Formats .....	1908
Analysis File Generation (HETRO,LPANAL, PVANAL, CVANAL, ATSA) .....	1908
File Queries (SNDINFO) .....	1920
File Conversion (DNOISE, PVLOOK, SDIF2AD, SRCONV) .....	1922
Other Csound Utilities (MAKECSD, CS) .....	1934
Cscore .....	1945
Events, Lists, and Operations .....	1945
Writing a Main Program .....	1946
More Advanced Examples .....	1952
Compiling a Cscore Program .....	1953
Extending Csound .....	1955
Adding Unit Generators .....	1955
Creating a Builtin Unit Generator .....	1955
Adding a Plugin Unit Generator .....	1959
OENTRY Reference .....	1959

---

# Orchestra Opcodes and Operators

# !=

!= -- Determines if one value is not equal to another.

!=

## Description

Determines if one value is not equal to another.

## Syntax

(a != b ? v1 : v2)

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

## Performance

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.

## Examples

Here is an example of the != opcode. It uses the files *notequal.orc* [examples/notequal.orc] and *notequal.sco* [examples/notequal.sco].

### Example 1. Example of the != opcode.

```
/* notequal.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
  ; Get the 4th p-field from the score.
  k1 = p4

  ; Is it not equal to 3? (1 = true, 0 = false)
  k2 = (p4 != 3 ? 1 : 0)

  ; Print the values of k1 and k2.
  printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin
/* notequal.orc */
```

```
/* notequal.sco */  
; Call Instrument #1 with a p4 = 2.  
i 1 0 0.5 2  
; Call Instrument #1 with a p4 = 3.  
i 1 1 0.5 3  
; Call Instrument #1 with a p4 = 4.  
i 1 2 0.5 4  
e  
/* notequal.sco */
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 1.000000  
k1 = 3.000000, k2 = 0.000000  
k1 = 4.000000, k2 = 1.000000
```

## See Also

`==, >=, >, <=, <`

## Credits

Example written by Kevin Conder.

# #define

`#define` -- Defines a macro.

`#define`

## 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.

## Syntax

```
#define NAME # replacement text #
```

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

## 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

Here is a simple example of the defining a macro. It uses the files *define.orc* [examples/define.orc] and *define.sco* [examples/define.sco].

### Example 2. Simple example of the define macro.

```
/* define.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410
```

```
ksmps = 10
nchnls = 1

; Define the macros.
#define VOLUME #5000#
#define FREQ #440#
#define TABLE #1#

; Instrument #1
instr 1
; Use the macros.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 oscil $VOLUME, $FREQ, $TABLE

; Send it to the output.
out a1
endin
/* define.orc */

/* define.sco */
; Define Table #1 with an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* define.sco */
```

Its output should include lines like this:

```
Macro definition for VOLUME
Macro definition for CPS
Macro definition for TABLE
```

Here is an example of the defining a macro with arguments. It uses the files *define\_args.orc* [examples/define\_args.orc] and *define\_args.sco* [examples/define\_args.sco].

### **Example 3. Example of the define macro with arguments.**

```
/* define_args.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Define the oscillator macro.
#define OSCMACRO(VOLUME'FREQ'TABLE) #oscil $VOLUME, $FREQ, $TABLE#

; Instrument #1
instr 1
; Use the oscillator macro.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 $OSCMACRO(5000'440'1)
```

```
    ; Send it to the output.  
    out a1  
endin  
/* define_args.orc */  
  
/* define_args.sco */  
; Define Table #1 with an ordinary sine wave.  
f 1 0 32768 10 1  
  
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* define_args.sco */
```

Its output should include lines like this:

Macro definition for OSCMACRO

## See Also

*\$NAME, #undef*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

Examples written by Kevin Conder.

New in Csound version 3.48

# #include

#include -- Includes an external file for processing.

#include

## Description

Includes an external file for processing.

## Syntax

**#include** "filename"

## Performance

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. *Note:* Csound versions prior to 4.19 had 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 defined 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.

## Examples

Here is an example of the include opcode. It uses the files *include.orc* [examples/include.orc], *include.sco* [examples/include.sco], and *table1.inc* [examples/table1.inc].



**Example 4. Example of the include opcode.**

```
/* include.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 440
  ifn = 1

  al oscil kamp, kcps, ifn
  out al
endin
/* include.orc */
```

```
/* table1.inc */
; Table #1, a sine wave.
f 1 0 16384 10 1
/* table1.inc */
```

```
/* include.sco */
; Include the file for Table #1.
#include "table1.inc"

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* include.sco */
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

Example written by Kevin Conder.

New in Csound version 3.48

# #undef

#undef -- Un-defines a macro.

#undef

## 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*.

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

## Syntax

**#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.

## See Also

*#define, \$NAME*

## Credits

Author: John fitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

New in Csound version 3.48

## #ifdef

`#ifdef` -- Conditional reading of code.

`#ifdef`

## Description

If a macro is defined then *#ifdef* can incorporate text into an orchestra upto the next *#end*. This is similar to, but independent of, the *macro system in the score language*.

## Syntax

`#ifdef` NAME

....

`#end`

## Performance

Note that the *#ifdef* construct cannot be nested, unlike the C preprocessor language.

## Examples

Here is a simple example of the conditional.

### Example 5. Simple example of the ifdef form.

```
#define debug
  instr 1
#ifdef debug
  print "calling oscil"
#end
  a1  oscil 32000,440,1
  out a1
endin
```

## See Also

*\$NAME*, *#define*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 2005

New in Csound5 (and 4.23f13)

# \$NAME

\$NAME -- Calls a defined macro.

\$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*.

*\$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.

## Syntax

**\$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

Here is an example of the calling a macro. It uses the files *define.orc* [examples/define.orc] and *define.sco* [examples/define.sco].

### Example 6. An example of the calling a macro.

```
/* define.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Define the macros.
#define VOLUME #5000#
#define FREQ #440#
#define TABLE #1#
```

```
; Instrument #1
instr 1
; Use the macros.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 oscil $VOLUME, $FREQ, $TABLE

; Send it to the output.
out a1
endin
/* define.orc */

/* define.sco */
; Define Table #1 with an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* define.sco */
```

Its output should include lines like this:

```
Macro definition for VOLUME
Macro definition for CPS
Macro definition for TABLE
```

Here is an example of the calling a macro with arguments. It uses the files *define\_args.orc* [examples/define\_args.orc] and *define\_args.sco* [examples/define\_args.sco].

### **Example 7. An example of the calling a macro with arguments.**

```
/* define_args.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Define the oscillator macro.
#define OSCMACRO(VOLUME'FREQ'TABLE) #oscil $VOLUME, $FREQ, $TABLE#

; Instrument #1
instr 1
; Use the oscillator macro.
; This will be expanded to "a1 oscil 5000, 440, 1".
a1 $OSCMACRO(5000'440'1)

; Send it to the output.
out a1
endin
/* define_args.orc */
```

```
/* define_args.sco */  
; Define Table #1 with an ordinary sine wave.  
f 1 0 32768 10 1  
  
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* define_args.sco */
```

Its output should include a line like this:

Macro definition for OSCMACRO

## See Also

*#define, #undef*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April, 1998

Example written by Kevin Conder.

New in Csound version 3.48

# %

% -- Modulus operator.

%

## Description

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  $\%$  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.

## Syntax



`a % b` (no rate restriction)

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

## Examples

Here is an example of the % operator. It uses the files *modulus.orc* [examples/modulus.orc] and *modulus.sco* [examples/modulus.sco].

### Example 8. Example of the % operator.

```
/* modulus.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = 5 % 3
  print i1
endin
/* modulus.orc */

/* modulus.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* modulus.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 2.000
```

## See Also

`-`, `+`, `&&`, `//`, `*`, `/`, `^`

## Credits

Example written by Kevin Conder.

## &&

&& -- Logical AND operator.

&&

## Description

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.

## Syntax

$a \&\& b$  (logical AND; not audio-rate)

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

## See Also

`-`, `+`, `//`, `*`, `/`, `^`, `%`

## >

> -- Determines if one value is greater than another.

>

## Description

Determines if one value is greater than another.

## Syntax

```
(a > b ? v1 : v2)
```

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

## Performance

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 "*a* = *b*".)

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.

## Examples

Here is an example of the > opcode. It uses the files *greaterthan.orc* [examples/greaterthan.orc] and *greaterthan.sco* [examples/greaterthan.sco].

### Example 9. Example of the > opcode.

```
/* greaterthan.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Get the 4th p-field from the score.
k1 = p4

; Is it greater than 3? (1 = true, 0 = false)
k2 = (p4 > 3 ? 1 : 0)

; Print the values of k1 and k2.
printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin
/* greaterthan.orc */
```

```
/* greaterthan.sco */  
; Call Instrument #1 with a p4 = 2.  
i 1 0 0.5 2  
; Call Instrument #1 with a p4 = 3.  
i 1 1 0.5 3  
; Call Instrument #1 with a p4 = 4.  
i 1 2 0.5 4  
e  
/* greaterthan.sco */
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 0.000000  
k1 = 3.000000, k2 = 0.000000  
k1 = 4.000000, k2 = 1.000000
```

## See Also

`==, >=, <=, <, !=`

## Credits

Example written by Kevin Conder.

## >=

>= -- Determines if one value is greater than or equal to another.

>=

## Description

Determines if one value is greater than or equal to another.

## Syntax

(a >= b ? v1 : v2)

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

## Performance

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.

## Examples

Here is an example of the >= opcode. It uses the files *greaterequal.orc* [examples/greaterequal.orc] and *greaterequal.sco* [examples/greaterequal.sco].

### Example 10. Example of the >= opcode.

```
/* greaterequal.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
  ; Get the 4th p-field from the score.
  k1 = p4

  ; Is it greater than or equal to 3? (1 = true, 0 = false)
  k2 = (p4 >= 3 ? 1 : 0)

  ; Print the values of k1 and k2.
  printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin
/* greaterequal.orc */
```

```
/* greaterequal.sco */  
; Call Instrument #1 with a p4 = 2.  
i 1 0 0.5 2  
; Call Instrument #1 with a p4 = 3.  
i 1 1 0.5 3  
; Call Instrument #1 with a p4 = 4.  
i 1 2 0.5 4  
e  
/* greaterequal.sco */
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 0.000000  
k1 = 3.000000, k2 = 1.000000  
k1 = 4.000000, k2 = 1.000000
```

## See Also

`==, >, <=, <, !=`

## Credits

Example written by Kevin Conder.

## &lt;

< -- Determines if one value is less than another.

&lt;

## Description

Determines if one value is less than another.

## Syntax

```
(a < b ? v1 : v2)
```

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

## Performance

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.

## Examples

Here is an example of the < opcode. It uses the files *lessthan.orc* [examples/lessthan.orc] and *lessthan.sco* [examples/lessthan.sco].

### Example 11. Example of the < opcode.

```
/* lessthan.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Get the 4th p-field from the score.
k1 = p4

; Is it less than 3? (1 = true, 0 = false)
k2 = (p4 < 3 ? 1 : 0)

; Print the values of k1 and k2.
printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin
/* lessthan.orc */
```



```
/* lessthan.sco */  
; Call Instrument #1 with a p4 = 2.  
i 1 0 0.5 2  
; Call Instrument #1 with a p4 = 3.  
i 1 1 0.5 3  
; Call Instrument #1 with a p4 = 4.  
i 1 2 0.5 4  
e  
/* lessthan.sco */
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 1.000000  
k1 = 3.000000, k2 = 0.000000  
k1 = 4.000000, k2 = 0.000000
```

## See Also

`==, >=, >, <=, !=`

## Credits

Example written by Kevin Conder.

## <=

<= -- Determines if one value is less than or equal to another.

<=

## Description

Determines if one value is less than or equal to another.

## Syntax

(a <= b ? v1 : v2)

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

## Performance

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.

## Examples

Here is an example of the <= opcode. It uses the files *lessequal.orc* [examples/lessequal.orc] and *lessequal.sco* [examples/lessequal.sco].

### Example 12. Example of the <= opcode.

```
/* lessequal.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Get the 4th p-field from the score.
k1 = p4

; Is it less than or equal to 3? (1 = true, 0 = false)
k2 = (p4 <= 3 ? 1 : 0)

; Print the values of k1 and k2.
printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin
/* lessequal.orc */
```

```
/* lessequal.sco */  
; Call Instrument #1 with a p4 = 2.  
i 1 0 0.5 2  
; Call Instrument #1 with a p4 = 3.  
i 1 1 0.5 3  
; Call Instrument #1 with a p4 = 4.  
i 1 2 0.5 4  
e  
/* lessequal.sco */
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 1.000000  
k1 = 3.000000, k2 = 1.000000  
k1 = 4.000000, k2 = 0.000000
```

## See Also

`==, >=, >, <, !=`

## Credits

Example written by Kevin Conder.

**\***

\* -- Multiplication operator.

\*

## Description

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 ||:

$a \&\& b - c \parallel d$

is taken as

$(a \&\& (b - c)) \parallel d$

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.

## Syntax

`a * b` (no rate restriction)

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

## Examples

Here is an example of the `*` operator. It uses the files *multiplies.orc* [examples/multiplies.orc] and *multiplies.sco* [examples/multiplies.sco].

### Example 13. Example of the `*` operator.

```
/* multiplies.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il = 24 * 8
  print il
endin
/* multiplies.orc */

/* multiplies.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* multiplies.sco */
```

Its output should include a line like this:

```
instr 1:  il = 192.000
```

## See Also

`-`, `+`, `&&`, `//`, `/`, `^`, `%`

## Credits

Example written by Kevin Conder.

**+**

+ -- Addition operator

+

## Description

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.

## Syntax

`+ a (no rate restriction)`

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

## Examples

Here is an example of the + operator. It uses the files *adds.orc* [examples/adds.orc] and *adds.sco* [examples/adds.sco].

### Example 14. Example of the + operator.

```
/* adds.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il = 24 + 8
  print il
endin
/* adds.orc */

/* adds.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* adds.sco */
```

Its output should include lines like:

```
instr 1:  il = 32.000
```

## See Also

-, &&, //, \*, /, ^, %

## Credits

Example written by Kevin Conder.

■

- -- Subtraction operator.

-

## Description

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.

## Syntax

`# a (no rate restriction)`

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



## Examples

Here is an example of the - operator. It uses the files *subtracts.orc* [examples/subtracts.orc] and *subtracts.sco* [examples/subtracts.sco].

### Example 15. Example of the - operator.

```
/* subtracts.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il = 24 - 8
  print il
endin
/* subtracts.orc */

/* subtracts.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* subtracts.sco */
```

Its output should include lines like this:

```
instr 1:  il = 16.000
```

## See Also

+, &&, //, \*, /, ^, %

## Credits

Example written by Kevin Conder.

**/**

/ -- Division operator.

/

## Description

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.

## Syntax

$a / b$  (no rate restriction)

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

## Examples

Here is an example of the `/` operator. It uses the files *divides.orc* [examples/divides.orc] and *divides.sco* [examples/divides.sco].

### Example 16. Example of the `/` operator.

```
/* divides.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il = 24 / 8
  print il
endin
/* divides.orc */

/* divides.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* divides.sco */
```

Its output should include lines like this:

```
instr 1:  il = 3.000
```

## See Also

`-`, `+`, `&&`, `//`, `*`, `^`, `%`

## Credits

Example written by Kevin Conder.

**=**

= -- Performs a simple assignment.

=

## Syntax

ares = xarg

ires = iarg

kres = karg

## Description

Performs a simple assignment.

## Initialization

= (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.

## Examples

Here is an example of the assign opcode. It uses the files *assign.orc* [examples/assign.orc] and *assign.sco* [examples/assign.sco].

### Example 17. Example of the assign opcode.

```
/* assign.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Assign a value to the variable i1.
  i1 = 1234

  ; Print the value of the i1 variable.
  print i1
endin
/* assign.orc */
```

```
/* assign.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* assign.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 1234.000
```

## See Also

*divz, init, tival*

## Credits

Example written by Kevin Conder.

**==**

== -- Compares two values for equality.

==

## Description

Compares two values for equality.

## Syntax

```
(a == b ? v1 : v2)
```

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

## Performance

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 "*a* == *b*".)

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.

## Examples

Here is an example of the == opcode. It uses the files *equal.orc* [examples/equal.orc] and *equal.sco* [examples/equal.sco].

### Example 18. Example of the == opcode.

```
/* equal.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
  ; Get the 4th p-field from the score.
  k1 = p4

  ; Is it equal to 3? (1 = true, 0 = false)
  k2 = (p4 == 3 ? 1 : 0)

  ; Print the values of k1 and k2.
  printks "k1 = %f, k2 = %f\\n", 1, k1, k2
endin
/* equal.orc */
```

```
/* equal.sco */  
; Call Instrument #1 with a p4 = 2.  
i 1 0 0.5 2  
; Call Instrument #1 with a p4 = 3.  
i 1 1 0.5 3  
; Call Instrument #1 with a p4 = 4.  
i 1 2 0.5 4  
e  
/* equal.sco */
```

Its output should include lines like this:

```
k1 = 2.000000, k2 = 0.000000  
k1 = 3.000000, k2 = 1.000000  
k1 = 4.000000, k2 = 0.000000
```

## See Also

`>=`, `>`, `<=`, `<`, `!=`

## Credits

Example written by Kevin Conder.

## ^

^ -- “Power of” operator.

^

## Description

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 *pow*. (New in Csound version 3.493.)

## Syntax

$a \wedge b$  ( $b$  not audio-rate)



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

## Examples

Here is an example of the `^` operator. It uses the files *raises.orc* [examples/raises.orc] and *raises.sco* [examples/raises.sco].

### Example 19. Example of the `^` operator.

```
/* raises.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = 2 ^ 12
  print i1
endin
/* raises.orc */

/* raises.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* raises.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 4096.000
```

## See Also

`-`, `+`, `&&`, `//`, `*`, `/`, `%`

## Credits

Example written by Kevin Conder.

# round

`round` -- Returns the integer value nearest to  $x$  ; if the fractional part of  $x$  is exactly 0.5, the direction of rounding is undefined.

`round`

## Description

The integer value nearest to  $x$  ; if the fractional part of  $x$  is exactly 0.5, the direction of rounding is undefined.

## Syntax

**round**( $x$ ) (*init-*, *control-*, or *audio-rate* arg allowed)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Author: Istvan Varga  
New in Csound 5  
2005

# ||

|| -- Logical OR operator.

||

## Description

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.

## Syntax

$a || b$  (logical OR; not audio-rate)

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

## See Also

`-`, `+`, `&&`, `*`, `/`, `^`, `%`

# 0dbfs

0dbfs -- Sets the value of 0 decibels using full scale amplitude.

0dbfs

## Description

Sets the value of 0 decibels using full scale amplitude.

## Syntax

**0dbfs** = iarg

## Initialization

*iarg* -- the value of 0 decibels using full scale amplitude.

## Performance

The default is 32767, so all existing orcs *should* work.

These calls should all work:

```
ipeak = 0dbfs
```

```
asig oscil 0dbfs,freq,1  
out asig * 0.3 * 0dbfs
```

and so on.

As for documentation: the usage should be obvious - the main thing is for people to start to code 0dbfs-relatively (and use the *ampdb()* opcodes a lot more!), rather than use explicit sample values.

Floats written to a file, when *0dbfs* = 1, will in effect go through no range translation at all. So the numbers in the file are exactly what the orc says they are.



### BIG NB

All the main sample formats are supported, but I haven't got around to dealing with the char formats. Probably it's straight-forward...

I have tried to cover the main utils - adsyn,lpanal etc. But there are bound to be things missing, sorry.

Some of the parsing code is a bit grungy because I have a variable with a leading digit!

## Examples

Here is an example of the 0dbfs opcode. It uses the files *0dbfs.orc* [examples/0dbfs.orc] and *0dbfs.sco* [examples/0dbfs.sco].

### Example 20. Example of the 0dbfs opcode.

```
/* 0dbfs.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Set the 0dbfs to the 16-bit maximum.
0dbfs = 32767

; Instrument #1.
instr 1
  ; Linearly increase the amplitude value "kamp" from
  ; 0 to 1 over the duration defined by p3.
  kamp line 0, p3, 1

  ; Generate a basic tone using our amplitude value.
  a1 oscil kamp, 440, 1

  ; Multiply the basic tone (with its amplitude between
  ; 0 and 1) by the full-scale 0dbfs value.
  out a1 * 0dbfs
endin
/* 0dbfs.orc */

/* 0dbfs.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
e
/* 0dbfs.sco */
```

## Credits

Author: Richard Dobson  
May 2002

Example written by Kevin Conder.

New in version 4.20

# &

& -- Bitwise AND operator.

&

## Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

## Syntax

`a & b` (bitwise AND)

where the arguments *a* and *b* may be further expressions. They are converted to the nearest integer to machine precision and then the operation is performed.

## See Also

`/`, `#`, `¬`

|

| -- Bitwise OR operator.

|

## Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

## Syntax

`a | b` (bitwise OR)

where the arguments *a* and *b* may be further expressions. They are converted to the nearest integer to machine precision and then the operation is performed.

## See Also

`&`, `#`, `⊖`



¬

¬ -- Bitwise NOT operator.

¬

## Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

## Syntax

```
~ a  (bitwise NOT)
```

where the argument *a* may be a further expression. It is converted to the nearest integer to machine precision and then the operation is performed.

## See Also

&, / #

# #

# -- Bitwise NON EQUIVALENCE operator.

#

## Description

The bitwise operators perform operations of bitwise AND, bitwise OR, bitwise NOT and bitwise non-equivalence.

The priority of these operators is less binding than the arithmetic ones, but more binding than the comparisons.

Parentheses may be used as above to force particular groupings.

## Syntax

`a # b (bitwise NON EQUIVALENCE)`

where the arguments *a* and *b* may be further expressions. They are converted to the nearest integer to machine precision and then the operation is performed.

## See Also

`&`, `/`, `¬`

## a

a -- Converts a k-rate parameter to an a-rate value with interpolation.

a

## Description

Converts a k-rate parameter to an a-rate value with interpolation.

## Syntax

**a**(x) (control-rate args only)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the a opcode. It uses the files *a.orc* [examples/a.orc] and *a.sco* [examples/a.sco].

### Example 21. Example of the a opcode.

```
/* a.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Create a sine wave at k-rate.
  kwave oscil 20000, 440, 1

  ; Convert the k-rate sine wave to the audio-rate.
  awave = a(kwave)

  ; Output the audio-rate version of sine wave.
  out awave
endin
/* a.orc */

/* a.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* a.sco */
```

## See Also

*i, k*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in version 4.21

# abetarand

abetarand -- Deprecated.

abetarand

## Description

Deprecated as of version 3.49. Use the *betarand* opcode instead.

## abexprnd

abexprnd -- Deprecated.

abexprnd

## Description

Deprecated as of version 3.49. Use the *bexprnd* opcode instead.

# abs

abs -- Returns an absolute value.

abs

## Description

Returns the absolute value of  $x$ .

## Syntax

**abs**( $x$ ) (no rate restriction)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the abs opcode. It uses the files *abs.orc* [examples/abs.orc] and *abs.sco* [examples/abs.sco].

### Example 22. Example of the abs opcode.

```
/* abs.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = -6
  i2 = abs(i1)

  print i2
endin
/* abs.orc */

/* abs.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* abs.sco */
```

Its output should include lines like:

```
instr 1:  i2 = 6.000
```

## See Also

*exp, frac, int, log, log10, i, sqrt*

## Credits

Example written by Kevin Conder.



## acauchy

acauchy -- Deprecated.

acauchy

## Description

Deprecated as of version 3.49. Use the *cauchy* opcode instead.

## active

`active` -- Returns the number of active instances of an instrument.

`active`

## Description

Returns the number of active instances of an instrument.

## Syntax

`ir active insnum`

`kres active kinsnum`

## Initialization

*insnum* -- number of the instrument to be reported

## Performance

*kinsnum* -- number of the instrument to be reported

*active* returns the number of active instances of instrument number *insnum*/*kinsnum*. As of Csound4.17 the output is updated at k-rate (if input arg is k-rate), to allow running count of instr instances.

## Examples

Here is a simple example of the `active` opcode. It uses the files *active.orc* [examples/active.orc] and *active.sco* [examples/active.sco].

### Example 23. Simple example of the active opcode.

```
/* active.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a noisy waveform.
instr 1
  ; Generate a really noisy waveform.
  anoisyrand 44100
  ; Turn down its amplitude.
  aoutput gain anoisyrand, 2500
  ; Send it to the output.
  out aoutput
endin

; Instrument #2 - counts active instruments.
instr 2
  ; Count the active instances of Instrument #1.
```

```
    icount active 1
    ; Print the number of active instances.
    print icount
  endin
/* active.orc */

/* active.sco */
; Start the first instance of Instrument #1 at 0:00 seconds.
i 1 0.0 3.0

; Start the second instance of Instrument #1 at 0:015 seconds.
i 1 1.5 1.5

; Play Instrument #2 at 0:01 seconds, when we have only
; one active instance of Instrument #1.
i 2 1.0 0.1

; Play Instrument #2 at 0:02 seconds, when we have
; two active instances of Instrument #1.
i 2 2.0 0.1
e
/* active.sco */
```

Its output should include lines like this:

```
instr 2:  icount = 1.000
instr 2:  icount = 2.000
```

Here is a more advanced example of the active opcode. It displays the results of the active opcode at k-rate instead of i-rate. It uses the files *active\_k.orc* [examples/active\_k.orc] and *active\_k.sco* [examples/active\_k.sco].

### Example 24. Example of the active opcode at k-rate.

```
/* active_k.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a noisy waveform.
instr 1
  ; Generate a really noisy waveform.
  anoisys rand 44100
  ; Turn down its amplitude.
  aoutput gain anoisys, 2500
  ; Send it to the output.
  out aoutput
endin

; Instrument #2 - counts active instruments at k-rate.
instr 2
  ; Count the active instances of Instrument #1.
  kcount active 1
```

```
    ; Print the number of active instances.
    printk2 kcount
  endin
/* active_k.orc */

/* active_k.sco */
; Start the first instance of Instrument #1 at 0:00 seconds.
i 1 0.0 3.0

; Start the second instance of Instrument #1 at 0:015 seconds.
i 1 1.5 1.5

; Play Instrument #2 at 0:01 seconds, when we have only
; one active instance of Instrument #1.
i 2 1.0 0.1

; Play Instrument #2 at 0:02 seconds, when we have
; two active instances of Instrument #1.
i 2 2.0 0.1
e
/* active_k.sco */
```

Its output should include lines like:

```
i2      1.00000
i2      2.00000
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

Examples written by Kevin Conder.

New in Csound version 3.57

## adsr

`adsr` -- Calculates the classical ADSR envelope using linear segments.

`adsr`

## Description

Calculates the classical ADSR envelope using linear segments.

## Syntax

```
ares adsr iatt, idec, islev, irel [, idel]
```

```
kres adsr iatt, idec, islev, irel [, idel]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

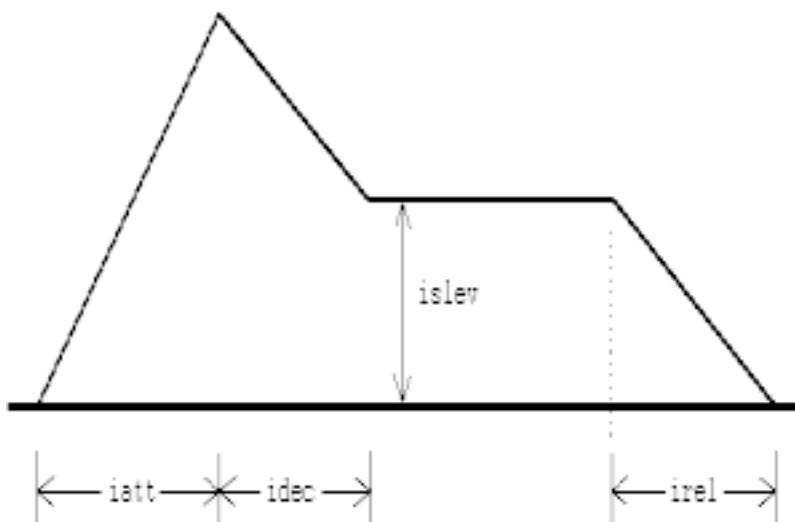
*islev* -- level for sustain phase

*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:



Picture of an ADSR envelope.

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.

*adsr* is new in Csound version 3.49.

## Examples

Here is an example of the *adsr* opcode. It uses the files *adsr.orc* [examples/adsr.orc] and *adsr.sco* [examples/adsr.sco].

### Example 25. Example of the *adsr* opcode.

```
/* adsr.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a simple instrument.
instr 1
  ; Set the amplitude.
  kamp init 20000
  ; Get the frequency from the fourth p-field.
  kcps = cpspch(p4)

  al vco kamp, kcps, 1
  out al
endin

; Instrument #2 - instrument with an ADSR envelope.
instr 2
  iatt = 0.05
  idec = 0.5
  islev = 0.08
  irel = 0.008

  ; Create an amplitude envelope.
  kenv adsr iatt, idec, islev, irel
  kamp = kenv * 20000

  ; Get the frequency from the fourth p-field.
  kcps = cpspch(p4)

  al vco kamp, kcps, 1
  out al
endin
/* adsr.orc */
```

```
/* adsr.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Set the tempo to 120 beats per minute.
t 0 120

; Play a melody with Instrument #1.
; p4 = frequency in pitch-class notation.
i 1 0 1 8.04
i 1 1 1 8.04
i 1 2 1 8.05
i 1 3 1 8.07
i 1 4 1 8.07
```

```
i 1 5 1 8.05
i 1 6 1 8.04
i 1 7 1 8.02
i 1 8 1 8.00
i 1 9 1 8.00
i 1 10 1 8.02
i 1 11 1 8.04
i 1 12 2 8.04
i 1 14 2 8.02

; Repeat the melody with Instrument #2.
; p4 = frequency in pitch-class notation.
i 2 16 1 8.04
i 2 17 1 8.04
i 2 18 1 8.05
i 2 19 1 8.07
i 2 20 1 8.07
i 2 21 1 8.05
i 2 22 1 8.04
i 2 23 1 8.02
i 2 24 1 8.00
i 2 25 1 8.00
i 2 26 1 8.02
i 2 27 1 8.04
i 2 28 2 8.04
i 2 30 2 8.02
e
/* adsr.sco */
```

## See Also

*madsr*, *mxadsr*, *xadsr*

## Credits

Author: John ffitch

Example written by Kevin Conder.

# adsyn

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

adsyn

## Description

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

## Syntax

ares **adsyn** kamod, kfmod, ksmod, ifilcod

## 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* or *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). *adsyn* control contains break-point amplitude- and frequency-envelope values organized for oscillator resynthesis, while *pvoc* control contains similar data organized for fft resynthesis. Memory usage depends on the size of the files involved, which are read and held entirely in memory during computation but are shared by multiple calls (see also *lpread*).

## Performance

*kamod* -- amplitude factor of the contributing partials.

*kfmod* -- frequency factor of the contributing partials. It 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.

*ksmod* -- speed factor of the contributing partials.

*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.



## Examples

Here is an example of the `adsyn` opcode. It uses the files *adsyn.orc* [examples/adsyn.orc], *adsyn.sco* [examples/adsyn.sco], and *kickroll.het* [examples/kickroll.het]. The file “kickroll.het” was created by using the *hetro* utility with the audio file *kickroll.wav* [examples/kickroll.wav].

### Example 26. Example of the `adsyn` opcode.

```
/* adsyn.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; If the modulation amounts are set to 1, adsyn
  ; will not perform any special modulation.
  kamod init 1
  kfmod init 1
  ksmod init 1

  ; Re-synthesizes the file "kickroll.het".
  al adsyn kamod, kfmod, ksmod, "kickroll.het"

  out al * 32768
endin
/* adsyn.orc */

/* adsyn.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* adsyn.sco */
```

## Credits

Example written by Kevin Conder.

# adsynt

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

adsynt

## Description

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

## Syntax

ares **adsynt** kamp, kcps, iwfn, ifreqfn, iampfn, icnt [, iphs]

## 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

Here is an example of the adsynt opcode. It uses the files *adsynt.orc* [examples/adsynt.orc] and *adsynt.sco* [examples/adsynt.sco]. 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.

### Example 27. Example of the adsynt opcode.

```
/* adsynt.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
```

```
ksmps = 10
nchnls = 1

; Generate a sinewave table.
giwave ftgen 1, 0, 1024, 10, 1
; Generate two empty tables for adsynt.
gifrqs ftgen 2, 0, 32, 7, 0, 32, 0
; A table for frequency and amp parameters.
giamps ftgen 3, 0, 32, 7, 0, 32, 0

; Generates parameters at init time
instr 1
; Generate 10 voices.
icnt = 10
; Init loop index.
index = 0

; Loop only executed at init time.
loop:
; Define non-harmonic partials.
ifreq pow index + 1, 1.5
; Define amplitudes.
iamp = 1 / (index+1)
; Write to tables.
tableiw ifreq, index, gifrqs
; Used by adsynt.
tableiw iamp, index, giamps

index = index + 1
; Do loop/
if (index < icnt) igoto loop

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

; Generates parameters every k-cycle.
instr 2
; Generate 10 voices.
icnt = 10
; Reset loop index.
kindex = 0

; Loop executed every k-cycle.
loop:
; Generate lfo for frequencies.
kspeed pow kindex + 1, 1.6
; Individual phase for each voice.
kphas phasorbnk kspeed * 0.7, kindex, icnt
klfo table kphas, giwave, 1
; Arbitrary parameter twiddling...
kdepth pow 1.4, kindex
kfreq pow kindex + 1, 1.5
kfreq = kfreq + klfo*0.006*kdepth

; Write freqs to table for adsynt.
tablew kfreq, kindex, gifrqs

; Generate lfo for amplitudes.
kspeed pow kindex + 1, 0.8
; Individual phase for each voice.
kphas phasorbnk kspeed*0.13, kindex, icnt, 2
klfo table kphas, giwave, 1
; Arbitrary parameter twiddling...
kamp pow 1 / (kindex + 1), 0.4
kamp = kamp * (0.3+0.35*(klfo+1))

; Write amps to table for adsynt.
tablew kamp, kindex, giamps
```

```
kindex = kindex + 1
; Do loop.
if (kindex < icnt) kgoto loop

asig adsynt 5000, 150, giwave, gifrqs, giamps, icnt
out asig
endin
/* adsynt.orc */
```

```
/* adsynt.sco */
; Play Instrument #1 for 2.5 seconds.
i 1 0 2.5
; Play Instrument #2 for 2.5 seconds.
i 2 3 2.5
e
/* adsynt.sco */
```

## Credits

Author: Peter Neubäcker  
Munich, Germany  
August, 1999

New in Csound version 3.58

## adsynt2

`adsynt2` -- Performs additive synthesis with an arbitrary number of partials -not necessarily harmonic- with interpolation.

`adsynt2`

## Description

Performs additive synthesis with an arbitrary number of partials, not necessarily harmonic. (see *adsynt* for detailed manual)

## Syntax

ar **adsynt2** *kamp*, *kcps*, *iwfn*, *ifreqfn*, *iampfn*, *icnt* [, *iphs*]

## 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.

*adsynt2* is identical to *adsynt* (by Peter Neubäcker), except it provides linear interpolation for amplitude envelopes of each partial. It is a bit slower than *adsynt*, but interpolation highly improves sound quality in fast amplitude envelope transients when  $kr < sr$  (i.e. when  $ksmps > 1$ ). No interpolation is provided for pitch envelopes, since in this case sound quality degradation is not so evident even with high values of  $ksmps$ . It is not recommended when  $kr=sr$ , in this case *adsynt* is better (since it is faster).

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

## **aexprand**

aexprand -- Deprecated.

aexprand

### **Description**

Deprecated as of version 3.49. Use the *exprand* opcode instead.

# aftouch

aftouch -- Get the current after-touch value for this channel.

aftouch

## Description

Get the current after-touch value for this channel.

## Syntax

kaft **aftouch** [*imin*] [, *imax*]

## Initialization

*imin* (optional, default=0) -- minimum limit on values obtained.

*imax* (optional, default=127) -- maximum limit on values obtained.

## Performance

Get the current after-touch value for this channel. 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.

## Examples

Here is an example of the aftouch opcode. It uses the files *aftouch.orc* [examples/aftouch.orc] and *aftouch.sco* [examples/aftouch.sco].

### Example 28. Example of the aftouch opcode.

```
/* aftouch.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  k1 aftouch

  printk2 k1
endin
/* aftouch.orc */

/* aftouch.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* aftouch.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.



## agauss

agauss -- Deprecated.

agauss

## Description

Deprecated as of version 3.49. Use the *gauss* opcode instead.

# agogobel

agogobel -- Deprecated.

agogobel

## Description

Deprecated as of version 3.52. Use the *gogobel* opcode instead.

# alinrand

alinrand -- Deprecated.

alinrand

## Description

Deprecated as of version 3.49. Use the *linrand* opcode instead.

# alpass

alpass -- Reverberates an input signal with a flat frequency response.

alpass

## Description

Reverberates an input signal with a flat frequency response.

## Syntax

ares **alpass** asig, krvt, ilpt [, iskip] [, insmps]

## Initialization

*ilpt* -- loop time in seconds, which determines the “echo density” of the reverberation. This in turn characterizes the “color” of the 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. The delay space is allocated and returned as in *delay*.

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

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*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).

This filter reiterates the 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 will begin to appear immediately.

## Examples

Here is an example of the alpass opcode. It uses the files *alpass.orc* [examples/alpass.orc] and *alpass.sco* [examples/alpass.sco].

### Example 29. Example of the alpass opcode.

```
/* alpass.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the audio mixer.
gamix init 0

; Instrument #1.
```

```
instr 1
; Generate a source signal.
a1 oscili 30000, cpspch(p4), 1
; Output the direct sound.
out a1

; Add the source signal to the audio mixer.
gamix = gamix + a1
endin

; Instrument #99 (highest instr number executed last)
instr 99
krvt = 1.5
ilpt = 0.1

; Filter the mixed signal.
a99 alpass gamix, krvt, ilpt
; Output the result.
out a99

; Empty the mixer for the next pass.
gamix = 0
endin
/* alpass.orc */

/* alpass.sco */
; Table #1, a sine wave.
f 1 0 128 10 1

; p4 = frequency (in a pitch-class)
; Play Instrument #1 for a tenth of a second, p4=7.00
i 1 0 0.1 7.00
; Play Instrument #1 for a tenth of a second, p4=7.02
i 1 1 0.1 7.02
; Play Instrument #1 for a tenth of a second, p4=7.04
i 1 2 0.1 7.04
; Play Instrument #1 for a tenth of a second, p4=7.06
i 1 3 0.1 7.06

; Make sure the filter remains active.
i 99 0 5
e
/* alpass.sco */
```

## See Also

*comb, reverb, valpass, vcomb*

## Credits

Author: William “Pete” Moss (*vcomb* and *valpass*)  
University of Texas at Austin  
Austin, Texas USA  
January 2002

Example written by Kevin Conder.

# ampdb

ampdb -- Returns the amplitude equivalent of the decibel value x.

ampdb

## Description

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

## Syntax

**ampdb**(x) (no rate restriction)

## Examples

Here is an example of the ampdb opcode. It uses the files *ampdb.orc* [examples/ampdb.orc] and *ampdb.sco* [examples/ampdb.sco].

### Example 30. Example of the ampdb opcode.

```
/* ampdb.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  idb = 90
  iamp = ampdb(idb)

  print iamp
endin
/* ampdb.orc */

/* ampdb.sco */
; Play Instrument #1 for one second.
i 1 0 1
```

```
e  
/* ampdb.sco */
```

Its output should include lines like:

```
instr 1:  iamp = 31622.764
```

## See Also

*ampdbfs*, *db*, *dbamp*, *dbfsamp*

## Credits

Example written by Kevin Conder.

# ampdbfs

ampdbfs -- Returns the amplitude equivalent of the decibel value x, which is relative to full scale amplitude.

ampdbfs

## Description

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 in Csound version 4.10.

## Syntax

**ampdbfs**(x) (no rate restriction)

## Examples

Here is an example of the ampdbfs opcode. It uses the files *ampdbfs.orc* [examples/ampdbfs.orc] and *ampdbfs.sco* [examples/ampdbfs.sco].

### Example 31. Example of the ampdbfs opcode.

```
/* ampdbfs.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  idb = -1
  iamp = ampdbfs(idb)

  print iamp
endin
/* ampdbfs.orc */

/* ampdbfs.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* ampdbfs.sco */
```

Its output should include lines like:

```
instr 1:  iamp = 29203.621
```



## See Also

*ampdb, dbamp, dbfsamp*

## Credits

Example written by Kevin Conder.

# ampmidi

ampmidi -- Get the velocity of the current MIDI event.

ampmidi

## Description

Get the velocity of the current MIDI event.

## Syntax

```
iamp ampmidi iscal [, ifn]
```

## Initialization

*iscal* -- i-time scaling factor

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

## Performance

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*.

## Examples

Here is an example of the ampmidi opcode. It uses the files *ampmidi.orc* [examples/ampmidi.orc] and *ampmidi.sco* [examples/ampmidi.sco].

### Example 32. Example of the ampmidi opcode.

```
/* ampmidi.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Scale the amplitude between 0 and 1.
il ampmidi 1

print il
endin
/* ampmidi.orc */

/* ampmidi.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
```

```
e  
/* ampmidi.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

## apcauchy

apcauchy -- Deprecated.

apcauchy

## Description

Deprecated as of version 3.49. Use the *pcauchy* opcode instead.

# apoisson

apoisson -- Deprecated.

apoisson

## Description

Deprecated as of version 3.49. Use the *poisson* opcode instead.

## **apow**

apow -- Deprecated.

apow

## **Description**

Deprecated as of version 3.48. Use the *pow* opcode instead.

# areson

*areson* -- A notch filter whose transfer functions are the complements of the *reson* opcode.

*areson*

## Description

A notch filter whose transfer functions are the complements of the *reson* opcode.

## Syntax

*ares* **areson** *asig*, *kcf*, *kbw* [, *iscl*] [, *iskip*]

## Initialization

*iscl* (optional, default=0) -- 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.

*iskip* (optional, default=0) -- 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

*ares* -- the output signal at audio rate.

*asig* -- the input signal at audio rate.

*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).

*areson* is a filter whose transfer functions is the complement of *reson*. Thus *areson* is a notch filter whose transfer functions represents the “filtered out” aspects of their complements. However, power scaling is not normalized in *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*.

## Examples

Here is an example of the *areson* opcode. It uses the files *areson.orc* [examples/*areson.orc*] and *areson.sco* [examples/*areson.sco*].

### Example 33. Example of the *areson* opcode.

```
/* areson.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1 - an unfiltered noise waveform.
instr 1
  ; Generate a white noise signal.
  asig rand 20000

  out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
  ; Generate a white noise signal.
  asig rand 20000

  ; Filter it using the areson opcode.
  kcf init 1000
  kbw init 100
  afilt areson asig, kcf, kbw

  ; Clip the filtered signal's amplitude to 85 dB.
  al clip afilt, 2, ampdb(85)
  out al
endin
/* areson.orc */

/* areson.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* areson.sco */
```

## See Also

*aresonk, atone, atonek, port, portk, reson, resonk, tone, tonek*



# aresonk

*aresonk* -- A notch filter whose transfer functions are the complements of the *reson* opcode.

*aresonk*

## Description

A notch filter whose transfer functions are the complements of the *reson* opcode.

## Syntax

*kres* **aresonk** *ksig*, *kcf*, *kbw* [, *iscl*] [, *iskip*]

## Initialization

*iscl* (optional, default=0) -- 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.

*iskip* (optional, default=0) -- 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

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*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).

*aresonk* is a filter whose transfer functions is the complement of *reson*. Thus *aresonk* is a notch filter whose transfer functions represents the “filtered out” aspects of their complements. However, power scaling is not normalized in *aresonk* but remains the true complement of the corresponding unit.

## See Also

*areson*, *atone*, *atonek*, *port*, *portk*, *reson*, *resonk*, *tone*, *tonek*

# atone

**atone** -- A hi-pass filter whose transfer functions are the complements of the *tone* opcode.

**atone**

## Description

A hi-pass filter whose transfer functions are the complements of the *tone* opcode.

## Syntax

**ares** **atone** *asig*, *khp* [, *iskip*]

## Initialization

*iskip* (optional, default=0) -- 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

*ares* -- the output signal at audio rate.

*asig* -- the input signal at audio rate.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*atone* is a filter whose transfer functions is the complement of *tone*. *atone* is thus a form of high-pass filter whose transfer functions represent the “filtered out” aspects of their complements. However, power scaling is not normalized in *atone* but remains the true complement of the corresponding unit. Thus an audio signal, filtered by parallel matching *tone* and *atone* 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*.

## Examples

Here is an example of the *atone* opcode. It uses the files *atone.orc* [examples/atone.orc] and *atone.sco* [examples/atone.sco].

### Example 34. Example of the *atone* opcode.

```
/* atone.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1
```

```
; Instrument #1 - an unfiltered noise waveform.
instr 1
  ; Generate a white noise signal.
  asig rand 20000

  out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
  ; Generate a white noise signal.
  asig rand 20000

  ; Filter it using the atone opcode.
  khp init 2000
  afilt atone asig, khp

  ; Clip the filtered signal's amplitude to 85 dB.
  al clip afilt, 2, ampdb(85)
  out al
endin
/* atone.orc */

/* atone.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* atone.sco */
```

## See Also

*areson, aresonk, atonek, port, portk, reson, resonk, tone, tonek*

# atonek

*atonek* -- A hi-pass filter whose transfer functions are the complements of the *tonek* opcode.

*atonek*

## Description

A hi-pass filter whose transfer functions are the complements of the *tonek* opcode.

## Syntax

```
kres atonek ksig, khp [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- 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

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*atonek* is a filter whose transfer functions is the complement of *tonek*. *atonek* is thus a form of high-pass filter whose transfer functions represent the “filtered out” aspects of their complements. However, power scaling is not normalized in *atonek* but remains the true complement of the corresponding unit.

## See Also

*areson*, *aresonk*, *atone*, *port*, *portk*, *reson*, *resonk*, *tone*, *tonek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# atonex

atonex -- Emulates a stack of filters using the atone opcode.

atonex

## Description

*atonex* is equivalent to a filter consisting of more layers of *atone* 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.

## Syntax

ares **atonex** asig, khp [, inumlayer] [, iskip]

## Initialization

*inumlayer* (optional) -- number of elements in the filter stack. Default value is 4.

*iskip* (optional, default=0) -- 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* -- input signal

*khp* -- the response curve's half-power point. Half power is defined as peak power / root 2.

## See Also

*resonx*, *tonex*

## Credits

Author: Gabriel Maldonado (adapted by John ffitch)  
Italy

New in Csound version 3.49

## atrirand

atrirand -- Deprecated.

atrirand

## Description

Deprecated as of version 3.49. Use the *trirand* opcode instead.

# ATSadd

ATSadd -- uses the data from an ATS analysis file to perform additive synthesis.

ATSadd

## Description

*ATSadd* reads from an ATS analysis file and uses the data to perform additive synthesis using an internal array of interpolating oscillators.

## Syntax

```
ar ATSadd ktimepnt, kfmod, iatsfile, ifn, ipartials[, ipartialoffset, ipartia
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ifn* – table number of a stored function containing a sine wave for *ATSadd* and a cosine for *ATSaddnz* (see examples below for more info)

*ipartials* – number of partials that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

*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 indices 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 the examples below.

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSadd* exactly the same as for *pvoc*.

*ATSadd* and *ATSaddnz* are based on *pvadd* by Richard Karpen and use files created by Juan Pampin's ATS (*Analysis - Transformation - Synthesis* [<http://www-ccrma.stanford.edu/~juan/ATS.html>]).

*kfmod* – A control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave. Used for *ATSadd* exactly the same as for *pvoc*.

*ATSadd* reads from an ATS analysis 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 partials will be used in the re-synthesis.

## Examples

```
ktime line 0, p3, 2.5  
asig atsadd ktime, 1, "clarinet.ats", 1, 20, 2
```

In the example above, *ipartials* is 20 and *ipartialoffset* is 2. This will synthesize the 3rd thru 22nd partials in the "clarinet.ats" analysis file. *kmod* is 1 so there will be no pitch transformation. Since the *ktimepnt* envelope moves from 0 to 2.5 over the duration of the note, the analysis file will be read from 0 to 2.5 seconds of the original duration of the analysis over the duration of the csound note, this way we can change the duration independent of the pitch.

```
ktime line 0, p3, 2.5  
asig atsadd ktime, 1.0125, "clarinet.ats", 1, 20, 0, 2
```

In the above example we synthesize 20 partials as in example 1 except this time we're using a *ipartialoffset* of 0 and *ipartialincr* of 2, which means that we'll start from the first partial and synthesize 20 partials total, skipping every other one (ie. partial 1, 3, 5,...). We've also increased the pitch of the result (*kfmod* is set to 1.0125).

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSaddnz*, *ATSinnoi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004



# ATSaddnz

ATSaddnz -- uses the data from an ATS analysis file to perform noise resynthesis.

ATSaddnz

## Description

*ATSaddnz* reads from an ATS analysis file and uses the data to perform additive synthesis using a modified randi function.

## Syntax

```
ar ATSaddnz ktimepnt, iatsfile, ifn, ibands[, ibandoffset, ibandincr]
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ifn* – table number of a stored function containing a sine wave for *ATSadd* and a cosine for *ATSaddnz* (see examples below for more info)

*ibands* – number of noise bands that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ibandoffset* (optional) – is the first noise band used (defaults to 0).

*ibandincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ibandoffset* for ibins components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSaddnz* exactly the same as for *pvoc* and *ATSadd*.

*ATSaddnz* and *ATSadd* are based on *pvadd* by Richard Karpen and use files created by Juan Pampin's *ATS (Analysis - Transformation - Synthesis)* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ATSaddnz* also reads from an ATS file but it resynthesizes the noise from noise energy data contained in the ATS file. It uses a modified randi function to create band limited noise and modulates that with a user supplied wave table (one period of a cosine wave), to synthesize a user specified selection of frequency bands. Modulating the noise is required to put the band limited noise in the correct place in the frequency spectrum.

## Examples

```
ktime line 0, p3, 2.5  
asig atsaddnz ktime, "clarinet.ats", 2, 25
```

In the example above we're synthesizing all 25 noise bands from the data contained in the ATS analysis file called "clarinet.ats", we're using function table 2, which should be a cosine ie:

```
f2 0 4096 9 1 1 90
```

```
ptime line 2.5, p3, 0
asig atsaddnz ptime, 1, "clarinet.ats", 2, 1, 24
```

Here we synthesize only the 25th noise band (*ibandoffset* of 24 and *ibands* of 1). Also our time pointer is going from 2.5 to 0 over the duration of the note so we're reading backwards from 2.5 seconds in the analysis file.

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSaddnz*, *ATSinnoi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSbufread

*ATSbufread* -- reads data from and ATS data file and stores it in an internal data table of frequency, amplitude pairs.

*ATSbufread*

## Description

*ATSbufread* reads data from and ATS data file and stores it in an internal data table of frequency, amplitude pairs.

## Syntax

**ATSbufread** *ktimepnt*, *kfmod*, *iatsfile*, *ipartials*[, *ipartialoffset*, *ipartialincr*]

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ipartials* – number of partials that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSbufread* exactly the same as for *pvoc*.

*kfmod* – an input for performing pitch transposition or frequency modulation on all of the synthesized partials, if no fm or pitch change is desired then use a 1 for this value.

*ATSbufread* is based on *pvbufread* by Richard Karpen. *ATScross*, *ATSinterpread* and *ATSpartialtap* are all dependent on *ATSbufread* just as *pvcross* and *pvinterp* are on *pvbufread*. *ATSbufread* reads data from and ATS data file and stores it in an internal data table of frequency, amplitude pairs. The data stored by an *ATSbufread* can only be accessed by other unit generators, and therefore, due to the architecture of Csound, an *ATSbufread* must come before (but not necessarily directly) any dependent unit generator. Besides the fact that *ATSbufread* doesn't output any data directly, it works almost exactly as *ATSadd*. The ugen uses a time pointer (*ktimepnt*) to index the data in time, *ipartials*, *ipartialoffset* and *ipartialincr* to select which partials to store in the table and *kfmod* to scale partials in frequency.

## Examples

See the examples for *ATScross*, *ATSinterpread* and *ATSpartialtap*

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSinnoi*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATScross

ATScross -- perform cross synthesis from ATS analysis files.

ATScross

## Description

*ATScross* uses data from an ATS analysis file and data from an *ATSbufread* to perform cross synthesis.

## Syntax

ar **ATScross** ktimepnt, kfmod, iatsfile, ifn, kmylev, kbuflev, ipartials[, iparti

## Initialization

*iatsfile* – integer or character-string denoting a control-file derived from ATS analysis of an audio signal. An integer denotes the suffix of a file ATS.m; a character-string (in double quotes) gives a filename, optionally a full pathname. If not full-path, the file is sought first in the current directory, then in the one given by the environment variable SADIR (if defined).

*ifn* – table number of a stored function containing a sine wave.

*ipartials* – number of partials that will be used in the resynthesis

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATScross* exactly the same as for *pvoc*.

*kfmod* – an input for performing pitch transposition or frequency modulation on all of the synthesized partials, if no fm or pitch change is desired then use a 1 for this value.

*kmylev* - scales the *ATScross* component of the frequency spectrum applied to the partials from the ATS file indicated by the *atscross* opcode. The frequency spectrum information comes from the *atscross* ATS file. A value of 1 (and 0 for *kbuflev*) gives the same results as *ATSadd*.

*kbuflev* - scales the *ATSbufread* component of the frequency spectrum applied to the partials from the ATS file indicated by the *ATScross* opcode. The frequency spectrum information comes from the *ATSbufread* ATS file. A value of 1 (and 0 for *kmylev*) results in partials that have frequency information from the ATS file given by the *ATScross*, but amplitudes imposed by data from the ATS file given by *ATSbufread*.

*ATScross* uses data from an ATS analysis file (indicated by *iatsfile*) and data from an *ATSbufread* to perform cross synthesis. *ATScross* uses *ktimepnt*, *kfmod*, *ipartials*, *ipartialoffset* and *ipartialincr* just like *ATSadd*. *ATScross* synthesizes a sine-wave for each partial selected by the user and uses the frequency of that partial (after scaling in frequency by *kfmod*) to index the table created by *ATSbufread*. Interpolation is used to get in-between values. *ATScross* uses the sum of the amplitude data from its ATS file (scaled by *kmylev*) and the amplitude data gained from an *ATSbufread* (scaled by *kbuflev*) to scale the amplitude of each partial it synthesizes. Setting *kmylev* to one and *kbuflev* to zero will make *ATScross* act exactly like *ATSadd*. Setting *kmylev* to zero and *kbuflev* to one will pro-

duce a sound that has all the partials selected by the *ATScross* ugen, but with amplitudes taken from an *ATSbufread*. The time pointers of the *ATSbufread* and *ATScross* do not need to be the same.

## Examples

```
ktime line 0, p3, 2.4
ktime2 line 0, p3, .5
kline expseg 0.001, .9, 1, p3-.9, 1
kline2 expseg .001, p3, 1
atsbufread ktime2, 1, "crt.ats", 20
aout atscross ktime, 1, "cl.ats", 1, kline, .001* (1 - kline2), 42
```

This example performs cross synthesis using two ATS files, "crt.ats" and "cl.ats". The result of this will be a sound that starts out with the shape (in frequency) of crt.ats, and ends with the shape of cl.ats. All the sine-wave frequencies come from cl.ats. The *kbuflev* value is scaled because the energy produced by applying crt.ats's frequency spectrum to cl.ats's partials is very large. Notice also that the time pointers of the *atsbufread* (crt.ats) and *atscross* (cl.ats) need not have the same value, this way you can read through the two ATS files at different rates.

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSinnoi*, *ATSbufread*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSinfo

ATSinfo -- reads data out of the header of an ATS file.

ATSinfo

## Description

*atsinfo* reads data out of the header of an ATS file.

## Syntax

```
idata ATSinfo iatsfile, ilocation
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ilocation* – indicates which location in the header file to return. The data in the header gives information about the data contained in the rest of the ATS file. The possible values for *ilocation* are given in the following list:

- 0 - Sample rate (Hz)
- 1 - Frame Size (samples)
- 2 - Window Size (samples)
- 3 - Number of Partial
- 4 - Number of Frames
- 5 - Maximum Amplitude
- 6 - Maximum Frequency (Hz)
- 7 - Duration (seconds)
- 8 - ATS file Type

## Performance

Macros can really improve the legibility of your csound code, I've provided my Macro Definitions below:

```
#define ATS_SAMP_RATE #0#  
#define ATS_FRAME_SZ #1#  
#define ATS_WIN_SZ #2#  
#define ATS_N_PARTIALS #3#  
#define ATS_N_FRAMES #4#  
#define ATS_AMP_MAX #5#  
#define ATS_FREQ_MAX #6#  
#define ATS_DUR #7#  
#define ATS_TYPE #8#
```

*ATSinfo* can be useful for writing generic instruments that will work with many ATS files, even if

they have different lengths and different numbers of partials etc. Example 2 is a simple application of this.

## Examples

1.

```
imax_freq atsinfo "cl.ats", $ATS_FREQ_MAX
```

In the example above we get the maximum frequency value from the ATS file "cl.ats" and store it in `imax_freq`. We use at Csound Macro (defined above) `$ATS_FREQ_MAX`, which is equivalent to the number 6.

2.

```
i_npartials atsinfo p4, $ATS_N_PARTIALS
i_dur      atsinfo p4, $ATS_DUR
ktimepnt line 0, p3, i_dur
aout      atsadd ktimepnt, 1, p4, 1, i_npartials
```

In the example above we use *ATSinfo* to retrieve the duration and number of partials in the ATS file indicated by `p4`. With this info we synthesize the partials using `atsadd`. Since the duration and number of partials are not "hard-coded" we can use this code with any ats file.

## See also

*ATSread*, *ATSreadnz*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*, *ATSinnoi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004



# ATSinterpread

ATSinterpread -- allows a user to determine the frequency envelope of any *ATSbufread*.

ATSinterpread

## Description

*ATSinterpread* allows a user to determine the frequency envelope of any *ATSbufread*.

## Syntax

```
kamp ATSinterpread kfreq
```

## Performance

*kfreq* - a frequency value (given in Hertz) used by *ATSinterpread* as in index into the table produced by an *ATSbufread*.

*ATSinterpread* takes a frequency value (*kfreq* in Hz). This frequency is used to index the data of an *ATSbufread*. The return value is an amplitude gained from the *ATSbufread* after interpolation. *ATSinterpread* allows a user to determine the frequency envelope of any *ATSbufread*. This data could be useful for an number of reasons, one might be performing cross synthesis of data from an ATS file and non ATS data.

## Examples

```
      ktime line 0, p3, 2.4
      atsbufread ktime, 1, "cl.ats", 42
      kamp atsinterpread p4
      aosc oscili kamp, p4, 1
```

This example shows how to use *ATSinterpread*. Here a frequency is given by the score (p4) and this frequency is given to an *ATSinterpread* (with a corresponding *ATSbufread*). The *ATSinterpread* uses this frequency to output a corresponding amplitude value, based on the atsfile given by the *ATSbufread* (cl.ats in this case). We then use that amplitude to scale a sine-wave that is synthesized with the same frequency (p4). You could extend this to include multiple sine-waves. This way you could synthesize any reasonable frequency (within the low and high frequencies of the indicated ATS file), and maintain the shape (in frequency) of the indicated atsfile (given by the *ATSbufread*).

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSSinnoi*, *ATSbufread*, *ATScross*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSread

ATSread -- reads data from an ATS file.

ATSread

## Description

*ATSread* returns the amplitude (*kamp*) and frequency (*kfreq*) information of a user specified partial contained in the ATS analysis file at the time indicated by the time pointer *ktimepnt*.

## Syntax

```
kfreq, kamp ATSread ktimepnt, iatsfile, ipartial
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ipartial* – the number of the analysis partial to return the frequency in Hz and amplitude.

## Performance

*kfreq*, *kamp* - outputs of the *ATSread* unit. These values represent the frequency and amplitude of a specific partial selected by the user using *ipartial*. The partials' informations are derived from an ATS analysis. *ATSread* linearly interpolates the frequency and amplitude between frames in the ATS analysis file at k-rate. The output is dependent on the data in the analysis file and the pointer *ktimepnt*.

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSread* exactly the same as for *pvoc* and *ATSadd*.

## Examples

```
ktime line 0, p3, 2.5
kfreq, kamp atsread ktime, "clarinet.ats", 2
aout oscili 1000000 * kamp, kfreq, 1
```

Here we're using *ATSread* to get the 2nd partial's frequency and amplitude data out of the 'clarinet.ats' ATS analysis file. We're using that data to drive an oscillator, but we could use it for anything else that can take a k-rate input, like the bandwidth and resonance of a filter etc.

## See also

*ATSreadnz*, *ATSinfo*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*, *ATSSinnoi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004



# ATSreadnz

ATSreadnz -- reads data from an ATS file.

ATSreadnz

## Description

*ATSreadnz* returns the energy (*kenergy*) of a user specified noise band (1-25 bands) at the time indicated by the time pointer *ktimepnt*.

## Syntax

```
kenergy ATSreadnz ktimepnt, iatsfile, iband
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ibands* – the number of the noise band to return the energy data.

## Performance

*kenergy* outputs the linearly interpolated energy of the noise band indicated in *iband*. The output is dependent on the data in the analysis file and the *ktimepnt*.

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSreadnz* exactly the same as for *pvoc* and *ATSadd*.

*ATSaddnz* reads from an ATS file and resynthesizes the noise from noise energy data contained in the ATS file. It uses a modified randi function to create band limited noise and modulates that with a user supplied wave table (one period of a cosine wave), to synthesize a user specified selection of frequency bands. Modulating the noise is required to put the band limited noise in the correct place in the frequency spectrum.

An ATS analysis differs from a pvanal in that ATS tracks the partials and computes the noise energy of the sound being analyzed. For more info about ATS analysis read Juan Pampin's description on the the *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>] web-page.

## Examples

```
ktime line 2.5, p3, 0
kenergy atsreadnz ktime, "clarinet.ats", 5
```

Here we are extracting the noise energy from band 5 in the 'clarinet.ats' ATS analysis file. We're actually reading backwards from 2.5 seconds to the beginning of the analysis file. We could use this to synthesize noise like this:

```
anoise randi sqrt(kenergy), 55
      aout      oscili 4000000000000000000000000000, 455, 2
aout = aout * anoise
```

Function table 2 used in the oscillator is a cosine, which is needed to shift the band limited noise in-

to the correct place in the frequency spectrum. The `randi` function creates a band of noise centered about 0 Hz that has a bandwidth of about 110 Hz; multiplying it by a cosine will shift it to be centered at 455 Hz, which is the center frequency of the 5th critical noise band. This is only an example, for synthesizing the noise you'd be better off just using *ATSaddnz* unless you want to use your own noise synthesis algorithm. Maybe you could use the noise energy for something else like applying a small amount of jitter to specific partials or for controlling something totally unrelated to the source sound?

## See also

*ATSread*, *ATSinfo*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*, *ATSinnoi*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSpartialtap

ATSpartialtap -- returns a frequency, amplitude pair from an *ATSbufread* opcode.

ATSpartialtap

## Description

*ATSpartialtap* takes a partial number and returns a frequency, amplitude pair. The frequency and amplitude data comes from an *atsbufread* *ATSbufread* opcode.

## Syntax

```
kfrq, kamp ATSpartialtap ipartialnum
```

## Initialization

*ipartialnum* - indicates the partial that the *ATSpartialtap* opcode should read from an *ATSbufread*.

## Performance

*kfrq* - returns the frequency value for the requested partial.

*kamp* - returns the amplitude value for the requested partial.

*ATSpartialtap* takes a partial number and returns a frequency, amplitude pair. The frequency and amplitude data comes from an *ATSbufread* opcode. This is more restricted version of *ATSread*, since each *ATSread* opcode has its own independent time pointer, and *ATSpartialtap* is restricted to the data given by an *ATSbufread*. Its simplicity is its attractive feature.

## Examples

```
ktime line 0, p3, 2.4
atsbufread ktime, 1, "crt.ats", 20
kfreq1, kamp1 atspartialtap 1
kfreq2, kamp2 atspartialtap 10
kfreq3, kamp3 atspartialtap 20
```

This example here uses an *ATSpartialtap*, and an *ATSbufread* to read partials 1, 10 and 20 from 'crt.ats'. These amplitudes and frequencies could be used to re-synthesize those partials, or something all together different.

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSSinnoi*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSadd*, *AT-Saddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004

# ATSSinnoi

ATSSinnoi -- uses the data from an ATS analysis file to perform resynthesis.

ATSSinnoi

## Description

*ATSSinnoi* reads data from an ATS data file and uses the information to synthesize sines and noise together.

## Syntax

```
ar ATSSinnoi ktimepnt, ksinlev, knzlev, kfmod, iatsfile, ipartial[, ipartial
```

## Initialization

*iatsfile* – the ATS number (n in ats.n) or the name in quotes of the analysis file made using *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>].

*ipartial* – number of partials that will be used in the resynthesis (the noise has a maximum of 25 bands)

*ipartialoffset* (optional) – is the first partial used (defaults to 0).

*ipartialincr* (optional) – sets an increment by which these synthesis opcodes counts up from *ipartialoffset* for ibins components in the re-synthesis (defaults to 1).

## Performance

*ktimepnt* – The time pointer in seconds used to index the ATS file. Used for *ATSSinnoi* exactly the same as for *pvoc*.

*ksinlev* - controls the level of the sines in the *ATSSinnoi* ugen. A value of 1 gives full volume sine-waves.

*knzlev* - controls the level of the noise components in the *ATSSinnoi* ugen. A value of 1 gives full volume noise.

*kfmod* – an input for performing pitch transposition or frequency modulation on all of the synthesized partials, if no fm or pitch change is desired then use a 1 for this value.

*ATSSinnoi* reads data from an ATS data file and uses the information to synthesize sines and noise together. The noise energy for each band is distributed equally among each partial that falls in that band. Each partial is then synthesized, along with that partial's noise component. Each noise component is then modulated by the corresponding partial to be put in the correct place in the frequency spectrum. The level of the noise and the partials are individually controllable. See the *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>] webpage for more info about the sinnoi synthesis. An ATS analysis differs from a pvanal in that ATS tracks the partials and computes the noise energy of the sound being analyzed. For more info about ATS analysis read Juan Pampin's description on the the *ATS* [<http://www-ccrma.stanford.edu/~juan/ATS.html>] web-page.

## Examples

```
ktime line 0, p3, 2.5
asig atssinnoi ktime, 1, 1, 1, "clarinet.ats", 42
```

Here we synthesize both the noise and the sinewaves (all 42 partials) contained in "clarinet.ats" together. The relative volumes of the noise and the partials are unaltered (each set to 1).

```
ptime line 0, p3, 2.5
knzfade expon 0.001, p3, 2.5
asig atssinnoi ptime, 1, knzfade, 1, "clarinet.ats", 42
```

This example here is like example 5 except that we use an envelope to control *knzlev* (the noise level). The result of this will be a clarinet sound that has its noise component fade in over the duration of the note.

## See also

*ATSread*, *ATSreadnz*, *ATSinfo*, *ATSbufread*, *ATScross*, *ATSinterpread*, *ATSpartialtap*, *ATSadd*, *ATSaddnz*

## Credits

Author: Alex Norman  
Seattle, Washington  
2004



# **aunirand**

aunirand -- Deprecated.

aunirand

## **Description**

Deprecated as of version 3.49. Use the *unirand* opcode instead.

# aweibull

aweibull -- Deprecated.

aweibull

## Description

Deprecated as of version 3.49. Use the *weibull* opcode instead.

# babo

babo -- A physical model reverberator.

babo

## Description

*babo* stands for *ball-within-the-box*. It is a physical model reverberator based on the 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.

## Syntax

```
a1, a2 babo asig, ksrcx, ksrcy, ksrcz, irx, iry, irz [, idiff] [, ifno]
```

## 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

Here is a simple example of the babo opcode. It uses the files *babo.orc* [examples/babo.orc], *babo.sco* [examples/babo.sco], and *beats.wav* [examples/beats.wav].

**Example 35. A simple example of the babo opcode.**

```
/* babo.orc */
/* Written by Nicola Bernardini */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; minimal babo instrument
;
instr 1
    ix      = p4 ; x position of source
    iy      = p5 ; y position of source
    iz      = p6 ; z position of source
    ixsize  = p7 ; width  of the resonator
    iysize  = p8 ; depth  of the resonator
    izsize  = p9 ; height of the resonator

    ainput soundin "beats.wav"

    al,ar    babo    ainput*0.7, ix, iy, iz, ixsize, iysize, izsize

    outs     al,ar
endin
/* babo.orc */

/* babo.sco */
/* Written by Nicola Bernardini */
; simple babo usage:
;
;p4      : x position of source
;p5      : y position of source
;p6      : z position of source
;p7      : width  of the resonator
;p8      : depth  of the resonator
;p9      : height of the resonator
;
i  1  0  10  6  4  3      14.39 11.86 10
;          ^^^^^^      ^^^^^^^^^^^^^^
;          |||||      ++++++ : optimal room dims according to
;          |||||      ++++++ : Milner and Bernard JASA 85(2), 1989
;          ++++++ : source position
e
/* babo.sco */
```

Here is an advanced example of the babo opcode. It uses the files *babo\_expert.orc* [examples/babo\_expert.orc], *babo\_expert.sco* [examples/babo\_expert.sco], and *beats.wav* [examples/beats.wav].

**Example 36. An advanced example of the babo opcode.**

```
/* babo_expert.orc */
```

---

```
/* Written by Nicola Bernardini */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; full blown babo instrument with movement
;
instr 2
  ixstart = p4 ; start x position of source (left-right)
  ixend = p7 ; end x position of source
  iystart = p5 ; start y position of source (front-back)
  iyend = p8 ; end y position of source
  izstart = p6 ; start z position of source (up-down)
  izend = p9 ; end z position of source
  ixsize = p10 ; width of the resonator
  iysize = p11 ; depth of the resonator
  izsize = p12 ; height of the resonator
  idiff = p13 ; diffusion coefficient
  iexpert = p14 ; power user values stored in this function

  ainput soundin "beats.wav"
  ksource_x line ixstart, p3, ixend
  ksource_y line iystart, p3, iyend
  ksource_z line izstart, p3, izend

  al,ar babo ainput*0.7, ksource_x, ksource_y, ksource_z, ixsize, iysize,
          outs al,ar
endin
/* babo_expert.orc */

/* babo_expert.sco */
/* Written by Nicola Bernardini */
; full blown instrument
;p4 : start x position of source (left-right)
;p5 : end x position of source
;p6 : start y position of source (front-back)
;p7 : end y position of source
;p8 : start z position of source (up-down)
;p9 : end z position of source
;p10 : width of the resonator
;p11 : depth of the resonator
;p12 : height of the resonator
;p13 : diffusion coefficient
;p14 : 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
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
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 0 10 6 4 3 6 4 3 14.39 11.86 10 1 6 ; defaults
i2 + 4 6 4 3 6 4 3 14.39 11.86 10 1 1 ; hear brightness 1
i2 + 4 6 4 3 -6 -4 3 14.39 11.86 10 1 2 ; hear brightness 2
i2 + 4 6 4 3 -6 -4 3 14.39 11.86 10 1 3 ; hear brightness 3
i2 + 3 .6 .4 .3 -.6 -.4 .3 1.439 1.186 1.0 0.0 4 ; hear diffusion 1
i2 + 3 .6 .4 .3 -.6 -.4 .3 1.439 1.186 1.0 1.0 4 ; hear diffusion 2
```

e

1999

2000

## New in Csound version 4.09

# balance

`balance` -- Adjust one audio signal according to the values of another.

`balance`

## Description

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

## Syntax

`ares balance asig, acomp [, ihp] [, iskip]`

## Initialization

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

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

## Performance

*asig* -- input audio signal

*acomp* -- the comparator signal

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

## Examples

Here is an example of the `balance` opcode. It uses the files *balance.orc* [examples/balance.orc] and *balance.sco* [examples/balance.sco].

### Example 37. Example of the `balance` opcode.

```
/* balance.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a band-limited pulse train.
  asrc buzz 30000, 440, sr/440, 1

  ; Send the source signal through 2 filters.
  a1 reson asrc, 1000, 100
  a2 reson a1, 3000, 500
```

```
; Balance the filtered signal with the source.
afin balance a2, asrc

    out afin
endin
/* balance.orc */


/* balance.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* balance.sco */
```

## See Also

*gain, rms*



# bamboo

**bamboo** -- Semi-physical model of a bamboo sound.

bamboo

## Description

*bamboo* is a semi-physical model of a bamboo sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **bamboo** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 1.25.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.9999 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.9999 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 0.05.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2800.

*ifreq1* (optional) -- the first resonant frequency. The default value is 2240.

*ifreq2* (optional) -- the second resonant frequency. The default value is 3360.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the bamboo opcode. It uses the files *bamboo.orc* [examples/bamboo.orc] and *bamboo.sco* [examples/bamboo.sco].

### Example 38. Example of the bamboo opcode.

```
/* bamboo.orc */
sr = 44100
```

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

instr 01 ;example of bamboo
a1 bamboo p4, 0.01
  out a1
endin
/* bamboo.orc */

/* bamboo.sco */
i1 0 1 20000
e
/* bamboo.sco */
```

## See Also

*dripwater, guiro, sleighbells, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John fitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# barmodel

**barmodel** -- Creates a tone similar to a stuck metal bar.

**barmodel**

## Description

Audio output is a tone similar to a stuck metal bar, using a physical model developed from solving the partial differential equation. There are controls over the boundary conditions as well as the bar characteristics.

## Syntax

```
ares barmodel kbcL, kbcR, iK, ib,  
      kscan, iT30, ipos, ivel, iwid
```

## Initialization

*iK* -- dimensionless siffness parameter. If this parameter is negative then the initialisation is skipped and the previous state of the bar is continued.

*ib* -- high-frequency loss parameter (keep this small)/

*iT30* -- 30 db decay time in seconds.

*ipos* -- position along the bar that the strike occurs.

*ivel* -- normalized strike velocity.

*iwid* -- spatial width of strike.

## Performance

A note is played on a metallic bar, with the arguments as below.

*kbcL* -- Boundary condition at left end of bar (1 is clamped, 2 pivoting and 3 free).

*kbcR* -- Boundary condition at right end of bar (1 is clamped, 2 pivoting and 3 free).

*kscan* -- Speed of scanning the output location.

Note that changing the boundary conditions during playing may lead to glitches and is made available as an experiment. The use of a non-zero *kscan* can give apparent re-introduction of sound due to modulation.

## Examples

Here is an example of the **barmodel** opcode. It uses the files *barmodel.orc* [examples/barmodel.orc] and *barmodel.sco* [examples/barmodel.sco].

### Example 39. Example of the barmodel opcode.

```
/* barmodel.orc */  
; Initialize the global variables.
```

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

; Instrument #1.
instr 1
  aq      barmodel    1, 1, p4, 0.001, 0.23, 5, p5, p6, p7
           out      aq
endin
/* barmodel.orc */

/* barmodel.sco */

i1 0.0 0.5  3 0.2 500  0.05
i1 0.5 0.5 -3 0.3 1000 0.05
i1 1.0 0.5 -3 0.4 1000 0.1
i1 1.5 4.0 -3 0.5 800  0.05
e
/* barmodel */
```

## Credits

Author: Stefan Bilbao  
University of Edinburgh, UK  
Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 5.01

## bbcutm

bbcutm -- Generates breakbeat-style cut-ups of a mono audio stream.

bbcutm

## Description

The BreakBeat Cutter automatically generates cut-ups of a source audio stream in the style of drum and bass/jungle breakbeat manipulations. There are two versions, for mono (*bbcutm*) or stereo (*bbcuts*) sources. Whilst originally based on breakbeat cutting, the opcode can be applied to any type of source audio.

The prototypical cut sequence favoured over one bar with eighth note subdivisions would be

3+ 3R + 2

where we take a 3 unit block from the source's start, repeat it, then 2 units from the 7th and 8th eighth notes of the source.

We talk of rendering phrases (a sequence of cuts before reaching a new phrase at the beginning of a bar) and units (as subdivision th notes).

The opcode comes most alive when multiple synchronised versions are used simultaneously.

## Syntax

a1 **bbcutm** asource, ibps, isubdiv, ibarlength, iphrasebars, inumrepeats [, istut

## Initialization

*ibps* -- Tempo to cut at, in beats per second.

*isubdiv* -- Subdivisions unit, for a bar. So 8 is eighth notes (of a 4/4 bar).

*ibarlength* -- How many beats per bar. Set to 4 for default 4/4 bar behaviour.

*iphrasebars* -- The output cuts are generated in phrases, each phrase is up to iphrasebars long

*inumrepeats* -- In normal use the algorithm would allow up to one additional repeat of a given cut at a time. This parameter allows that to be changed. Value 1 is normal- up to one extra repeat. 0 would avoid repeating, and you would always get back the original source except for enveloping and stuttering.

*istutterspeed* -- (optional, default=1) The stutter can be an integer multiple of the subdivision speed. For instance, if subdiv is 8 (quavers) and stutterspeed is 2, then the stutter is in semiquavers (sixteenth notes= subdiv 16). The default is 1.

*istutterchance* -- (optional, default=0) The tail of a phrase has this chance of becoming a single repeating one unit cell stutter (0.0 to 1.0). The default is 0.

*ienvchoice* -- (optional, default=1) choose 1 for on (exponential envelope for cut grains) or 0 for off. Off will cause clicking, but may give good noisy results, especially for percussive sources. The default is 1, on.

## Performance

*asource* -- The audio signal to be cut up. This version runs in real-time without knowledge of future audio.

## Examples

Here is a simple example of the *bbcutm* opcode. It uses the files *bbcutm.orc* [examples/bbcutm.orc], *bbcutm.sco* [examples/bbcutm.sco], and *beats.wav* [examples/beats.wav].

### Example 40. A simple example of the *bbcutm* opcode.

```
/* bbcutm.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - Play an audio file normally.
instr 1
  asource soundin "beats.wav"
  out asource
endin

; Instrument #2 - Cut-up an audio file.
instr 2
  asource soundin "beats.wav"

  ibps = 4
  isubdiv = 8
  ibarlength = 4
  iphrasebars = 1
  inumrepeats = 2

  a1 bbcutm asource, ibps, isubdiv, ibarlength, iphrasebars, inumrepeats

  out a1
endin
/* bbcutm.orc */

/* bbcutm.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 3 2
e
/* bbcutm.sco */
```

Here are some more advanced examples...

### Example 41. First steps- mono and stereo versions

```
<CsoundSynthesizer>
<CsInstruments>
```

```
sr      =      44100
kr      =      4410
ksmps   =      10
nchnls  =      2

instr 1
  asource diskin "break7.wav",1,0,1    ; a source breakbeat sample, wraparound

  ; cuts in eighth notes per 4/4 bar, up to 4 bar phrases, up to 1
  ; repeat in total (standard use) rare stuttering at 16 note speed,
  ; no enveloping
  asig bbcutm asource, 2.6937, 8,4,4,1,    2,0.1,0

  outs      asig,asig
endin

instr 2 ;stereo version
  asource1,asource2 diskin "break7stereo.wav",1,0,1    ; a source breakbeat sa

  ; cuts in eighth notes per 4/4 bar, up to 4 bar phrases, up to 1
  ; repeat in total (standard use) rare stuttering at 16 note speed,
  ; no enveloping
  asig1,asig2 bbcuts asource1, asource2, 2.6937, 8,4,4,1,    2,0.1,0

  outs  asig1,asig2
endin

</CsInstruments>
<CsScore>
i1 0 10
i2 11 10
e
</CsScore>
</CsoundSynthesizer>
```

## Example 42. Multiple simultaneous synchronised breaks

```
<CsoundSynthesizer>
<CsInstruments>
sr      =      44100
kr      =      4410
ksmps   =      10
nchnls  =      1

instr 1
  ibps   = 2.6937
  iplaybackspeed = ibps/p5
  asource diskin p4,ipplaybackspeed,0,1

  asig bbcutm asource, 2.6937, p6,4,4,p7,    2,0.1,1

  out  asig
endin

</CsInstruments>
<CsScore>

;   source      bps cut repeats
i1 0 10 "break1.wav" 2.3 8    2 //2.3 is the source original tempo
i1 0 10 "break2.wav" 2.4 8    3
i1 0 10 "break3.wav" 2.5 16   4
e
```

```
</CsScore>
</CsoundSynthesizer>
```

**Example 43. Cutting up any old audio- much more interesting noises than this should be possible!**

```
<CsoundSynthesizer>
<CsInstruments>
sr      =      44100
kr      =      4410
ksmps   =      10
nchnls  =      1

instr 1
  asource oscil 20000,70,1
  ; ain, bps, subdiv, barlength, phrasebars, numrepeats,
  ; stutterspeed, stutterchance, envelopingon
  asig bbcutm asource, 2, 32,1,1,2, 4,0.6,1
  outs asig
endin

</CsInstruments>
<CsScore>
f1 0 256 10 1
i1 0 10
e
</CsScore>
</CsoundSynthesizer>
```

**Example 44. Constant stuttering- faked, not possible since can only stutter in last half bar could make extra stuttering option parameter**

```
<CsoundSynthesizer>
<CsInstruments>
sr      =      44100
kr      =      4410
ksmps   =      10
nchnls  =      1

instr 1
  asource diskin "break7.wav",1,0,1

  ;16th note cuts- but cut size 2 over half a beat.
  ;each half beat will eiather survive intact or be turned into
  ;the first sixteenth played twice in succession

  asig bbcutm asource,2.6937,2,0.5,1,2, 2,1.0,0
  outs asig
endin

</CsInstruments>
<CsScore>
i1 0 30
e
```



```
</CsScore>  
</CsoundSynthesizer>
```

## See Also

*bbcuts*

## Credits

Author: Nick Collins  
London  
August 2001

New in version 4.13

## bbcuts

`bbcuts` -- Generates breakbeat-style cut-ups of a stereo audio stream.

`bbcuts`

## Description

The BreakBeat Cutter automatically generates cut-ups of a source audio stream in the style of drum and bass/jungle breakbeat manipulations. There are two versions, for mono (*bbcutm*) or stereo (*bbcuts*) sources. Whilst originally based on breakbeat cutting, the opcode can be applied to any type of source audio.

The prototypical cut sequence favoured over one bar with eighth note subdivisions would be

$3 + 3R + 2$

where we take a 3 unit block from the source's start, repeat it, then 2 units from the 7th and 8th eighth notes of the source.

We talk of rendering phrases (a sequence of cuts before reaching a new phrase at the beginning of a bar) and units (as subdivision th notes).

The opcode comes most alive when multiple synchronised versions are used simultaneously.

## Syntax

`a1,a2 bbcuts asource1, asource2, ibps, isubdiv, ibarlength, iphrasebars, inumrepeats`

## Initialization

*ibps* -- Tempo to cut at, in beats per second.

*isubdiv* -- Subdivisions unit, for a bar. So 8 is eighth notes (of a 4/4 bar).

*ibarlength* -- How many beats per bar. Set to 4 for default 4/4 bar behaviour.

*iphrasebars* -- The output cuts are generated in phrases, each phrase is up to *iphrasebars* long

*inumrepeats* -- In normal use the algorithm would allow up to one additional repeat of a given cut at a time. This parameter allows that to be changed. Value 1 is normal- up to one extra repeat. 0 would avoid repeating, and you would always get back the original source except for enveloping and stuttering.

*istutterspeed* -- (optional, default=1) The stutter can be an integer multiple of the subdivision speed. For instance, if *subdiv* is 8 (quavers) and *stutterspeed* is 2, then the stutter is in semiquavers (sixteenth notes= subdiv 16). The default is 1.

*istutterchance* -- (optional, default=0) The tail of a phrase has this chance of becoming a single repeating one unit cell stutter (0.0 to 1.0). The default is 0.

*ienvchoice* -- (optional, default=1) choose 1 for on (exponential envelope for cut grains) or 0 for off. Off will cause clicking, but may give good noisy results, especially for percussive sources. The default is 1, on.

## Performance

*asource* -- The audio signal to be cut up. This version runs in real-time without knowledge of future audio.

## Examples

See the advanced examples for the *bbcutm* opcode.

## See Also

*bbcutm*

## Credits

Author: Nick Collins  
London  
August 2001

New in version 4.13

# betarand

betarand -- Beta distribution random number generator (positive values only).

betarand

## Description

Beta distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

ares **betarand** krange, kalpha, kbeta

ires **betarand** krange, kalpha, kbeta

kres **betarand** krange, kalpha, kbeta

## Performance

*krange* -- range of the random numbers (0 - *krange*).

*kalpha* -- alpha value. If *kalpha* is smaller than one, smaller values favor values near 0.

*kbeta* -- beta value. 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.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the betarand opcode. It uses the files *betarand.orc* [examples/betarand.orc] and *betarand.sco* [examples/betarand.sco].

### Example 45. Example of the betarand opcode.

```
/* betarand.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a number between 0 and 1 with a
```

```
; uniform distribution.
; krange = 1
; kalpha = 1
; kbeta = 1

i1 betarand 1, 1, 1

print i1
endin
/* betarand.orc */

/* betarand.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* betarand.sco */
```

Its output should include lines like:

```
instr 1:  i1 = 24583.412
```

## See Also

*seed, bexprnd, cauchy, exprnd, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# bexprnd

bexprnd -- Exponential distribution random number generator.

bexprnd

## Description

Exponential distribution random number generator. This is an x-class noise generator.

## Syntax

ares **bexprnd** krange

ires **bexprnd** krange

kres **bexprnd** krange

## Performance

*krange* -- the range of the random numbers (*-krange* to *+krange*)

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the bexprnd opcode. It uses the files *bexprnd.orc* [examples/bexprnd.orc] and *bexprnd.sco* [examples/bexprnd.sco].

### Example 46. Example of the bexprnd opcode.

```
/* bexprnd.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number between -1 and 1.
  ; krange = 1

  i1 bexprnd 1

  print i1
endin
/* bexprnd.orc */
```

```
/* bexprnd.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* bexprnd.sco */
```

Its output should include lines like:

```
instr 1:  i1 = 1.141
```

## See Also

*seed, betarand, cauchy, exprand, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# bformenc

bformenc -- Codes a signal into the ambisonic B format

bformenc

## Description

Codes a signal into the ambisonic B format

## Syntax

*aw, ax, ay, az* **bformenc** *asig, kalpha, kbeta, kord0, kord1*

*aw, ax, ay, az, ar, as, at, au, av* **bformenc** *asig, kalpha, kbeta, kord0, kord1 ,*

*aw, ax, ay, az, ar, as, at, au, av, ak, al, am, an, ao, ap, aq* **bformenc** *asig, k*

## Performance

*aw, ax, ay, ...* -- output cells of the B format.

*asig* -- input signal.

*kalpha* — azimuth angle in degrees (clockwise).

*kbeta* -- altitude angle in degrees.

*kord0* -- linear gain of the zero order B format.

*kord1* -- linear gain of the first order B format.

*kord2* -- linear gain of the second order B format.

*kord3* -- linear gain of the third order B format.

## Example

```
instr 1
    ; generate pink noise
    anoise pinkish 1000

    ; two full turns
    kalpha line 0, p3, 720
    kbeta = 0

    ; fade ambisonic order from 2nd to 0th during second turn
    kord0 = 1
    kord1 linseg 1, p3 / 2, 1, p3 / 2, 0
    kord2 linseg 1, p3 / 2, 1, p3 / 2, 0

    ; generate B format
    aw, ax, ay, az, ar, as, at, au, av bformenc anoise, kalpha, kbeta, kord0, kord1, kord2, kord3

    ; decode B format for 8 channel circle loudspeaker setup
    a1, a2, a3, a4, a5, a6, a7, a8 bformdec 4, aw, ax, ay, az, ar, as, at, au, av

    ; write audio out
```



```
        out a1, a2, a3, a4, a5, a6, a7, a8  
endin
```

## Credits

Author: Samuel Groner  
2005

# bformdec

bformdec -- Decodes an ambisonic B format signal

bformdec

## Description

Decodes an ambisonic B format signal into loudspeaker specific signals.

## Syntax

ao1, ao2 **bformdec** isetup, aw, ax, ay, az [, ar, as, at, au, av [, abk, al, am, a

ao1, ao2, ao3, ao4 **bformdec** isetup, aw, ax, ay, az [, ar, as, at, au, av [, abk

ao1, ao2, ao3, ao4, ao5 **bformdec** isetup, aw, ax, ay, az [, ar, as, at, au, av [

ao1, ao2, ao3, ao4, ao5, ao6, ao7, ao8 **bformdec** isetup, aw, ax, ay, az [, ar, a

## Initialization

*isetup* — loudspeaker setup. There are five supported setups: 1 denotes stereo setup. There must be two output cells with loudspeaker positions (azimuth angle clockwise/altitude angle) assumed to be (330/0, 30/0).

2 denotes quad setup. There must be four output cells. Loudspeaker positions assumed to be (45°/0), (135°/0), (225/0), (315/0).

3 is a 5.1 surround setup. There must be five output cells. LFE channel is not supported. Loudspeaker positions assumed to be (330/0), (30/0), (0/0), (250/0), (110/0).

4 denotes eight loudspeaker circle setup. There must be eight output cells. Loudspeaker positions assumed to be (22.5/0), (67.5/0), (112.5/0), (157.5/0), (202.5/0), (247.5/0), (292.5/0), (337.5/0).

5 means an eight loudspeaker cubic setup. There must be eight output cells. Loudspeaker positions assumed to be (45/0), (45/30), (135/0), (135/30), (225/0), (225/30), (315/0), (315/30).

## Performance

*aw, ax, ay, ...* -- input signal in the B format.

*ao1 .. ao8* — loudspeaker specific output signals.

## Example

```
instr 1
  ; generate pink noise
  anoise pinkish 1000

  ; two full turns
  kalpha line 0, p3, 720
  kbeta = 0

  ; fade ambisonic order from 2nd to 0th during second turn
```

```
kord0 = 1
kord1 linseg 1, p3 / 2, 1, p3 / 2, 0
kord2 linseg 1, p3 / 2, 1, p3 / 2, 0

; generate B format
aw, ax, ay, az, ar, as, at, au, av bformenc anoise, kalpha, kbeta, kord

; decode B format for 8 channel circle loudspeaker setup
a1, a2, a3, a4, a5, a6, a7, a8 bformdec 4, aw, ax, ay, az, ar, as, at, a

; write audio out
out a1, a2, a3, a4, a5, a6, a7, a8
endin
```

## Credits

Author: Samuel Groner  
2005

# binit

`binit` -- PVS tracks to amplitude+frequency conversion.

`binit`

## Description

The `binit` opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by `partials`) and converts it into a equal-bandwidth bin-frame containing amplitude and frequency pairs (PVS\_AMP\_FREQ), suitable for overlap-add resynthesis (such as performed by `pvsynth`) or further PVS streaming phase vocoder signal transformations. For each frequency bin, it will look for a suitable track signal to fill it; if not found, the bin will be empty (0 amplitude). If more than one track fits a certain bin, the one with highest amplitude will be chosen. This means that not all of the input signal is actually 'binned', the operation is lossy. However, in many situations this loss is not perceptually relevant.

## Syntax

```
fsig binit fin, isize
```

## Performance

*fsig* -- output pv stream in PVS\_AMP\_FREQ format

*fin* -- input pv stream in TRACKS format

*isize* -- FFT size of output (N).

## Examples

### Example 47. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
fbins binit fst, 2048 ; convert it back to bins
aout pvsynth fbins ; overlap-add resynthesis
out aout
```

The example above shows partial tracking of an ifd-analysis signal, conversion to bin frames and overlap-add resynthesis.

## Credits

Author: Victor Lazzarini;  
February 2006

# biquad

biquad -- A sweepable general purpose biquadratic digital filter.

biquad

## Description

A sweepable general purpose biquadratic digital filter.

## Syntax

ares **biquad** asig, kb0, kb1, kb2, ka0, ka1, ka2 [, iskip]

## Initialization

*iskip* (optional, default=0) -- if non-zero, initialization will be skipped. Default value 0. (New in Csound version 3.50)

## Performance

*asig* -- input signal

*biquad* is a general purpose biquadratic digital filter of the form:

$$a_0*y(n) + a_1*y[n-1] + a_2*y[n-2] = b_0*x[n] + b_1*x[n-1] + b_2*x[n-2]$$

This filter has the following frequency response:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b_0 + b_1*Z^{-1} + b_2*Z^{-2}}{a_0 + a_1*Z^{-1} + a_2*Z^{-2}}$$

This type of filter is often encountered in digital signal processing literature. It allows six user-defined k-rate coefficients.

## Examples

Here is an example of the biquad opcode. It uses the files *biquad.orc* [examples/biquad.orc] and *biquad.sco* [examples/biquad.sco].

### Example 48. Example of the biquad opcode.

```
/* biquad.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
```

```
; Instrument #1.
instr 1
; Get the values from the score.
idur = p3
iamp = p4
icps = cpspch(p5)
kfco = p6
krez = p7

; Calculate the biquadratic filter's coefficients
kfcon = 2*3.14159265*kfco/sr
kalpha = 1-2*krez*cos(kfcon)*cos(kfcon)+krez*krez*cos(2*kfcon)
kbeta = krez*krez*sin(2*kfcon)-2*krez*cos(kfcon)*sin(kfcon)
kgama = 1+cos(kfcon)
kml = kalpha*kgama+kbeta*sin(kfcon)
km2 = kalpha*kgama-kbeta*sin(kfcon)
kden = sqrt(kml*kml+km2*km2)
kb0 = 1.5*(kalpha*kalpha+kbeta*kbeta)/kden
kb1 = kb0
kb2 = 0
ka0 = 1
ka1 = -2*krez*cos(kfcon)
ka2 = krez*krez

; Generate an input signal.
axn vco 1, icps, 1

; Filter the input signal.
ayn biquad axn, kb0, kb1, kb2, ka0, ka1, ka2
outs ayn*iamp/2, ayn*iamp/2
endin
/* biquad.orc */

/* biquad.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

;      Sta  Dur  Amp    Pitch Fco    Rez
i 1  0.0  1.0  20000  6.00  1000  .8
i 1  1.0  1.0  20000  6.03  2000  .95
e
/* biquad.sco */
```

## See Also

*biquada, moogvcf, rezzy*

## Credits

Author: Hans Mikelson  
October 1998

New in Csound version 3.49

# biquada

**biquada** -- A sweepable general purpose biquadratic digital filter with a-rate parameters.

**biquada**

## Description

A sweepable general purpose biquadratic digital filter.

## Syntax

**ares biquada** *asig*, *ab0*, *ab1*, *ab2*, *aa0*, *aa1*, *aa2* [, *iskip*]

## Initialization

*iskip* (optional, default=0) -- if non-zero, initialization will be skipped. Default value 0. (New in Csound version 3.50)

## Performance

*asig* -- input signal

*biquada* is a general purpose biquadratic digital filter of the form:

$$a_0*y(n) + a_1*y[n-1] + a_2*y[n-2] = b_0*x[n] + b_1*x[n-1] + b_2*x[n-2]$$

This filter has the following frequency response:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b_0 + b_1*Z^{-1} + b_2*Z^{-2}}{a_0 + a_1*Z^{-1} + a_2*Z^{-2}}$$

This type of filter is often encountered in digital signal processing literature. It allows six user-defined a-rate coefficients.

## See Also

*biquad*

## Credits

Author: Hans Mikelson  
October 1998

New in Csound version 3.49

# birnd

birnd -- Returns a random number in a bi-polar range.

birnd

## Description

Returns a random number in a bi-polar range.

## Syntax

**birnd**(x) (init- or control-rate only)

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

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.

## Examples

Here is an example of the birnd opcode. It uses the files *birnd.orc* [examples/birnd.orc] and *birnd.sco* [examples/birnd.sco].

### Example 49. Example of the birnd opcode.

```
/* birnd.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number from -1 to 1.
  il = birnd(1)
  print il
endin
/* birnd.orc */
```

```
/* birnd.sco */
; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #1 for one second.
i 1 1 1
e
/* birnd.sco */
```



Its output should include lines like:

```
instr 1:  i1 = 0.947  
instr 1:  i1 = -0.721
```

## See Also

*rnd*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachussetts  
1997

Example written by Kevin Conder.

# bqrez

bqrez -- A second-order multi-mode filter.

bqrez

## Description

A second-order multi-mode filter.

## Syntax

```
ares bqrez asig, xfco, xres [, imode]  
      [, iskip]
```

## Initialization

*imode* (optional, default=0) -- The mode of the filter. Choose from one of the following:

- 0 = low-pass (default)
- 1 = high-pass
- 2 = band-pass
- 3 = band-reject
- 4 = all-pass

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*ares* -- output audio signal.

*asig* -- input audio signal.

*xfco* -- filter cut-off frequency in Hz. May be i-time, k-rate, a-rate.

*xres* -- amount of resonance. Values of 1 to 100 are typical. Resonance should be one or greater. A value of 100 gives a 20dB gain at the cutoff frequency. May be i-time, k-rate, a-rate.

All filter modes can be frequency modulated as well as the resonance can also be frequency modulated.

*bqrez* is a resonant low-pass filter created using the Laplace s-domain equations for low-pass, high-pass, and band-pass filters normalized to a frequency. The bi-linear transform was used which contains a frequency transform constant from s-domain to z-domain to exactly match the frequencies together. A lot of trigonometric identities were used to simplify the calculation. It is very stable across the working frequency range up to the Nyquist frequency.

## Examples

Here is an example of the *bqrez* opcode. It uses the files *bqrez.orc* [examples/bqrez.orc] and *bqrez.sco* [examples/bqrez.sco].

**Example 50. Example of the bqrez opcode borrowed from the “rezzy” opcode in Kevin Conder's manual.**

```
/* bqrez.orc */
/* Written by Matt Gerassimof from example by Kevin Conder */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Use a nice sawtooth waveform.
asig vco 16000, 220, 1

; Vary the filter-cutoff frequency from .2 to 2 KHz.
kfco line 200, p3, 2000

; Set the resonance amount.
kres init 0.99

a1 bqrez asig, kfco, kres

out a1
endin
/* bqrez.orc */

/* bqrez.sco */
/* Written by Matt Gerassimof from example by Kevin Conder */
; Table #1, a sine wave for the vco opcode.
f 1 0 16384 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
e
/* bqrez.sco */
```

## See Also

*biquad, moogvcf, rezzy*

## Credits

Author: Matt Gerassimoff  
New in version 4.32  
Written in November 2002.

# butbp

butbp -- Same as the butterbp opcode.

butbp

## Description

Same as the *butterbp* opcode.

## Syntax

ares **butbp** asig, kfreq, kband [, iskip]

## butbr

butbr -- Same as the butterbr opcode.

butbr

## Description

Same as the *butterbr* opcode.

## Syntax

ares **butbr** asig, kfreq, kband [, iskip]

# buthp

buthp -- Same as the butterhp opcode.

buthp

## Description

Same as the *butterhp* opcode.

## Syntax

ares **buthp** asig, kfreq [, iskip]

# butlp

butlp -- Same as the butterlp opcode.

butlp

## Description

Same as the *butterlp* opcode.

## Syntax

ares **butlp** asig, kfreq [, iskip]

# butterbp

butterbp -- A band-pass Butterworth filter.

butterbp

## Description

Implementation of a second-order band-pass Butterworth filter. This opcode can also be written as *butbp*.

## Syntax

ares **butterbp** asig, kfreq, kband [, iskip]

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## 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.

*kband* -- Bandwidth of the bandpass and bandreject filters.

## Examples

Here is an example of the butterbp opcode. It uses the files *butterbp.orc* [examples/butterbp.orc] and *butterbp.sco* [examples/butterbp.sco].

### Example 51. Example of the butterbp opcode.

```
/* butterbp.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1 - an unfiltered noise waveform.
instr 1
  ; White noise signal
  asig rand 22050

  out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
```



```
; White noise signal
asig rand 22050

; Filter it, passing only 1950 to 2050 Hz.
abp butterbp asig, 2000, 100

out abp
endin
/* butterbp.orc */

/* butterbp.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* butterbp.sco */
```

## See Also

*butterbr, butterhp, butterlp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# butterbr

butterbr -- A band-reject Butterworth filter.

butterbr

## Description

Implementation of a second-order band-reject Butterworth filter. This opcode can also be written as *butbr*.

## Syntax

ares **butterbr** asig, kfreq, kband [, iskip]

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## 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.

*kband* -- Bandwidth of the bandpass and bandreject filters.

## Examples

Here is an example of the butterbr opcode. It uses the files *butterbr.orc* [examples/butterbr.orc] and *butterbr.sco* [examples/butterbr.sco].

### Example 52. Example of the butterbr opcode.

```
/* butterbr.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1 - an unfiltered noise waveform.
instr 1
  ; White noise signal
  asig rand 22050

  out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
```

```
; White noise signal
asig rand 22050

; Filter it, cutting 2000 to 6000 Hz.
abr butterbr asig, 4000, 2000

out abr
endin
/* butterbr.orc */

/* butterbr.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* butterbr.sco */
```

## See Also

*butterbp, butterhp, butterlp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# butterhp

butterhp -- A high-pass Butterworth filter.

butterhp

## Description

Implementation of second-order high-pass Butterworth filter. This opcode can also be written as *buthp*.

## Syntax

ares **butterhp** asig, kfreq [, iskip]

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## 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.

## Examples

Here is an example of the butterhp opcode. It uses the files *butterhp.orc* [examples/butterhp.orc] and *butterhp.sco* [examples/butterhp.sco].

### Example 53. Example of the butterhp opcode.

```
/* butterhp.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1 - an unfiltered noise waveform.
instr 1
  ; White noise signal
  asig rand 22050

  out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
  ; White noise signal
  asig rand 22050
```

```
    ; Filter it, passing frequencies above 250 Hz.
    ahp butterhp asig, 250

    out ahp
endin
/* butterhp.orc */
```

```
/* butterhp.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* butterhp.sco */
```

## See Also

*butterbp, butterbr, butterlp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# butterlp

butterlp -- A low-pass Butterworth filter.

butterlp

## Description

Implementation of a second-order low-pass Butterworth filter. This opcode can also be written as *butlp*.

## Syntax

ares **butterlp** asig, kfreq [, iskip]

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

## 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.

## Examples

Here is an example of the butterlp opcode. It uses the files *butterlp.orc* [examples/butterlp.orc] and *butterlp.sco* [examples/butterlp.sco].

### Example 54. Example of the butterlp opcode.

```
/* butterlp.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1 - an unfiltered noise waveform.
instr 1
  ; White noise signal
  asig rand 22050

  out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
  ; White noise signal
  asig rand 22050
```

```
    ; Filter it, cutting frequencies above 1 KHz.
    alp butterlp asig, 1000

    out alp
endin
/* butterlp.orc */
```

```
/* butterlp.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* butterlp.sco */
```

## See Also

*butterbp, butterbr, butterhp*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# button

button -- Sense on-screen controls.

button

## Description

Sense on-screen controls. Requires Winsound or TCL/TK.

## Syntax

kres **button** knum

## Performance

*kres* -- value of the button control. If the button has been pushed since the last k-period, then return 1, otherwise return 0.

*knum* -- the number of the button. If it does not exist, it is made on-screen at initialization.

## See Also

*checkbox*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
September 2000

New in Csound version 4.08



# **buzz**

**buzz** -- Output is a set of harmonically related sine partials.

**buzz**

## **Description**

Output is a set of harmonically related sine partials.

## **Syntax**

**ares** **buzz** *xamp*, *xcps*, *knh*, *ifn* [, *iphs*]

## **Initialization**

*ifn* -- table number of a stored function containing a sine wave. A large table of at least 8192 points is recommended.

*iphs* (optional, default=0) -- 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**

*xamp* -- amplitude

*xcps* -- frequency in cycles per second

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.

**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* = *kmul* = 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.  $knh = \text{int}(sr / 2 / \text{fundamental freq.})$ , the result is a real pulse train of amplitude *xamp*.)

Although *knh* may be varied during performance, its internal value is necessarily integer and may cause “pops” due to discontinuities in the output. **buzz** can be amplitude- and/or frequency-modulated by either control or audio signals.

N.B. This unit has its analog in **GENII**, 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.

## **Examples**

Here is an example of the **buzz** opcode. It uses the files *buzz.orc* [examples/buzz.orc] and *buzz.sco* [examples/buzz.sco].

### **Example 55. Example of the buzz opcode.**

```
/* buzz.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 20000
  kcps = 440
  knh = 3
  ifn = 1

  al buzz kamp, kcps, knh, ifn
  out al
endin
/* buzz.orc */
```

```
/* buzz.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* buzz.sco */
```

## See Also

*gbuzz*

## Credits

September 2003. Thanks to Kanata Motohashi for correcting the mentions of the *kmul* parameter.

Example written by Kevin Conder.

# cabasa

*cabasa* -- Semi-physical model of a cabasa sound.

*cabasa*

## Description

*cabasa* is a semi-physical model of a cabasa sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **cabasa** *iamp*, *idettack* [, *inum*] [, *idamp*] [, *imaxshake*]

## 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

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 512.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.997 which means that the default value of *idamp* is -0.5. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the cabasa opcode. It uses the files *cabasa.orc* [examples/cabasa.orc] and *cabasa.sco* [examples/cabasa.sco].

### Example 56. Example of the cabasa opcode.

```
/* cabasa.orc */
;orchestra -----

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

    instr 01
a1      cabasa p4, 0.01
        out a1
        endin
/* cabasa.orc */
;an example of a cabasa
```

```
/* cabasa.sco */
;score -----

      i1 0 1 26000
      e
/* cabasa.sco */
```

## See Also

*crunch, sandpaper, sekere, stix*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# cauchy

cauchy -- Cauchy distribution random number generator.

cauchy

## Description

Cauchy distribution random number generator. This is an x-class noise generator.

## Syntax

ares **cauchy** kalpha

ires **cauchy** kalpha

kres **cauchy** kalpha

## Performance

*kalpha* -- controls the spread from zero (big *kalpha* = big spread). Outputs both positive and negative numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the cauchy opcode. It uses the files *cauchy.orc* [examples/cauchy.orc] and *cauchy.sco* [examples/cauchy.sco].

### Example 57. Example of the cauchy opcode.

```
/* cauchy.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number, spread from 10.
  ; kalpha = 10

  i1 cauchy 10

  print i1
endin
```

```
/* cauchy.orc */

/* cauchy.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cauchy.sco */
```

Its output should include lines like:

```
instr 1:  i1 = -0.106
```

## See Also

*seed, betarand, bexprnd, exprand, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# ceil

ceil -- Returns the smallest integer not less than  $x$

ceil

## Description

Returns the smallest integer not less than  $x$

## Syntax

**ceil**( $x$ ) (init-, control-, or audio-rate arg allowed)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Author: Istvan Varga  
New in Csound 5  
2005

# cent

`cent` -- Calculates a factor to raise/lower a frequency by a given amount of cents.

`cent`

## Description

Calculates a factor to raise/lower a frequency by a given amount of cents.

## Syntax

`cent` (*x*)

This function works at a-rate, i-rate, and k-rate.

## Initialization

*x* -- a value expressed in cents.

## Performance

The value returned by the `cent` function is a factor. You can multiply a frequency by this factor to raise/lower it by the given amount of cents.

## Examples

Here is an example of the `cent` opcode. It uses the files `cent.orc` [examples/cent.orc] and `cent.sco` [examples/cent.sco].

### Example 58. Example of the `cent` opcode.

```
/* cent.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; The root note is A above middle-C (440 Hz)
  iroot = 440

  ; Raise the root note by 300 cents to C.
  icents = 300

  ; Calculate the new note.
  ifactor = cent(icents)
  inew = iroot * ifactor

  ; Print out of all of the values.
  print iroot
  print ifactor
  print inew
endin
/* cent.orc */
```



```
/* cent.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* cent.sco */
```

Its output should include lines like:

```
instr 1:  iroot = 440.000  
instr 1:  ifactor = 1.189  
instr 1:  inew = 523.229
```

## See Also

*db, octave, semitone*

## Credits

Example written by Kevin Conder.

New in version 4.16

## cggoto

cggoto -- Conditionally transfer control on every pass.

cggoto

## Description

Transfer control to *label* on every pass. (Combination of *cigoto* and *ckgoto*)

## Syntax

**cggoto** condition, label

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

## Examples

Here is an example of the cggoto opcode. It uses the files *cggoto.orc* [examples/cggoto.orc] and *cggoto.sco* [examples/cggoto.sco].

### Example 59. Example of the cggoto opcode.

```
/* cggoto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il = 1

  ; If il is equal to one, play a high note.
  ; Otherwise play a low note.
  cggoto (il == 1), highnote

lownote:
  al oscil 10000, 220, 1
  goto playit

highnote:
  al oscil 10000, 440, 1
  goto playit

playit:
  out al
endin
/* cggoto.orc */
```

```
/* cggoto.sco */
; Table #1: a simple sine wave.
f 1 0 32768 10 1
```

```
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* cggoto.sco */
```

## See Also

*cigoto, ckgoto, cngoto, if, igoto, kgoto, tigoto, timeout*

## Credits

Added a note by Jim Aikin.

Example written by Kevin Conder.

# chanctrl

chanctrl -- Get the current value of a MIDI channel controller.

chanctrl

## Description

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

## Syntax

ival **chanctrl** ichnl, ictlno [, ilow] [, ihigh]

kval **chanctrl** ichnl, ictlno [, ilow] [, ihigh]

## Initialization

*ichnl* -- the MIDI channel (1-16).

*ictlno* -- the MIDI controller number (0-127).

*ilow, ihigh* -- low and high ranges for mapping

## Credits

Author: Mike Berry  
Mills College  
May, 1997

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# changed

changed -- k-rate signal change detector.

changed

## Description

This opcode outputs a trigger signal that informs when any one of its k-rate arguments has changed. Useful with valuator widgets or MIDI controllers.

## Syntax

ktrig **changed** kvar1 [, kvar2,..., kvarN]

## Performance

*ktrig* - Outputs a value of 1 when any of the k-rate signals has changed, otherwise outputs 0.

*kvar1* [, *kvar2*,..., *kvarN*] - k-rate variables to watch for changes.

## Examples

Here is an example of the changed opcode. It uses the file *changed.csd* [examples/changed.csd].

### Example 60. Example of the changed opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr      =      44100
kr      =      100
ksmps  =      441
nchnls =      2

      instr      1
ksig oscil 2,0.5,1
kint = int(ksig)
ktrig changed kint
printk 0.2, kint
printk2 ktrig
      endin

</CsInstruments>
<CsScore>
f 1 0 1024 10 1
i 1 0 20

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# chani

chani -- Reads data from the software bus

chani

## Description

Reads data from a channel of the inward software bus.

## Syntax

```
kval chani kchan
```

```
aval chani kchan
```

## Initialization

## Performance

*kchan* -- a positive integer that indicates which channel of the software bus to read

Note that the inward and outward software busses are independent, and are not mixer buses. The last value remains until a new value is written. There is no imposed limit to the number of busses but they use memory so small numbers are to be preferred.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values

```
sr = 44100
ksmps = 100
nchnls = 1

instr 1
  kc chani 1
  a1 oscil p4, p5, 100
  a2 lowpass2 a1, kc, 200
  out a2
endin
```

## Credits

Author: John ffitch  
2005

# chano

chano -- Send data to the outwards software bus

chano

## Description

Send data to a channel of the outward software bus.

## Syntax

**chano** kval, kchan

**chano** aval, kchan

## Initialization

## Performance

*xval* --- value to transmit

*kchan* -- a positive integer that indicates which channel of the software bus to write

Note that the inward and outward software busses are independent, and are not mixer buses. The last value remains until a new value is written. There is no imposed limit to the number of busses but they use memory so small numbers are to be preferred.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values

```
sr = 44100
ksmps = 100
nchnls = 1

instr    1
  a1    oscil    p4, p5, 100
        chano    1, a1
endin
```

## Credits

Author: John ffitch  
2005



# checkbox

checkbox -- Sense on-screen controls.

checkbox

## Description

Sense on-screen controls. Requires Winsound or TCL/TK.

## Syntax

kres **checkbox** knum

## Performance

*kres* -- value of the checkbox control. If the checkbox is set (pushed) then return 1, if not, return 0.

*knun* -- the number of the checkbox. If it does not exist, it is made on-screen at initialization.

## Examples

Here is a simple example of the checkbox opcode. It uses the files *checkbox.orc* [examples/checkbox.orc] and *checkbox.sco* [examples/checkbox.sco].

### Example 61. Simple example of the checkbox opcode.

```
/* checkbox.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1
; Get the value from the checkbox.
k1 checkbox 1

; If the checkbox is selected then k2=440, otherwise k2=880.
k2 = (k1 == 0 ? 440 : 880)

a1 oscil 10000, k2, 1
out a1
endin
/* checkbox.orc */
```

```
/* checkbox.sco */
; Just generate a nice, ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* checkbox.sco */
```

## See Also

*button*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
September, 2000

Example written by Kevin Conder.

New in Csound version 4.08

# chn

**chn** -- Declare a channel of the named software bus.

chn

## Description

Declare a channel of the named software bus, with setting optional parameters in the case of a control channel. If the channel does not exist yet, it is created, with an initial value of zero or empty string. Otherwise, the type (control, audio, or string) of the existing channel must match the declaration, or an init error occurs. The input/output mode of an existing channel is updated so that it becomes the bitwise OR of the previous and the newly specified value.

## Syntax

**chn\_k** Sname, imode[, itype, idflt, imin, imax]

**chn\_a** Sname, imode

**chn\_s** Sname, imode

## Initialization

*imode* -- sum of at least one of 1 for input and 2 for output.

*itype* (optional, defaults to 0) -- channel subtype for control channels only. Possible values are:

- 0: default/unspecified (idflt, imin, and imax are ignored)
- 1: integer values only
- 2: linear scale
- 3: exponential scale

*idflt* (optional, defaults to 0) -- default value, for control channels with non-zero itype only. Must be greater than or equal to imin, and less than or equal to imax.

*imin* (optional, defaults to 0) -- minimum value, for control channels with non-zero itype only. Must be non-zero for exponential scale (itype = 3).

*imax* (optional, defaults to 0) -- maximum value, for control channels with non-zero itype only. Must be greater than imin. In the case of exponential scale, it should also match the sign of imin.

## Notes

The channel parameters (imode, itype, idflt, imin, and imax) are only hints for the host application or external software accessing the bus through the API, and do not actually restrict reading from or writing to the channel in any way. Also, the initial value of a newly created control channel is zero, regardless of the setting of idflt.

For communication with external software, using `chnexport` may be preferred, as it allows direct access to orchestra variables exported as channels of the bus, eliminating the need for using `chnset` and `chnget` to send or receive data.

## Performance

**chn\_k**, **chn\_a**, and **chn\_S** declare a control, audio, or string channel, respectively.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values.

```
sr = 44100
ksmps = 100
nchnls = 1

chn_k "cutoff", 1, 3, 1000, 500, 2000

instr    1
  kc     chnget    "cutoff"
  a1     oscil     p4, p5, 100
  a2     lowpass2  a1, kc, 200
  out
endin
```

## Credits

Author: Istvan Varga  
2005

# chnclear

chnclear -- Clears an audio output channel of the named software bus.

chnclear

## Description

Clears an audio channel of the named software bus to zero. Implies declaring the channel with imode=2 (see also chn\_a).

## Syntax

**chnclear** Sname

## Initialization

*Sname* -- a string that indicates which named channel of the software bus to write.

## Credits

Author: Istvan Varga  
2006

# chnexport

chnexport -- Export a global variable as a channel of the bus.

chnexport

## Description

Export a global variable as a channel of the bus; the channel should not already exist, otherwise an init error occurs. This opcode is normally called from the orchestra header, and allows the host application to read or write orchestra variables directly, without having to use chnget or chnset to copy data.

## Syntax

```
gival chnexport Sname, imode[, itype, idflt, imin, imax]
```

```
gkval chnexport Sname, imode[, itype, idflt, imin, imax]
```

```
gaval chnexport Sname, imode
```

```
gSval chnexport Sname, imode
```

## Initialization

*imode* -- sum of at least one of 1 for input and 2 for output.

*itype* (optional, defaults to 0) -- channel subtype for control channels only. Possible values are:

- 0: default/unspecified (*idflt*, *imin*, and *imax* are ignored)
- 1: integer values only
- 2: linear scale
- 3: exponential scale

*idflt* (optional, defaults to 0) -- default value, for control channels with non-zero *itype* only. Must be greater than or equal to *imin*, and less than or equal to *imax*.

*imin* (optional, defaults to 0) -- minimum value, for control channels with non-zero *itype* only. Must be non-zero for exponential scale (*itype* = 3).

*imax* (optional, defaults to 0) -- maximum value, for control channels with non-zero *itype* only. Must be greater than *imin*. In the case of exponential scale, it should also match the sign of *imin*.

## Notes

The channel parameters (*imode*, *itype*, *idflt*, *imin*, and *imax*) are only hints for the host application or external software accessing the bus through the API, and do not actually restrict reading from or writing to the channel in any way.

While the global variable is used as output argument, **chnexport** does not actually change it, and always runs at i-time only. If the variable is not previously declared, it is created by **Csound** with an

initial value of zero or empty string.

## Performance

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values.

```
sr = 44100
ksmps = 100
nchnls = 1

gkc init 1000 ; set default value
gkc chnexport "cutoff", 1, 3, i(gkc), 500, 2000

instr 1
  a1 oscil p4, p5, 100
  a2 lowpass2 a1, gkc, 200
  out a2
endin
```

## Credits

Author: Istvan Varga  
2005

# chnget

chnget -- Reads data from the software bus.

chnget

## Description

Reads data from a channel of the inward named software bus. Implies declaring the channel with imode=1 (see also chn\_k, chn\_a, and chn\_S).

## Syntax

ival **chnget** Sname

kval **chnget** Sname

aval **chnget** Sname

Sval **chnget** Sname

## Initialization

*Sname* -- a string that identifies a channel of the named software bus to read.

## Performance

*ival* -- the control value read at i-time.

*kval* -- the control value read at performance time.

*aval* -- the audio signal read at performance time.

*Sval* -- the string value read at i-time.

## Example

The example shows the software bus being used as an asynchronous control signal to select a filter cutoff. It assumes that an external program that has access to the API is feeding the values.

```
sr = 44100
ksmps = 100
nchnls = 1

instr    1
  kc    chnget    "cutoff"
  a1    oscil     p4, p5, 100
  a2    lowpass2  a1, kc, 200
  out                    a2
endin
```



## Credits

Author: Istvan Varga  
2005

# chnmix

chnmix -- Writes audio data to the named software bus, mixing to the previous output.

chnmix

## Description

Adds an audio signal to a channel of the named software bus. Implies declaring the channel with `imode=2` (see also `chn_a`).

## Syntax

**chnmix** *aval*, *Sname*

## Initialization

*Sname* -- a string that indicates which named channel of the software bus to write.

## Performance

*aval* -- the audio signal to write at performance time.

## Credits

Author: Istvan Varga  
2006

# chnparams

chnparams -- Query parameters of a channel.

chnparams

## Description

Query parameters of a channel (if it does not exist, all returned values are zero).

## Syntax

*itype*, *imode*, *ictltype*, *idflt*, *imin*, *imax* **chnparams**

## Initialization

*itype* -- channel data type (1: control, 2: audio, 3: string)

*imode* -- sum of 1 for input and 2 for output

*ictltype* -- special parameter for control channel only; if not available, set to zero.

*idflt* -- special parameter for control channel only; if not available, set to zero.

*imin* -- special parameter for control channel only; if not available, set to zero.

*imax* -- special parameter for control channel only; if not available, set to zero.

## Performance

## Example

## Credits

Author: Istvan Varga  
2005

# chnset

chnset -- Writes data to the named software bus.

chnset

## Description

Write to a channel of the named software bus. Implies declaring the channel with imode=2 (see also chn\_k, chn\_a, and chn\_S).

## Syntax

**chnset** ival, Sname

**chnset** kval, Sname

**chnset** aval, Sname

**chnset** Sval, Sname

## Initialization

*Sname* -- a string that indicates which named channel of the software bus to write.

## Performance

*ival* -- the control value to write at i-time.

*kval* -- the control value to write at performance time.

*aval* -- the audio signal to write at performance time.

*Sval* -- the string value to write at i-time.

## Example

The example shows the software bus being used to write pitch information to a controlling program.

```
sr = 44100
ksmps = 100
nchnls = 1

instr 1
  al in
  kp,ka pitchamdf al
  chnset kp, "pitch"
endin
```

## Credits

Author: Istvan Varga  
2005

# cigoto

cigoto -- Conditionally transfer control during the i-time pass.

cigoto

## Description

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

## Syntax

**cigoto** condition, label

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

## Examples

Here is an example of the cigoto opcode. It uses the files *cigoto.orc* [examples/cigoto.orc] and *cigoto.sco* [examples/cigoto.sco].

### Example 62. Example of the cigoto opcode.

```
/* cigoto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Get the value of the 4th p-field from the score.
iparam = p4

; If iparam is 1 then play the high note.
; If not then play the low note.
cigoto (iparam ==1), highnote
igoto lownote

highnote:
ifreq = 880
goto playit

lownote:
ifreq = 440
goto playit

playit:
; Print the values of iparam and ifreq.
print iparam
print ifreq

a1 oscil 10000, ifreq, 1
out a1
endin
/* cigoto.orc */
```

```
/* cigoto.sco */
; Table #1: a simple sine wave.
f 1 0 32768 10 1

; p4: 1 = high note, anything else = low note
; Play Instrument #1 for one second, a low note.
i 1 0 1 0
; Play a Instrument #1 for one second, a high note.
i 1 1 1 1
e
/* cigoto.sco */
```

Its output should include lines like:

```
instr 1:  iparam = 0.000
instr 1:  ifreq = 440.000
instr 1:  iparam = 1.000
instr 1:  ifreq = 880.000
```

## See Also

*cggoto, ckgoto, cngoto, goto, if, kgoto, rigoto, tigoto, timeout*

## Credits

Added a note by Jim Aikin.

Example written by Kevin Conder.

## ckgoto

ckgoto -- Conditionally transfer control during the p-time passes.

ckgoto

## Description

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

## Syntax

**ckgoto** condition, label

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

## Examples

Here is an example of the ckgoto opcode. It uses the files *ckgoto.orc* [examples/ckgoto.orc] and *ckgoto.sco* [examples/ckgoto.sco].

### Example 63. Example of the ckgoto opcode.

```
/* ckgoto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Change kval linearly from 0 to 2 over
; the period set by the third p-field.
kval line 0, p3, 2

; If kval is greater than or equal to 1 then play the high note.
; If not then play the low note.
ckgoto (kval >= 1), highnote
    kgoto lownote

highnote:
    kfreq = 880
    goto playit

lownote:
    kfreq = 440
    goto playit

playit:
; Print the values of kval and kfreq.
printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

    a1 oscil 10000, kfreq, 1
    out a1
endin
/* ckgoto.orc */
```



```
/* ckgoto.sco */  
; Table: a simple sine wave.  
f 1 0 32768 10 1  
  
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* ckgoto.sco */
```

Its output should include lines like:

```
kval = 0.000000, kfreq = 440.000000  
kval = 0.999732, kfreq = 440.000000  
kval = 1.999639, kfreq = 880.000000
```

## See Also

*cggoto, cigoto, cngoto, goto, if, igoto, tigoto, timeout*

## Credits

Added a note by Jim Aikin.

Example written by Kevin Conder.

# clear

`clear` -- Zeroes a list of audio signals.

`clear`

## Description

*clear* zeroes a list of audio signals.

## Syntax

**clear** *avar1* [, *avar2*] [, *avar3*] [...]

## Performance

*avar1*, *avar2*, *avar3*, ... -- 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).

## Examples

See the *fout* opcode for an example.

## See Also

*vincr*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# clfilt

clfilt -- Implements low-pass and high-pass filters of different styles.

clfilt

## Description

Implements the classical standard analog filter types: low-pass and high-pass. They are implemented with the four classical kinds of filters: Butterworth, Chebyshev Type I, Chebyshev Type II, and Elliptical. The number of poles may be any even number from 2 to 80.

## Syntax

ares **clfilt** asig, kfreq, itype, inpol [, ikind] [, ipbr] [, isba] [, iskip]

## Initialization

*itype* -- 0 for low-pass, 1 for high-pass.

*inpol* -- The number of poles in the filter. It must be an even number from 2 to 80.

*ikind* (optional) -- 0 for Butterworth, 1 for Chebyshev Type I, 2 for Chebyshev Type II, 3 for Elliptical. Defaults to 0 (Butterworth)

*ipbr* (optional) -- The pass-band ripple in dB. Must be greater than 0. It is ignored by Butterworth and Chebyshev Type II. The default is 1 dB.

*isba* (optional) -- The stop-band attenuation in dB. Must be less than 0. It is ignored by Butterworth and Chebyshev Type I. The default is -60 dB.

*iskip* (optional) -- 0 initializes all filter internal states to 0. 1 skips initialization. The default is 0.

## Performance

*asig* -- The input audio signal.

*kfreq* -- The corner frequency for low-pass or high-pass.

## Examples

Here is an example of the clfilt opcode as a low-pass filter. It uses the files *clfilt\_lowpass.orc* [examples/clfilt\_lowpass.orc] and *clfilt\_lowpass.sco* [examples/clfilt\_lowpass.sco].

### Example 64. Example of the clfilt opcode as a low-pass filter.

```
/* clfilt_lowpass.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1 - an unfiltered noise waveform.
instr 1
```

```
    ; White noise signal
    asig rand 22050

    out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
    ; White noise signal
    asig rand 22050

    ; Lowpass filter signal asig with a
    ; 10-pole Butterworth at 500 Hz.
    a1 clfilt asig, 500, 0, 10

    out a1
endin
/* clfilt_lowpass.orc */
```

```
/* clfilt_lowpass.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* clfilt_lowpass.sco */
```

Here is an example of the `clfilt` opcode as a high-pass filter. It uses the files *clfilt\_highpass.orc* [examples/clfilt\_highpass.orc] and *clfilt\_highpass.sco* [examples/clfilt\_highpass.sco].

### **Example 65. Example of the `clfilt` opcode as a high-pass filter.**

```
/* clfilt_highpass.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1 - an unfiltered noise waveform.
instr 1
    ; White noise signal
    asig rand 22050

    out asig
endin

; Instrument #2 - a filtered noise waveform.
instr 2
    ; White noise signal
    asig rand 22050

    ; Highpass filter signal asig with a 6-pole Chebyshev
    ; Type I at 20 Hz with 3 dB of passband ripple.
    a1 clfilt asig, 20, 1, 6, 1, 3
```

```
    out a1
endin
/* clfilt_highpass.orc */

/* clfilt_highpass.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* clfilt_highpass.sco */
```

## Credits

Author: Erik Spjut

New in version 4.20

# clip

clip -- Clips a signal to a predefined limit.

clip

## Description

Clips an a-rate signal to a predefined limit, in a “soft” manner, using one of three methods.

## Syntax

ares **clip** asig, imeth, ilimit [, iarg]

## 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, default=0.5) -- 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:

$$|x| > a: \quad f(x) = \sin(x) * (a + (x-a) / (1 + ((x-a) / (1-a))^2) \quad |x| > 1: f(x) = \sin(x) *$$

This method requires that *asig* be normalized to 1.

The second method (*imeth* = 1) is the sine clip:

$$|x| < limit: f(x) = limit * \sin(\#x / (2 * limit)) \quad f(x) = limit * \sin(x)$$

The third method (*imeth* = 2) 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.

## Examples

Here is an example of the clip opcode. It uses the files *clip.orc* [examples/clip.orc] and *clip.sco* [examples/clip.sco].

### Example 66. Example of the clip opcode.

```
/* clip.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a noisy waveform.
  arnd rand 44100
  ; Clip the noisy waveform's amplitude to 20,000
  a1 clip arnd, 2, 20000

  out a1
endin
/* clip.orc */
```

```
/* clip.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* clip.sco */
```

## Credits

Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK  
August, 2000

New in Csound version 4.07

## clock

clock -- Deprecated.

clock

## Description

Deprecated. Use the *rtclock* opcode instead.



# clockoff

clockoff -- Stops one of a number of internal clocks.

clockoff

## Description

Stops one of a number of internal clocks.

## Syntax

**clockoff** inum

## Initialization

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

## Performance

Between a *clockon* and a *clockoff* opcode, the CPU time used is accumulated in the clock. The precision is machine dependent but is the millisecond range on UNIX and Windows systems. The *readclock* opcode reads the current value of a clock at initialization time.

## Examples

See the *readclock* opcode for an example.

## See Also

*clockon*, *readclock*

## Credits

Author: John fitch  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

New in Csound version 3.56

# clockon

clockon -- Starts one of a number of internal clocks.

clockon

## Description

Starts one of a number of internal clocks.

## Syntax

**clockon** inum

## Initialization

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

## Performance

Between a *clockon* and a *clockoff* opcode, the CPU time used is accumulated in the clock. The precision is machine dependent but is the millisecond range on UNIX and Windows systems. The *readclock* opcode reads the current value of a clock at initialization time.

## Examples

See the *readclock* opcode for an example.

## See Also

*clockoff*, *readclock*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

New in Csound version 3.56

# cngoto

cngoto -- Transfers control on every pass when a condition is not true.

cngoto

## Description

Transfers control on every pass when the condition is *not* true.

## Syntax

**cngoto** condition, label

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

## Examples

Here is an example of the cngoto opcode. It uses the files *cngoto.orc* [examples/cngoto.orc] and *cngoto.sco* [examples/cngoto.sco].

### Example 67. Example of the cngoto opcode.

```
/* cngoto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Change kval linearly from 0 to 2 over
; the period set by the third p-field.
kval line 0, p3, 2

; If kval *is not* greater than or equal to 1 then play
; the high note. Otherwise, play the low note.
cngoto (kval >= 1), highnote
kgoto lownote

highnote:
kfreq = 880
goto playit

lownote:
kfreq = 440
goto playit

playit:
; Print the values of kval and kfreq.
printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

a1 oscil 10000, kfreq, 1
out a1
endin
/* cngoto.orc */
```

```
/* cngoto.sco */  
; Table: a simple sine wave.  
f 1 0 32768 10 1  
  
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* cngoto.sco */
```

Its output should include lines like:

```
kval = 0.000000, kfreq = 880.000000  
kval = 0.999732, kfreq = 880.000000  
kval = 1.999639, kfreq = 440.000000
```

## See Also

*cggoto, cigoto, ckgoto, goto, if, igoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

New in version 4.21

# comb

comb -- Reverberates an input signal with a “colored” frequency response.

comb

## Description

Reverberates an input signal with a “colored” frequency response.

## Syntax

ares **comb** asig, krvt, ilpt [, iskip] [, insmps]

## 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. Delay space is allocated and returned as in *delay*.

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

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*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).

This filter reiterates 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.

## Examples

Here is an example of the comb opcode. It uses the files *comb.orc* [examples/comb.orc] and *comb.sco* [examples/comb.sco].

### Example 68. Example of the comb opcode.

```
/* comb.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the audio mixer.
gamix init 0

; Instrument #1.
```

```
instr 1
; Generate a source signal.
a1 oscili 30000, cpspch(p4), 1
; Output the direct sound.
out a1

; Add the source signal to the audio mixer.
gamix = gamix + a1
endin

; Instrument #99 (highest instr number executed last)
instr 99
krvt = 1.5
ilpt = 0.1

; Comb-filter the mixed signal.
a99 comb gamix, krvt, ilpt
; Output the result.
out a99

; Empty the mixer for the next pass.
gamix = 0
endin
/* comb.orc */

/* comb.sco */
; Table #1, a sine wave.
f 1 0 128 10 1

; p4 = frequency (in a pitch-class)
; Play Instrument #1 for a tenth of a second, p4=7.00
i 1 0 0.1 7.00
; Play Instrument #1 for a tenth of a second, p4=7.02
i 1 1 0.1 7.02
; Play Instrument #1 for a tenth of a second, p4=7.04
i 1 2 0.1 7.04
; Play Instrument #1 for a tenth of a second, p4=7.06
i 1 3 0.1 7.06

; Make sure the comb-filter remains active.
i 99 0 5
e
/* comb.sco */
```

## See Also

*alpass, reverb, valpass, vcomb*

## Credits

Author: William “Pete” Moss (*vcomb* and *valpass*)  
University of Texas at Austin  
Austin, Texas USA  
January 2002

Example written by Kevin Conder.

# compress

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

compress

## Description

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

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

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

## Syntax

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

## Initialization

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

## Performance

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

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

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

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

## Examples

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

## Credits

Written by Barry L. Vercoe for Extended Csound and released in csound5.



# control

`control` -- Configurable slider controls for realtime user input.

`control`

## Description

Configurable slider controls for realtime user input. Requires Winsound or TCL/TK. *control* reads a slider's value.

## Syntax

`kres control knum`

## Performance

*knun* -- number of the slider to be read.

Calling *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*.

## Examples

See the *setctrl* opcode for an example.

## See Also

*setctrl*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
May, 2000

New in Csound version 4.06

## convle

convle -- Same as the convolve opcode.

convle

## Description

Same as the *convolve* opcode.

## Syntax

```
ar1 [, ar2] [, ar3] [, ar4] convle ain, ifilcod [, ichannel]
```

# convolve

convolve -- Convolves a signal and an impulse response.

convolve

## 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 as *convle*.

## Syntax

```
ar1 [, ar2] [, ar3] [, ar4] convolve ain, ifilcod [, ichannel]
```

## 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.

*ichannel* (optional) -- which channel to use from the impulse response data file.

## Performance

*ain* -- input audio signal.

*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.

## Examples

Create frequency domain impulse response file using the *cvanal* utility:

```
csound -Ucvanal 11_44.wav 11_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:

$$\text{duration} = (\text{sample frames}) / (\text{sample rate of soundfile})$$

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.

```
sndinfo 11_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  $\leq IRdur$  (1.379s), so:

$$\begin{aligned} \text{Delay1} &= \text{ceil}(IRdur * kr) / kr \\ &= \text{ceil}(608.14) / 441 \\ &= 609/441 \\ &= 1.38s \end{aligned}$$

Accounting for the initial delay:

Delay2 = 0.06s

$$\begin{aligned} \text{Total delay} &= \text{delay1} - \text{delay2} \\ &= 1.38 - 0.06 \\ &= 1.32s \end{aligned}$$

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,"ll_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
```

## Credits

Author: Greg Sullivan  
1996

## COS

`cos` -- Performs a cosine function.

`cos`

## Description

Returns the cosine of  $x$  ( $x$  in radians).

## Syntax

`cos(x)` (no rate restriction)

## Examples

Here is an example of the `cos` opcode. It uses the files *cos.orc* [examples/cos.orc] and *cos.sco* [examples/cos.sco].

### Example 69. Example of the cos opcode.

```
/* cos.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 25
  il = cos(irad)

  print il
endin
/* cos.orc */

/* cos.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cos.sco */
```

Its output should include lines like this:

```
instr 1:  il = 0.991
```

## See Also

*cosh, cosinv, sin, sinh, sininv, tan, tanh, taninv*

## Credits

Example written by Kevin Conder.

# cosh

cosh -- Performs a hyperbolic cosine function.

cosh

## Description

Returns the hyperbolic cosine of  $x$  ( $x$  in radians).

## Syntax

**cosh**( $x$ ) (no rate restriction)

## Examples

Here is an example of the cosh opcode. It uses the files *cosh.orc* [examples/cosh.orc] and *cosh.sco* [examples/cosh.sco].

### Example 70. Example of the cosh opcode.

```
/* cosh.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 1
  il = cosh(irad)

  print il
endin
/* cosh.orc */

/* cosh.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cosh.sco */
```

Its output should include lines like this:

```
instr 1:  il = 1.543
```

## See Also



*cos, cosinv, sin, sinh, sininv, tan, tanh, taninv*

## Credits

Author: John ffitch

Example written by Kevin Conder.

# cosinv

cosinv -- Performs a arccosine function.

cosinv

## Description

Returns the arccosine of  $x$  ( $x$  in radians).

## Syntax

**cosinv**( $x$ ) (no rate restriction)

## Examples

Here is an example of the cosinv opcode. It uses the files *cosinv.orc* [examples/cosinv.orc] and *cosinv.sco* [examples/cosinv.sco].

### Example 71. Example of the cosinv opcode.

```
/* cosinv.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 0.5
  il = cosinv(irad)

  print il
endin
/* cosinv.orc */

/* cosinv.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cosinv.sco */
```

Its output should include lines like this:

```
instr 1:  il = 1.047
```

## See Also

*cos, cosh, sin, sinh, sininv, tan, tanh, taninv*

## Credits

Author: John ffitch

Example written by Kevin Conder.

# cps2pch

`cps2pch` -- Converts a pitch-class value into cycles-per-second for equal divisions of the octave.

`cps2pch`

## Description

Converts a pitch-class value into cycles-per-second (Hz) for equal divisions of the octave.

## Syntax

`icps cps2pch ipch, iequal`

## 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.



### Note

1. The following are essentially the same

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

2. These are opcodes not functions
3. Negative values of `ipch` are allowed.

## Examples

Here is an example of the `cps2pch` opcode. It uses the files `cps2pch.orc` [examples/cps2pch.orc] and `cps2pch.sco` [examples/cps2pch.sco].

### Example 72. Example of the `cps2pch` opcode.

```
/* cps2pch.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
```

```
; Use a normal twelve-tone scale.
ipch = 8.02
iequal = 12

icps cps2pch ipch, iequal

print icps
endin
/* cps2pch.orc */

/* cps2pch.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cps2pch.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 293.666
```

Here is an example of the `cps2pch` opcode using a table of frequency multipliers. It uses the files `cps2pch_ftable.orc` [examples/cps2pch\_ftable.orc] and `cps2pch_ftable.sco` [examples/cps2pch\_ftable.sco].

### **Example 73. Example of the `cps2pch` opcode using a table of frequency multipliers.**

```
/* cps2pch_ftable.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ipch = 8.02

  ; Use Table #1, a table of frequency multipliers.
  icps cps2pch ipch, -1

  print icps
endin
/* cps2pch_ftable.orc */

/* cps2pch_ftable.sco */
; Table #1: a table of frequency multipliers.
; Creates a 10-note scale of unequal divisions.
f 1 0 16 -2 1 1.1 1.2 1.3 1.4 1.6 1.7 1.8 1.9

; Play Instrument #1 for one second.
```

```
i 1 0 1
e
/* cps2pch_ftable.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 313.951
```

Here is an example of the `cps2pch` opcode using a 19ET scale. It uses the files *cps2pch\_19et.orc* [examples/cps2pch\_19et.orc] and *cps2pch\_19et.sco* [examples/cps2pch\_19et.sco].

### Example 74. Example of the `cps2pch` opcode using a 19ET scale.

```
/* cps2pch_19et.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use 19ET scale.
  ipch = 8.02
  iequal = 19

  icps cps2pch ipch, iequal

  print icps
endin
/* cps2pch_19et.orc */

/* cps2pch_19et.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cps2pch_19et.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 281.429
```

## See Also

*cpspch*, *cpsxpch*

## Credits

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

New in Csound version 3.492

# cpsmidi

cpsmidi -- Get the note number of the current MIDI event, expressed in cycles-per-second.

cpsmidi

## Description

Get the note number of the current MIDI event, expressed in cycles-per-second.

## Syntax

icps **cpsmidi**

## Performance

Get the note number of the current MIDI event, expressed in cycles-per-second units, for local processing.

## Examples

Here is an example of the cpsmidi opcode. It uses the files *cpsmidi.orc* [examples/cpsmidi.orc] and *cpsmidi.sco* [examples/cpsmidi.sco].

### Example 75. Example of the cpsmidi opcode.

```
/* cpsmidi.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il cpsmidi

  print il
endin
/* cpsmidi.orc */

/* cpsmidi.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* cpsmidi.sco */
```

## See Also



*aftouch, ampmidi, cpsmidib, cpstmid, midictrl, notnum, octmidi, octmidib, pchbend, pchmidi, pchmidib, veloc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

## cpsmidib

`cpsmidib` -- Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in cycles-per-second.

`cpsmidib`

## Description

Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in cycles-per-second.

## Syntax

```
icps cpsmidib [irange]
```

```
kcps cpsmidib [irange]
```

## Initialization

*irange* (optional) -- the pitch bend range in semitones.

## Performance

Get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in cycles-per-second units. Available as an i-time value or as a continuous k-rate value.

## Examples

Here is an example of the `cpsmidib` opcode. It uses the files *cpsmidib.orc* [examples/cpsmidib.orc] and *cpsmidib.sco* [examples/cpsmidib.sco].

### Example 76. Example of the `cpsmidib` opcode.

```
/* cpsmidib.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il cpsmidib

  print il
endin
/* cpsmidib.orc */
```

```
/* cpsmidib.sco */
```

```
; Play Instrument #1 for 12 seconds.  
i 1 0 12  
e  
/* cpsmidib.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# cpsoct

`cpsoct --` Converts an octave-point-decimal value to cycles-per-second.

`cpsoct`

## Description

Converts an octave-point-decimal value to cycles-per-second.

## Syntax

`cpsoct (oct) (no rate restriction)`

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

## Performance

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:

**Table 1. Pitch and Frequency Values**

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

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

## Examples

Here is an example of the `cpsoct` opcode. It uses the files *cpsoct.orc* [examples/cpsoct.orc] and *cpsoct.sco* [examples/cpsoct.sco].

### Example 77. Example of the `cpsoct` opcode.

```
/* cpsoct.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Convert an octave-point-decimal value into a
  ; cycles-per-second value.
  ioct = 8.75
  icps = cpsoct(ioct)

  print icps
endin
/* cpsoct.orc */

/* cpsoct.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cpsoct.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 440.000
```

## See Also

*cpspch*, *octcps*, *octpch*, *pchoct*

## Credits

Example written by Kevin Conder.

# cpspch

cpspch -- Converts a pitch-class value to cycles-per-second.

cpspch

## Description

Converts a pitch-class value to cycles-per-second.

## Syntax

**cpspch** (pch) (init- or control-rate args only)

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

## Performance

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:

**Table 2. Pitch and Frequency Values**

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

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

## Examples

Here is an example of the `cpspch` opcode. It uses the files *cpspch.orc* [examples/cpspch.orc] and *cpspch.sco* [examples/cpspch.sco].

### Example 78. Example of the `cpspch` opcode.

```
/* cpspch.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Convert a pitch-class value into a
  ; cycles-per-second value.
  ipch = 8.09
  icps = cpspch(ipch)

  print icps
endin
/* cpspch.orc */

/* cpspch.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cpspch.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 440.000
```

## See Also

*cps2pch*, *cpsoct*, *cpsxpch*, *octcps*, *octpch*, *pchoct*

## Credits

Example written by Kevin Conder.

# cpstmid

cpstmid -- Get a MIDI note number (allows customized micro-tuning scales).

cpstmid

## Description

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

## Syntax

icps **cpstmid** ifn

## Initialization

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

## Performance

Init-rate only

*cpstmid* 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 *GEN02*, 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 interval basefreq basekeymidi tuning ratios (equal temp)
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 interval basefreq basekeymidi tuning-ratios (equal temp)
f1 0 64 -2 24 1.5 440 48 1 1.01 1.02 1.03 ..etc...
```

## Examples



Here is an example of the `cpstmid` opcode. It uses the files *cpstmid.orc* [examples/cpstmid.orc] and *cpstmid.sco* [examples/cpstmid.sco].

### Example 79. Example of the `cpstmid` opcode.

```
/* cpstmid.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Table #1, a normal 12-tone equal temperament scale.
; numgrades = 12 (twelve tones)
; interval = 2 (one octave)
; basefreq = 261.659 (Middle C)
; basekeymidi = 60 (Middle C)
gitemp ftgen 1, 0, 64, -2, 12, 2, 261.659, 60, 1.00, \
          1.059, 1.122, 1.189, 1.260, 1.335, 1.414, \
          1.498, 1.588, 1.682, 1.782, 1.888, 2.000

; Instrument #1.
instr 1
  ; Use Table #1.
  ifn = 1
  il cpstmid ifn

  print il
endin
/* cpstmid.orc */

/* cpstmid.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* cpstmid.sco */
```

## See Also

*cpsmidi*, *GEN02*

## Credits

Author: Gabriel Maldonado  
Italy  
1998

Example written by Kevin Conder.

New in Csound version 3.492

# cpstun

cpstun -- Returns micro-tuning values at k-rate.

cpstun

## Description

Returns micro-tuning values at k-rate.

## Syntax

kcps **cpstun** ktrig, kindex, kfn

## Performance

*kcps* -- Return value in cycles per second.

*ktrig* -- A trigger signal used to trigger the evaluation.

*kindex* -- An integer number denoting an index of scale.

*kfn* -- Function table containing the parameters (numgrades, interval, basefreq, basekeymidi) and the tuning ratios.

These opcodes are similar to cpstmid, but work without necessity of MIDI.

*cpstun* works at k-rate. It allows fully customized micro-tuning scales. It requires a function table number containing the tuning ratios, and some other parameters stored in the function table itself.

*kindex* arguments should be filled with integer numbers expressing the grade of given scale to be converted in cps. In *cpstun*, a new value is evaluated only when *ktrig* contains a non-zero value. The function table *kfn* should be generated by *GEN02* and the first four values stored in this function are parameters that express:

- numgrades -- The number of grades of the micro-tuning scale.
- interval -- The frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etcetera.
- basefreq -- The base frequency of the scale in cycles per second.
- basekey -- The integer index of the scale to which to assign basefreq unmodified.

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

```
;          numgrades    basefreq    tuning-ratios (eq.temp) .....  
;  
f1 0 64 -2 12      2      261      60      1      1.059463 1.12246 1.18920 ..etc...
```

Another example with a 24-grade 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		basekey		
f1	0	64	-2	24	1.5	440	48
						1	1.01 1.02 1.03 ..etc.

## Examples

Here is an example of the `cpstun` opcode. It uses the files `cpstun.orc` [examples/cpstun.orc] and `cpstun.sco` [examples/cpstun.sco].

### Example 80. Example of the `cpstun` opcode.

```
/* cpstun.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Table #1, a normal 12-tone equal temperament scale.
; numgrades = 12 (twelve tones)
; interval = 2 (one octave)
; basefreq = 261.659 (Middle C)
; basekeymidi = 60 (Middle C)
gitemp ftgen 1, 0, 64, -2, 12, 2, 261.659, 60, 1.00, \
              1.059, 1.122, 1.189, 1.260, 1.335, 1.414, \
              1.498, 1.588, 1.682, 1.782, 1.888, 2.000

; Instrument #1.
instr 1
; Set the trigger.
ktrig init 1

; Use Table #1.
kfn init 1

; If the base key (note #60) is C, then 9 notes
; above it (note #60 + 9 = note #69) should be A.
kindex init 69

k1 cpstun ktrig, kindex, kfn

printk2 k1
endin
/* cpstun.orc */

/* cpstun.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cpstun.sco */
```

Its output should include lines like this:

```
i1    440.11044
```

## See Also

*cpstmid*, *cpstuni*, *GEN02*

## Credits

Example written by Kevin Conder.

# cpstuni

cpstuni -- Returns micro-tuning values at init-rate.

cpstuni

## Description

Returns micro-tuning values at init-rate.

## Syntax

icps **cpstuni** index, ifn

## Initialization

*icps* -- Return value in cycles per second.

*index* -- An integer number denoting an index of scale.

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

## Performance

These opcodes are similar to *cpstmid*, but work without necessity of MIDI.

*cpstuni* works at init-rate. It allows fully customized micro-tuning scales. It requires a function table number containing the tuning ratios, and some other parameters stored in the function table itself.

The *index* argument should be filled with integer numbers expressing the grade of given scale to be converted in cps. The function table *ifn* should be generated by *GEN02* and the first four values stored in this function are parameters that express:

- numgrades -- The number of grades of the micro-tuning scale.
- interval -- The frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etcetera.
- basefreq -- The base frequency of the scale in cycles per second.
- basekey -- The integer index of the scale to which to assign basefreq unmodified.

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

```
;          numgrades    basefreq    tuning-ratios (eq.temp) .....  
;          interval     basekey  
f1 0 64 -2 12          2        261    60      1    1.059463 1.12246 1.18920 ..etc...
```

Another example with a 24-grade 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		basekey		
f1	0	64	-2	24	1.5	440	48
						1	1.01 1.02 1.03 ..etc.

## Examples

Here is an example of the `cpstuni` opcode. It uses the files *cpstuni.orc* [examples/cpstuni.orc] and *cpstuni.sco* [examples/cpstuni.sco].

### Example 81. Example of the `cpstuni` opcode.

```
/* cpstuni.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Table #1, a normal 12-tone equal temperament scale.
; numgrades = 12 (twelve tones)
; interval = 2 (one octave)
; basefreq = 261.659 (Middle C)
; basekeymidi = 60 (Middle C)
gitemp ftgen 1, 0, 64, -2, 12, 2, 261.659, 60, 1.00, \
              1.059, 1.122, 1.189, 1.260, 1.335, 1.414, \
              1.498, 1.588, 1.682, 1.782, 1.888, 2.000

; Instrument #1.
instr 1
  ; Use Table #1.
  ifn = 1

  ; If the base key (note #60) is C, then 9 notes
  ; above it (note #60 + 9 = note #69) should be A.
  index = 69

  il cpstuni index, ifn

  print il
endin
/* cpstuni.orc */

/* cpstuni.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cpstuni.sco */
```

Its output should include lines like this:

```
instr 1:  i1 = 440.110
```

## See Also

*cpstmid*, *cpstun*, *GEN02*

## Credits

Written by Gabriel Maldonado.

Example written by Kevin Conder.

# cpsxpch

`cpsxpch` -- Converts a pitch-class value into cycles-per-second (Hz) for equal divisions of any interval.

`cpsxpch`

## Description

Converts a pitch-class value into cycles-per-second (Hz) for equal divisions of any interval. There is a restriction of no more than 100 equal divisions.

## Syntax

`icps cpsxpch ipch, iequal, irepeat, ibase`

## 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

1. The following are essentially the same

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

2. These are opcodes not functions
3. Negative values of *ipch* are allowed, but not negative *irepeat*, *iequal* or *ibase*.

## Examples

Here is an example of the `cpsxpch` opcode. It uses the files *cpsxpch.orc* [examples/cpsxpch.orc] and *cpsxpch.sco* [examples/cpsxpch.sco].

### Example 82. Example of the `cpsxpch` opcode.



```
/* cpsxpch.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a normal twelve-tone scale.
  ipch = 8.02
  iequal = 12
  irepeat = 2
  ibase = 1.02197503906

  icps cpsxpch ipch, iequal, irepeat, ibase

  print icps
endin
/* cpsxpch.orc */

/* cpsxpch.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cpsxpch.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 293.666
```

Here is an example of the cpsxpch opcode using a 10.5 ET scale. It uses the files *cpsxpch\_105et.orc* [examples/cpsxpch\_105et.orc] and *cpsxpch\_105et.sco* [examples/cpsxpch\_105et.sco].

### **Example 83. Example of the cpsxpch opcode using a 10.5 ET scale.**

```
/* cpsxpch_105et.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a 10.5ET scale.
  ipch = 4.02
  iequal = 21
  irepeat = 4
  ibase = 16.35160062496

  icps cpsxpch ipch, iequal, irepeat, ibase

  print icps
endin
```

```
/* cpsxpch_105et.orc */

/* cpsxpch_105et.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cpsxpch_105et.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 4776.824
```

Here is an example of the `cpsxpch` opcode using a Pierce scale centered on middle A. It uses the files `cpsxpch_pierce.orc` [examples/cpsxpch\_pierce.orc] and `cpsxpch_pierce.sco` [examples/cpsxpch\_pierce.sco].

#### **Example 84. Example of the `cpsxpch` opcode using a Pierce scale centered on middle A.**

```
/* cpsxpch_pierce.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a Pierce scale centered on middle A.
  ipch = 2.02
  iequal = 12
  irepeat = 3
  ibase = 261.62561

  icps cpsxpch ipch, iequal, irepeat, ibase

  print icps
endin
/* cpsxpch_pierce.orc */

/* cpsxpch_pierce.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* cpsxpch_pierce.sco */
```

Its output should include lines like this:

```
instr 1:  icps = 2827.762
```

## See Also

*cpspch*, *cps2pch*

## Credits

Author: John fitch  
University of Bath/Codemist Ltd.  
Bath, UK  
1997

New in Csound version 3.492

# cpuprc

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

cpuprc

## Description

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

## Syntax

**cpuprc** *insnum*, *ipercnt*

## Initialization

*insnum* -- instrument number

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

## Performance

*cpuprc* sets the cpu processing-time percent usage of an instrument, in order to avoid buffer under-run in realtime performances, enabling a sort of polyphony threshold. The user must set *ipercnt* value for each instrument to be activated in realtime. 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 realtime polyphony in different computers.

For example, if *ipercnt* is set to 5% for instrument 1, the maximum number of voices that can be allocated in realtime, 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 *ipercnt* = 0.0% (i.e. zero processing time or rather infinite cpu processing-speed). This setting is OK for deferred-time sessions.

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

## Examples

Here is an example of the *cpuprc* opcode. It uses the files *cpuprc.orc* [examples/cpuprc.orc] and *cpuprc.sco* [examples/cpuprc.sco].

### Example 85. Example of the cpuprc opcode.

```
/* cpuprc.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Limit Instrument #1 to 5% of the CPU processing time.
cpuprc 1, 5

; Instrument #1
instr 1
    al oscil 10000, 440, 1
    out al
endin
/* cpuprc.orc */

/* cpuprc.sco */
; Just generate a nice, ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* cpuprc.sco */
```

## See Also

*maxalloc, prealloc*

## Credits

Author: Gabriel Maldonado  
Italy  
July, 1999

Example written by Kevin Conder.

New in Csound version 3.57

## cross2

cross2 -- Cross synthesis using FFT's.

cross2

## Description

This is an implementation of cross synthesis using FFT's.

## Syntax

ares **cross2** ain1, ain2, isize, ioverlap, iwin, kbias

## 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 function table that contains the window to be used in the analysis. One can use the *GEN20* routine to create this window.

## 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

Here is an example of the cross2 opcode. It uses the files *cross2.orc* [examples/cross2.orc], *cross2.sco* [examples/cross2.sco] and *beats.wav* [examples/beats.wav].

### Example 86. Example of the cross2 opcode.

```
/* cross2.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - Play an audio file.
instr 1
  ; Use the "beats.wav" audio file.
  aout soundin "beats.wav"
  out aout
endin

; Instrument #2 - Cross-synthesize!
```

```
instr 2
; Use the "ahh" sound stored in Table #1.
ain1 loscil 30000, 1, 1, 1
; Use the "beats.wav" audio file.
ain2 soundin "beats.wav"

isize = 4096
ioverlap = 2
iwin = 2
kbias init 1

aout cross2 ain1, ain2, isize, ioverlap, iwin, kbias

out aout
endin
/* cross2.orc */

/* cross2.sco */
; Table #1: An audio file.
f 1 0 128 1 "ahh.aiff" 0 4 0
; Table #2: A windowing function.
f 2 0 2048 20 2

; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 2 2
e
/* cross2.sco */
```

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1997

Example written by Kevin Conder.

# crunch

`crunch` -- Semi-physical model of a crunch sound.

`crunch`

## Description

*crunch* is a semi-physical model of a crunch sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **crunch** *iamp*, *idettack* [, *inum*] [, *idamp*] [, *imaxshake*]

## 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

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 7.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.99806 which means that the default value of *idamp* is 0.03. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the crunch opcode. It uses the files *crunch.orc* [examples/crunch.orc] and *crunch.sco* [examples/crunch.sco].

### Example 87. Example of the crunch opcode.

```
/* crunch.orc */
;orchestra -----

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

instr 01
al      crunch p4, 0.01      ;an example of a crunch
      out al
      endin
/* crunch.orc */
```



```
/* crunch.sco */
;score -----

      i1 0 1 26000
      e
/* crunch.sco */
```

## See Also

*cabasa, sandpaper, sekere, stix*

## Credits

Author: Perry Cook, part of the PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds)

Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# ctrl14

ctrl14 -- Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

ctrl14

## Description

Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

```
idest ctrl14 ichan, ictlno1, ictlno2, imin, imax [, ifn]
```

```
kdest ctrl14 ichan, ictlno1, ictlno2, kmin, kmax [, ifn]
```

## Initialization

*idest* -- output signal

*ichan* -- MIDI channel number (1-16)

*ictlno1* -- most-significant byte controller number (0-127)

*ictlno2* -- least-significant byte controller number (0-127)

*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

*ctrl14* (i- and k-rate 14 bit MIDI control) allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range. The minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires two MIDI controllers as input.

*ctrl14* differs from *midic14* because it can be included in score-oriented instruments without Csound crashes. It needs the additional parameter *ichan* containing the MIDI channel of the controller. MIDI channel is the same for all the controllers used in a single *ctrl14* opcode.

## See Also

*ctrl7*, *ctrl21*, *initc7*, *initc14*, *initc21*, *midic7*, *midic14*, *midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# ctrl21

ctrl21 -- Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

ctrl21

## Description

Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

*idest* **ctrl21** *ichan*, *ictlno1*, *ictlno2*, *ictlno3*, *imin*, *imax* [, *ifn*]

*kdest* **ctrl21** *ichan*, *ictlno1*, *ictlno2*, *ictlno3*, *kmin*, *kmax* [, *ifn*]

## Initialization

*idest* -- output signal

*ichan* -- MIDI channel number (1-16)

*ictlno* -- MIDI controller number (0-127)

*ictlno1* -- most-significant byte controller number (0-127)

*ictlno2* -- mid-significant byte controller number (0-127)

*ictlno3* -- least-significant byte controller number (0-127)

*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

*ctrl21* (i- and k-rate 21 bit MIDI control) allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range. Minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires three MIDI controllers as input.

*ctrl21* differs from *midic21* because it can be included in score oriented instruments without Csound crashes. It needs the additional parameter *ichan* containing the MIDI channel of the controller. MIDI channel is the same for all the controllers used in a single *ctrl21* opcode.

## See Also

*ctrl7*, *ctrl14*, *initc7*, *initc14*, *initc21*, *midic7*, *midic14*, *midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

## ctrl7

ctrl7 -- Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

ctrl7

## Description

Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

*idest* **ctrl7** *ichan*, *ictlno*, *imin*, *imax* [, *ifn*]

*kdest* **ctrl7** *ichan*, *ictlno*, *kmin*, *kmax* [, *ifn*]

## Initialization

*idest* -- output signal

*ichan* -- MIDI channel (1-16)

*ictlno* -- MIDI controller number (0-127)

*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

*ctrl7* (i- and k-rate 7 bit MIDI control) allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range. It also allows optional non-interpolated table indexing. Minimum and maximum values can be varied at k-rate.

*ctrl7* differs from *midic7* because it can be included in score-oriented instruments without Csound crashes. It also needs the additional parameter *ichan* containing the MIDI channel of the controller.

## See Also

*ctrl14*, *ctrl21*, *initc7*, *initc14*, *initc21*, *midic7*, *midic14*, *midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# ctrlinit

ctrlinit -- Sets the initial values for a set of MIDI controllers.

ctrlinit

## Description

Sets the initial values for a set of MIDI controllers.

## Syntax

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

## Initialization

*ichnl* -- MIDI channel number (1-16)

*ictlno1*, *ictlno1*, etc. -- MIDI controller numbers (0-127)

*ival1*, *ival2*, etc. -- initial value for corresponding MIDI controller number

## Performance

Sets the initial values for a set of MIDI controllers.

## See Also

*massign*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT, Cambridge, Mass.

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.



# cuserrnd

cuserrnd -- Continuous USER-defined-distribution RaNDom generator.

cuserrnd

## Description

Continuous USER-defined-distribution RaNDom generator.

## Syntax

aout **cuserrnd** kmin, kmax, ktableNum

iout **cuserrnd** imin, imax, itableNum

kout **cuserrnd** kmin, kmax, ktableNum

## Initialization

*imin* -- minimum range limit

*imax* -- maximum range limit

*itableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

## Performance

*ktableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

*kmin* -- minimum range limit

*kmax* -- maximum range limit

*cuserrnd* (continuous user-defined-distribution random generator) generates random values according to a continuous random distribution created by the user. In this case the shape of the distribution histogram can be drawn or generated by any GEN routine. The table containing the shape of such histogram must then be translated to a distribution function by means of GEN40 (see GEN40 for more details). Then such function must be assigned to the XtableNum argument of cuserrnd. The output range can then be rescaled according to the Xmin and Xmax arguments. cuserrnd linearly interpolates between table elements, so it is not recommended for discrete distributions (GEN41 and GEN42).

For a tutorial about random distribution histograms and functions see:

- D. Lorrain. "A panoply of stochastic cannons". In C. Roads, ed. 1989. Music machine. Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## See Also

*duserrnd*, *urd*

## Credits

Author: Gabriel Maldonado

New in Version 4.16

# dam

dam -- A dynamic compressor/expander.

dam

## Description

This opcode dynamically modifies a gain 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.

## Syntax

ares **dam** asig, kthreshold, icompl, icomp2, irtime, iftime

## Initialization

*icompl* -- 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 to be modified

*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).

## Examples

Because the results of the *dam* opcode can be subtle, I recommend looking at them in a graphical audio editor program like *audacity*. *audacity* is available for Linux, Windows, and the MacOS and may be downloaded from <http://audacity.sourceforge.net> [<http://audacity.sourceforge.net/>].

Here is an example of the *dam* opcode. It uses the files *dam.orc* [examples/dam.orc], *dam.sco* [examples/dam.sco], and *beats.wav* [examples/beats.wav].

### Example 88. An example of the dam opcode compressing an audio signal.

```
/* dam.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1, uncompressed signal.
instr 1
  ; Use the "beats.wav" audio file.
  asig soundin "beats.wav"

  out asig
endin

; Instrument #2, compressed signal.
instr 2
  ; Use the "beats.wav" audio file.
  asig soundin "beats.wav"

  ; Compress the audio signal.
  kthreshold init 25000
  icomp1 = 0.5
  icomp2 = 0.763
  irtime = 0.1
  iftime = 0.1
  a1 dam asig, kthreshold, icomp1, icomp2, irtime, iftime

  out a1
endin
/* dam.orc */

/* dam.sco */
; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 2 2
e
/* dam.sco */
```

This example compresses the audio file “beats.wav”. You should hear a drum pattern repeat twice. The second time, the sound should be quieter (compressed) than the first.

Here is another example of the dam opcode. It uses the files *dam\_expanded.orc* [examples/dam\_expanded.orc], *dam\_expanded.sco* [examples/dam\_expanded.sco], and *mary.wav* [examples/mary.wav].

### **Example 89. An example of the dam opcode expanding an audio signal.**

```
/* dam_expanded.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1, normal audio signal.
instr 1
  ; Use the "mary.wav" audio file.
  asig soundin "mary.wav"

  out asig
endin
```

```
; Instrument #2, expanded audio signal.
instr 2
; Use the "mary.wav" audio file.
asig soundin "mary.wav"

; Expand the audio signal.
kthreshold init 7500
icompl = 2.25
icomp2 = 2.25
irtime = 0.1
iftime = 0.6
a1 dam asig, kthreshold, icomp1, icomp2, irtime, iftime

out a1
endin
/* dam_expanded.orc */

/* dam_expanded.sco */
; Play Instrument #1.
i 1 0.0 3.5
; Play Instrument #2.
i 2 3.5 3.5
e
/* dam_expanded.sco */
```

This example expands the audio file “mary.wav”. You should hear a melody repeat twice. The second time, the sound should be louder (expanded) than the first.

## Credits

Author: Marc Resibois  
Belgium  
1997

Examples written by Kevin Conder.

# db

db -- Returns the amplitude equivalent for a given decibel amount.

db

## Description

Returns the amplitude equivalent for a given decibel amount. This opcode is the same as *db*.

## Syntax

**db**(*x*)

This function works at a-rate, i-rate, and k-rate.

## Initialization

*x* -- a value expressed in decibels.

## Performance

Returns the amplitude for a given decibel amount.

## Examples

Here is an example of the db opcode. It uses the files *db.orc* [examples/db.orc] and *db.sco* [examples/db.sco].

### Example 90. Example of the db opcode.

```
/* db.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Calculate the amplitude of 40 decibels.
  idecibels = 40
  iamp = db(idecibels)

  print iamp
endin
/* db.orc */
```

```
/* db.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* db.sco */
```

Its output should include lines like:

```
instr 1:  iamp = 100.000
```

## See Also

*ampdb, cent, octave, semitone*

## Credits

Example written by Kevin Conder.

New in version 4.16

# dbamp

dbamp -- Returns the decibel equivalent of the raw amplitude x.

dbamp

## Description

Returns the decibel equivalent of the raw amplitude x.

## Syntax

**dbamp**(x) (init-rate or control-rate args only)

## Examples

Here is an example of the dbamp opcode. It uses the files *dbamp.orc* [examples/dbamp.orc] and *dbamp.sco* [examples/dbamp.sco].

### Example 91. Example of the dbamp opcode.

```
/* dbamp.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iamp = 30000
  idb = dbamp(iamp)

  print idb
endin
/* dbamp.orc */

/* dbamp.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* dbamp.sco */
```

Its output should include lines like this:

```
instr 1:  idb = 89.542
```

## See Also



*ampdb, ampdbfs, dbfsamp*

## Credits

Example written by Kevin Conder.

# dbfsamp

dbfsamp -- Returns the decibel equivalent of the raw amplitude x, relative to full scale amplitude.

dbfsamp

## Description

Returns the decibel equivalent of the raw amplitude x, relative to full scale amplitude. Full scale is assumed to be 16 bit. New is Csound version 4.10.

## Syntax

**dbfsamp**(x) (init-rate or control-rate args only)

## Examples

Here is an example of the dbfsamp opcode. It uses the files *dbfsamp.orc* [examples/dbfsamp.orc] and *dbfsamp.sco* [examples/dbfsamp.sco].

### Example 92. Example of the dbfsamp opcode.

```
/* dbfsamp.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iamp = 30000
  idb = dbfsamp(iamp)

  print idb
endin
/* dbfsamp.orc */

/* dbfsamp.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* dbfsamp.sco */
```

Its output should include lines like this:

```
instr 1:  idb = -0.767
```

## See Also

*ampdb, ampdbfs, dbamp*

## Credits

Example written by Kevin Conder.

# dcblock

dcblock -- A DC blocking filter.

dcblock

## Description

Implements the DC blocking filter

$$Y[i] = X[i] - X[i-1] + (igain * Y[i-1])$$

Based on work by Perry Cook.

## Syntax

ares **dcblock** ain [, igain]

## Initialization

*igain* -- the gain of the filter, which defaults to 0.99

## Performance

*ain* -- audio signal input

## Examples

Here is an example of the dcblock opcode. It uses the files *dcblock.orc* [examples/dcblock.orc], *dcblock.sco* [examples/dcblock.sco], and *beats.wav* [examples/beats.wav].

### Example 93. Example of the dcblock opcode.

```
/* dcblock.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 -- normal audio signal.
instr 1
  asig soundin "beats.wav"
  out asig
endin

; Instrument #2 -- dcblock-ed audio signal.
instr 2
  asig soundin "beats.wav"

  igain = 0.75
  al dcblock asig, igain
```

```
    out a1
endin
/* dcblock.orc */

/* dcblock.sco */
; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 2 2
e
/* dcblock.sco */
```

## Credits

Author: John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.49

February 2003: Thanks to a note from Anders Andersson, corrected the formula.

# dconv

dconv -- A direct convolution opcode.

dconv

## Description

A direct convolution opcode.

## Syntax

ares **dconv** asig, isize, ifn

## Initialization

*isize* -- the size of the convolution buffer to use. if the buffer size is smaller than the size of ifn, then only the first isize values will be used from the table.

*ifn* -- table number of a stored function containing the impulse response for convolution.

## Performance

Rather than the analysis/resynthesis method of the convolve opcode, *dconv* uses direct convolution to create the result. For small tables it can do this quite efficiently, however larger table require much more time to run. *dconv* does (isize \* ksmpts) multiplies on every k-cycle. Therefore, reverb and delay effects are best done with other opcodes (unless the times are short).

*dconv* was designed to be used with time varying tables to facilitate new realtime filtering capabilities.

## Examples

Here is an example of the dconv opcode. It uses the files *dconv.orc* [examples/dconv.orc] and *dconv.sco* [examples/dconv.sco].

### Example 94. Example of the dconv opcode.

```
/* dconv.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

#define RANDI(A) #kout randi 1, kfq, $A*.001+iseed, 1
               tablew kout, $A, itable#

instr 1
itable init 1
iseed init .6
isize init ftlen(itable)
kfq line 1, p3, 10

$RANDI(0)
$RANDI(1)
$RANDI(2)
```

```
$RANDI(3)
$RANDI(4)
$RANDI(5)
$RANDI(6)
$RANDI(7)
$RANDI(8)
$RANDI(9)
$RANDI(10)
$RANDI(11)
$RANDI(12)
$RANDI(13)
$RANDI(14)
$RANDI(15)

asig      rand      10000, .5, 1
asig      butlp     asig, 5000
asig      dconv     asig, isize, itable

          out       asig *.5
endin
/* dconv.orc */

/* dconv.sco */
f1 0 16 10 1
i1 0 10
e
/* dconv.sco */
```

## Credits

Author: William “Pete” Moss  
2001

New in version 4.12

# delay

delay -- Delays an input signal by some time interval.

delay

## Description

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

## Syntax

ares **delay** asig, idlt [, iskip]

## 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, default=0) -- initial disposition of delay-loop data space (see *reson*). The default value is 0.

## Performance

*asig* -- audio signal

*delay* is a composite of *delayr* and *delayw*, 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.

## Examples

Here is an example of the delay opcode. It uses the files *delay.orc* [examples/delay.orc] and *delay.sco* [examples/delay.sco].

### Example 95. Example of the delay opcode.

```
/* delay.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Instrument #1 -- Delayed beeps.
instr 1
  ; Make a basic sound.
  abeep vco 20000, 440, 1

  ; Delay the beep by .1 seconds.
  idlt = 0.1
  adel delay abeep, idlt

  ; Send the beep to the left speaker and
```



```
    ; the delayed beep to the right speaker.  
    outs abeep, adel  
endin  
/* delay.orc */
```

```
/* delay.sco */  
; Table #1, a sine wave.  
f 1 0 16384 10 1  
  
; Keep the score running for 2 seconds.  
f 0 2  
  
; Play Instrument #1.  
i 1 0.0 0.2  
i 1 0.5 0.2  
e  
/* delay.sco */
```

## See Also

*delayl, delayr, delayw*

## Credits

Example written by Kevin Conder.

# delay1

delay1 -- Delays an input signal by one sample.

delay1

## Description

Delays an input signal by one sample.

## Syntax

```
ares delay1 asig [, iskip]
```

## Initialization

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

## Performance

*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 the *delay* opcode but is more efficient in both time and space. This unit is particularly useful in the fabrication of generalized non-recursive filters.

## See Also

*delay*, *delayr*, *delayw*

# delayk

delayk -- Delays an input signal by some time interval.

delayk

## Description

k-rate delay opcodes

## Syntax

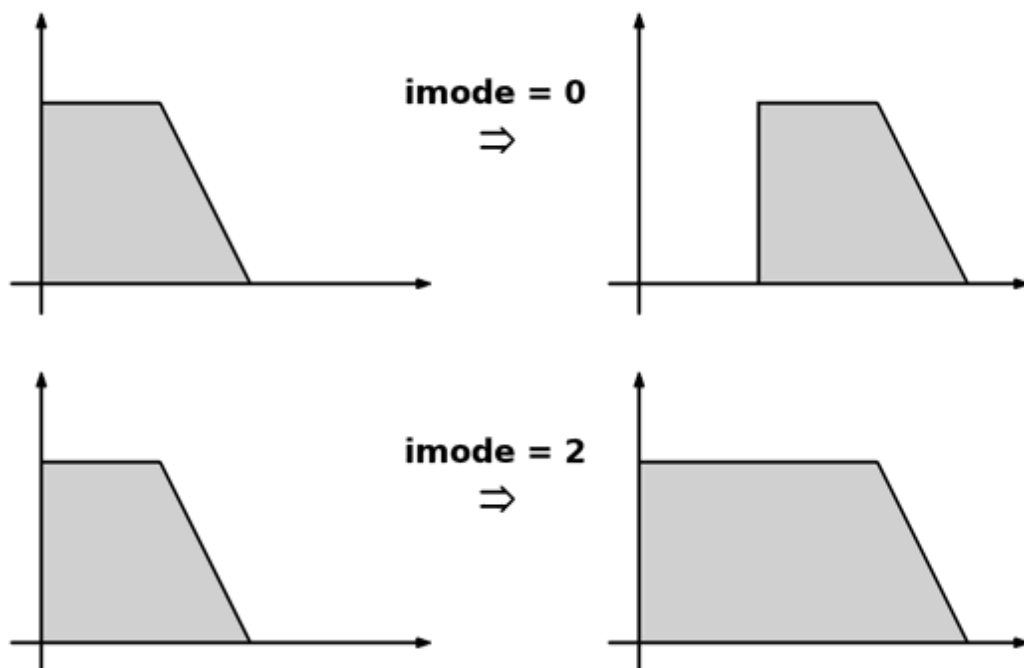
```
kr delayk    ksig, idel[, imode]
```

```
kr vdel_k    ksig, kdel, imdel[, imode]
```

## Initialization

*idel* -- delay time (in seconds) for delayk. It is rounded to the nearest integer multiple of a k-cycle (i.e.  $1/kr$ ).

*imode* -- sum of 1 for skipping initialization (e.g. in tied notes) and 2 for holding the first input value during the initial delay, instead of outputting zero. This is mainly of use when delaying envelopes that do not start at zero.



*imdel* -- maximum delay time for vdel\_k, in seconds.

## Performance

*kr* -- the output signal. Note: neither of the opcodes interpolate the output.

*ksig* -- the input signal.

*kdel* -- delay time (in seconds) for *vdel\_k*. It is rounded to the nearest integer multiple of a k-cycle (i.e.  $1/kr$ ).

## Credits

Istvan Varga.

# delayr

*delayr* -- Reads from an automatically established digital delay line.

*delayr*

## Description

Reads from an automatically established digital delay line.

## Syntax

```
ares delayr idlt [, iskip]
```

## 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, default=0) -- 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.

## Examples

See the example for *delayw*.

## See Also

*delay*, *delay1*, *delayw*

# delayw

delayw -- Writes the audio signal to a digital delay line.

delayw

## Description

Writes the audio signal to a digital delay line.

## Syntax

**delayw** asig

## Performance

*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  $1/kr$ ).

## Examples

Here is an example of the *delayw* opcode. It uses the files *delayw.orc* [examples/delayw.orc] and *delayw.sco* [examples/delayw.sco].

### Example 96. Example of the *delayw* opcode.

```
/* delayw.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Instrument #1 -- Delayed beeps.
instr 1
; Make a basic sound.
abep vco 20000, 440, 1

; Set up a delay line.
idlt = 0.1
adel delayr idlt

; Write the beep to the delay line.
delayw abep

; Send the beep to the left speaker and
; the delayed beep to the right speaker.
outs abep, adel
endin
/* delayw.orc */
```

```
/* delayw.sco */
; Table #1, a sine wave.
```

```
f 1 0 16384 10 1

; Keep the score running for 2 seconds.
f 0 2

; Play Instrument #1.
i 1 0.0 0.2
i 1 0.5 0.2
e
/* delayw.sco */
```

## See Also

*delay, delay1, delayr*

## Credits

Example written by Kevin Conder.

# deltap

deltap -- Taps a delay line at variable offset times.

deltap

## Description

Tap a delay line at variable offset times.

## Syntax

ares **deltap** kdl t

## Performance

*kdl t* -- specifies 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.

*deltap* extracts sound by reading the stored samples directly.

This opcode 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.

This opcode 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 97. deltap 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 98. deltap 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
```

**See Also**

*deltap3, deltapi, deltapn*

# deltap3

deltap3 -- Taps a delay line at variable offset times, uses cubic interpolation.

deltap

## Description

Taps a delay line at variable offset times, uses cubic interpolation.

## Syntax

ares **deltap3** xdl t

## Performance

*xdl t* -- specifies 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 *xdl t* argument in *deltap3* implies that an audio-varying delay is permitted there.

*deltap3* is experimental, and uses cubic interpolation. (New in Csound version 3.50.)

This opcode 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.

This opcode 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 ffitich).

*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 99. deltap 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 100. *deltap* 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
```

**See Also**

*deltap*, *deltapi*, *deltapn*

# deltapi

deltapi -- Taps a delay line at variable offset times, uses interpolation.

deltapi

## Description

Taps a delay line at variable offset times, uses interpolation.

## Syntax

ares **deltapi** xdlrt

## Performance

*xdlrt* -- specifies 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 *xdlrt* argument in *deltapi* implies that an audio-varying delay is permitted there.

*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.

This opcode 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.

This opcode 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 fitch).

*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 101. deltap 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    deltap     atime                   ; move sound source past
          delayw     asource                ; the listener
          out        amove * ampfac
```

**Example 102. *deltap* 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
```

**See Also**

*deltap*, *deltap3*, *deltapn*

# deltapn

deltapn -- Taps a delay line at variable offset times.

deltapn

## Description

Tap a delay line at variable offset times.

## Syntax

ares **deltapn** xnumsamps

## Performance

*xnumsamps* -- specifies the tapped delay time in number of samples. 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.

*deltapn* is identical to *deltapi*, except delay time is specified in number of samples, instead of seconds (Hans Mikelson).

This opcode 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.

This opcode 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 ffitich).

*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 103. deltap 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 104. *deltap* 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
```

**See Also**

*deltap*, *deltap3*, *deltapi*

# deltapx

deltapx -- Read to or write from a delay line with interpolation.

deltapx

## Description

*deltapx* is similar to *deltapi* or *deltap3*. However, it allows higher quality interpolation. This opcode can read from and write to a *delayr/delayw* delay line with interpolation.

## Syntax

*aout* **deltapx** *adel*, *iws*<sub>size</sub>

## Initialization

*iws*<sub>size</sub> -- interpolation window size in samples. Allowed values are integer multiplies of 4 in the range 4 to 1024. *iws*<sub>size</sub> = 4 uses cubic interpolation. Increasing *iws*<sub>size</sub> improves sound quality at the expense of CPU usage, and minimum delay time.

## Performance

*aout* -- Output signal

*adel* -- Delay time in seconds.

```
a1      delayr idlr
        deltapxw a2, adl1, iws1
a3      deltapx adl2, iws2
        deltapxw a4, adl3, iws3
        delayw a5
```

Minimum and maximum delay times:

$idlr \geq 1/kr$	Delay line length
$adl1 \geq (iws1/2)/sr$	Write before read
$adl1 \leq idlr - (1 + iws1/2)/sr$	(allows shorter delays)
$adl2 \geq 1/kr + (iws2/2)/sr$	Read time
$adl2 \leq idlr - (1 + iws2/2)/sr$	
$adl2 \geq adl1 + (iws1 + iws2) / (2*sr)$	
$adl2 \geq 1/kr + adl3 + (iws2 + iws3) / (2*sr)$	
$adl3 \geq (iws3/2)/sr$	Write after read
$adl3 \leq idlr - (1 + iws3/2)/sr$	(allows feedback)





## Note

Window sizes for opcodes other than `deltapx` are: `deltap`, `deltapn`: 1, `deltapi`: 2 (linear), `deltap3`: 4 (cubic)

## Examples

```
a1      phasor 300.0
a1      = a1 - 0.5
a_      delayr 1.0
adel    phasor 4.0
adel    = sin (2.0 * 3.14159265 * adel) * 0.01 + 0.2
        deltapxw a1, adel, 32
adel    phasor 2.0
adel    = sin (2.0 * 3.14159265 * adel) * 0.01 + 0.2
        deltapxw a1, adel, 32
adel    = 0.3
a2      deltapx adel, 32
a1      = 0
        delayw a1

        out a2 * 20000.0
```

## See Also

*deltapxw*

## Credits

Author: Istvan Varga  
August 2001

New in version 4.13

# deltapxw

deltapxw -- Mixes the input signal to a delay line.

deltapxw

## Description

*deltapxw* mixes the input signal to a delay line. This opcode can be mixed with reading units (*deltap*, *deltapn*, *deltapi*, *deltap3*, and *deltapx*) in any order; the actual delay time is the difference of the read and write time. This opcode can read from and write to a *delayr/delayw* delay line with interpolation.

## Syntax

**deltapxw** ain, adel, iwsiz

## Initialization

*iwsiz* -- interpolation window size in samples. Allowed values are integer multiplies of 4 in the range 4 to 1024. *iwsiz* = 4 uses cubic interpolation. Increasing *iwsiz* improves sound quality at the expense of CPU usage, and minimum delay time.

## Performance

*ain* -- Input signal

*adel* -- Delay time in seconds.

```
a1      delayr idlr
        deltapxw a2, adl1, iws1
a3      deltapx adl2, iws2
        deltapxw a4, adl3, iws3
        delayw a5
```

Minimum and maximum delay times:

$idlr \geq 1/kr$	Delay line length
$adl1 \geq (iws1/2)/sr$	Write before read
$adl1 \leq idlr - (1 + iws1/2)/sr$	(allows shorter delays)
$adl2 \geq 1/kr + (iws2/2)/sr$	Read time
$adl2 \leq idlr - (1 + iws2/2)/sr$	
$adl2 \geq adl1 + (iws1 + iws2) / (2*sr)$	
$adl2 \geq 1/kr + adl3 + (iws2 + iws3) / (2*sr)$	
$adl3 \geq (iws3/2)/sr$	Write after read
$adl3 \leq idlr - (1 + iws3/2)/sr$	(allows feedback)



## Note

Window sizes for opcodes other than `deltapx` are: `deltap`, `deltapn`: 1, `deltapi`: 2 (linear), `deltap3`: 4 (cubic)

## Examples

```
a1      phasor 300.0
a1      = a1 - 0.5
a_      delayr 1.0
adel    phasor 4.0
adel    = sin (2.0 * 3.14159265 * adel) * 0.01 + 0.2
        deltapxw a1, adel, 32
adel    phasor 2.0
adel    = sin (2.0 * 3.14159265 * adel) * 0.01 + 0.2
        deltapxw a1, adel, 32
        = 0.3
a2      deltapx adel, 32
a1      = 0
        delayw a1

        out a2 * 20000.0
```

## See Also

*deltapx*

## Credits

Author: Istvan Varga  
August 2001

New in version 4.13

# denorm

denorm -- Mixes low level noise to a list of a-rate signals

denorm

## Description

Mixes low level (~1e-20 for floats, and ~1e-56 for doubles) noise to a list of a-rate signals. Can be used before IIR filters and reverbs to avoid denormalized numbers which may otherwise result in significantly increased CPU usage.

## Syntax

**denorm** *a1[ , a2[ , a3[ , ... ]]*

## Performance

*a1[ , a2[ , a3[ , ... ]]* -- signals to mix noise with

## Credits

Author: Istvan Varga  
2005

# diff

diff -- Modify a signal by differentiation.

diff

## Description

Modify a signal by differentiation.

## Syntax

ares **diff** asig [, iskip]

kres **diff** ksig [, iskip]

## Initialization

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

## Performance

*integ* and *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.

## Examples

Here is an example of the diff opcode. It uses the files *diff.orc* [examples/diff.orc] and *diff.sco* [examples/diff.sco].

### Example 105. Example of the diff opcode.

```
/* diff.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 -- a normal instrument.
instr 1
  ; Generate a band-limited pulse train.
  asrc buzz 20000, 440, 20, 1

  out asrc
endin

; Instrument #2 -- a differentiated instrument.
instr 2
  ; Generate a band-limited pulse train.
  asrc buzz 20000, 440, 20, 1
```

```
; Emphasize the highs.
a1 diff asrc

out a1
endin
/* diff.orc */

/* diff.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 1 1
e
/* diff.sco */
```

## See Also

*downsamp, integ, interp, samphold, upsamp*

## Credits

Example written by Kevin Conder.

# diskin

diskin -- Reads audio data from an external device or stream and can alter its pitch.

diskin

## Description

Reads audio data from an external device or stream and can alter its pitch.

## Syntax

```
ar1 [, ar2 [, ar3 [, ... ar24]]] diskin  
    ifilcod, kpitch [, iskiptim] [, iwraparound] [, iformat] [, iskipinit]
```

## 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*.

*iskiptim* (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
- 7 = 8-bit unsigned int (not available in Csound versions older than 5.00)
- 8 = 24-bit int (not available in Csound versions older than 5.00)
- 9 = 64-bit doubles (not available in Csound versions older than 5.00)

*iwraparound* -- 1 = on, 0 = off (wraps around to end of file either direction)

*iskipinit* switches off all initialisation if non zero (default =0). This was introduced in 4\_23f13 and csound5.

If *iformat* = 0 it is taken from the soundfile header, and if no header from the Csound -o command-line flag. The default value is 0.

## Performance

*kpitch* -- can be any real number. a negative number signifies backwards playback. The given number is a pitch ratio, where:

- $1$  = normal pitch
- $2$  = 1 octave higher
- $3$  = 12th higher, etc.
- $.5$  = 1 octave lower
- $.25$  = 2 octaves lower, etc.
- $-1$  = normal pitch backwards
- $-2$  = 1 octave higher backwards, etc.

*diskin* is identical to *soundin* except that it can alter the pitch of the sound that is being read.



### Note to Windows users

Windows users typically use back-slashes, “\”, when specifying the paths of their files. As an example, a Windows user might use the path “c:\music\samples\loop001.wav”. This is problematic because back-slashes are normally used to specify special characters.

To correctly specify this path in Csound, one may alternately:

- Use forward slashes: c:/music/samples/loop001.wav
- Use back-slash special characters, “\\”: c:\\music\\samples\\loop001.wav

## Examples

Here is an example of the *diskin* opcode. It uses the files *diskin.orc* [examples/diskin.orc], *diskin.sco* [examples/diskin.sco], *beats.wav* [examples/beats.wav].

### Example 106. Example of the *diskin* opcode.

```
/* diskin.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1 - play an audio file.
instr 1
  ; Play the audio file backwards.
  asig diskin "beats.wav", -1
  out asig
endin
/* diskin.orc */
```

```
/* diskin.sco */
; Play Instrument #1, the audio file, for three seconds.
i 1 0 3
```



```
e  
/* diskin.sco */
```

## See Also

*in, inh, ino, inq, ins, soundin*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

Example written by Kevin Conder.

Warning to Windows users added by Kevin Conder, April 2002

# diskin2

diskin2 -- Reads audio data from a file, and can alter its pitch using one of several available interpolation types, as well as convert the sample rate to match the orchestra sr setting.

diskin2

## Description

Reads audio data from a file, and can alter its pitch using one of several available interpolation types, as well as convert the sample rate to match the orchestra sr setting. diskin2 can also read multichannel files with any number of channels in the range 1 to 24. diskin2 allows more control and higher sound quality than diskin, but there is also the disadvantage of higher CPU usage.

## Syntax

```
a1[, a2[, ... a24]] diskin2 ifilcod, kpitch[, iskiptim[, iwrap[, iformat [, iws
```

## Initialization

*ifilcod* -- integer or character-string denoting the source soundfile name. An integer denotes the file soundin.ifilcod; 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 those given by the environment variable SSDIR (if defined) then by SFDIR. See also GEN01. Note: files longer than  $2^{31}-1$  sample frames may not be played correctly on 32 bit platforms; this means a maximum length about 3 hours with a sample rate of 192000 Hz.

*iskiptim* (optional, defaults to zero) -- time in seconds of input sound to be skipped, assuming *kpitch*=1. Can be negative, to add -iskiptim/kpitch seconds of delay instead of skipping sound.

*iwrap* (optional, defaults to zero) -- if set to any non-zero value, read locations that are negative or are beyond the end of the file are wrapped to the duration of the sound file instead of assuming zero samples. Useful for playing a file in a loop.



### Note

If *iwrap* is enabled, the file length should not be shorter than the interpolation window size (see below), otherwise there may be clicks in the sound output.

*iformat* (optional, defaults to zero) -- sample format, for raw (headerless) files only. This parameter is ignored if the file has a header. Allowed values are:

- 0: 16-bit short integers
- 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
- 7: 8-bit unsigned int

- 8: 24-bit int
- 9: 64-bit doubles

*iwsiz*e (optional, defaults to zero) -- interpolation window size, in samples. Can be one of the following:

- 1: round to nearest sample (no interpolation, for *kpitch*=1)
- 2: linear interpolation
- 4: cubic interpolation
- >= 8: *iwsiz*e point sinc interpolation with anti-aliasing (slow)

Zero or negative values select the default, which is cubic interpolation.



### Note

If interpolation is used, *kpitch* is automatically scaled by the ratio of the sample rate of the sound file and the orchestra, so that the file will always be played at the original pitch if *kpitch* is 1. However, the sample rate conversion is disabled if *iwsiz*e is 1.

*ibufsize* (optional, defaults to 0) -- buffer size in mono samples (not sample frames). This is only the suggested value, the actual setting will be rounded so that the number of sample frames is an integer power of two and is in the range 128 (or *iwsiz*e if greater than 128) to 1048576. The default, which is 4096, and is enabled by zero or negative values, should be suitable for most uses, but for non-realtime mixing of many large sound files, a high buffer setting is recommended to improve the efficiency of disk reads. For real time audio output, reading the files from a fast RAM file system (on platforms where this option is available) with a small buffer size may be preferred.

*iskipinit* (optional, defaults to 0) -- skip initialization if set to any non-zero value.

## Performance

*a1* ... *a24* -- output signals, in the range -0dbfs to 0dbfs. Any samples before the beginning (i.e. negative location) and after the end of the file are assumed to be zero, unless *iwrap* is non-zero. The number of output arguments must be the same as the number of sound file channels - which can be determined with the *filenchnls* opcode, otherwise an init error will occur.



### Note

It is more efficient to read a single file with many channels, than many files with only a single channel, especially with high *iwsiz*e settings.

*kpitch* -- transpose the pitch of input sound by this factor (e.g. 0.5 means one octave lower, 2 is one octave higher, and 1 is the original pitch). Fractional and negative values are allowed (the latter results in playing the file backwards, however, in this case the skip time parameter should be set to some positive value, e.g. the length of the file, or *iwrap* should be non-zero, otherwise nothing would be played). If interpolation is enabled, and the sample rate of the file differs from the orchestra sample rate, the transpose ratio is automatically adjusted to make sure that *kpitch*=1 plays at the original pitch. Using a high *iwsiz*e setting (40 or more) can significantly improve sound quality when transposing up, although at the expense of high CPU usage.

## Example

```
<CsoundSynthesizer>
```

```
<CsOptions>
; set this to a directory where beats.aiff can be found
--env:SSDIR+=/Csound/Documentation/manual/examples
</CsOptions>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2

        instr 1

ktrans  linseg 1, 5, 2, 10, -2
a1      diskin2 "beats.aiff", ktrans, 0, 1, 0, 32
        outs a1, a1
        endin

</CsInstruments>
<CsScore>

i 1 0 15
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
2005

# dispfft

dispfft -- Displays the Fourier Transform of an audio or control signal.

displayfft

## 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.

## Syntax

**dispfft** xsig, iprd, iwsiz [, iwtyp] [, idbout] [, iwtflg]

## 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, default=0) -- window type. 0 = rectangular, 1 = Hanning. The default value is 0 (rectangular).

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

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

## Performance

*dispfft* -- displays the Fourier Transform of an audio or control signal (*asig* or *ksig*) every *iprd* seconds using the Fast Fourier Transform method.

## Examples

Here is an example of the dispfft opcode. It uses the files *dispfft.orc* [examples/dispfft.orc], *dispfft.sco* [examples/dispfft.sco] and *beats.wav* [examples/beats.wav].

### Example 107. Example of the dispfft opcode.

```
/* dispfft.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  asig soundin "beats.wav"
```

```
    dispfft asig, 1, 512
    out asig
endin
/* dispfft.orc */
```

```
/* dispfft.sco */
; Play Instrument #1 for three seconds.
i 1 0 3
e
/* dispfft.sco */
```

## See Also

*display, print*

## Credits

Comments about the *inprds* parameter contributed by Rasmus Ekman.

Example written by Kevin Conder.

# display

`display` -- Displays the audio or control signals as an amplitude vs. time graph.

`display`

## 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.

## Syntax

**display** *xsig*, *iprd* [, *inprds*] [, *iwtflg*]

## Initialization

*iprd* -- the period of display in seconds.

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

*inprds* (optional, default=1) -- a scaling factor for the displayed waveform, controlling how many *iprd*-sized frames of samples are drawn in the window (the default and minimum value is 1.0). Higher *inprds* values are slower to draw (more points to draw) but will show the waveform scrolling through the window, which is useful with low *iprd* values.

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

## Performance

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

## Examples

Here is an example of the `display` opcode. It uses the files *display.orc* [examples/display.orc] and *display.sco* [examples/display.sco].

### Example 108. Example of the `display` opcode.

```
/* display.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Go from 1000 to 0 linearly, over the period defined by p3.
  klin line 1000, p3, 0
```

```
    ; Create a new display each second, wait for the user.  
    display klin, 1, 1, 1  
endin  
/* display.orc */
```

```
/* display.sco */  
; Play Instrument #1 for 5 seconds.  
i 1 0 5  
e  
/* display.sco */
```

## See Also

*dispfft, print*

## Credits

Comments about the *inprds* parameter contributed by Rasmus Ekman.

Example written by Kevin Conder.



# distort

distort -- Distort an audio signal via waveshaping and optional clipping.

distort

## Description

## Syntax

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

## Initialization

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

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

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

## Performance

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

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

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

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

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

## Examples

```
gifn  ftgen      0,0, 257, 9, .5,1,270      ; define a sigmoid, c
gifn  ftgen      0,0, 257, 9, .5,1,270,1.5,.33,90,2.5,.2,270,3.5,.1
kdist line      0, 10, 1.2                  ; and over 10 seconds
aout  distort    asig, kdist, gifn          ; gradually increase
```

## Credits

Written by Barry L. Vercoe for Extended Csound and released in csound5.

# distort1

distort1 -- Modified hyperbolic tangent distortion.

distort1

## Description

Implementation of modified hyperbolic tangent distortion. *distort1* can be used to generate wave shaping distortion based on a modification of the *tanh* function.

$$\text{aout} = \frac{\exp(\text{asig} * (\text{shape1} + \text{pregain})) - \exp(\text{asig} * (\text{shape2} - \text{pregain}))}{\exp(\text{asig} * \text{pregain}) + \exp(-\text{asig} * \text{pregain})}$$

## Syntax

ares **distort1** asig, kpregain, kpostgain, kshape1, kshape2[, imode]

## Initialization

*imode* (Csound version 5.00 and later only; optional, defaults to 0) -- scales kpregain, kpostgain, kshape1, and kshape2 for use with audio signals in the range -32768 to 32768 (*imode*=0), -0dbfs to 0dbfs (*imode*=1), or disables scaling of kpregain and kpostgain and scales kshape1 by kpregain and kshape2 by -kpregain (*imode*=2).

## 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.

## Examples

Here is an example of the distort1 opcode. It uses the files *distort1.orc* [examples/distort1.orc] and *distort1.sco* [examples/distort1.sco].

### Example 109. Example of the distort1 opcode.

```
/* distort1.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
```

```
ksmps = 10
nchnls = 2

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 distortl gadist, kpre, kpost, kshap1, kshap2

  outs aout, aout

  gadist = 0
endin
/* distortl.orc */

/* distortl.sco */
; 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
/* distortl.sco */
```

## Credits

Author: Hans Mikelson  
December 1998

New in Csound version 3.50

## divz

divz -- Safely divides two numbers.

divz

## Syntax

ares **divz** xa, xb, ksubst

ires **divz** ia, ib, isubst

kres **divz** ka, kb, ksubst

## Description

Safely divides two numbers.

## Initialization

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.

## Examples

Here is an example of the divz opcode. It uses the files *divz.orc* [examples/divz.orc] and *divz.sco* [examples/divz.sco].

### Example 110. Example of the divz opcode.

```
/* divz.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Define the numbers to be divided.
ka init 200
; Linearly change the value of kb from 200 to 0.
kb line 0, p3, 200
; If a "divide by zero" error occurs, substitute -1.
ksubst init -1

; Safely divide the numbers.
kresults divz ka, kb, ksubst

; Print out the results.
printks "%f / %f = %f\\n", 0.1, ka, kb, kresults
endin
/* divz.orc */
```

```
/* divz.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* divz.sco */
```

Its output should include lines like:

```
200.000000 / 0.000000 = -1.000000  
200.000000 / 19.999887 = 10.000056  
200.000000 / 40.000027 = 4.999997
```

## See Also

*=, init, tival*

## Credits

Author: John ffitch after an idea by Barry L. Vercoe

Example written by Kevin Conder.

# downsamp

downsamp -- Modify a signal by down-sampling.

downsamp

## Description

Modify a signal by down-sampling.

## Syntax

```
kres downsamp asig [, iwlen]
```

## 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.

## 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 fold-over.

## Examples

Here is an example of the *downsamp* opcode. It uses the files *downsamp.orc* [examples/downsamp.orc] and *downsamp.sco* [examples/downsamp.sco].

### Example 111. Example of the *downsamp* opcode.

```
/* downsamp.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Create a noise signal at a-rate.
  anoise noise 20000, 0.2

  ; Downsample the noise signal to k-rate.
  knoise downsamp anoise

  ; Use the noise signal at k-rate.
  al oscil 30000, knoise, 1
  out anoise
endin
/* downsamp.orc */
```

```
/* downsamp.sco */  
; Table #1, a sine wave.  
f 1 0 16384 10 1  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* downsamp.sco */
```

## See Also

*diff, integ, interp, samphold, upsamp*

## Credits

Example written by Kevin Conder.



# dripwater

dripwater -- Semi-physical model of a water drop.

dripwater

## Description

*dripwater* is a semi-physical model of a water drop. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **dripwater** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1] [, ifreq2]

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 10.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.996 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.996 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 2.0.

The recommended range for *idamp* is usually below 75% of the maximum value. Rasmus Ekman suggests a range of 1.4-1.75. He also suggests a maximum value of 1.9 instead of the theoretical limit of 2.0.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 450.

*ifreq1* (optional) -- the first resonant frequency. The default value is 600.

*ifreq2* (optional) -- the second resonant frequency. The default value is 750.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the dripwater opcode. It uses the files *dripwater.orc* [examples/dripwater.orc] and *dripwater.sco* [examples/dripwater.sco].

### Example 112. Example of the dripwater opcode.

```
/* dripwater.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 01 ;example of a water drip
a1 line 5, p3, 5 ;preset an amplitude boost
a2 dripwater p4, 0.01, 0, .9 ;dripwater needs a little amplitude help at thes
a3 product a1, a2 ;increase amplitude
    out a3
endin
/* dripwater.orc */

/* dripwater.sco */
i1 0 1 20000
e
/* dripwater.sco */
```

## See Also

*bamboo, guiro, sleighbells, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# dssiactivate

`dssiactivate` -- Activates or deactivates a DSSI or LADSPA plugin.

`dssiactivate`

## Syntax

**dssiactivate** *ihandle*, *ktoggle*

## Description

*dssiactivate* is used to activate or deactivate a DSSI or LADSPA plugin. It calls the plugin's `activate()` and `deactivate()` functions if they are provided.

## Initialization

*ihandle* - the number which identifies the plugin, generated by *dssiinit*.

## Performance

*ktoggle* - Selects between activation (*ktoggle*=1) and deactivation (*ktoggle*=0).

*dssiactivate* is used to turn on and off plugins if they provide this facility. This may help conserve CPU processing in some cases. For consistency, all plugins must be activated to produce sound. An inactive plugin produces silence.

Depending on the plugin's implementation, this may cause interruptions in the realtime audio process, so use with caution.

*dssiactivate* may cause audio stream breakups when used in realtime, so it is recommended to load all plugins to be used before playing.



### Warning

Please note that even if `activate()` and `deactivate()` functions are not present in a plugin, *dssiactivate* must be called for the plugin to produce sound.

## Credits

2005

By: Andres Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dssiaudio

dssiaudio -- Processes audio using a LADSPA or DSSI plugin.

dssiaudio

## Syntax

```
aout1 [, aout2, aout3, aout4] dssiaudio ihandle, ain1 [,ain2, ain3, ain4]
```

## Description

*dssiaudio* generates audio by processing an input signal through a LADSPA plugin.

## Initialization

### Initialization

*ihandle* - handle for the plugin returned by *dssiinit*

*aout1, aout2, etc* - Audio output generated by the plugin

*ain1, ain2, etc* - Audio provided to the plugin for processing

*dssiaudio* runs a plugin on the provided audio and produces audio output. Currently upto four inputs and outputs are provided. You should provide signal for all the plugins audio inputs, otherwise unpredictable results may occur. If the plugin doesn't have any input (e.g Noise generator) you must still provide at least one input variable, which will be ignored with a message.

Only one *dssiaudio* should be executed once per plugin, or strange results may occur.

## Credits

2005

By: Andres Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dssictls

dssictls -- Send control information to a LADSPA or DSSI plugin.

dssictls

## Syntax

```
dssictls ihandle, iport, kvalue, ktrigger
```

## Description

*dssictls* sends control values to a plugin's control port

## Initialization

### Initialization

*ihandle* - handle for the plugin returned by *dssiinit*

*iport* - control port number

*kvalue* - value to be assigned to the port

*ktrigger* - determines whether the control information will be sent (*ktrigger* = 1) or not. This is useful for thinning control information, generating *ktrigger* with *metro*

*dssictls* sends control information to a LADSPA or DSSI plugin's control port. The valid control ports and ranges are given by *dssiinit*. Using values outside the ranges may produce unspecified behaviour.

## Credits

2005

By: Andres Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dssiinit

dssiinit -- Loads a DSSI or LADSPA plugin.

dssiinit

## Syntax

```
ihandle dssiinit ilibraryname, ipluginindex [, iverbose]
```

## Description

*dssiinit* is used to load a DSSI or LADSPA plugin into memory for use with the other dssi4cs opcodes. Both LADSPA effects and DSSI instruments can be used.

## Initialization

*ihandle* - the number which identifies the plugin, to be passed to other dssi4cs opcodes.

*ilibraryname* - the name of the .so (shared object) file to load.

*ipluginindex* - The index of the plugin to be used.

*iverbose* (optional) - show plugin information and parameters when loading. (default = 1)

*dssiinit* looks for *ilibraryname* on LADSPA\_PATH and DSSI\_PATH. One of these variables must be set, otherwise *dssiinit* will return an error. LADSPA and DSSI libraries may contain more than one plugin which must be referenced by its index. *dssiinit* then attempts to find plugin index *ipluginindex* in the library and load the plugin into memory if it is found. To find out which plugins you have available and their index numbers you can use: *dssilist*.

If *iverbose* is not 0 (the default), information about the plugin detailing its characteristics and its ports will be shown. This information is important for opcodes like *dssictls*.

Plugins are set to inactive by default, so you *\*must\** use *dssiactivate* to get the plugin to produce sound. This is required even if the plugin doesn't provide an activate() function.

*dssiinit* may cause audio stream breakups when used in realtime, so it is recommended to load all plugins to be used before playing.

## Examples

Here is an example of the dssinit opcode. It uses the file *dssi4cs.csd* [examples/dssi4cs.csd].

### Example 113. Example of the dssiinit opcode. (Remember to change the Library name)

```
<CsoundSynthesizer>

<CsOptions>
;use appropriate realtime options
</CsOptions>

<CsInstruments>
ksmps = 256
nchnls = 2
```

```
dssilist

gihandle dssiinit "amp.so", 0, 1
;gihandle dssiinit "cmt.so", 30 , 2
;gihandle2 dssiinit "cmt.so", 8 , 1
;gihandle dssiinit "delayorama_1402", 0
gihandle2 dssiinit "cmt.so", 49 , 1
;gihandle dssiinit "freq_tracker_1418.so", 0 , 1, 1
;gihandle dssiinit "g2reverb.so", 0, 1
;gihandle2 dssiinit "declip_1195.so", 0, 1
;gihandle2 dssiinit "revdelay_1605.so", 0, 1
;gihandle2 dssiinit "tap_chorusflanger.so", 0, 1
;gihandle2 dssiinit "plate_1423.so", 0, 1
gihandle3 dssiinit "gate_1410.so", 0, 1
;gihandle3 dssiinit "hexter.so", 0, 1

instr 1
print p4
dssiactivate gihandle, p4
dssiactivate gihandle2, p4
dssiactivate gihandle3, p4
endin

instr 2
ain1 inch 1
ain2 inch 2
;aout1,aout2 dssiaudio gihandle, ain1, ain2
aout1 dssiaudio gihandle, ain1
outs aout1,aout1
endin

instr 3
kval linen 1, p3 /3, p3, p3/ 3
dssictls gihandle, p4, kval, 1
endin

instr 4
ain1 inch 1
aout1 dssiaudio gihandle2, ain1
outs aout1,aout1
endin

</CsInstruments>

<CsScore>

i 1 1 1 1

i 2 2 15 ;plugin 1

i 3 3 12 0 ;Control port 0

i 4 8 2 ;plugin 2
e
</CsScore>

</CsoundSynthesizer>
```

## Credits

2005

By: Andres Cabrera

Uses code from Richard Furse's LADSPA sdk.



# dssilist

dssilist -- Lists all available DSSI and LADSPA plugins.

dssilist

## Syntax

**dssilist**

## Description

*dssilist* checks the variables DSSI\_PATH and LADSPA\_PATH and lists all plugins available in all plugin libraries there.

LADSPA and DSSI libraries may contain more than one plugin which must be referenced by the index provided by *dssilist*.

This opcode produces a long printout which may interrupt realtime audio output, so it should be run at the start of a performance.

## Credits

2005

By: Andres Cabrera

Uses code from Richard Furse's LADSPA sdk.

# dumpk

dumpk -- Periodically writes an orchestra control-signal value to an external file.

dumpk

## Description

Periodically writes an orchestra control-signal value to a named external file in a specific format.

## Syntax

**dumpk** ksig, ifilename, iformat, iprd

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats, 7=ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. The output file contains no header information.

*iprd* -- the period of *ksig* output in seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will create an output file sampled at the orchestra control rate.

## Performance

*ksig* -- a control-rate signal

This opcode allows a generated control signal value to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

```
knum      =          knum+1
ktemp      tempest    krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995
kocot      specptrk    wsig, 6, .9, 0
```

```
; at each
; estimate
; and the
```

```
dumpk3      knum, ktemp, cpsoct(koct), "what happened when", 8 0 ;& save
```

## See Also

*dumpk2, dumpk3, dumpk4, readk, readk2, readk3, readk4*

# dumpk2

dumpk2 -- Periodically writes two orchestra control-signal values to an external file.

dumpk2

## Description

Periodically writes two orchestra control-signal values to a named external file in a specific format.

## Syntax

**dumpk2** ksig1, ksig2, ifilename, iformat, iprd

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats, 7=ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. 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.

## Performance

*ksig1*, *ksig2* -- control-rate signals.

This opcode allows two generated control signal values to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk2* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

```
knum      =      knum+1
ktemp      tempest  krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995
kocot      specptrk  wsig, 6, .9, 0
```

```
; at each
;estimated
;and the
```

```
dumpk3      knum, ktemp, cpsoct(koct), "what happened when", 8 0 ;& save
```

## See Also

*dumpk*, *dumpk3*, *dumpk4*, *readk*, *readk2*, *readk3*, *readk4*

# dumpk3

dumpk3 -- Periodically writes three orchestra control-signal values to an external file.

dumpk3

## Description

Periodically writes three orchestra control-signal values to a named external file in a specific format.

## Syntax

**dumpk3** ksig1, ksig2, ksig3, ifilename, iformat, iprd

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats, 7=ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. 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.

## Performance

*ksig1*, *ksig2*, *ksig3* -- control-rate signals

This opcode allows three generated control signal values to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk3* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

```
knum      =      knum+1
ktemp      tempest  krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995
kocot      specptrk  wsig, 6, .9, 0
```

```
; at each
;estimated
;and the
```

```
dumpk3      knum, ktemp, cpsoct(koct), "what happened when", 8 0 ;& save
```

## See Also

*dumpk*, *dumpk2*, *dumpk4*, *readk*, *readk2*, *readk3*, *readk4*

# dumpk4

dumpk4 -- Periodically writes four orchestra control-signal values to an external file.

dumpk4

## Description

Periodically writes four orchestra control-signal values to a named external file in a specific format.

## Syntax

**dumpk4** ksig1, ksig2, ksig3, ksig4, ifilename, iformat, iprd

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats, 7=ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. 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.

## Performance

*ksig1*, *ksig2*, *ksig3*, *ksig4* -- control-rate signals

This opcode allows four generated control signal values to be saved in a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *dumpk4* opcodes in an instrument or orchestra but each must write to a different file.

## Examples

```
knum      =      knum+1
ktemp      tempest  krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995
kocot      specptrk wsig, 6, .9, 0
```

```
; at each
;estimated
;and the
```



```
dumpk3      knum, ktemp, cpsoct(koct), "what happened when", 8 0 ;& save
```

## See Also

*dumpk*, *dumpk2*, *dumpk3*, *readk*, *readk2*, *readk3*, *readk4*

# duserrnd

duserrnd -- Discrete USER-defined-distribution RaNDom generator.

duserrnd

## Description

Discrete USER-defined-distribution RaNDom generator.

## Syntax

aout **duserrnd** ktableNum

iout **duserrnd** itableNum

kout **duserrnd** ktableNum

## Initialization

*itableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

## Performance

*ktableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

*duserrnd* (discrete user-defined-distribution random generator) generates random values according to a discrete random distribution created by the user. The user can create the discrete distribution histogram by using GEN41. In order to create that table, the user has to define an arbitrary amount of number pairs, the first number of each pair representing a value and the second representing its probability (see GEN41 for more details).

When used as a function, the rate of generation depends by the rate type of input variable XtableNum. In this case it can be embedded into any formula. Table number can be varied at k-rate, allowing to change the distribution histogram during the performance of a single note. *duserrnd* is designed be used in algorithmic music generation.

*duserrnd* can also be used to generate values following a set of ranges of probabilities by using distribution functions generated by GEN42 (See GEN42 for more details). In this case, in order to simulate continuous ranges, the length of table XtableNum should be reasonably big, as *duserrnd* does not interpolate between table elements.

For a tutorial about random distribution histograms and functions see:

- D. Lorrain. "A panoply of stochastic cannons". In C. Roads, ed. 1989. Music machine. Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## See Also

*cuserrnd*, *urd*

## Credits

Author: Gabriel Maldonado

New in Version 4.16

# else

else -- Executes a block of code when an "if...then" condition is false.

else

## Description

Executes a block of code when an "if...then" condition is false.

## Syntax

**else**

## Performance

*else* is used inside of a block of code between the *"if...then"* and *endif* opcodes. It defines which statements are executed when a "if...then" condition is false. Only one *else* statement may occur and it must be the last conditional statement before the *endif* opcode.

## Examples

See the example for the *if* opcode.

## See Also

*elseif*, *endif*, *goto*, *if*, *igoto*, *kgoto*, *tigoto*, *timeout*

## Credits

New in version 4.21

# elseif

elseif -- Defines another "if...then" condition when a "if...then" condition is false.

elseif

## Description

Defines another "if...then" condition when a "if...then" condition is false.

## Syntax

**elseif** *xa* *R* *xb* **then**

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*).

## Performance

*elseif* is used inside of a block of code between the "*if...then*" and *endif* opcodes. When a "if...then" condition is false, it defines another "if...then" condition to be met. Any number of *elseif* statements are allowed.

## Examples

See the example for the *if* opcode.

## See Also

*else*, *endif*, *goto*, *if*, *igoto*, *kgoto*, *tigoto*, *timeout*

## Credits

New in version 4.21

# endif

endif -- Closes a block of code that begins with an "if...then" statement.

endif

## Description

Closes a block of code that begins with an *"if...then"* statement.

## Syntax

**endif**

## Performance

Any block of code that begins with an *"if...then"* statement must end with an *endif* statement.

## Examples

See the example for the *if* opcode.

## See Also

*elseif, else, goto, if, igoto, kgoto, tigoto, timeout*

## Credits

New in version 4.21

# endin

endin -- Ends the current instrument block.

endin

## Description

Ends the current instrument block.

## Syntax

**endin**

## Initialization

Ends the current instrument block.

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).



### Note

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

## Examples

Here is an example of the endin opcode. It uses the files *endin.orc* [examples/endin.orc] and *endin.sco* [examples/endin.sco].

### Example 114. Example of the endin opcode.

```
/* endin.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iamp = 10000
  icps = 440
  iphs = 0

  a1 oscils iamp, icps, iphs
  out a1
endin
/* endin.orc */
```

```
/* endin.sco */
```

```
; Play Instrument #1 for 2 seconds.  
i 1 0 2  
e  
/* endin.sco */
```

## See Also

*instr*

## Credits

Example written by Kevin Conder.



# endop

endop -- Marks the end of an user-defined opcode block.

endop

## Description

Marks the end of an user-defined opcode block.

## Syntax

**endop**

## Performance

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmps]
```

## Examples

See the example for the *opcode* opcode.

## See Also

*opcode*, *setksmps*, *xin*, *xout*

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

# envlpx

envlpx -- Applies an envelope consisting of 3 segments.

envlpx

## Description

*envlpx* -- apply an envelope consisting of 3 segments:

1. stored function rise shape
2. modified exponential pseudo steady state
3. exponential decay

## Syntax

ares **envlpx** xamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod]

kres **envlpx** kamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod]

## 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.

*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 greater than 1 causes an exponential growth and a factor less than 1 creates an exponential decay. A factor of 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

*kamp*, *xamp* -- input amplitude signal.

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 modified by the first exponential pattern. If the rise and decay periods overlap then 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, tending asymptotically to zero.

## Examples

Here is an example of the `envlpx` opcode. It uses the files `envlpx.orc` [examples/envlpx.orc] and `envlpx.sco` [examples/envlpx.sco].

### Example 115. Example of the `envlpx` opcode.

```
/* envlpx.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a simple instrument.
instr 1
  ; Set the amplitude.
  kamp init 20000
  ; Get the frequency from the fourth p-field.
  kcps = cspch(p4)

  al vco kamp, kcps, 1
  out al
endin

; Instrument #2 - instrument with an amplitude envelope.
instr 2
  kamp = 20000
  irise = 0.05
  idur = p3 - .01
  idec = 0.5
  ifn = 2
  iatss = 1
  iatdec = 0.01

  ; Create an amplitude envelope.
  kenv envlpx kamp, irise, idur, idec, ifn, iatss, iatdec

  ; Get the frequency from the fourth p-field.
  kcps = cspch(p4)

  al vco kenv, kcps, 1
  out al
endin
/* envlpx.orc */
```

```
/* envlpx.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1
; Table #2, a rising envelope.
f 2 0 129 -7 0 128 1

; Set the tempo to 120 beats per minute.
t 0 120

; Make sure the score plays for 33 seconds.
f 0 33

; Play a melody with Instrument #1.
```

```
; p4 = frequency in pitch-class notation.
i 1 0 1 8.04
i 1 1 1 8.04
i 1 2 1 8.05
i 1 3 1 8.07
i 1 4 1 8.07
i 1 5 1 8.05
i 1 6 1 8.04
i 1 7 1 8.02
i 1 8 1 8.00
i 1 9 1 8.00
i 1 10 1 8.02
i 1 11 1 8.04
i 1 12 2 8.04
i 1 14 2 8.02

; Repeat the melody with Instrument #2.
; p4 = frequency in pitch-class notation.
i 2 16 1 8.04
i 2 17 1 8.04
i 2 18 1 8.05
i 2 19 1 8.07
i 2 20 1 8.07
i 2 21 1 8.05
i 2 22 1 8.04
i 2 23 1 8.02
i 2 24 1 8.00
i 2 25 1 8.00
i 2 26 1 8.02
i 2 27 1 8.04
i 2 28 2 8.04
i 2 30 2 8.02
e
/* envlpx.sco */
```

## See Also

*envlpxr*, *linen*, *linenr*

## Credits

Thanks goes to Luis Jure for pointing out a mistake with *iatss*.

Example written by Kevin Conder.

# envlpxr

envlpxr -- The envlpx opcode with a final release segment.

envlpxr

## Description

*envlpxr* is the same as *envlpx* except that the final segment is entered only on sensing a MIDI note release. The note is then extended by the decay time.

## Syntax

ares **envlpxr** xamp, irise, idec, ifn, iatss, iatdec [, ixmod] [,irind]

kres **envlpxr** kamp, irise, idec, ifn, iatss, iatdec [, ixmod] [,irind]

## Initialization

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

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

*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 greater than 1 causes an exponential growth and a factor less than 1 creates an exponential decay. A factor of 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).

*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.

## Performance

*kamp*, *xamp* -- input amplitude signal.

*envlpxr* is an example of the special Csound “r” units that contain a note-off sensor and release time extender. 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 unless it is made independent by *irind*. Then it will begin a decay 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*.

## See Also

*envlpx*, *linen*, *linenr*

## Credits

Thanks goes to Luis Jure for pointing out a mistake with *iatss*.

# event

event -- Generates a score event from an instrument.

event

## Description

Generates a score event from an instrument.

## Syntax

**event** "scorechar", kinsnum, kdelay, kdur, [, kp4] [, kp5] [, ...]

**event** "scorechar", "insname", kdelay, kdur, [, kp4] [, kp5] [, ...]

## Initialization

*"scorechar"* -- A string (in double-quotes) representing the first p-field in a score statement. This is usually *"e"*, *"f"*, or *"i"*.

*"insname"* -- A string (in double-quotes) representing a named instrument.

## Performance

*kinsnum* -- The instrument to use for the event. This corresponds to the first p-field, p1, in a score statement.

*kdelay* -- When (in seconds) the event will occur from the current performance time. This corresponds to the second p-field, p2, in a score statement.

*kdur* -- How long (in seconds) the event will happen. This corresponds to the third p-field, p3, in a score statement.

*kp4*, *kp5*, ... (optional) -- Parameters representing additional p-field in a score statement. It starts with the fourth p-field, p4.

## Examples

Here is an example of the event opcode. It uses the files *event.orc* [examples/event.orc] and *event.sco* [examples/event.sco].

### Example 116. Example of the event opcode.

```
/* event.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - an oscillator with a high note.
instr 1
  ; Create a trigger and set its initial value to 1.
  ktrigger init 1
```

```
; If the trigger is equal to 0, continue playing.
; If not, schedule another event.
if (ktrigger == 0) goto contin
  ; kscoreop="i", an i-statement.
  ; kinsnum=2, play Instrument #2.
  ; kwhen=1, start at 1 second.
  ; kdur=0.5, play for a half-second.
  event "i", 2, 1, 0.5

  ; Make sure the event isn't triggered again.
  ktrigger = 0

contin:
  a1 oscils 10000, 440, 1
  out a1
endin

; Instrument #2 - an oscillator with a low note.
instr 2
  a1 oscils 10000, 220, 1
  out a1
endin
/* event.orc */

/* event.sco */
; Make sure the score plays for two seconds.
f 0 2

; Play Instrument #1 for a half-second.
i 1 0 0.5
e
/* event.sco */
```

Here is an example of the event opcode using a named instrument. It uses the files *event\_named.orc* [examples/event\_named.orc] and *event\_named.sco* [examples/event\_named.sco].

### **Example 117. Example of the event opcode using a named instrument.**

```
/* event_named.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - an oscillator with a high note.
instr 1
  ; Create a trigger and set its initial value to 1.
  ktrigger init 1

  ; If the trigger is equal to 0, continue playing.
  ; If not, schedule another event.
  if (ktrigger == 0) goto contin
    ; kscoreop="i", an i-statement.
    ; kinsnum="low_note", instrument named "low_note".
    ; kwhen=1, start at 1 second.
```



```
; kdur=0.5, play for a half-second.
event "i", "low_note", 1, 0.5

; Make sure the event isn't triggered again.
ktrigger = 0

contin:
  al oscils 10000, 440, 1
  out al
endin

; Instrument "low_note" - an oscillator with a low note.
instr low_note
  al oscils 10000, 220, 1
  out al
endin
/* event_named.orc */

/* event_named.sco */
; Make sure the score plays for two seconds.
f 0 2

; Play Instrument #1 for a half-second.
i 1 0 0.5
e
/* event_named.sco */
```

## Credits

Examples written by Kevin Conder.

New in version 4.17

Thanks goes to Matt Ingalls for helping to fix the example.

Thanks goes to Matt Ingalls for helping clarify the kwhen/kdelay parameter.

## event\_i

`event_i` -- Generates a score event from an instrument.

`event_i`

## Description

Generates a score event from an instrument.

## Syntax

**event\_i** "scorechar", iinsnum, idelay, idur, [, ip4] [, ip5] [, ...]

**event** "scorechar", "insname", idelay, idur, [, ip4] [, ip5] [, ...]

## Initialization

*"scorechar"* -- A string (in double-quotes) representing the first p-field in a score statement. This is usually *"e"*, *"f"*, or *"i"*.

*"insname"* -- A string (in double-quotes) representing a named instrument.

*iinsnum* -- The instrument to use for the event. This corresponds to the first p-field, p1, in a score statement.

*idelay* -- When (in seconds) the event will occur from the current performance time. This corresponds to the second p-field, p2, in a score statement.

*idur* -- How long (in seconds) the event will happen. This corresponds to the third p-field, p3, in a score statement.

*ip4, ip5, ...* (optional) -- Parameters representing additional p-field in a score statement. It starts with the fourth p-field, p4.

## Performance

The event is added to the queue at initialisation time.

## Credits

Written by Istvan Varga.

New in Csound5

# exitnow

exitnow -- Exit csound as fast as possible, with no cleaning up.

exitnow

## Description

In Csound4 calls an exit function to leave csound as fast as possible. On Csound5 exits back to the driving code.

## Syntax

`exitnow`

## Performance

Stops Csound on the initialisation cycle.

# exp

`exp` -- Returns *e* raised to the *x*-th power.

`exp`

## Description

Returns *e* raised to the *x*th power.

## Syntax

**exp**(*x*) (no rate restriction)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the `exp` opcode. It uses the files *exp.orc* [examples/exp.orc] and *exp.sco* [examples/exp.sco].

### Example 118. Example of the `exp` opcode.

```
/* exp.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = exp(8)
  print i1
endin
/* exp.orc */

/* exp.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* exp.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 2980.958
```

## See Also

*abs, frac, int, log, log10, i, sqrt*

## Credits

Example written by Kevin Conder.

New in version 4.21

# expon

expon -- Trace an exponential curve between specified points.

expon

## Description

Trace an exponential curve between specified points.

## Syntax

ares **expon** ia, idur1, ib

kres **expon** ia, idur1, ib

## 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.

## 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.

## Examples

Here is an example of the expon opcode. It uses the files *expon.orc* [examples/expon.orc] and *expon.sco* [examples/expon.sco].

### Example 119. Example of the expon opcode.

```
/* expon.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Define kcps as a frequency value that exponentially declines
; from 880 to 220. It declines over the period set by p3.
kcps expon 880, p3, 220

a1 oscil 20000, kcps, 1
out a1
```

```
endin
/* expon.orc */

/* expon.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* expon.sco */
```

## See Also

*expseg, expsegr, line, linseg, linsegr*

## Credits

Example written by Kevin Conder.

# exprand

exprand -- Exponential distribution random number generator (positive values only).

exprand

## Description

Exponential distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

ares **exprand** krange

ires **exprand** krange

kres **exprand** krange

## Performance

*krange* -- the range of the random numbers (0 - *krange*). Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the exprand opcode. It uses the files *exprand.orc* [examples/exprand.orc] and *exprand.sco* [examples/exprand.sco].

### Example 120. Example of the exprand opcode.

```
/* exprand.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random between 0 and 1.
  ; krange = 1

  i1 exprand 1

  print i1
endin
```



```
/* exprand.orc */

/* exprand.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* exprand.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 0.174
```

## See Also

*seed, betarand, bexprnd, cauchy, gauss, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

## expseg

expseg -- Trace a series of exponential segments between specified points.

expseg

## Description

Trace a series of exponential segments between specified points.

## Syntax

```
ares expseg ia, idur1, ib [, idur2] [, ic] [...]
```

```
kres expseg ia, idur1, ib [, idur2] [, ic] [...]
```

## Initialization

*ia* -- starting value. Zero is illegal for exponentials.

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

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

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

## Performance

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

Note that the *expseg* opcode does not operate correctly at audio rate when segments are shorter than a k-period. Try the *expsega* opcode instead.

## Examples

Here is an example of the *expseg* opcode. It uses the files *expseg.orc* [examples/expseg.orc] and *expseg.sco* [examples/expseg.sco].

### Example 121. Example of the expseg opcode.

```
/* expseg.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1
```

```
; Instrument #1.
instr 1
  ; p4 = frequency in pitch-class notation.
  kcps = cpspch(p4)

  ; Create an amplitude envelope.
  kenv expseg 0.01, p3*0.25, 1, p3*0.75, 0.01
  kamp = kenv * 30000

  a1 oscil kamp, kcps, 1
  out a1
endin
/* expseg.orc */

/* expseg.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for a half-second, p4=8.00
i 1 0 0.5 8.00
; Play Instrument #1 for a half-second, p4=8.01
i 1 1 0.5 8.01
; Play Instrument #1 for a half-second, p4=8.02
i 1 2 0.5 8.02
; Play Instrument #1 for a half-second, p4=8.03
i 1 3 0.5 8.03
e
/* expseg.sco */
```

## See Also

*expon, expsega, expsegr, line, linseg, linsegr transeg*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Csound 3.57

## expsega

expsega -- An exponential segment generator operating at a-rate.

expsega

## Description

An exponential segment generator operating at a-rate. This unit 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.

## Syntax

ares **expsega** ia, idur1, ib [, idur2] [, ic] [...]

## Initialization

*ia* -- starting value. Zero is illegal.

*ib*, *ic*, etc. -- value after *idur1* seconds, etc. must be non-zero and must agree in sign with *ia*.

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

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

## Performance

These units generate control or audio signals whose values can pass through two 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.

## Examples

Here is an example of the expsega opcode. It uses the files *expsega.orc* [examples/expsega.orc] and *expsega.sco* [examples/expsega.sco].

### Example 122. Example of the expsega opcode.

```
/* expsega.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Define a short percussive amplitude envelope that
; goes from 0.01 to 20,000 and back.
aenv expsega 0.01, 0.1, 20000, 0.1, 0.01
```

```
    al oscil aenv, 440, 1
    out al
endin
/* expsega.orc */

/* expsega.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #1 for one second.
i 1 1 1
; Play Instrument #1 for one second.
i 1 2 1
; Play Instrument #1 for one second.
i 1 3 1
e
/* expsega.sco */
```

## See Also

*expseg*, *expsegr*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Csound 3.57

## expsegr

expsegr -- Trace a series of exponential segments between specified points including a release segment.

expsegr

## Description

Trace a series of exponential segments between specified points including a release segment.

## Syntax

ares **expsegr** ia, idur1, ib [, idur2] [, ic] [...], irel, iz

kres **expsegr** ia, idur1, ib [, idur2] [, ic] [...], irel, iz

## 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.

*expsegr* is 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. For two or more extenders in an instrument, extension is by the greatest period.

## Examples

Here is an example of the *expsegr* opcode. It uses the files *expsegr.orc* [examples/expsegr.orc] and *expsegr.sco* [examples/expsegr.sco].

### Example 123. Example of the expsegr opcode.

```
/* expsegr.orc */
```

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; p4 = frequency in pitch-class notation.
  kcps = cpspch(p4)

  ; Use an amplitude envelope with second-long release.
  kenv expsegr 0.01, p3/2, 1, p3/2, 0.01, 1, 1
  kamp = kenv * 30000

  a1 oscil kamp, kcps, 1
  out a1
endin
/* expsegr.orc */

/* expsegr.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Make sure the score lasts for four seconds.
f 0 4

; p4 = frequency (in pitch-class notation).
; Play Instrument #1 for a half-second, p4=8.00
i 1 0 0.5 8.00
; Play Instrument #1 for a half-second, p4=8.01
i 1 1 0.5 8.01
; Play Instrument #1 for a half-second, p4=8.02
i 1 2 0.5 8.02
; Play Instrument #1 for a half-second, p4=8.03
i 1 3 0.5 8.03
e
/* expsegr.sco */
```

## See Also

*expon, expseg, expsega, line, linseg, linsegr*

## Credits

Author: Barry L. Vercoe

Example written by Kevin Conder.

New in Csound 3.47

# ficlose

ficlose -- Closes a previously opened file.

ficlose

## Description

*ficlose* can be used to close a file which was opened with *fiopen*.

## Syntax

```
ficlose ihandle
```

```
ficlose Sfilename
```

## Initialization

*ihandle* -- a number which identifies this file (generated by a previous *fiopen*).

*Sfilename* -- A string in double quotes or string variable with the filename. The full path must be given if the file directory is not in the system PATH and is not present in the current directory.

## Performance

*ficlose* closes a file which was previously opened with *fiopen*. *ficlose* is only needed if you need to read a file written to during the same csound performance, since only when csound ends a performance does it close and save data in all open files. The opcode *ficlose* is useful for instance if you want to save presets within files which you want to be accessible without having to terminate csound.



### Note

If you don't need this functionality it is safer not to call *ficlose*, and just let csound close the files when it exits.

If a file closed with *ficlose* is being accessed by another opcode (like *fout* or *foutk*, it will be closed later when it is no longer being used.



### Warning

This opcode should be used with care, as the file handle will become invalid, and will cause an init error when an opcode tries to access the closed file.

## See Also

*fout*, *fout*, *fouti*, *foutir*, *foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 5.02



# filelen

filelen -- Returns the length of a sound file.

filelen

## Description

Returns the length of a sound file.

## Syntax

```
ir filelen ifilcod
```

## Initialization

*ifilcod* -- sound file to be queried

## Performance

*filelen* returns the length of the sound file *ifilcod* in seconds.

## Examples

Here is an example of the filelen opcode. It uses the files *filelen.orc* [examples/filelen.orc], *filelen.sco* [examples/filelen.sco], and *mary.wav* [examples/mary.wav].

### Example 124. Example of the filelen opcode.

```
/* filelen.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the length of the audio file
  ; "mary.wav" in seconds.
  ilen filelen "mary.wav"
  print ilen
endin
/* filelen.orc */

/* filelen.sco */
; Play Instrument #1 for 1 second.
i 1 0 1
e
/* filelen.sco */
```

The audio file “mary.wav” is 3.5 seconds long. So *filelen*'s output should include a line like this:

```
instr 1:  ilen = 3.501
```

## See Also

*filenchnls*, *filepeak*, *filesr*

## Credits

Author: Matt Ingalls  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# filenchnls

`filenchnls` -- Returns the number of channels in a sound file.

`filenchnls`

## Description

Returns the number of channels in a sound file.

## Syntax

```
ir filenchnls ifilcod
```

## Initialization

*ifilcod* -- sound file to be queried

## Performance

*filenchnls* returns the number of channels in the sound file *ifilcod*.

## Examples

Here is an example of the `filenchnls` opcode. It uses the files *filenchnls.orc* [examples/filenchnls.orc], *filenchnls.sco* [examples/filenchnls.sco], and *mary.wav* [examples/mary.wav].

### Example 125. Example of the `filenchnls` opcode.

```
/* filenchnls.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the number of channels in the
  ; audio file "mary.wav".
  ichnls filenchnls "mary.wav"
  print ichnls
endin
/* filenchnls.orc */
```

```
/* filenchnls.sco */
; Play Instrument #1 for 1 second.
i 1 0 1
e
/* filenchnls.sco */
```

The audio file “mary.wav” is monoaural (1 channel). So *filenchnls*'s output should include a line like this:

```
instr 1:  ichnls = 1.000
```

## See Also

*filelen*, *filepeak*, *filesr*

## Credits

Author: Matt Ingalls  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# filepeak

filepeak -- Returns the peak absolute value of a sound file.

filepeak

## Description

Returns the peak absolute value of a sound file.

## Syntax

```
ir filepeak ifilcod [, ichnl]
```

## Initialization

*ifilcod* -- sound file to be queried

*ichnl* (optional, default=0) -- 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

*filepeak* returns the peak absolute value of the sound file *ifilcod*. Currently, *filepeak* supports only AIFF-C float files.

## Examples

Here is an example of the filepeak opcode. It uses the files *filepeak.orc* [examples/filepeak.orc], *filepeak.sco* [examples/filepeak.sco], and *mary.wav* [examples/mary.wav].

### Example 126. Example of the filepeak opcode.

```
/* filepeak.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the peak absolute value of the
  ; audio file "mary.wav".
  ipeak filepeak "mary.wav"
  print ipeak
endin
/* filepeak.orc */
```

```
/* filepeak.sco */  
; Play Instrument #1 for 1 second.  
i 1 0 1  
e  
/* filepeak.sco */
```

The peak absolute value of the audio file “mary.wav” is 0.306902. So *filepeak*'s output should include a line like this:

```
instr 1:  ipeak = 0.307
```

## See Also

*filelen*, *filenchnls*, *filesr*

## Credits

Author: Matt Ingalls  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# filesr

filesr -- Returns the sample rate of a sound file.

filesr

## Description

Returns the sample rate of a sound file.

## Syntax

```
ir filesr ifilcod
```

## Initialization

*ifilcod* -- sound file to be queried

## Performance

*filesr* returns the sample rate of the sound file *ifilcod*.

## Examples

Here is an example of the filesr opcode. It uses the files *filesr.orc* [examples/filesr.orc], *filesr.sco* [examples/filesr.sco], and *mary.wav* [examples/mary.wav].

### Example 127. Example of the filesr opcode.

```
/* filesr.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the sampling rate of the
  ; audio file "mary.wav".
  isr filesr "mary.wav"
  print isr
endin
/* filesr.orc */
```

```
/* filesr.sco */
; Play Instrument #1 for 1 second.
i 1 0 1
e
/* filesr.sco */
```

The audio file “mary.wav” was sampled at 44.1 KHz. So *filesr*'s output should include a line like this:

```
instr 1:  isr = 44100.000
```

## See Also

*filelen*, *filenchnls*, *filepeak*

## Credits

Author: Matt Ingalls  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57



## filter2

`filter2` -- Performs filtering using a transposed form-II digital filter lattice with no time-varying control.

`filter2`

## Description

General purpose custom filter with time-varying pole control. The filter coefficients implement the following difference equation:

$$(1)*y(n) = b0*x[n] + b1*x[n-1] + \dots + bM*x[n-M] - a1*y[n-1] - \dots - aN*y[n-N]$$

the system function for which is represented by:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + \dots + bM*Z^{-M}}{1 + a1*Z^{-1} + \dots + aN*Z^{-N}}$$

## Syntax

`ares filter2 asig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN`

`kres filter2 ksig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN`

## 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*.

## Performance

The *filter2* opcodes perform filtering using a transposed form-II digital filter lattice with no time-varying control.

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.

## Examples

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
```

## See Also

*zfilter2*

## Credits

Author: Michael A. Casey  
M.I.T.  
Cambridge, Mass.  
1997

# fin

`fin` -- Read signals from a file at a-rate.

`fin`

## Description

Read signals from a file at a-rate.

## Syntax

```
fin ifilename, iskipframes, iformat, ain1 [, ain2] [, ain3] [...]
```

## 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.

- 0 - 32 bit floating points without header
- 1 - 16 bit integers without header

## 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.



### Note

Please note that since this opcode generates its output using input parameters (on the right side of the opcode), these variables must be initialized before use, otherwise a 'used before defined' error will occur. You can use the *init* opcode for this.

## See Also

*fini*, *fink*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# fini

*fini* -- Read signals from a file at i-rate.

*fini*

## Description

Read signals from a file at i-rate.

## Syntax

```
fini ifilename, iskipframes, iformat, in1 [, in2] [, in3] [, ...]
```

## 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.

- 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

*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 and the end of file is reached, the file pointer is zeroed. This restarts the scan from the beginning. When *iformat* is set to 1 or 2, no looping is enabled and at the end of file the corresponding variables will be filled with zeroes.



### Note

Please note that since this opcode generates its output using input parameters (on the right side of the opcode), these variables must be initialized before use, otherwise a 'used before defined' error will occur. You can use the *init* opcode for this.

## See Also

*fin*, *fink*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# fink

*fink* -- Read signals from a file at k-rate.

*fink*

## Description

Read signals from a file at k-rate.

## Syntax

```
fink ifilename, iskipframes, iformat, kin1 [, kin2] [, kin3] [...]
```

## 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.

- 0 - 32 bit floating points without header
- 1 - 16 bit integers without header

## Performance

*fink* is the same as *fin* but operates at k-rate.



### Note

Please note that since this opcode generates its output using input parameters (on the right side of the opcode), these variables must be initialized before use, otherwise a 'used before defined' error will occur. You can use the *init* opcode for this.

## See Also

*fin*, *fini*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# fiopen

fiopen -- Opens a file in a specific mode.

fiopen

## Description

*fiopen* can be used to open a file in one of the specified modes.

## Syntax

```
ihandle fiopen ifilename, imode
```

## Initialization

*ihandle* -- a number which specifies this file.

*ifilename* -- the output file's name (in double-quotes).

*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

*fiopen* opens a file to be used by the *fout* family of opcodes. It is safer to use it in the header section, external to any instruments. It returns a number, *ihandle*, which unequivocally refers to the opened file.

If *fiopen* is called on an already open file, it just returns the same handle again, and does not close the 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.

## See Also

*ficlose fout, fouti, foutir, foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# flanger

flanger -- A user controlled flanger.

flanger

## Description

A user controlled flanger.

## Syntax

ares **flanger** asig, adel, kfeedback [, imaxd]

## 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 realtime performance. This unit is very similar to *wguide1*, the only difference is *flanger* does not have the lowpass filter.

## Examples

Here is an example of the flanger opcode. It uses the files *flanger.orc* [examples/flanger.orc], *flanger.sco* [examples/flanger.sco], and *beats.wav* [examples/beats.wav].

### Example 128. Example of the flanger opcode.

```
/* flanger.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use the "beat.wav" audio file.
  asig soundin "beats.wav"

  ; Vary the delay amount from 0 to 0.01 seconds.
  adel line 0, p3, 0.01
  kfeedback = 0.7
```

```
; Apply flange to the input signal.
aflang flanger asig, adel, kfeedback

; It can get loud, so clip its amplitude to 30,000.
a1 clip aflang, 1, 30000
out a1
endin
/* flanger.orc */

/* flanger.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* flanger.sco */
```

## Credits

Author: Gabriel Maldonado  
Italy

Example written by Kevin Conder.

New in Csound version 3.49



# flashtxt

flashtxt -- Allows text to be displayed from instruments like sliders

flashtxt

## Description

Allows text to be displayed from instruments like sliders etc. (only on Unix and Windows at present)

## Syntax

**flashtxt** *iwhich*, *String*

## Initialization

*iwhich* -- the number of the window.

*String* -- the string to be displayed.

## Performance

A window is created, identified by the *iwhich* argument, with the text string displayed. If the text is replaced by a number then the window id deleted. Note that the text windows are globally numbered so different instruments can change the text, and the window survives the instance of the instrument.

## Examples

Here is an example of the flashtxt opcode. It uses the files *flashtxt.orc* [examples/flashtxt.orc] and *flashtxt.sco* [examples/flashtxt.sco].

### Example 129. Example of the flashtxt opcode.

```
/* flashtxt.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1
  flashtxt 1, "Instr 1 live"
  ao oscil 4000, 440, 1
  out ao
endin
/* flashtxt.orc */
```

```
/* flashtxt.sco */
; Table 1: an ordinary sine wave.
f 1 0 32768 10 1
```

```
; Play Instrument #1 for three seconds.  
i 1 0 3  
e  
/* flashtxt.sco */
```

# FLbox

FLbox -- A FLTK widget that displays text inside of a box.

FLbox

## Description

A FLTK widget that displays text inside of a box.

## Syntax

```
ihandle FLbox "label", itype, ifont, isize, iwidth, iheight, ix, iy [, image]
```

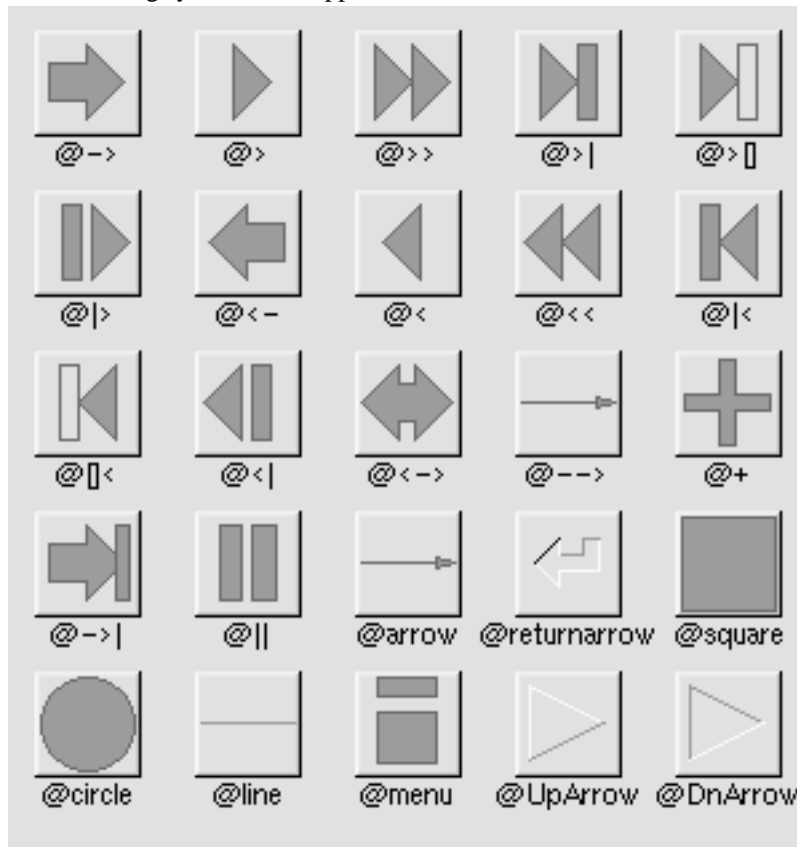
## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLbox* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near corresponding widget.

Notice that with *FLbox*, it is not necessary to call the *FLsetTextType* opcode at all in order to use a symbol. In this case, it is sufficient to set a label starting with "@" followed by the proper formatting string.

The following symbols are supported:



FLTK label supported symbols.

The @ sign may be followed by the following optional “formatting” characters, in this order:

1. “#” forces square scaling rather than distortion to the widget's shape.
2. +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
3. [1-9] rotates by a multiple of 45 degrees. “6” does nothing, the others point in the direction of that key on a numeric keypad.

*itype* -- an integer number denoting the appearance of the widget.

The following values are legal for *itype*:

- 1 - flat box
- 2 - up box
- 3 - down box
- 4 - thin up box
- 5 - thin down box
- 6 - engraved box
- 7 - embossed box
- 8 - border box
- 9 - shadow box
- 10 - rounded box
- 11 - rounded box with shadow
- 12 - rounded flat box
- 13 - rounded up box
- 14 - rounded down box
- 15 - diamond up box
- 16 - diamond down box
- 17 - oval box
- 18 - oval shadow box
- 19 - oval flat box

*ifont* -- an integer number denoting the font of *FLbox*.

*ifont* argument to set the font type. The following values are legal for *ifont*:

- 1 - helvetica (same as "Arial" under Windows)
- 2 - helvetica bold

- 3 - helvetica italic
- 4 - helvetica bold italic
- 5 - courier
- 6 - courier bold
- 7 - courier italic
- 8 - courier bold italic
- 9 - times
- 10 - times bold
- 11 - times italic
- 12 - times bold italic
- 13 - symbol
- 14 - screen
- 15 - screen bold
- 16 - dingbats

*isize* -- size of the font.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of the upper left corner of the valuator, relative to the upper left corner of corresponding window. (Expressed in pixels.)

*iy* -- vertical position of the upper left corner of the valuator, relative to the upper left corner of corresponding window. (Expressed in pixels.)

*image* -- a handle referring to an eventual image opened with *bmopen* opcode. If it is set, it allows a skin for that widget.



### Note about the *bmopen* opcode

Although the documentation mentions the *bmopen* opcode, it has not been implemented in Csound 4.22.

## Performance

*FLbox* is useful to show some text in a window. The text is bounded by a box, whose aspect depends on *itype* argument.

Note that *FLbox* is not a valuator and its value is fixed. Its value cannot be modified.

## Examples

Here is an example of the *FLbox* opcode. It uses the files *FLbox.orc* [examples/FLbox.orc] and *FLbox.sco* [examples/FLbox.sco].

**Example 130. Example of the FLbox opcode.**

```
/* flbox.orc */
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Text Box", 700, 400, 50, 50
; Box border type (7=embossed box)
itype = 7
; Font type (10='Times Bold')
ifont = 10
; Font size
isize = 20
; Width of the flbox
iwidth = 400
; Height of the flbox
iheight = 30
; Distance of the left edge of the flbox
; from the left edge of the panel
ix = 150
; Distance of the upper edge of the flbox
; from the upper edge of the panel
iy = 100

ih3 FLbox "Use Text Boxes For Labelling", itype, ifont, isize, iwidth, iheight
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
endin
/* flbox.orc */

/* flbox.sco */
; Real-time performance for 1 hour.
f 0 3600
e
/* flbox.sco */
```

## See Also

*FLbutBank, FLbutton, FLprintk, FLprintk2, FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLbutBank

FLbutBank -- A FLTK widget opcode that creates a bank of buttons.

FLbutBank

## Description

A FLTK widget opcode that creates a bank of buttons.

## Syntax

`kout, ihandle FLbutBank itype, inumx, inumy, iwidth, iheight, ix, iy, iopcode [`

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLbutBank* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*itype* -- an integer number denoting the appearance of the widget. Its meaning is different for different types of widget.

*inumx* -- number of buttons in each row of the bank.

*inumy* -- number of buttons in each column of the bank

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window, expressed in pixels

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window, expressed in pixels

*iopcode* -- score opcode type. You have to provide the ascii code of the letter corresponding to the score opcode. At present time only "i" (ascii code 105) score statements are supported. A zero value refers to a default value of "i". So both 0 and 105 activates the *i* opcode. A value of -1 disables this opcode feature.

## Performance

*kout* -- output value

*kp1, kp2, ..., kpN* -- arguments of the activated instruments.

The *FLbutBank* opcode creates a bank of buttons. For example, the following line:

```
gkButton,ihbl FLbutBank 12, 8, 8, 380, 180, 50, 350, 0, 7, 0, 0, 5000, 6000
```

will create the this bank:



**FLbutBank.**

A click to a button checks that button. It may also uncheck a previous checked button belonging to the same bank. So the behaviour is always that of radio-buttons. Notice that each button is labeled with a progressive number. The *kout* argument is filled with that number when corresponding button is checked.

*FLbutBank* not only outputs a value but can also activate (or schedule) an instrument provided by the user each time a button is pressed. If the *iopcode* argument is set to a negative number, no instrument is activated so this feature is optional. In order to activate an instrument, *iopcode* must be set to 0 or to 105 (the ascii code of character “i”, referring to the *i* score opcode). P-fields of the activated instrument are *kp1* (instrument number), *kp2* (action time), *kp3* (duration) and so on with user p-fields.

The *itype* argument sets the type of buttons identically to the *FLbutton* opcode. By adding 10 to the *itype* argument (i.e. by setting 11 for type 1, 12 for type 2, 13 for type 3 and 14 for type 4), it is possible to skip the current *FLbutBank* value when getting/setting snapshots (see *General FLTK Widget-related Opcodes*).

*FLbutBank* is very useful to retrieve snapshots.

## See Also

*FLbox*, *FLbutton*, *FLprintk*, *FLprintk2*, *FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22



# FLbutton

FLbutton -- A FLTK widget opcode that creates a button.

FLbutton

## Description

A FLTK widget opcode that creates a button.

## Syntax

kout, ihandle **FLbutton** "label", ion, ioff, itype, iwidth, iheight, ix, iy, iopco

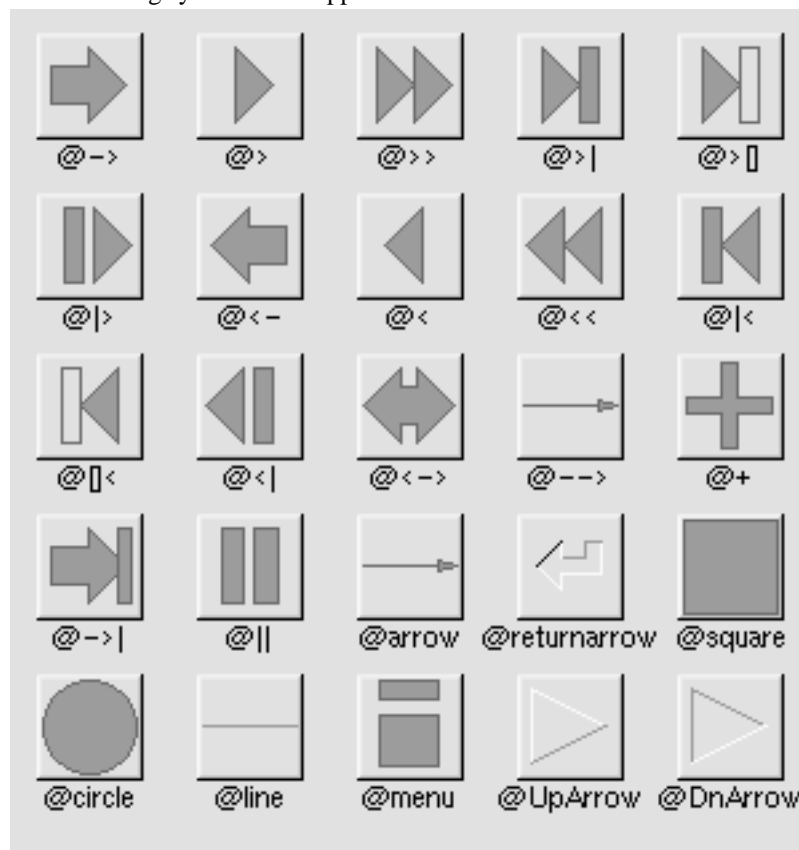
## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLbutton* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

Notice that with *FLbutton*, it is not necessary to call the *FLsetTextType* opcode at all in order to use a symbol. In this case, it is sufficient to set a label starting with "@" followed by the proper formatting string.

The following symbols are supported:



FLTK label supported symbols.

The @ sign may be followed by the following optional “formatting” characters, in this order:

1. “#” forces square scaling rather than distortion to the widget's shape.
2. +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
3. [1-9] rotates by a multiple of 45 degrees. “6” does nothing, the others point in the direction of that key on a numeric keypad.

*ion* -- value output when the button is checked.

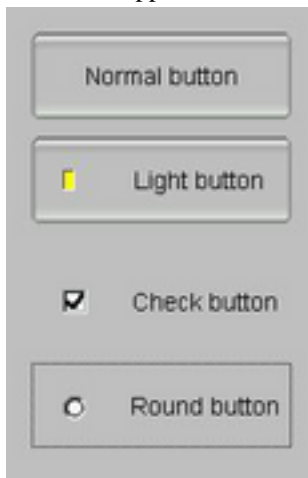
*ioff* -- value output when the button is unchecked.

*itype* -- an integer number denoting the appearance of the widget.

Several kind of buttons are possible, according to the value of *itype* argument:

- 1 - normal button
- 2 - light button
- 3 - check button
- 4 - round button

This is the appearance of the buttons:



FLbutton.

*ewidth* -- width of widget.

*height* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iopcode* -- score opcode type. You have to provide the ascii code of the letter corresponding to the score opcode. At present time only “i” (ascii code 105) score statements are supported. A zero value refers to a default value of “i”. So both 0 and 105 activates the *i* opcode. A value of -1 disables this

opcode feature.

## Performance

*kout* -- output value

*kp1*, *kp2*, ..., *kpN* -- arguments of the activated instruments.



### Note

Please note that FLbuttons of type 1 switch their *kout* values to *ion*, but do not reset to *ioff* (This is done due to synchronization issues between the FLTK window and the csound process), please note that it is the orchestra's job to reset the value to 0 after it has been read.

Buttons of type 2, 3, and 4 also output (*kout* argument) the value contained in the *ion* argument when checked (or lit), and that contained in *ioff* argument when unchecked (or unlit).

By adding 10 to *itype* argument (i.e. by setting 11 for type 1, 12 for type 2, 13 for type 3 and 14 for type 4) it is possible to skip the button value when getting/setting snapshots (see later section). *FLbutton* not only outputs a value, but can also activate (or schedule) an instrument provided by the user each time a button is pressed.

If the *iopcode* argument is set to a negative number, no instrument is activated. So this feature is optional. In order to activate an instrument, *iopcode* must be set to 0 or to 105 (the ascii code of character "i", referring to the *i* score opcode).

P-fields of the activated instrument are *kp1* (instrument number), *kp2* (action time), *kp3* (duration) and so on with user p-fields. Notice that in dual state buttons (light button, check button and round button), the instrument is activated only when button state changes from unchecked to checked (not when passing from checked to unchecked).

## Examples

Here is an example of the FLbutton opcode. It uses the files *FLbutton.orc* [examples/FLbutton.orc], *FLbutton.sco* [examples/FLbutton.sco], and *beats.wav* [examples/beats.wav].

### Example 131. Example of the FLbutton opcode.

```
/* flbutton.orc */
; Using fl-buttons to create on screen controls for play,
; stop, fast forward and fast rewind of a sound file
; This example also makes use of a preset graphic for buttons.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

FLpanel "Buttons", 320, 120, 100, 100
  ion = 0
  ioff = 0
  itype = 1
  iwidth = 50
  iheight = 50
  ix = 50
  iy = 35
  iopcode = 0
  istarttim = 0
  idur = -1

; Normal speed forwards
gkplay, ihbl FLbutton "@>", ion, ioff, itype, iwidth, iheight, ix, iy, iopco
```

```
    ; Stationary
    gkstop, ihb2 FLbutton "@square", ion,ioff, itype, iwidth, iheight, ix+55, i
    ; Double speed backwards
    gkrew, ihb2 FLbutton "@<<", ion, ioff, itype, iwidth, iheight, ix+110, iy, i
    ; Double speed forwards
    gkff, ihb2 FLbutton "@>>", ion, ioff, itype, iwidth, iheight, ix+165, iy, i
FLpanelEnd
FLrun

; Ensure that only 1 instance of instr 1
; plays even if the play button is clicked repeatedly
insnum = 1
icount = 1
maxalloc insnum, icount

instr 1
    asig diskin "beats.wav", p4, 0, 1
    out asig
endin
/* flbutton.orc */

/* flbutton.sco */
; A sine wave
f 1 0 131072 10 1

; Real-time performance for 1 hour.
f 0 3600
e
/* flbutton.sco */
```

## See Also

*FLbox*, *FLbutBank*, *FLprintk*, *FLprintk2*, *FLvalue*.

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLcolor

FLcolor -- A FLTK opcode that sets the primary colors.

FLcolor

## Description

Sets the primary colors to RGB values given by the user.

## Syntax

**FLcolor** *ired*, *igreen*, *ibblue*

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*ibblue* -- The blue color of the target widget. The range for each RGB component is 0-255

## Performance

These opcodes modify the appearance of other widgets. There are two types of such opcodes, those that don't contain the *ihandle* argument which affect all subsequently declared widgets, and those without *ihandle* which affect only a target widget previously defined.

*FLcolor* sets the primary colors to RGB values given by the user. This opcode affects the primary color of (almost) all widgets defined next its location. User can put several instances of *FLcolor* in front of each widget he intend to modify. However, to modify a single widget, it would be better to use the opcode belonging to the second type (i.e. those containing *ihandle* argument).

*FLcolor* is designed to modify the colors of a group of related widgets that assume the same color. The influence of *FLcolor* on subsequent widgets can be turned off by using -1 as the only argument of the opcode. Also, using -2 (or -3) as the only value of *FLcolor* makes all next widget colors randomly selected. The difference is that -2 selects a light random color, while -3 selects a dark random color.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLcolor2

FLcolor2 -- A FLTK opcode that sets the secondary (selection) color.

FLcolor2

## Description

*FLcolor2* is the same of *FLcolor* except it affects the secondary (selection) color.

## Syntax

**FLcolor2** *ired*, *igreen*, *ibblue*

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*ibblue* -- The blue color of the target widget. The range for each RGB component is 0-255

## Performance

These opcodes modify the appearance of other widgets. There are two types of such opcodes: those that don't contain the *ihandle* argument which affect all subsequently declared widgets, and those without *ihandle* which affect only a target widget previously defined.

*FLcolor2* is the same of *FLcolor* except it affects the secondary (selection) color. Setting it to -1 turns off the influence of *FLcolor2* on subsequent widgets. A value of -2 (or -3) makes all next widget secondary colors randomly selected. The difference is that -2 selects a light random color, while -3 selects a dark random color.

## See Also

*FLcolor*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLcount

FLcount -- A FLTK widget opcode that creates a counter.

FLcount

## Description

Allows the user to increase/decrease a value with mouse clicks on a corresponding arrow button.

## Syntax

`kout, ihandle FLcount "label", imin, imax, istep1, istep2, itype, iwidth, iheight`

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. Used by further opcodes that changes some valuator's properties. It is automatically set by the corresponding valuator.

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range

*imax* -- maximum value of output range

*istep1* -- a floating-point number indicating the increment of valuator value corresponding to of each mouse click. *istep1* is for coarse adjustments.

*istep2* -- a floating-point number indicating the increment of valuator value corresponding to of each mouse click. *istep2* is for fine adjustments.

*itype* -- an integer number denoting the appearance of the valuator.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

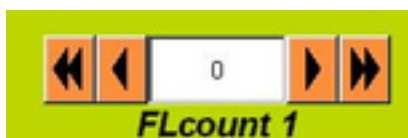
*iopcode* -- score opcode type. You have to provide the ascii code of the letter corresponding to the score opcode. At present time only "i" (ascii code 105) score statements are supported. A zero value refers to a default value of "i". So both 0 and 105 activates the *i* opcode. A value of -1 disables this opcode feature.

## Performance

*kout* -- output value

*kp1, kp2, ..., kpN* -- arguments of the activated instruments.

*FLcount* allows the user to increase/decrease a value with mouse clicks on corresponding arrow buttons:



FLcount.

There are two kind of arrow buttons, for larger and smaller steps. Notice that *FLcount* not only outputs a value and a handle, but can also activate (schedule) an instrument provided by the user each time a button is pressed. P-fields of the activated instrument are *kp1* (instrument number), *kp2* (action time), *kp3* (duration) and so on with user p-fields. If the *iopcode* argument is set to a negative number, no instrument is activated. So this feature is optional.

## Examples

Here is an example of the FLcount opcode. It uses the files *FLcount.orc* [examples/FLcount.orc] and *FLcount.sco* [examples/FLcount.sco].

### Example 132. Example of the FLcount opcode.

```
/* flcount.orc */
; Demonstration of the flcount opcode
; clicking on the single arrow buttons
; increments the oscillator in semitone steps
; clicking on the double arrow buttons
; increments the oscillator in octave steps
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Counter", 900, 400, 50, 50
    ; Minimum value output by counter
    imin = 6
    ; Maximum value output by counter
    imax = 12
    ; Single arrow step size (semitones)
    istep1 = 1/12
    ; Double arrow step size (octave)
    istep2 = 1
    ; Counter type (1=double arrow counter)
    itype = 1
    ; Width of the counter in pixels
    iwidth = 200
    ; Height of the counter in pixels
    iheight = 30
    ; Distance of the left edge of the counter
    ; from the left edge of the panel
    ix = 50
    ; Distance of the top edge of the counter
    ; from the top edge of the panel
    iy = 50
    ; Score event type (-1=ignored)
    iopcode = -1

    gkobj, ihandle FLcount "pitch in oct format", imin, imax, istep1, istep2, i
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    iamp = 15000
    ifn = 1
```



```
        asig oscili iamp, cpsoct(gkcoct), ifn
        out asig
    endin
/* flcount.orc */

/* flcount.sco */
; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
/* flcount.sco */
```

## See Also

*FLjoy, FLkeyb, FLknob, FLroller, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLgetsnap

FLgetsnap -- Retrieves a previously stored FLTK snapshot.

FLgetsnap

## Description

Retrieves a previously stored snapshot (in memory), i.e. sets all valuator to the corresponding values stored in that snapshot.

## Syntax

`inumsnap FLgetsnap index`

## Initialization

*inumsnap* -- current number of snapshots.

*index* -- a number referring unequivocally to a snapshot. Several snapshots can be stored in the same bank.

## Performance

*FLgetsnap* retrieves a previously stored snapshot (in memory), i.e. sets all valuator to the corresponding values stored in that snapshot. The *index* argument unequivocally must refer to an already existing snapshot. If the *index* argument refers to an empty snapshot or to a snapshot that doesn't exist, no action is done. *FLsetsnap* outputs the current number of snapshots (*inumsnap* argument).

## See Also

*FLloadsnap*, *FLrun*, *FLsavesnap*, *FLsetsnap*, *FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLgroup

FLgroup -- A FLTK container opcode that groups child widgets.

FLgroup

## Description

A FLTK container opcode that groups child widgets.

## Syntax

```
FLgroup "label", iwidth, iheight, ix, iy [, iborder] [, image]
```

## Initialization

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iborder* (optional, default=0) -- border type of the container. It is expressed by means of an integer number chosen from the following:

- 0 - no border
- 1 - down box border
- 2 - up box border
- 3 - engraved border
- 4 - embossed border
- 5 - black line border
- 6 - thin down border
- 7 - thin up border

If the integer number doesn't match any of the previous values, no border is provided as the default.

*image* (optional) -- a handle referring to an eventual image opened with the *bmopen* opcode. If it is set, it allows a skin for that widget.



### Note about the *bmopen* opcode

Although the documentation mentions the *bmopen* opcode, it has not been implemented in Csound 4.22.

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLgroupEnd

FLgroupEnd -- Marks the end of a group of FLTK child widgets.

FLgroupEnd

## Description

Marks the end of a group of FLTK child widgets.

## Syntax

**FLgroupEnd**

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLgroupEnd

FLgroupEnd -- Marks the end of a group of FLTK child widgets.

FLgroup\_end

## Description

Marks the end of a group of FLTK child widgets. This is another name for **FLgroupEnd** provides for compatibility. See *FLgroupEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLhide

FLhide -- Hides the target FLTK widget.

FLhide

## Description

Hides the target FLTK widget, making it invisible.

## Syntax

**FLhide** *ihandle*

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLbutBank* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

## Performance

*FLhide* hides target widget, making it invisible.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLjoy

FLjoy -- A FLTK opcode that acts like a joystick.

FLjoy

## Description

*FLjoy* is a squared area that allows the user to modify two output values at the same time. It acts like a joystick.

## Syntax

`koutx, kouty, ihandlex, ihandley FLjoy "label", iminx, imaxx, iminy, imaxy, iexp`

## Initialization

*ihandlex* -- a handle value (an integer number) that unequivocally references a corresponding widget. Used by further opcodes that changes some valuator's properties. It is automatically set by the corresponding valuator.

*ihandley* -- a handle value (an integer number) that unequivocally references a corresponding widget. Used by further opcodes that changes some valuator's properties. It is automatically set by the corresponding valuator.

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iminx* -- minimum x value of output range

*imaxx* -- maximum x value of output range

*iminy* -- minimum y value of output range

*imaxy* -- maximum y value of output range

*iwidth* -- width of widget.

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*idisy* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iexp* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexp* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.

*iexpy* -- an integer number denoting the behaviour of valuator:



- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexpy* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



## IMPORTANT!

Notice that the tables used by valuator must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not placed in the score. In fact, tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*koutx* -- x output value

*kouty* -- y output value

## Examples

Here is an example of the FLjoy opcode. It uses the files *FLjoy.orc* [examples/FLjoy.orc] and *FLjoy.sco* [examples/FLjoy.sco].

### Example 133. Example of the FLjoy opcode.

```
/* fljoy.orc */
; Demonstration of the flpanel opcode
; Horizontal click-dragging controls the frequency of the oscillator
; Vertical click-dragging controls the amplitude of the oscillator
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "X Y Panel", 900, 400, 50, 50
; Minimum value output by x movement (frequency)
iminx = 200
; Maximum value output by x movement (frequency)
imaxx = 5000
; Minimum value output by y movement (amplitude)
iminy = 0
; Maximum value output by y movement (amplitude)
imaxy = 15000
; Logarithmic change in x direction
iexpx = -1
; Linear change in y direction
iexpy = 0
; Display handle x direction (-1=not used)
```

```
idispx = -1
; Display handle y direction (-1=not used)
idispy = -1
; Width of the x y panel in pixels
iwidth = 800
; Height of the x y panel in pixels
iheight = 300
; Distance of the left edge of the x y panel from
; the left edge of the panel
ix = 50
; Distance of the top edge of the x y
; panel from the top edge of the panel
iy = 50

gkfreqx, gkampy, ihandlex, ihandley FLjoy "X - Frequency Y - Amplitude", im
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    ifn = 1
        asig oscili gkampy, gkfreqx, ifn
        out asig
    endin
/* fljoy.orc */

/* fljoy.sco */
; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
/* fljoy.sco */
```

## See Also

*FLcount, FLkeyb, FLknob, FLroller, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLkeyb

FLkeyb -- Experimental, no documentation exists. May be deprecated in future versions.

FLkeyb

## Description

Experimental, no documentation exists. May be deprecated in future versions.

## Syntax

kout **FLkeyb** kparam1 [, kparam2] ... [, kparamN]

## See Also

*FLcount, FLjoy, FLknob, FLroller, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLknob

FLknob -- A FLTK widget opcode that creates a knob.

FLknob

## Description

A FLTK widget opcode that creates a knob.

## Syntax

kout, ihandle **FLknob** "label", imin, imax, iexp, itype, idisp, iwidth, ix, iy[,

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLknob* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range.

*imax* -- maximum value of output range.

*iexp* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexp* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



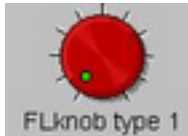
### IMPORTANT!

Notice that the tables used by valuator must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not be placed in the score. In fact, tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

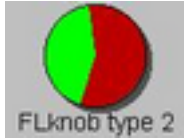
*itype* -- an integer number denoting the appearance of the valuator.

The *itype* argument can be set to the following values:

- 1 - a 3-D knob
- 2 - a pie-like knob
- 3 - a clock-like knob
- 4 - a flat knob



A 3-D knob.



A pie knob.



A clock knob.



A flat knob.

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*icursorsize* (optional) -- If *FLknob's* *itype* is set to 1 (3D knob), this parameter controls the size of knob cursor.

## Performance

*kout* -- output value

*FLknob* puts a knob in the corresponding container.

## Examples

Here is an example of the *FLknob* opcode. It uses the files *FLknob.orc* [examples/FLknob.orc] and *FLknob.sco* [examples/FLknob.sco].

### Example 134. Example of the *FLknob* opcode.

```
/* flknob.orc */  
; A sine with oscillator with flknob controlled frequency
```

```
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Frequency Knob", 900, 400, 50, 50
    ; Minimum value output by the knob
    imin = 200
    ; Maximum value output by the knob
    imax = 5000
    ; Logarithmic type knob selected
    iexp = -1
    ; Knob graphic type (1=3D knob)
    itype = 1
    ; Display handle (-1=not used)
    idisp = -1
    ; Width of the knob in pixels
    iwidth = 70
    ; Height of the knob in pixels
    iheight = 70
    ; Distance of the left edge of the knob
    ; from the left edge of the panel
    ix = 125

    gkfreq, ihandle FLknob "Frequency", imin, imax, iexp, itype, idisp, iwidth,
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    iamp = 15000
    ifn = 1
    asig oscili iamp, gkfreq, ifn
    out asig
endin
/* flknob.orc */

/* flknob.sco */
; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
/* flknob.sco */
```

## See Also

*FLcount, FLjoy, FLkeyb, FLroller, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLlabel

FLlabel -- A FLTK opcode that modifies the appearance of a text label.

FLlabel

## Description

Modifies a set of parameters related to the text label appearance of a widget (i.e. size, font, alignment and color of corresponding text).

## Syntax

**FLlabel** *isize, ifont, ialign, ired, igreen, iblue*

## Initialization

*isize* -- size of the font of the target widget. Normal values are in the order of 15. Greater numbers enlarge font size, while smaller numbers reduce it.

*ifont* -- sets the the font type of the label of a widget.

Legal values for ifont argument are:

- 1 - Helvetica (same as Arial under Windows)
- 2 - Helvetica Bold
- 3 - Helvetica Italic
- 4 - Helvetica Bold Italic
- 5 - Courier
- 6 - Courier Bold
- 7 - Courier Italic
- 8 - Courier Bold Italic
- 9 - Times
- 10 - Times Bold
- 11 - Times Italic
- 12 - Times Bold Italic
- 13 - Symbol
- 14 - Screen
- 15 - Screen Bold
- 16 - Dingbats

*ialign* -- sets the alignment of the label text of the widget.

Legal values for `ialign` argument are:

- 1 - align center
- 2 - align top
- 3 - align bottom
- 4 - align left
- 5 - align right
- 6 - align top-left
- 7 - align top-right
- 8 - align bottom-left
- 9 - align bottom-right

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*ibblue* -- The blue color of the target widget. The range for each RGB component is 0-255

## Performance

*FLlabel* modifies a set of parameters related to the text label appearance of a widget, i.e. size, font, alignment and color of corresponding text. This opcode affects (almost) all widgets defined next its location. A user can put several instances of *FLlabel* in front of each widget he intends to modify. However, to modify a particular widget, it is better to use the opcode belonging to the second type (i.e. those containing the *ihandle* argument).

The influence of *FLlabel* on the next widget can be turned off by using -1 as its only argument. *FLlabel* is designed to modify text attributes of a group of related widgets.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22



# FLloadsnap

FLloadsnap -- Loads all snapshots into the memory bank of the current orchestra.

FLloadsnap

## Description

*FLloadsnap* loads all the snapshots contained in a file into the memory bank of the current orchestra.

## Syntax

**FLloadsnap** "filename"

## Initialization

"filename" -- a double-quoted string corresponding to a file to load a bank of snapshots.

## Performance

*FLloadsnap* loads all snapshots contained in filename into the memory bank of current orchestra.

## See Also

*FLgetsnap*, *FLrun*, *FLsavesnap*, *FLsetsnap*, *FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# flooper

flooper -- Function-table-based crossfading looper.

flooper

## Description

This opcode reads audio from a function table and plays it back in a loop with user-defined start time, duration and crossfade time. It also allows the pitch of the loop to be controlled, including reversed playback. It accepts non-power-of-two tables, such as deferred-allocation GEN01 tables.

## Syntax

```
asig flooper kamp, kpitch, istart, idur, ifad, ifn
```

## Initialisation

*istart* -- loop start pos in seconds

*idur* -- loop duration in seconds

*ifad* -- crossfade duration in seconds

*ifn* -- function table number, generally created using GEN01

## Performance

*asig* -- output sig

*kon* -- amplitude control

*kpitch* -- pitch control (transposition ratio); negative values play the loop back in reverse

## Examples

### Example 135. Example

```
aout flooper 16000, 1, 1, 4, 0.05 ; loop starts at 1 sec, for 4 secs 0.05 cro  
out aout
```

The example above shows the basic operation of flooper. Pitch can be controlled at the k-rate, as well as amplitude. The example assumes the table to contain at least 5.05 seconds of audio (4 secs loop duration, starting 1 sec into the table, using 0.05 secs after the loop end for the crossfade).

## Credits

Author: Victor Lazzarini;  
April 2005

New plugin in version 5

April 2005.

# floor

floor -- Returns the largest integer not greater than  $x$

floor

## Description

Returns the largest integer not greater than  $x$

## Syntax

**floor**( $x$ ) (init-, control-, or audio-rate arg allowed)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Author: Istvan Varga  
New in Csound 5  
2005

# FLpack

FLpack -- Provides the functionality of compressing and aligning FLTK widgets.

FLpack

## Description

*FLpack* provides the functionality of compressing and aligning widgets.

## Syntax

**FLpack** *iwidth, iheight, ix, iy, itype, ispace, iborder*

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*itype* -- an integer number that modifies the appearance of the target widget.

The *itype* argument expresses the type of packing:

- 0 - vertical
- 1 - horizontal

*ispace* -- sets the space between the widgets.

*iborder* -- border type of the container. It is expressed by means of an integer number chosen from the following:

- 0 - no border
- 1 - down box border
- 2 - up box border
- 3 - engraved border
- 4 - embossed border
- 5 - black line border
- 6 - thin down border
- 7 - thin up border

## Performance

*FLpack* provides the functionality of compressing and aligning widgets.

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

## Examples

The following example:

```
FLpanel "Panel1",450,300,100,100
FLpack 400,300, 10,40,0,15,3
gk1,ihs1 FLslider "FLslider 1", 500, 1000, 2 ,1, -1, 300,15, 20,5
gk2,ihs2 FLslider "FLslider 2", 300, 5000, 2 ,3, -1, 300,15, 20,1
gk3,ihs3 FLslider "FLslider 3", 350, 1000, 2 ,5, -1, 300,15, 20,1
gk4,ihs4 FLslider "FLslider 4", 250, 5000, 1 ,11, -1, 300,30, 20,
gk5,ihs5 FLslider "FLslider 5", 220, 8000, 2 ,1, -1, 300,15, 20,2
gk6,ihs6 FLslider "FLslider 6", 1, 5000, 1 ,13, -1, 300,15, 20,30
gk7,ihs7 FLslider "FLslider 7", 870, 5000, 1 ,15, -1, 300,30, 20,
FLpackEnd
FLpanelEnd
```

...will produce this result, when resizing the window:



*FLpack*.

## See Also

*FLgroup, FLgroupEnd, FLpackEnd, FLpanel, FLpanelEnd, FLscroll, FLscrollEnd, FLtabs, FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLpackEnd

FLpackEnd -- Marks the end of a group of compressed or aligned FLTK widgets.

FLpackEnd

## Description

Marks the end of a group of compressed or aligned FLTK widgets.

## Syntax

**FLpackEnd**

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22



## FLpack\_end

FLpack\_end -- Marks the end of a group of compressed or aligned FLTK widgets.

FLpack\_End

## Description

Marks the end of a group of compressed or aligned FLTK widgets. This is another name for **FLpanelEnd** provided for compatibility. See *FLpanel\_end*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLpanel

FLpanel -- Creates a window that contains FLTK widgets.

FLpanel

## Description

Creates a window that contains FLTK widgets.

## Syntax

```
FLpanel "label", iwidth, iheight [, ix] [, iy] [, iborder]
```

## Initialization

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* (optional) -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* (optional) -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iborder* (optional) -- border type of the container. It is expressed by means of an integer number chosen from the following:

- 0 - no border
- 1 - down box border
- 2 - up box border
- 3 - engraved border
- 4 - embossed border
- 5 - black line border
- 6 - thin down border
- 7 - thin up border

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

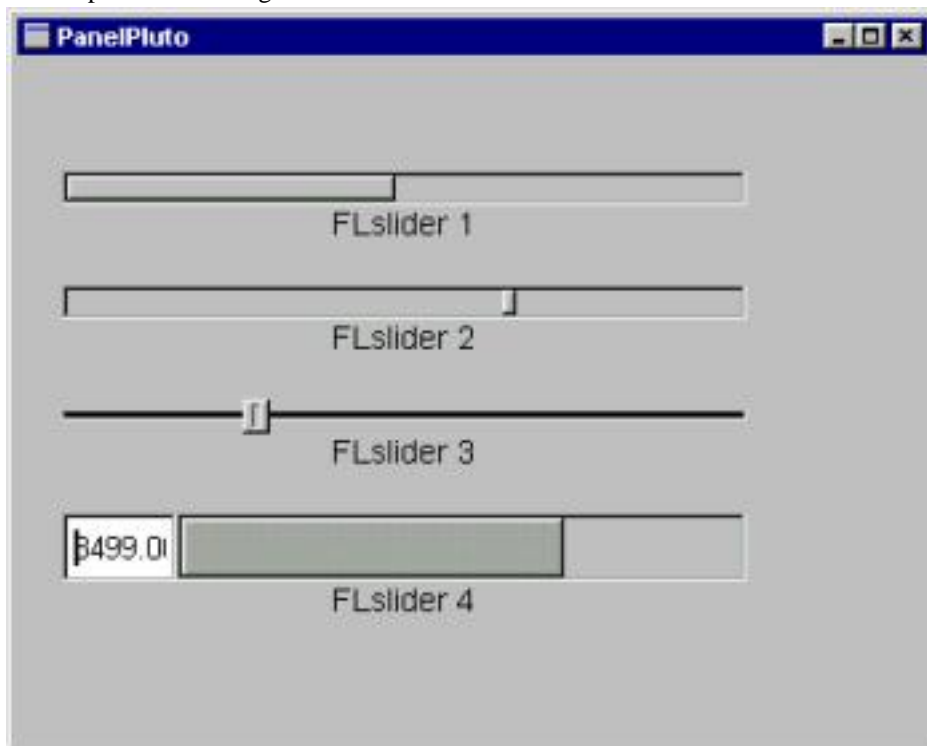
There are no k-rate arguments in containers.

*FLpanel* creates a window. It must be followed by the opcode *FLpanelEnd* when all widgets intern-

al to it are declared. For example:

```
FLpanel    "PanelPluto",450,550,100,100 ;***** start of container
gk1,ih1 FLslider  "FLslider 1", 500, 1000, 2 ,1, -1, 300,15, 20,50
gk2,ih2 FLslider  "FLslider 2", 300, 5000, 2 ,3, -1, 300,15, 20,100
gk3,ih3 FLslider  "FLslider 3", 350, 1000, 2 ,5, -1, 300,15, 20,150
gk4,ih4 FLslider  "FLslider 4", 250, 5000, 1 ,11,-1, 300,30, 20,200
FLpanelEnd ;***** end of container
```

will output the following result:



FLpanel.

## Examples

Here is an example of the FLpanel opcode. It uses the files *FLpanel.orc* [examples/FLpanel.orc] and *FLpanel.sco* [examples/FLpanel.sco].

### Example 136. Example of the FLpanel opcode.

```
/* flpanel.orc */
; Creates an empty window panel
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

; Panel height in pixels
ipanelheight = 900
; Panel width in pixels
ipanelwidth = 400
; Horizontal position of the panel on screen in pixels
ix = 50
```

```
; Vertical position of the panel on screen in pixels
iy = 50

FLpanel "A Window Panel", ipanelheight, ipanelwidth, ix, iy
; End of panel contents
FLpanelEnd

;Run the widget thread!
FLrun

instr 1
endin
/* flpanel.orc */

/* flpanel.sco */
; 'Dummy' score event of 1 hour.
f 0 3600
e
/* flpanel.sco */
```

## See Also

*FLgroup, FLgroupEnd, FLpack, FLpackEnd, FLpanelEnd, FLscroll, FLscrollEnd, FLtabs, FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLpanelEnd

FLpanelEnd -- Marks the end of a group of FLTK widgets contained inside of a window (panel).

FLpanelEnd

## Description

Marks the end of a group of FLTK widgets contained inside of a window (panel).

## Syntax

**FLpanelEnd**

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLscroll*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLpanel\_end

FLpanel\_end -- Marks the end of a group of FLTK widgets contained inside of a window (panel).

FLpanel\_end

## Description

Marks the end of a group of FLTK widgets contained inside of a window (panel). This is another name for **FLpanelEnd** provided for compatibility. See *FLpanelEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLprintk

FLprintk -- A FLTK opcode that prints a k-rate value at specified intervals.

FLprintk

## Description

*FLprintk* is similar to *printk* but shows values of a k-rate signal in a text field instead of on the console.

## Syntax

**FLprintk** *itime*, *kval*, *idisp*

## Initialization

*itime* -- how much time in seconds is to elapse between updated displays.

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

## Performance

*kval* -- k-rate signal to be displayed.

*FLprintk* is similar to *printk*, but shows values of a k-rate signal in a text field instead of showing it in the console. The *idisp* argument must be filled with the *ihandle* return value of a previous *FLvalue* opcode. While *FLvalue* should be placed in the header section of an orchestra inside an *FLpanel/FLpanelEnd* block, *FLprintk* must be placed inside an instrument to operate correctly. For this reason, it slows down performance and should be used for debugging purposes only.

## See Also

*FLbox*, *FLbutBank*, *FLbutton*, *FLprintk2*, *FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLprintk2

FLprintk2 -- A FLTK opcode that prints a new value every time a control-rate variable changes.

FLprintk2

## Description

*FLprintk2* is similar to *FLprintk* but shows a k-rate variable's value only when it changes.

## Syntax

**FLprintk2** kval, idisp

## Initialization

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

## Performance

*kval* -- k-rate signal to be displayed.

*FLprintk2* is similar to *FLprintk*, but shows the k-rate variable's value only each time it changes. Useful for monitoring MIDI control changes when using sliders. It should be used for debugging purposes only, since it slows-down performance.

## See Also

*FLbox*, *FLbutBank*, *FLbutton*, *FLprintk*, *FLvalue*

## Credits

Author: Gabriel Maldonado

New in version 4.22



# FLroller

FLroller -- A FLTK widget that creates a transversal knob.

FLroller

## Description

*FLroller* is a sort of knob, but put transversally.

## Syntax

kout, ihandle **FLroller** "label", imin, imax, istep, iexp, itype, idisp, iwidth,

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLroller* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range.

*imax* -- maximum value of output range.

*istep* -- a floating-point number indicating the increment of valuator value corresponding to of each mouse click. The *istep* argument allows the user to arbitrarily slow roller's motion, enabling arbitrary precision.

*iexp* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexp* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



### IMPORTANT!

Notice that the tables used by valuator must be created with the *figen* opcode and placed in the orchestra before the corresponding valuator. They can not placed in the score. In fact, tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

*itype* -- an integer number denoting the appearance of the valuator.

The *itype* argument can be set to the following values:

- 1 - horizontal roller
- 2 - vertical roller

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*kout* -- output value

*FLroller* is a sort of knob, but put transversally:



FLroller.

## Examples

Here is an example of the FLroller opcode. It uses the files *FLroller.orc* [examples/FLroller.orc] and *FLroller.sco* [examples/FLroller.sco].

### Example 137. Example of the FLroller opcode.

```
/* flroller.orc */
; A sine with oscillator with flroller controlled frequency
sr = 44100
kr = 441
ksmps = 100
nchnls = 1
```

```
FLpanel "Frequency Roller", 900, 400, 50, 50
; Minimum value output by the roller
imin = 200
; Maximum value output by the roller
imax = 5000
; Increment with each pixel
istep = 1
; Logarithmic type roller selected
iexp = -1
; Roller graphic type (1=horizontal)
itype = 1
; Display handle (-1=not used)
idisp = -1
; Width of the roller in pixels
iwidth = 300
; Height of the roller in pixels
iheight = 50
; Distance of the left edge of the knob
; from the left edge of the panel
ix = 300
; Distance of the top edge of the knob
; from the top edge of the panel
iy = 50
```

```
        gkfreq, ihandle FLroller "Frequency", imin, imax, istep, iexp, itype, idisp
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    iamp = 15000
    ifn = 1
    asig oscili iamp, gkfreq, ifn
    out asig
endin
/* flroller.orc */

/* flroller.sco */
; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
/* flroller.sco */
```

## See Also

*FLcount, FLjoy, FLkeyb, FLknob, FLslider, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLrun

FLrun -- Starts the FLTK widget thread.

FLrun

## Description

Starts the FLTK widget thread.

## Syntax

**FLrun**

## Performance

This opcode must be located at the end of all widget declarations. It has no arguments, and its purpose is to start the thread related to widgets. Widgets would not operate if *FLrun* is missing.

## See Also

*FLgetsnap, FLloadsnap, FLsavesnap, FLsetsnap, FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsavesnap

FLsavesnap -- Saves all snapshots currently created into a file.

FLsavesnap

## Description

*FLsavesnap* saves all snapshots currently created (i.e. the entire memory bank) into a file.

## Syntax

**FLsavesnap** "filename"

## Initialization

"filename" -- a double-quoted string corresponding to a file to store a bank of snapshots.

## Performance

*FLsavesnap* saves all snapshots currently created (i.e. the entire memory bank) into a file whose name is *filename*. Since the file is a text file, snapshot values can also be edited manually by means of a text editor. The format of the data stored in the file is the following (at present time, this could be changed in next Csound version):

```
----- 0 -----
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLslider 331.946 80 5000 -1 "frequency of the first oscillator"
FLslider 385.923 80 5000 -1 "frequency of the second oscillator"
FLslider 80 80 5000 -1 "frequency of the third oscillator"
FLcount 0 0 10 0 "this index must point to the location number where snapshot i
FLbutton 0 0 1 0 "Store snapshot to current index"
FLbutton 0 0 1 0 "Save snapshot bank to disk"
FLbutton 0 0 1 0 "Load snapshot bank from disk"
FLbox 0 0 1 0 ""
----- 1 -----
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLvalue 0 0 1 0 ""
FLslider 819.72 80 5000 -1 "frequency of the first oscillator"
FLslider 385.923 80 5000 -1 "frequency of the second oscillator"
FLslider 80 80 5000 -1 "frequency of the third oscillator"
FLcount 1 0 10 0 "this index must point to the location number where snapshot i
FLbutton 0 0 1 0 "Store snapshot to current index"
FLbutton 0 0 1 0 "Save snapshot bank to disk"
FLbutton 0 0 1 0 "Load snapshot bank from disk"
FLbox 0 0 1 0 ""
----- 2 -----
..... etc...
----- 3 -----
..... etc...
-----
```

As you can see, each snapshot contain several lines. Each snapshot is separated from previous and

next snapshot by a line of this kind:

"----- snapshot Num -----"

Then there are several lines containing data. Each of these lines corresponds to a widget.

The first field of each line is an unquoted string containing opcode name corresponding to that widget. Second field is a number that expresses current value of a snapshot. In current version, this is the only field that can be modified manually. The third and fourth fields shows minimum and maximum values allowed for that valuator. The fifth field is a special number that indicates if the valuator is linear (value 0), exponential (value -1), or is indexed by a table interpolating values (negative table numbers) or non-interpolating (positive table numbers). The last field is a quoted string with the label of the widget. Last line of the file is always

"-----"

.

## See Also

*FLgetsnap, FLloadsnap, FLrun, FLsetsnap, FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLscroll

FLscroll -- A FLTK opcode that adds scroll bars to an area.

FLscroll

## Description

*FLscroll* adds scroll bars to an area.

## Syntax

**FLscroll** *iwidth*, *iheight* [, *ix*] [, *iy*]

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* (optional) -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* (optional) -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

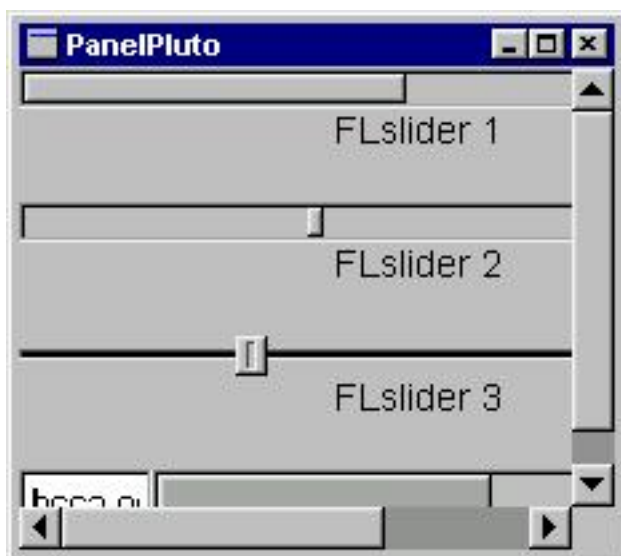
Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

*FLscroll* adds scroll bars to an area. Normally you must set arguments *iwidth* and *iheight* equal to that of the parent window or other parent container. *ix* and *iy* are optional since they normally are set to zero. For example the following code:

```
FLpanel    "PanelPluto", 400, 300, 100, 100
FLscroll   400, 300
gk1, ih1 FLslider "FLslider 1", 500, 1000, 2, 1, -1, 300, 15, 20, 50
gk2, ih2 FLslider "FLslider 2", 300, 5000, 2, 3, -1, 300, 15, 20, 100
gk3, ih3 FLslider "FLslider 3", 350, 1000, 2, 5, -1, 300, 15, 20, 150
gk4, ih4 FLslider "FLslider 4", 250, 5000, 1, 11, -1, 300, 30, 20, 200
FLscrollEnd
FLpanelEnd
```

will show scroll bars, when the main window size is reduced:



FLscroll.

## Examples

Here is an example of the FLscroll opcode. It uses the files *FLscroll.orc* [examples/FLscroll.orc] and *FLscroll.sco* [examples/FLscroll.sco].

### Example 138. Example of the FLscroll opcode.

```
/* flscroll.orc */
; Demonstration of the flscroll opcode which enables
; the use of widget sizes and placings beyond the
; dimensions of the containing panel
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Text Box", 420, 200, 50, 50
    iwidth = 420
    iheight = 200
    ix = 0
    iy = 0
    FLscroll iwidth, iheight, ix, iy
    ih3 FLbox "DRAG THE SCROLL BAR TO THE RIGHT IN ORDER TO READ THE REST OF TH
    FLscrollEnd
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
endin
/* flscroll.orc */

/* flscroll.sco */
; 'Dummy' score event of 1 hour.
f 0 3600
e
```



```
/* flscroll.sco */
```

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscrollEnd*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLscrollEnd

FLscrollEnd -- A FLTK opcode that marks the end of an area with scrollbars.

FLscrollEnd

## Description

A FLTK opcode that marks the end of an area with scrollbars.

## Syntax

**FLscrollEnd**

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLtabs*, *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLscroll\_end

FLscroll\_end -- A FLTK opcode that marks the end of an area with scrollbars.

FLscroll\_end

## Description

A FLTK opcode that marks the end of an area with scrollbars. This is another name for **FLscrollEnd** provided for compatibility. See *FLscrollEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetAlign

FLsetAlign -- Sets the text alignment of a label of a FLTK widget.

FLsetAlign

## Description

*FLsetAlign* sets the text alignment of the label of the target widget.

## Syntax

**FLsetAlign** *ialign*, *ihandle*

## Initialization

*ialign* -- sets the alignment of the label text of widgets.

The legal values for the *ialign* argument are:

- 1 - align center
- 2 - align top
- 3 - align bottom
- 4 - align left
- 5 - align right
- 6 - align top-left
- 7 - align top-right
- 8 - align bottom-left
- 9 - align bottom-right

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetBox

FLsetBox -- Sets the appearance of a box surrounding a FLTK widget.

FLsetBox

## Description

*FLsetBox* sets the appearance of a box surrounding the target widget.

## Syntax

**FLsetBox** *itype*, *ihandle*

## Initialization

*itype* -- an integer number that modify the appearance of the target widget.

Legal values for the *itype* argument are:

- 1 - flat box
- 2 - up box
- 3 - down box
- 4 - thin up box
- 5 - thin down box
- 6 - engraved box
- 7 - embossed box
- 8 - border box
- 9 - shadow box
- 10 - rounded box
- 11 - rounded box with shadow
- 12 - rounded flat box
- 13 - rounded up box
- 14 - rounded down box
- 15 - diamond up box
- 16 - diamond down box
- 17 - oval box
- 18 - oval shadow box
- 19 - oval flat box

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetColor

FLsetColor -- Sets the primary color of a FLTK widget.

FLsetColor

## Description

*FLsetColor* sets the primary color of the target widget.

## Syntax

**FLsetColor** *ired, igreen, iblue, ihandle*

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*iblue* -- The blue color of the target widget. The range for each RGB component is 0-255

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## Examples

Here is an example of the FLsetcolor opcode. It uses the files *FLsetcolor.orc* [examples/FLsetcolor.orc] and *FLsetcolor.sco* [examples/FLsetcolor.sco].

### Example 139. Example of the FLsetcolor opcode.

```
/* flsetcolor.orc */
; Using the opcode flsetcolor to change from the
; default colours for widgets
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Coloured Sliders", 900, 360, 50, 50
  gkfreq, ihandle FLslider "A Red Slider", 200, 5000, -1, 5, -1, 750, 30, 85,
  ired1 = 255
  igreen1 = 0
  iblue1 = 0
  FLsetColor ired1, igreen1, iblue1, ihandle

  gkfreq, ihandle FLslider "A Green Slider", 200, 5000, -1, 5, -1, 750, 30, 85,
  ired1 = 0
  igreen1 = 255
  iblue1 = 0
  FLsetColor ired1, igreen1, iblue1, ihandle

  gkfreq, ihandle FLslider "A Blue Slider", 200, 5000, -1, 5, -1, 750, 30, 85,
  ired1 = 0
```

```
        igreen1 = 0
        iblue1 = 255
        FLsetColor ired1, igreen1, iblue1, ihandle
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
endin
/* flsetcolor.orc */
```

```
/* flsetcolor.sco */
; 'Dummy' score event for 1 hour.
f 0 3600
e
/* flsetcolor.sco */
```

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.



# FLsetColor2

FLsetColor2 -- Sets the secondary (or selection) color of a FLTK widget.

FLsetColor2

## Description

*FLsetColor2* sets the secondary (or selection) color of the target widget.

## Syntax

**FLsetColor2** *ired*, *igreen*, *ibblue*, *ihandle*

## Initialization

*ired* -- The red color of the target widget. The range for each RGB component is 0-255

*igreen* -- The green color of the target widget. The range for each RGB component is 0-255

*ibblue* -- The blue color of the target widget. The range for each RGB component is 0-255

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetFont

FLsetFont -- Sets the font type of a FLTK widget.

FLsetFont

## Description

*FLsetFont* sets the font type of the target widget.

## Syntax

**FLsetFont** ifont, ihandle

## Initialization

*ifont* -- sets the the font type of the label of a widget.

Legal values for ifont argument are:

- 1 - Helvetica (same as Arial under Windows)
- 2 - Helvetica Bold
- 3 - Helvetica Italic
- 4 - Helvetica Bold Italic
- 5 - Courier
- 6 - Courier Bold
- 7 - Courier Italic
- 8 - Courier Bold Italic
- 9 - Times
- 10 - Times Bold
- 11 - Times Italic
- 12 - Times Bold Italic
- 13 - Symbol
- 14 - Screen
- 15 - Screen Bold
- 16 - Dingbats

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2, FLhide, FLlabel, FLsetAlign, FLsetBox, FLsetColor, FLsetColor2, FLsetFont, FLsetPosition, FLsetSize, FLsetText, FLsetTextColor, FLsetTextSize, FLsetTextType, FLsetVal\_i, FLsetVal, FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetPosition

FLsetPosition -- Sets the position of a FLTK widget.

FLsetPosition

## Description

*FLsetPosition* sets the position of the target widget according to the *ix* and *iy* arguments.

## Syntax

**FLsetPosition** *ix*, *iy*, *ihandle*

## Initialization

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetSize

FLsetSize -- Resizes a FLTK widget.

FLsetSize

## Description

*FLsetSize* resizes the target widget (not the size of its text) according to the *iwidth* and *iheight* arguments.

## Syntax

**FLsetSize** *iwidth*, *iheight*, *ihandle*

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetsnap

FLsetsnap -- Stores the current status of all FLTK valutors into a snapshot location.

FLsetsnap

## Description

*FLsetsnap* stores the current status of all valutors present in the orchestra into a snapshot location (in memory).

## Syntax

```
inumsnap, inumval FLsetsnap index [, ifn]
```

## Initialization

*inumsnap* -- current number of snapshots.

*inumval* -- number of valutors (whose value is stored in a snapshot) present in current orchestra.

*index* -- a number referring unequivocally to a snapshot. Several snapshots can be stored in the same bank.

*ifn* (optional) -- optional argument referring to an already allocated table, to store values of a snapshot.

## Performance

The *FLsetsnap* opcode stores current status of all valutors present in the orchestra into a snapshot location (in memory). Any number of snapshots can be stored in the current bank. Banks are structures that only exist in memory, there are no other reference to them other that they can be accessed by *FLsetsnap*, *FLsavesnap*, *FLloadsnap* and *FLgetsnap* opcodes. Only a single bank can be present in memory.

If the optional *ifn* argument refers to an already allocated and valid table, the snapshot will be stored in the table instead of in the bank. So that table can be accessed from other Csound opcodes.

The *index* argument unequivocally refers to a determinate snapshot. If the value of *index* refers to a previously stored snapshot, all its old values will be replaced with current ones. If *index* refers to a snapshot that doesn't exist, a new snapshot will be created. If the *index* value is not adjacent with that of a previously created snapshot, some empty snapshots will be created. For example, if a location with *index* 0 contains the only and unique snapshot present in a bank and the user stores a new snapshot using *index* 5, all locations between 1 and 4 will automatically contain empty snapshots. Empty snapshots don't contain any data and are neutral.

*FLsetsnap* outputs the current number of snapshots (the *inumsnap* argument) and the total number of values stored in each snapshot (*inumval*). *inumval* is equal to the number of valutors present in the orchestra.

## See Also

*FLgetsnap*, *FLloadsnap*, *FLrun*, *FLsavesnap*, *FLupdate*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetText

FLsetText -- Sets the label of a FLTK widget.

FLsetText

## Description

*FLsetText* sets the label of the target widget to the double-quoted text string provided with the *itext* argument.

## Syntax

**FLsetText** "itext", ihandle

## Initialization

*"itext"* -- a double-quoted string denoting the text of the label of the widget.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22



# FLsetTextColor

FLsetTextColor -- Sets the color of the text label of a FLTK widget.

FLsetTextColor

## Description

*FLsetTextColor* sets the color of the text label of the target widget.

## Syntax

**FLsetTextColor** *isize, ihandle*

## Initialization

*isize* -- size of the font of the target widget. Normal values are in the order of 15. Greater numbers enlarge font size, while smaller numbers reduce it.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2, FLhide, FLlabel, FLsetAlign, FLsetBox, FLsetColor, FLsetColor2, FLsetFont, FLsetPosition, FLsetSize, FLsetText, FLsetTextColor, FLsetTextSize, FLsetTextType, FLsetVal\_i, FLsetVal, FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetTextSize

FLsetTextSize -- Sets the size of the text label of a FLTK widget.

FLsetTextSize

## Description

*FLsetTextSize* sets the size of the text label of the target widget.

## Syntax

**FLsetTextSize** *isize*, *ihandle*

## Initialization

*isize* -- size of the font of the target widget. Normal values are in the order of 15. Greater numbers enlarge font size, while smaller numbers reduce it.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetTextType

FLsetTextType -- Sets some font attributes of the text label of a FLTK widget.

FLsetTextType

## Description

*FLsetTextType* sets some attributes related to the fonts of the text label of the target widget.

## Syntax

**FLsetTextType** *itype*, *ihandle*

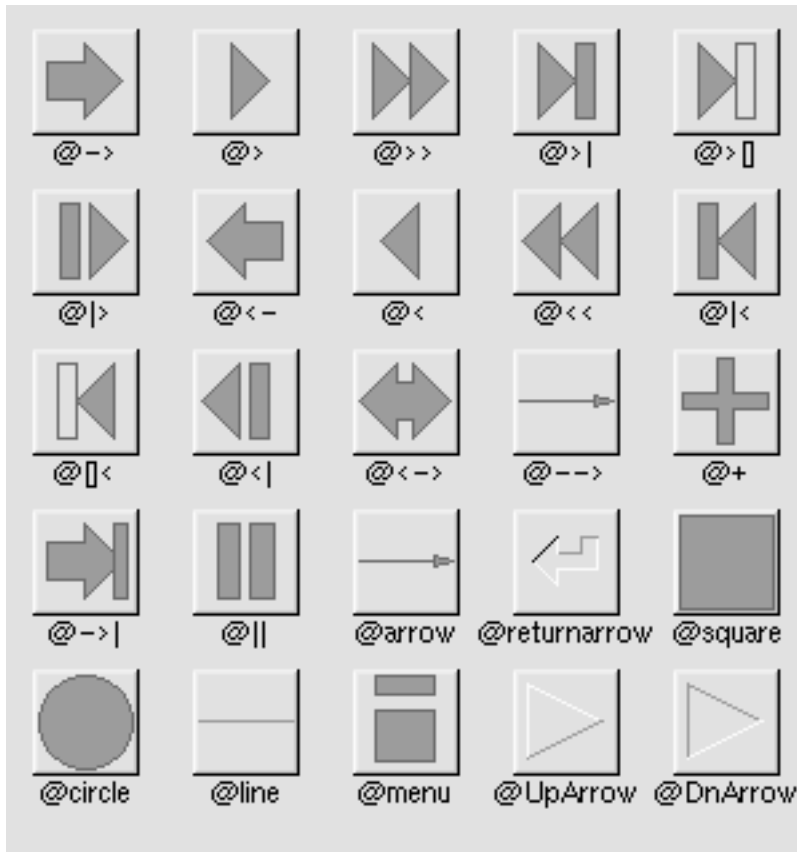
## Initialization

*itype* -- an integer number that modify the appearance of the target widget.

The legal values of *itype* are:

- 0 - normal label
- 1 - no label (hides the text)
- 2 - symbol label (see below)
- 3 - shadow label
- 4 - engraved label
- 5- embossed label
- 6- bitmap label (not implemented yet)
- 7- pixmap label (not implemented yet)
- 8- image label (not implemented yet)
- 9- multi label (not implemented yet)
- 10- free-type label (not implemented yet)

When using *itype*=3 (symbol label), it is possible to assign a graphical symbol instead of the text label of the target widget. In this case, the string of the target label must always start with “@”. If it starts with something else (or the symbol is not found), the label is drawn normally. The following symbols are supported:



FLTK label supported symbols.

The @ sign may be followed by the following optional “formatting” characters, in this order:

1. “#” forces square scaling rather than distortion to the widget's shape.
2. +[1-9] or -[1-9] tweaks the scaling a little bigger or smaller.
3. [1-9] rotates by a multiple of 45 degrees. “6” does nothing, the others point in the direction of that key on a numeric keypad.

Notice that with *FLbox* and *FLbutton*, it is not necessary to call *FLsetTextType* opcode at all in order to use a symbol. In this case, it is sufficient to set a label starting with “@” followed by the proper formatting string.

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetVal\_i

FLsetVal\_i -- Sets the value of a FLTK valuator to a number provided by the user.

FLsetVal\_i

## Description

*FLsetVal\_i* forces the value of a valuator to a number provided by the user.

## Syntax

**FLsetVal\_i** kvalue, ihandle

## Initialization

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## Performance

*kvalue* -- not implemented yet.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLsetVal

FLsetVal -- Sets the value of a FLTK valuator at control-rate.

FLsetVal

## Description

*FLsetVal* is almost identical to *FLsetVal\_i*. Except it operates at k-rate and it affects the target valuator only when *ktrig* is set to a non-zero value.

## Syntax

**FLsetVal** *ktrig*, *kvalue*, *ihandle*

## Initialization

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## Performance

*ktrig* -- not implemented yet.

*kvalue* -- not implemented yet.

## See Also

*FLcolor*, *FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLshow

FLshow -- Restores the visibility of a previously hidden FLTK widget.

FLshow

## Description

*FLshow* restores the visibility of a previously hidden widget.

## Syntax

**FLshow** *ihandle*

## Initialization

*ihandle* -- an integer number (used as unique identifier) taken from the output of a previously located widget opcode (which corresponds to the target widget). It is used to unequivocally identify the widget when modifying its appearance with this class of opcodes. The user must not set the *ihandle* value directly, otherwise a Csound crash will occur.

## See Also

*FLcolor2*, *FLhide*, *FLlabel*, *FLsetAlign*, *FLsetBox*, *FLsetColor*, *FLsetColor2*, *FLsetFont*, *FLsetPosition*, *FLsetSize*, *FLsetText*, *FLsetTextColor*, *FLsetTextSize*, *FLsetTextType*, *FLsetVal\_i*, *FLsetVal*, *FLshow*

## Credits

Author: Gabriel Maldonado

New in version 4.22



# FLslidBnk

FLslidBnk -- A FLTK widget containing a bank of horizontal sliders.

FLslidBnk

## Description

*FLslidBnk* is a widget containing a bank of horizontal sliders.

## Syntax

**FLslidBnk** "names", inumsliders [, ioutable] [, iwidth] [, iheight] [, ix] [, iy]

## Initialization

*"names"* -- a double-quoted string containing the names of each slider. Each slider can have a different name. Separate each name with "@" character, for example: "frequency@amplitude@cutoff". It is possible to not provide any name by giving a single space ". In this case, the opcode will automatically assign a progressive number as a label for each slider.

*inumsliders* -- the number of sliders.

*ioutable* (optional, default=0) -- number of a previously-allocated table in which to store output values of each slider. The user must be sure that table size is large enough to contain all output cells, otherwise a segfault will crash Csound. By assigning zero to this argument, the output will be directed to the zak space in the k-rate zone. In this case, the zak space must be previously allocated with the *zakinit* opcode and the user must be sure that the allocation size is big enough to cover all sliders. The default value is zero (i.e. store output in zak space).

*istart\_index* (optional, default=0) -- an integer number referring to a starting offset of output cell locations. It can be positive to allow multiple banks of sliders to output in the same table or in the zak space. The default value is zero (no offset).

*iminmaxtable* (optional, default=0) -- number of a previously-defined table containing a list of min-max pairs, referred to each slider. A zero value defaults to the 0 to 1 range for all sliders without necessity to provide a table. The default value is zero.

*ixptable* (optional, default=0) -- number of a previously-defined table containing a list of identifiers (i.e. integer numbers) provided to modify the behaviour of each slider independently. Identifiers can assume the following values:

- -1 -- exponential curve response
- 0 -- linear response
- number > than 0 -- follow the curve of a previously-defined table to shape the response of the corresponding slider. In this case, the number corresponds to table number.

You can assume that all sliders of the bank have the same response curve (exponential or linear). In this case, you can assign -1 or 0 to *ixptable* without worrying about previously defining any table. The default value is zero (all sliders have a linear response, without having to provide a table).

*ityetable* (optional, default=0) -- number of a previously-defined table containing a list of identifiers (i.e. integer numbers) provided to modify the aspect of each individual slider independently. Identifiers can assume the following values:

- 0 = Nice slider
- 1 = Fill slider
- 3 = Normal slider
- 5 = Nice slider
- 7 = Nice slider with down-box

You can assume that all sliders of the bank have the same aspect. In this case, you can assign a negative number to *ityetable* without worrying about previously defining any table. Negative numbers have the same meaning of the corresponding positive identifiers with the difference that the same aspect is assigned to all sliders. You can also assign a random aspect to each slider by setting *ityetable* to a negative number lower than -7. The default value is zero (all sliders have the aspect of nice sliders, without having to provide a table).

*iwidth* (optional) -- width of the rectangular area containing all sliders of the bank, excluding text labels, that are placed to the left of that area.

*iheight* (optional) -- height of the rectangular area containing all sliders of the bank, excluding text labels, that are placed to the left of that area.

*ix* (optional) -- horizontal position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, at the left of that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

*iy* (optional) -- vertical position of the upper left corner of the rectangular area containing all sliders belonging to the bank. You have to leave enough space, at the left of that rectangle, in order to make sure labels of sliders to be visible. This is because the labels themselves are external to the rectangular area.

## Performance

There are no k-rate arguments, even if cells of the output table (or the zak space) are updated at k-rate.

*FLslidBnk* is a widget containing a bank of horizontal sliders. Any number of sliders can be placed into the bank (*inumsliders* argument). The output of all sliders is stored into a previously allocated table or into the zak space (*ioutable* argument). It is possible to determine the first location of the table (or of the zak space) in which to store the output of the first slider by means of *istart\_index* argument.

Each slider can have an individual label that is placed to the left of it. Labels are defined by the “*names*” argument. The output range of each slider can be individually set by means of an external table (*iminmaxtable* argument). The curve response of each slider can be set individually, by means of a list of identifiers placed in a table (*ixptable* argument). It is possible to define the aspect of each slider independently or to make all sliders have the same aspect (*ityetable* argument).

The *iwidth*, *iheight*, *ix*, and *iy* arguments determine width, height, horizontal and vertical position of the rectangular area containing sliders. Notice that the label of each slider is placed to the left of them and is not included in the rectangular area containing sliders. So the user should leave enough space to the left of the bank by assigning a proper *ix* value in order to leave labels visible.

## See Also

*FLslider*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLslider

FLslider -- Puts a slider into the corresponding FLTK container.

FLslider

## Description

*FLslider* puts a slider into the corresponding container.

## Syntax

kout, ihandle **FLslider** "label", imin, imax, iexp, itype, idisp, iwidth, iheight

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLslider* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*imin* -- minimum value of output range.

*imax* -- maximum value of output range.

The *imin* argument may be greater than *imax* argument. This has the effect of “reversing” the object so the larger values are in the opposite direction. This also switches which end of the filled sliders is filled.

*iexp* -- an integer number denoting the behaviour of valuator:

- 0 = valuator output is linear
- -1 = valuator output is exponential

All other positive numbers for *iexp* indicate the number of an existing table that is used for indexing. Linear interpolation is provided in table indexing. A negative table number suppresses interpolation.



### IMPORTANT!

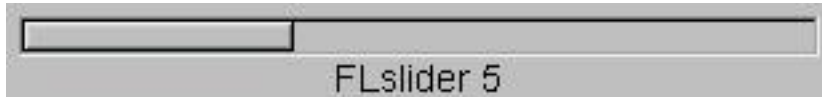
Notice that the tables used by valuator must be created with the *ftgen* opcode and placed in the orchestra before the corresponding valuator. They can not be placed in the score. In fact, tables placed in the score are created later than the initialization of the opcodes placed in the header section of the orchestra.

*itype* -- an integer number denoting the appearance of the valuator.

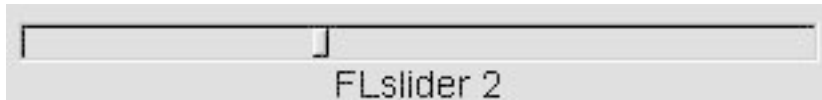
The *itype* argument can be set to the following values:

- 1 - shows a horizontal fill slider
- 2 - a vertical fill slider

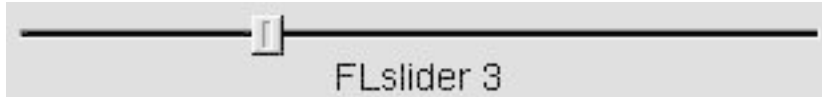
- 3 - a horizontal engraved slider
- 4 - a vertical engraved slider
- 5 - a horizontal nice slider
- 6 - a vertical nice slider
- 7 - a horizontal up-box nice slider
- 8 - a vertical up-box nice slider



FLslider - a horizontal fill slider (itype=1).



FLslider - a horizontal engraved slider (itype=3).



FLslider - a horizontal nice slider (itype=5).

*idisp* -- a handle value that was output from a previous instance of the *FLvalue* opcode to display the current value of the current valuator in the *FLvalue* widget itself. If the user doesn't want to use this feature that displays current values, it must be set to a negative number by the user.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*kout* -- output value

## Examples

Here is an example of the FLslider opcode. It uses the files *FLslider.orc* [examples/FLslider.orc] and *FLslider.sco* [examples/FLslider.sco].

### Example 140. Example of the FLslider opcode.

```
/* flslider.orc */
; A sine with oscillator with flslider controlled frequency
sr = 44100
kr = 441
ksmps = 100
```

```
nchnls = 1

FLpanel "Frequency Slider", 900, 400, 50, 50
; Minimum value output by the slider
imin = 200
; Maximum value output by the slider
imax = 5000
; Logarithmic type slider selected
iexp = -1
; Slider graphic type (5='nice' slider)
itype = 5
; Display handle (-1=not used)
idisp = -1
; Width of the slider in pixels
iwidth = 750
; Height of the slider in pixels
iheight = 30
; Distance of the left edge of the slider
; from the left edge of the panel
ix = 125
; Distance of the top edge of the slider
; from the top edge of the panel
iy = 50

    gkfreq, ihandle FLslider "Frequency", imin, imax, iexp, itype, idisp, iwidth
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    iamp = 15000
    ifn = 1
    asig oscili iamp, gkfreq, ifn
    out asig
endin
/* flslider.orc */

/* flslider.sco */
; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
/* flslider.sco */
```

## See Also

*FLcount, FLjoy, FLkeyb, FLknob, FLroller, FLslidBnk, FLtext*

## Credits

Author: Gabriel Maldonado

New in version 4.22

February 2004. Thanks to a note from Dave Phillips, deleted the extraneous istep parameter.

Example written by Iain McCurdy, edited by Kevin Conder.

# FLtabs

FLtabs -- Creates a tabbed FLTK interface.

FLtabs

## Description

*FLtabs* is the “file card tabs” interface that allows useful to display several areas containing widgets in the same windows, alternatively. It must be used together with *FLgroup*, another container that groups child widgets.

## Syntax

**FLtabs** *iwidth, iheight, ix, iy*

## Initialization

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window. Expressed in pixels.

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window. Expressed in pixels.

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

*FLtabs* is a “file card tabs” interface that is useful to display several alternate areas containing widgets in the same window.



FLtabs.

It must be used together with *FLgroup*, another FLTK container opcode that groups child widgets.

## Examples

The following example code:

```
FLpanel "Panel1",450,550,100,100
FLscroll 450,550,0,0
```



```

        FLtabs 400,550, 5,5
        FLgroup "sliders",380,500, 10,40,1
gk1,ihs FLslider "FLslider 1", 500, 1000, 2 ,1, -1, 300,15, 20,50
gk2,ihs FLslider "FLslider 2", 300, 5000, 2 ,3, -1, 300,15, 20,100
gk3,ihs FLslider "FLslider 3", 350, 1000, 2 ,5, -1, 300,15, 20,150
gk4,ihs FLslider "FLslider 4", 250, 5000, 1 ,11, -1, 300,30, 20,200
gk5,ihs FLslider "FLslider 5", 220, 8000, 2 ,1, -1, 300,15, 20,250
gk6,ihs FLslider "FLslider 6", 1, 5000, 1 ,13, -1, 300,15, 20,300
gk7,ihs FLslider "FLslider 7", 870, 5000, 1 ,15, -1, 300,30, 20,350
gk8,ihs FLslider "FLslider 8", 20, 20000, 2 ,6, -1, 30,400, 350,50
        FLgroupEnd

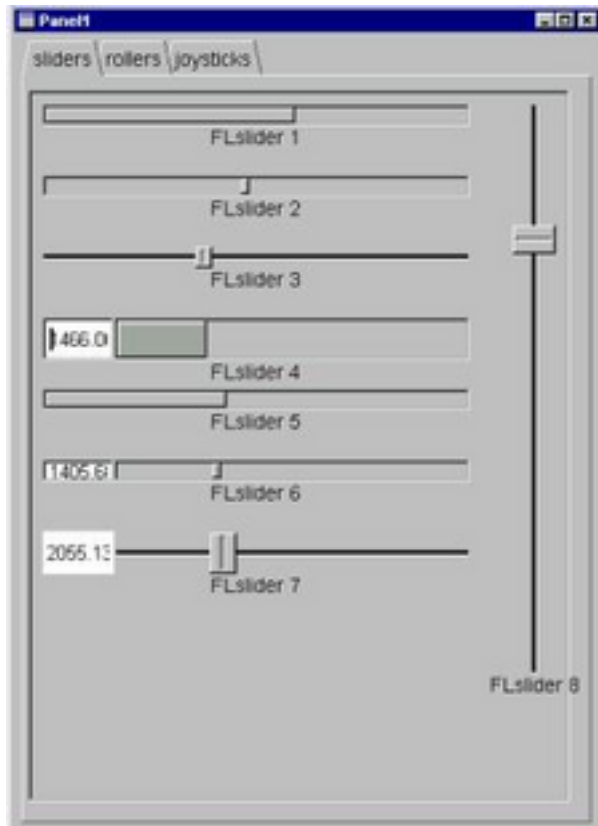
        FLgroup "rollers",380,500, 10,30,2
gk1,ihr FLroller "FLroller 1", 50, 1000,.1,2 ,1 , -1, 200,22, 20,50
gk2,ihr FLroller "FLroller 2", 80, 5000,1,2 ,1 , -1, 200,22, 20,100
gk3,ihr FLroller "FLroller 3", 50, 1000,.1,2 ,1 , -1, 200,22, 20,150
gk4,ihr FLroller "FLroller 4", 80, 5000,1,2 ,1 , -1, 200,22, 20,200
gk5,ihr FLroller "FLroller 5", 50, 1000,.1,2 ,1 , -1, 200,22, 20,250
gk6,ihr FLroller "FLroller 6", 80, 5000,1,2 ,1 , -1, 200,22, 20,300
gk7,ihr FLroller "FLroller 7",50, 5000,1,1 ,2 , -1, 30,300, 280,50
        FLgroupEnd

        FLgroup "joysticks",380,500, 10,40,3
gk1,gk2,ihj1,ihj2 FLjoy "FLjoy", 50, 18000, 50, 18000,2,2,-1,-1,300,300,30,60
        FLgroupEnd

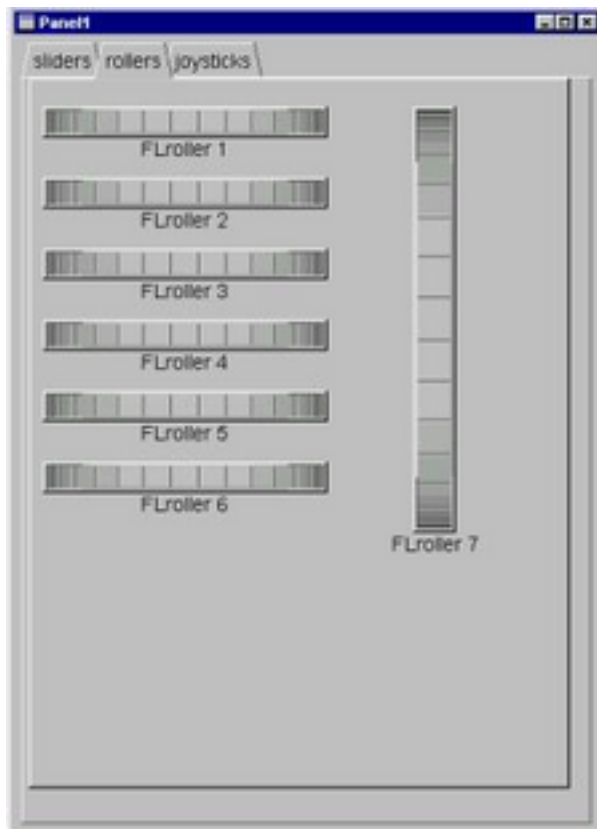
        FLtabsEnd
        FLscrollEnd
        FLpanelEnd

```

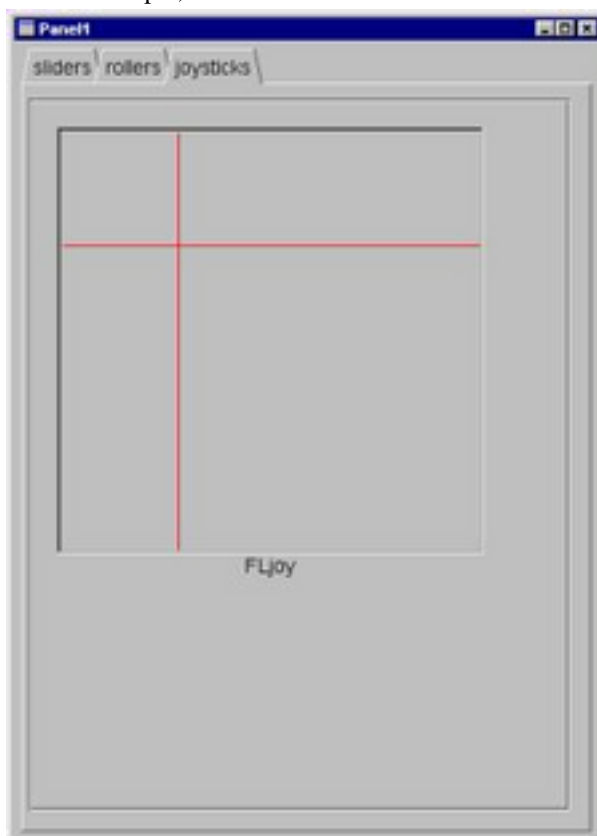
...will produce the following result:



FLtabs example, sliders tab.



FLtabs example, rollers tab.



FLtabs example, joysticks tab.  
(Each picture shows a different tab selection inside the same window.)

## Examples

Here is an example of the FLtabs opcode. It uses the files *FLtabs.orc* [examples/FLtabs.orc] and *FLtabs.sco* [examples/FLtabs.sco].

### Example 141. Example of the FLtabs opcode.

```
/* fltabs.orc */
; A single oscillator with frequency, amplitude and
; panning controls on separate file tab cards
sr = 44100
kr = 441
ksmps = 100
nchnls = 2

FLpanel "Tabs", 300, 350, 100, 100
itabswidth = 280
itabsheight = 330
ix = 5
iy = 5
FLtabs itabswidth,itabsheight, ix,iy

    itablwidth = 280
    itablheight = 300
    itablx = 10
    itably = 40
    FLgroup "Tab 1", itablwidth, itablheight, itablx, itably
        gkfreq, i1 FLknob "Frequency", 200, 5000, -1, 1, -1, 70, 70, 130
        FLsetVal_i 400, i1
    FLgroupEnd

    itab2width = 280
    itab2height = 300
    itab2x = 10
    itab2y = 40
    FLgroup "Tab 2", itab2width, itab2height, itab2x, itab2y
        gkamp, i2 FLknob "Amplitude", 0, 15000, 0, 1, -1, 70, 70, 130
        FLsetVal_i 15000, i2
    FLgroupEnd

    itab3width = 280
    itab3height = 300
    itab3x = 10
    itab3y = 40
    FLgroup "Tab 3", itab3width, itab3height, itab3x, itab3y
        gkpan, i3 FLknob "Pan position", 0, 1, 0, 1, -1, 70, 70, 130
        FLsetVal_i 0.5, i3
    FLgroupEnd
FLtabsEnd
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    ifn = 1
    asig oscili gkamp, gkfreq, ifn
    outs asig*(1-gkpan), asig*gkpan
endin
/* fltabs.orc */

/* fltabs.sco */
; Function table that defines a single cycle
```

---

```
; of a sine wave.  
f 1 0 1024 10 1  
  
; Instrument 1 will play a note for 1 hour.  
i 1 0 3600  
e  
/* fltabs.sco */
```

## See Also

*FLgroup, FLgroupEnd, FLpack, FLpackEnd, FLpanel, FLpanelEnd, FLscroll, FLscrollEnd, FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLtabsEnd

FLtabsEnd -- Marks the end of a tabbed FLTK interface.

FLtabsEnd

## Description

Marks the end of a tabbed FLTK interface.

## Syntax

**FLtabsEnd**

## Performance

Containers are useful to format the graphic appearance of the widgets. The most important container is *FLpanel*, that actually creates a window. It can be filled with other containers and/or valuator or other kinds of widgets.

There are no k-rate arguments in containers.

## See Also

*FLgroup*, *FLgroupEnd*, *FLpack*, *FLpackEnd*, *FLpanel*, *FLpanelEnd*, *FLscroll*, *FLscrollEnd*, *FLtabs*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLtabs\_end

FLtabs\_end -- Marks the end of a tabbed FLTK interface.

FLtabs\_end

## Description

Marks the end of a tabbed FLTK interface. This is another name for **FLtabsEnd** provided for compatibility. See *FLtabsEnd*

## Credits

Author: Gabriel Maldonado

New in version 4.22

# FLtext

FLtext -- A FLTK widget opcode that creates a textbox.

FLtext

## Description

FLtext allows the user to modify a parameter value by directly typing it into a text field.

## Syntax

kout, ihandle **FLtext** "label", imin, imax, istep, itype, iwidth, iheight, ix, iy

## Initialization

*ihandle* -- a handle value (an integer number) that unequivocally references a corresponding widget. This is used by other opcodes that modify a widget's properties (see *Modifying FLTK Widget Appearance*). It is automatically output by *FLtext* and must not be set by the user label. (The user label is a double-quoted string containing some user-provided text placed near the widget.)

*"label"* -- a double-quoted string containing some user-provided text, placed near corresponding widget.

*imin* -- minimum value of output range.

*imax* -- maximum value of output range.

*istep* -- a floating-point number indicating the increment of valuator value corresponding to of each mouse click. The *istep* argument allows the user to arbitrarily slow roller's motion, enabling arbitrary precision.

*itype* -- an integer number denoting the appearance of the valuator.

The *itype* argument can be set to the following values:

- 1 - normal behaviour
- 2 - dragging operation is suppressed, instead it will appear two arrow buttons. A mouse-click on one of these buttons can increase/decrease the output value.
- 3 - text editing is suppressed, only mouse dragging modifies the output value.

*iwidth* -- width of widget.

*iheight* -- height of widget.

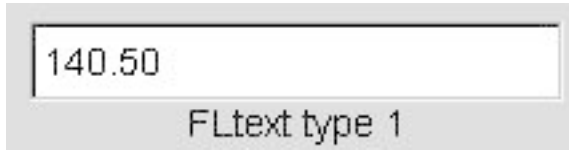
*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

*kout* -- output value

*FLtext* allows the user to modify a parameter value by directly typing it into a text field:



*FLtext*.

Its value can also be modified by clicking on it and dragging the mouse horizontally. The *istep* argument allows the user to arbitrarily set the response on mouse dragging.

## Examples

Here is an example of the *FLtext* opcode. It uses the files *FLtext.orc* [examples/*FLtext.orc*] and *FLtext.sco* [examples/*FLtext.sco*].

### Example 142. Example of the *FLtext* opcode.

```
/* fltext.orc */
; A sine with oscillator with fltext box controlled
; frequency either click and drag or double click and
; type to change frequency value
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Frequency Text Box", 270, 600, 50, 50
    ; Minimum value output by the text box
    imin = 200
    ; Maximum value output by the text box
    imax = 5000
    ; Step size
    istep = 1
    ; Text box graphic type
    itype = 1
    ; Width of the text box in pixels
    iwidth = 70
    ; Height of the text box in pixels
    iheight = 30
    ; Distance of the left edge of the text box
    ; from the left edge of the panel
    ix = 100
    ; Distance of the top edge of the text box
    ; from the top edge of the panel
    iy = 300

    gkfreq,ihandle FLtext "Enter the frequency", imin, imax, istep, itype, iwidth
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
    iamp = 15000
    ifn = 1
    asig oscili iamp, gkfreq, ifn
    out asig
endin
/* fltext.orc */
```



```
/* fltext.sco */
; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
/* fltext.sco */
```

## See Also

*FLcount*, *FLjoy*, *FLkeyb*, *FLknob*, *FLroller*, *FLslider*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# FLupdate

FLupdate -- Same as the FLrun opcode.

FLupdate

## Description

Same as the *FLrun* opcode.

## Syntax

**FLupdate**

# fluidAllOut

fluidAllOut -- Collects all audio from all Fluidsynth engines in a performance

fluidAllOut

## Syntax

aleft, aright **fluidAllOut**

## Description

Collects all audio from all Fluidsynth engines in a performance

## Initialization

*aleft* -- Left channel audio output.

*aright* -- Right channel audio output.

## Performance

Invoke fluidAllOut in an instrument definition numbered higher than any fluidcontrol instrument definitions. All SoundFonts send their audio output to this one opcode. Send a note with an indefinite duration to this instrument to turn the SoundFonts on for as long as required.

In this implementation, SoundFont effects such as chorus or reverb are used if and only if they are defaults for the preset. There is no means of turning such effects on or off, or of changing their parameters, from Csound.

## Examples

Here is an example of the fluidsynth opcodes. It uses the file *fluidAllOut.orc* [examples/fluidAllOut.orc].

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
0dbfs = 32767

; LOAD SOUND FONTS
ienginenum1 fluidEngine
ienginenum2 fluidEngine
isfnum1 fluidLoad "Piano Steinway Grand Model C (21,738KB).sf2", ienginenum1, 1
fluidProgramSelect ienginenum1, 1, isfnum1, 0, 1 ; Bright Steinway, program 1,
fluidProgramSelect ienginenum1, 3, isfnum1, 0, 2 ; Concert Steinway with reverb
isfnum2 fluidLoad "63.3mg The Sound Site Album Bank V1.0.SF2", ienginenum2, 1 f

; SEND NOTES TO STEINWAY SOUND FONT

instr 1 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault 60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel = 1
ikey = p4
```

```
ivelocity      =      p5
istatus        =      144
fluidControl   iengineum1, istatus, ichannel, ikey, ivelocity
endin

instr 2 ; GM soundfont
; INITIALIZATION
mididefault    60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey   p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel        =      2
ikey            =      p4
ivelocity       =      p5
istatus         =      144
fluidNote iengineum2, ichannel, ikey, ivelocity
endin

instr 3 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault    60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey   p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel        =      3
ikey            =      p4
ivelocity       =      p5
istatus         =      144
fluidNote iengineum1, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUNDFONTS

instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude      =      ampdb(p5) * (10000.0 / 0.1)
; AUDIO
aleft, aright    fluidAllOut
outs             aleft * iamplitude, aright * iamplitude
endin
```

Here is another more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluidcomplex.csd* [examples/fluidcomplex.csd].

```
<CsoundSynthesizer>
<CsOptions>
-d -m229 -o dac -T -F midifile.mid
</CsOptions>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn     = 1
lp1:
    massign ichn, 0
    loop_le ichn, 1, 16, lp1
    pgmassign 0, 0

; initialize FluidSynth
```

```
gifld    fluidEngine
gisf2    fluidLoad "2gmgsmt.sf2", gifld, 1

; k-rate version of fluidProgramSelect

        opcode fluidProgramSelect_k, 0, kkkkk

keng, kchn, ksf2, kbnk, kpre    xin
        igoto skipInit
doInit:
        fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
        reinit doInit
        rireturn
skipInit:

        endop

        instr 1

; initialize channels

kchn    init 1
        if (kchn == 1) then
lp2:
        fluidControl gifld, 192, kchn - 1, 0, 0
        fluidControl gifld, 176, kchn - 1, 7, 100
        fluidControl gifld, 176, kchn - 1, 10, 64
        loop_le kchn, 1, 16, lp2
        endif

; send any MIDI events received to FluidSynth

nxt:
kst, kch, kd1, kd2    midiin
        if (kst != 0) then
            if (kst != 192 || kch != 10) then
                fluidControl gifld, kst, kch - 1, kd1, kd2
            else
                fluidProgramSelect_k gifld, kch - 1, gisf2, 128, kd1
            endif
            kgoto nxt
        endif

; get audio output from FluidSynth

aL, aR    fluidOut gifld
outs aL, aR

        endin

</CsInstruments>
<CsScore>

i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Opcode by Michael Gogins (gogins at pipeline dot com). Thanks to Peter Hanappe for Fluidsynth, and to Steven Yi for seeing that it is necessary to break up the Fluidsynth into several different Csound opcodes.

## fluidCCi

fluidCCi -- Sends a MIDI controller data message to fluid.

fluidCCi

## Syntax

```
fluidCCi iEngineNumber, iChannelNumber,  
          iControllerNumber, iValue
```

## Description

Sends a MIDI controller data (MIDI controller number and value to use) message to a fluid engine by number on the user specified MIDI channel number.

## Initialization

*iEngineNumber* -- engine number assigned from fluidEngine

*iChannelNumber* -- MIDI channel number to which the Fluidsynth program is assigned: from 0 to 255. MIDI channels numbered 16 or higher are virtual channels.

*iControllerNumber* -- MIDI controller number to use for this message

*iValue* -- value to set for controller (usually 0-127)

## Performance

This opcode is useful for setting controller values at init time. For continuous changes, use fluidCCk.

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

## fluidCCk

fluidCCk -- Sends a MIDI controller data message to fluid.

fluidCCk

## Syntax

```
fluidCCk iEngineNumber, iChannelNumber,  
          iControllerNumber, kValue
```

## Description

Sends a MIDI controller data (MIDI controller number and value to use) message to a fluid engine by number on the user specified MIDI channel number.

## Initialization

*iEngineNumber* -- engine number assigned from fluidEngine

*iChannelNumber* -- MIDI channel number to which the Fluidsynth program is assigned: from 0 to 255. MIDI channels numbered 16 or higher are virtual channels.

*iControllerNumber* -- MIDI controller number to use for this message

## Performance

*kValue* -- value to set for controller (usually 0-127)

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidControl

fluidControl -- Sends MIDI note on, note off, and other messages to a SoundFont preset.

fluidControl

## Syntax

**fluidControl** *ienginenum*, *kstatus*, *kchannel*, *kdata1*, *data2*

## Description

The fluid opcodes provide a simple Csound opcode wrapper around Peter Hanappe's Fluidsynth SoundFont2 synthesizer. This implementation accepts any MIDI note on, note off, controller, pitch bend, or program change message at k-rate. Maximum polyphony is 4096 simultaneously sounding voices. Any number of SoundFonts may be loaded and played simultaneously.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine

*kstatus* -- MIDI channel message status byte: 128 for note off, 144 for note on, 176 for control change, 192 for program change, or 224 for pitch bend. Note off messages need not be specified, as one is automatically generated when each Csound note expires or is released.

*kchannel* -- MIDI channel number to which the Fluidsynth program is assigned: from 0 to 255. MIDI channels numbered 16 or higher are virtual channels.

*kdata1* -- For note on, MIDI key number: from 0 (lowest) to 127 (highest), where 60 is middle C. For continuous controller messages, controller number.

*kdata2* -- For note on, MIDI key velocity: from 0 (no sound) to 127 (loudest). For continuous controller messages, controller value.

## Performance

Invoke fluidControl in instrument definitions that actually play notes and send control messages. Each instrument definition must consistently use one MIDI channel that was assigned to a Fluidsynth program using fluidload.

In this implementation, SoundFont effects such as chorus or reverb are used if and only if they are defaults for the preset. There is no means of turning such effects on or off, or of changing their parameters, from Csound.

## Examples

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
0dbfs = 32767
```

```
; LOAD SOUNDFONTS
ienginenum1 fluidEngine
ienginenum2 fluidEngine
isfnum1 fluidLoad "Piano Steinway Grand Model C (21,738KB).sf2", ienginenum1, 1
fluidProgramSelect ienginenum1, 1, isfnum1, 0, 1 ; Bright Steinway, program 1, 0
```



```
fluidProgramSelect ienginenum1, 3, isfnum1, 0, 2 ; Concert Steinway with reverb
isfnum2 fluidLoad "63.3mg The Sound Site Album Bank V1.0.SF2", ienginenum2, 1 f

; SEND NOTES TO STEINWAY SOUNDFONT

instr 1 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel             = 1
ikey                 = p4
ivelocity            = p5
istatus              = 144
fluidControl         ienginenum1, istatus, ichannel, ikey, ivelocity
endin

instr 2 ; GM soundfont
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel             = 2
ikey                 = p4
ivelocity            = p5
istatus              = 144
fluidNote ienginenum2, ichannel, ikey, ivelocity
endin

instr 3 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel             = 3
ikey                 = p4
ivelocity            = p5
istatus              = 144
fluidNote ienginenum1, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUNDFONTS

instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude1          = ampdb(p5) * (10000.0 / 0.1)
iamplitude2          = ampdb(p6) * (10000.0 / 0.1)

; AUDIO
aleft1, aright1      fluidOut ienginenum1
aleft2, aright2      fluidOut ienginenum2
outs (aleft1 * iamplitude1) + (aleft2 * iamplitude2), (aright1 * iamplitude1) +
endin
```

## Credits

Opcodes by Michael Gogins (gogins at pipeline dot com). Thanks to Peter Hanappe for Fluidsynth, and to Steven Yi for seeing that it is necessary to break up the Fluidsynth into several different Csound opcodes.

# fluidEngine

fluidEngine -- Instantiates a fluidsynth engine.

fluidEngine

## Syntax

ienginenum **fluidEngine**

## Description

Instantiates a fluidsynth engine, returning a number to identify the engine. ienginenum is used in conjunction with other opcodes for loading and playing SoundFonts and gathering the generated sound.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine

## Examples

Here is an example of the fluidsynth opcodes. It uses the file *fluidAllOut.orc* [examples/fluidAllOut.orc].

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
0dbfs = 32767

; LOAD SOUNDFONTS
ienginenum1 fluidEngine
ienginenum2 fluidEngine
isfnum1 fluidLoad "Piano Steinway Grand Model C (21,738KB).sf2", ienginenum1, 1
fluidProgramSelect ienginenum1, 1, isfnum1, 0, 1 ; Bright Steinway, program 1, 0
fluidProgramSelect ienginenum1, 3, isfnum1, 0, 2 ; Concert Steinway with reverb
isfnum2 fluidLoad "63.3mg The Sound Site Album Bank V1.0.SF2", ienginenum2, 1 f

; SEND NOTES TO STEINWAY SOUNDFONT

instr 1 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault 60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel = 1
ikey = p4
ivelocity = p5
istatus = 144
fluidControl ienginenum1, istatus, ichannel, ikey, ivelocity
endin

instr 2 ; GM soundfont
; INITIALIZATION
mididefault 60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel = 2
ikey = p4
```

```
ivelocity      =                                p5
istatus        =                                144
fluidNote iengineum2, ichannel, ikey, ivelocity
endin

instr 3 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault      60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey    p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel         =                                3
ikey             =                                p4
ivelocity        =                                p5
istatus          =                                144
fluidNote iengineum1, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUND FONTS

instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude       =                                ampdb(p5) * (10000.0 / 0.1)
; AUDIO
aleft, aright    fluidAllOut
outs             aleft * iamplitude, aright * iamplitude
endin
```

Here is another more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluidcomplex.csd* [examples/fluidcomplex.csd].

```
<CsoundSynthesizer>
<CsOptions>
-d -m229 -o dac -T -F midifile.mid
</CsOptions>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn     = 1
lp1:
    massign ichn, 0
    loop_le ichn, 1, 16, lp1
    pgmassign 0, 0

; initialize FluidSynth

gifld    fluidEngine
gisf2    fluidLoad "2gmgsmt.sf2", gifld, 1

; k-rate version of fluidProgramSelect

    opcode fluidProgramSelect_k, 0, kkkkk

keng, kchn, ksf2, kbnk, kpre  xin
    igoto skipInit
doInit:
    fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
```

```
        reinit doInit
        rireturn
skipInit:

        endop

        instr 1

; initialize channels

kchn  init 1
      if (kchn == 1) then
lp2:   fluidControl gifld, 192, kchn - 1, 0, 0
        fluidControl gifld, 176, kchn - 1, 7, 100
        fluidControl gifld, 176, kchn - 1, 10, 64
        loop_le kchn, 1, 16, lp2
      endif

; send any MIDI events received to FluidSynth

nxt:
kst, kch, kd1, kd2  midiin
      if (kst != 0) then
        if (kst != 192 || kch != 10) then
          fluidControl gifld, kst, kch - 1, kd1, kd2
        else
          fluidProgramSelect_k gifld, kch - 1, gisf2, 128, kd1
        endif
        kgoto nxt
      endif

; get audio output from FluidSynth

aL, aR  fluidOut gifld
      outs aL, aR

      endin

</CsInstruments>
<CsScore>

i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidLoad

fluidLoad -- Loads a SoundFont into a fluidEngine, optionally listing SoundFont contents.

fluidLoad

## Syntax

```
isfnum fluidLoad soundfont, ienginenum[, ilistpresets]
```

## Description

Loads a SoundFont into an instance of a fluidEngine, optionally listing banks and presets for SoundFont.

## Initialization

*isfnum* -- Number assigned to just-loaded soundfont.

*soundfont* -- String specifying a SoundFont filename. Note that any number of SoundFonts may be loaded (obviously, by different invocations of fluidLoad).

*ienginenum* -- engine number assigned from fluidEngine

*ilistpresets* -- optional, if specified, lists all Fluidsynth programs for the just-loaded SoundFont. A Fluidsynth program is a combination of SoundFont ID, bank number, and preset number that is assigned to a MIDI channel.

## Performance

Invoke fluidLoad in the orchestra header, any number of times. The same SoundFont may be invoked to assign programs to MIDI channels any number of times; the SoundFont is only loaded the first time.

## Examples

Here is an example of the fluidsynth opcodes. It uses the file *fluidAllOut.orc* [examples/fluidAllOut.orc].

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
0dbfs = 32767
```

```
; LOAD SOUNDFONTS
ienginenum1 fluidEngine
ienginenum2 fluidEngine
isfnum1 fluidLoad "Piano Steinway Grand Model C (21,738KB).sf2", ienginenum1, 1
fluidProgramSelect ienginenum1, 1, isfnum1, 0, 1 ; Bright Steinway, program 1,
fluidProgramSelect ienginenum1, 3, isfnum1, 0, 2 ; Concert Steinway with reverb
isfnum2 fluidLoad "63.3mg The Sound Site Album Bank V1.0.SF2", ienginenum2, 1 f
```

```
; SEND NOTES TO STEINWAY SOUNDFONT
```

```
instr 1 ; FluidSynth Steinway Rev
```

```
; INITIALIZATION
```

```
mididefault 60, p3 ; Default duration of 60 -- overridden by s
```

```
midinoteonkey      p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel           = 1
ikey               = p4
ivelocity          = p5
istatus            = 144
fluidControl       ienginenum1, istatus, ichannel, ikey, ivelocity
endin

instr 2 ; GM soundfont
; INITIALIZATION
mididefault        60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey      p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel           = 2
ikey               = p4
ivelocity          = p5
istatus            = 144
fluidNote ienginenum2, ichannel, ikey, ivelocity
endin

instr 3 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault        60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey      p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel           = 3
ikey               = p4
ivelocity          = p5
istatus            = 144
fluidNote ienginenum1, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUND FONTS

instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude         = ampdb(p5) * (10000.0 / 0.1)
; AUDIO
aleft, aright      fluidAllOut
outs               aleft * iamplitude, aright * iamplitude
endin
```

Here is another more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluidcomplex.csd* [examples/fluidcomplex.csd].

```
<CsoundSynthesizer>
<CsOptions>
-d -m229 -o dac -T -F midifile.mid
</CsOptions>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn     = 1
lp1:
    massign ichn, 0
```

```
        loop_le ichn, 1, 16, lp1
        pgmassign 0, 0

; initialize FluidSynth

gifld    fluidEngine
gisf2    fluidLoad "2gmgsmt.sf2", gifld, 1

; k-rate version of fluidProgramSelect

        opcode fluidProgramSelect_k, 0, kkkkk

keng, kchn, ksf2, kbnk, kpre    xin
        igoto skipInit
doInit:
        fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
        reinit doInit
        rireturn
skipInit:

        endop

        instr 1

; initialize channels

kchn    init 1
        if (kchn == 1) then
lp2:
        fluidControl gifld, 192, kchn - 1, 0, 0
        fluidControl gifld, 176, kchn - 1, 7, 100
        fluidControl gifld, 176, kchn - 1, 10, 64
        loop_le kchn, 1, 16, lp2
        endif

; send any MIDI events received to FluidSynth

nxt:
kst, kch, kd1, kd2    midiin
        if (kst != 0) then
            if (kst != 192 || kch != 10) then
                fluidControl gifld, kst, kch - 1, kd1, kd2
            else
                fluidProgramSelect_k gifld, kch - 1, gisf2, 128, kd1
            endif
            kgoto nxt
        endif

; get audio output from FluidSynth

aL, aR    fluidOut gifld
        outs aL, aR

        endin

</CsInstruments>
<CsScore>

i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.



# fluidNote

fluidNote -- Plays a note on a channel in a fluidSynth engine.

fluidNote

## Syntax

**fluidNote** ienginenum, ichannelnum, imidikey, imidivel

## Description

Plays a note at imidikey pitch and imidivel velocity on ichannelnum channel of number ienginenum fluidEngine.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine

*ichannelnum* -- which channel number to play a note on in the given fluidEngine

*imidikey* -- MIDI key for note (0-127)

*imidivel* -- MIDI velocity for note (0-127)

## Examples

Here is an example of the fluidsynth opcodes. It uses the file *fluid.orc* [examples/fluid.orc].

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
0dbfs = 32767

; LOAD SOUNDFONTS
ienginenum1 fluidEngine
ienginenum2 fluidEngine
isfnum1 fluidLoad "Piano Steinway Grand Model C (21,738KB).sf2", ienginenum1, 1
fluidProgramSelect ienginenum1, 1, isfnum1, 0, 1 ; Bright Steinway, program 1, c
fluidProgramSelect ienginenum1, 3, isfnum1, 0, 2 ; Concert Steinway with reverb
isfnum2 fluidLoad "63.3mg The Sound Site Album Bank V1.0.SF2", ienginenum2, 1 f

; SEND NOTES TO STEINWAY SOUNDFONT

instr 1 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault 60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel = 1
ikey = p4
ivelocity = p5
istatus = 144
fluidControl ienginenum1, istatus, ichannel, ikey, ivelocity
endin

instr 2 ; GM soundfont
; INITIALIZATION
mididefault 60, p3 ; Default duration of 60 -- overridden by s
```

```
midinoteonkey          p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel               =                2
ikey                   =                p4
ivelocity              =                p5
istatus                =               144
fluidNote ienginenum2, ichannel, ikey, ivelocity
endin

instr 3 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault            60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey          p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel               =                3
ikey                   =                p4
ivelocity              =                p5
istatus                =               144
fluidNote ienginenum1, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUND FONTS

instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude1            =                ampdb(p5) * (10000.0 / 0.1)
iamplitude2            =                ampdb(p6) * (10000.0 / 0.1)

; AUDIO
aleft1, aright1        fluidOut ienginenum1
aleft2, aright2        fluidOut ienginenum2
outs (aleft1 * iamplitude1) + (aleft2 * iamplitude2), (aright1 * iamplitude1) +
endin
```

Here is another more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluidcomplex.csd* [examples/fluidcomplex.csd].

```
<CsoundSynthesizer>
<CsOptions>
-d -m229 -o dac -T -F midifile.mid
</CsOptions>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn     = 1
lp1:
    massign ichn, 0
    loop_le ichn, 1, 16, lp1
    pgmassign 0, 0

; initialize FluidSynth

gifld    fluidEngine
gisf2    fluidLoad "2gmgsmt.sf2", gifld, 1

; k-rate version of fluidProgramSelect
```

```
        opcode fluidProgramSelect_k, 0, kkkkkk
keng, kchn, ksf2, kbnk, kpre  xin
    igoto skipInit
doInit:
    fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
    reinit doInit
    rireturn
skipInit:

    endop

    instr 1

; initialize channels

kchn  init 1
    if (kchn == 1) then
lp2:
    fluidControl gifld, 192, kchn - 1, 0, 0
    fluidControl gifld, 176, kchn - 1, 7, 100
    fluidControl gifld, 176, kchn - 1, 10, 64
    loop_le kchn, 1, 16, lp2
    endif

; send any MIDI events received to FluidSynth

nxt:
kst, kch, kd1, kd2  midiin
    if (kst != 0) then
        if (kst != 192 || kch != 10) then
            fluidControl gifld, kst, kch - 1, kd1, kd2
        else
            fluidProgramSelect_k gifld, kch - 1, gisf2, 128, kd1
        endif
        kgoto nxt
    endif

; get audio output from FluidSynth

aL, aR  fluidOut gifld
    outs aL, aR

    endin

</CsInstruments>
<CsScore>

i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidOut

fluidOut -- Outputs sound from a given fluidEngine

fluidOut

## Syntax

*aleft*, *aright* **fluidOut** *ienginenum*

## Description

Outputs the sound from a fluidEngine.

Invoke fluidOut in an instrument definition numbered higher than any fluidcontrol instrument definitions. All SoundFonts send their audio output to this one opcode. Send a note with an indefinite duration to this instrument to turn the SoundFonts on for as long as required.

## Initialization

*ienginenum* -- engine number assigned from fluidEngine

## Performance

*aleft* -- Left channel audio output.

*aright* -- Right channel audio output.

## Examples

Here is an example of the fluidsynth opcodes. It uses the file *fluid.orc* [examples/fluid.orc].

```
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
0dbfs = 32767

; LOAD SOUND FONTS
ienginenum1 fluidEngine
ienginenum2 fluidEngine
isfnum1 fluidLoad "Piano Steinway Grand Model C (21,738KB).sf2", ienginenum1, 1
fluidProgramSelect ienginenum1, 1, isfnum1, 0, 1 ; Bright Steinway, program 1, c
fluidProgramSelect ienginenum1, 3, isfnum1, 0, 2 ; Concert Steinway with reverb
isfnum2 fluidLoad "63.3mg The Sound Site Album Bank V1.0.SF2", ienginenum2, 1 f

; SEND NOTES TO STEINWAY SOUND FONT

instr 1 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel              = 1
ikey                  = p4
ivelocity              = p5
istatus               = 144
fluidControl          ienginenum1, istatus, ichannel, ikey, ivelocity
endin
```

```
instr 2 ; GM soundfont
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel             =                2
ikey                  =                p4
ivelocity             =                p5
istatus              =                144
fluidNote ienginenum2, ichannel, ikey, ivelocity
endin

instr 3 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel             =                3
ikey                  =                p4
ivelocity             =                p5
istatus              =                144
fluidNote ienginenum1, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUND FONTS

instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude1           =                ampdb(p5) * (10000.0 / 0.1)
iamplitude2           =                ampdb(p6) * (10000.0 / 0.1)

; AUDIO
aleft1, aright1       fluidOut ienginenum1
aleft2, aright2       fluidOut ienginenum2
outs (aleft1 * iamplitude1) + (aleft2 * iamplitude2), (aright1 * iamplitude1) +
endin
```

Here is another more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluidcomplex.csd* [examples/fluidcomplex.csd].

```
<CsoundSynthesizer>
<CsOptions>
-d -m229 -o dac -T -F midifile.mid
</CsOptions>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn     = 1
lp1:
    massign ichn, 0
    loop_le ichn, 1, 16, lp1
    pgmassign 0, 0

; initialize FluidSynth

gifld    fluidEngine
```

```
gisf2    fluidLoad "2gmgsmt.sf2", gifld, 1
; k-rate version of fluidProgramSelect
        opcode fluidProgramSelect_k, 0, kkkkk
keng, kchn, ksf2, kbnk, kpre    xin
    igoto skipInit
doInit:
    fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
    reinit doInit
    rireturn
skipInit:
    endop
    instr 1
; initialize channels
kchn    init 1
    if (kchn == 1) then
lp2:
    fluidControl gifld, 192, kchn - 1, 0, 0
    fluidControl gifld, 176, kchn - 1, 7, 100
    fluidControl gifld, 176, kchn - 1, 10, 64
    loop_le kchn, 1, 16, lp2
    endif
; send any MIDI events received to FluidSynth
nxt:
kst, kch, kd1, kd2    midiin
    if (kst != 0) then
        if (kst != 192 || kch != 10) then
            fluidControl gifld, kst, kch - 1, kd1, kd2
        else
            fluidProgramSelect_k gifld, kch - 1, gisf2, 128, kd1
        endif
        kgoto nxt
    endif
; get audio output from FluidSynth
aL, aR    fluidOut gifld
    outs aL, aR
    endin
</CsInstruments>
<CsScore>
i 1 0 3600
e
</CsScore>
</CsoundSynthesizer>
```

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# fluidProgramSelect

fluidProgramSelect -- Assigns a preset from a SoundFont to a channel on a fluidEngine.

fluidProgramSelect

## Syntax

```
fluidProgramSelect ienginenum,  
                    ichannelnum, isfnum, ibanknum, ipresetnum
```

## Description

Assigns a preset from a SoundFont to a channel on a fluidEngine.

## Initialization

*ienginenum* --

*ichannelnum* --

*isfnum* --

*ibanknum* --

*ipresetnum* --

## Examples

Here is an example of the fluidsynth opcodes. It uses the file *fluid.orc* [examples/fluid.orc].

```
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 2  
0dbfs = 32767  
  
; LOAD SOUNDFONTS  
ienginenum1 fluidEngine  
ienginenum2 fluidEngine  
isfnum1 fluidLoad "Piano Steinway Grand Model C (21,738KB).sf2", ienginenum1, 1  
fluidProgramSelect ienginenum1, 1, isfnum1, 0, 1 ; Bright Steinway, program 1,  
fluidProgramSelect ienginenum1, 3, isfnum1, 0, 2 ; Concert Steinway with reverb  
isfnum2 fluidLoad "63.3mg The Sound Site Album Bank V1.0.SF2", ienginenum2, 1 f  
  
; SEND NOTES TO STEINWAY SOUNDFONT  
  
instr 1 ; FluidSynth Steinway Rev  
; INITIALIZATION  
mididefault 60, p3 ; Default duration of 60 -- overridden by s  
midinoteonkey p4, p5 ; Channels MIDI input to pfields.  
; Use channel assigned in fluidload.  
ichannel = 1  
ikey = p4  
ivelocity = p5  
istatus = 144  
fluidControl ienginenum1, istatus, ichannel, ikey, ivelocity  
endin  
  
instr 2 ; GM soundfont
```

```
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel             =                2
ikey                  =                p4
ivelocity             =                p5
istatus              =                144
fluidNote iengineum2, ichannel, ikey, ivelocity
endin

instr 3 ; FluidSynth Steinway Rev
; INITIALIZATION
mididefault          60, p3 ; Default duration of 60 -- overridden by s
midinoteonkey        p4, p5 ; Channels MIDI input to pfields.
; Use channel assigned in fluidload.
ichannel             =                3
ikey                  =                p4
ivelocity             =                p5
istatus              =                144
fluidNote iengineum1, ichannel, ikey, ivelocity
endin

; COLLECT AUDIO FROM ALL SOUND FONTS

instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude1          =                ampdb(p5) * (10000.0 / 0.1)
iamplitude2          =                ampdb(p6) * (10000.0 / 0.1)

; AUDIO
aleft1, aright1      fluidOut iengineum1
aleft2, aright2      fluidOut iengineum2
outs (aleft1 * iamplitude1) + (aleft2 * iamplitude2), (aright1 * iamplitude1) +
endin
```

Here is another more complex example of the fluidsynth opcodes written by Istvan Varga. It uses the file *fluidcomplex.csd* [examples/fluidcomplex.csd].

```
<CsoundSynthesizer>
<CsOptions>
-d -m229 -o dac -T -F midifile.mid
</CsOptions>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

; Example by Istvan Varga

; disable triggering of instruments by MIDI events

ichn     = 1
lp1:
    massign ichn, 0
    loop_le ichn, 1, 16, lp1
    pgmassign 0, 0

; initialize FluidSynth

gifld    fluidEngine
gisf2    fluidLoad "2gmgsmt.sf2", gifld, 1
```



```
; k-rate version of fluidProgramSelect
    opcode fluidProgramSelect_k, 0, kkkkk

keng, kchn, ksf2, kbnk, kpre  xin
    igoto skipInit
doInit:
    fluidProgramSelect i(keng), i(kchn), i(ksf2), i(kbnk), i(kpre)
    reinit doInit
    rireturn
skipInit:

    endop

    instr 1

; initialize channels

kchn  init 1
    if (kchn == 1) then
lp2:
    fluidControl gifld, 192, kchn - 1, 0, 0
    fluidControl gifld, 176, kchn - 1, 7, 100
    fluidControl gifld, 176, kchn - 1, 10, 64
    loop_le kchn, 1, 16, lp2
    endif

; send any MIDI events received to FluidSynth

nxt:
kst, kch, kd1, kd2  midiin
    if (kst != 0) then
        if (kst != 192 || kch != 10) then
            fluidControl gifld, kst, kch - 1, kd1, kd2
        else
            fluidProgramSelect_k gifld, kch - 1, gisf2, 128, kd1
        endif
        kgoto nxt
    endif

; get audio output from FluidSynth

aL, aR  fluidOut gifld
    outs aL, aR

    endin

</CsInstruments>
<CsScore>

i 1 0 3600
e

</CsScore>
</CsoundSynthesizer>
```

## Credits

Michael Gogins (gogins at pipeline dot com), Steven Yi. Thanks to Peter Hanappe for Fluidsynth.

# FLvalue

FLvalue -- Shows the current value of a FLTK valuator.

FLvalue

## Description

*FLvalue* shows current the value of a valuator in a text field.

## Syntax

```
ihandle FLvalue "label", iwidth, iheight, ix, iy
```

## Initialization

*ihandle* -- handle value (an integer number) that unequivocally references the corresponding valuator. It can be used for the *idisp* argument of a valuator.

*"label"* -- a double-quoted string containing some user-provided text, placed near the corresponding widget.

*iwidth* -- width of widget.

*iheight* -- height of widget.

*ix* -- horizontal position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

*iy* -- vertical position of upper left corner of the valuator, relative to the upper left corner of corresponding window (expressed in pixels).

## Performance

Note that *FLvalue* is not a valuator and its value is fixed. Its value cannot be modified.

*FLvalue* shows the current values of a valuator in a text field. It outputs *ihandle* that can then be used for the *idisp* argument of a valuator (see the *FLTK Valuators section*). In this way, the values of that valuator will be dynamically be shown in a text field.

## Examples

Here is an example of the FLvalue opcode. It uses the files *FLvalue.orc* [examples/FLvalue.orc] and *FLvalue.sco* [examples/FLvalue.sco].

### Example 143. Example of the FLvalue opcode.

```
/* flvalue.orc */
; Using the opcode flvalue to display the output of a slider
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

FLpanel "Value Display Box", 900, 200, 50, 50
```

```
; Width of the value display box in pixels
iwidth = 50
; Height of the value display box in pixels
iheight = 20
; Distance of the left edge of the value display
; box from the left edge of the panel
ix = 65
; Distance of the top edge of the value display
; box from the top edge of the panel
iy = 55

idisp FLvalue "Hertz", iwidth, iheight, ix, iy
gkfreq, ihandle FLslider "Frequency", 200, 5000, -1, 5, idisp, 750, 30, 125
FLsetVal_i 500, ihandle
; End of panel contents
FLpanelEnd
; Run the widget thread!
FLrun

instr 1
  iamp = 15000
  ifn = 1
  asig oscili iamp, gkfreq, ifn
  out asig
endin
/* flvalue.orc */

/* flvalue.sco */
; Function table that defines a single cycle
; of a sine wave.
f 1 0 1024 10 1

; Instrument 1 will play a note for 1 hour.
i 1 0 3600
e
/* flvalue.sco */
```

## See Also

*FLbox*, *FLbutBank*, *FLbutton*, *FLprintk*, *FLprintk2*

## Credits

Author: Gabriel Maldonado

New in version 4.22

Example written by Iain McCurdy, edited by Kevin Conder.

# fmb3

fmb3 -- Uses FM synthesis to create a Hammond B3 organ sound.

fmb3

## Description

Uses FM synthesis to create a Hammond B3 organ sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

ares **fmb3** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn

## Initialization

*fmb3* takes 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.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- sine wave

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Total mod index
- *kc2* -- Crossfade of two modulators
- *Algorithm* -- 4

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmb3 opcode. It uses the files *fmb3.orc* [examples/fmb3.orc] and *fmb3.sco* [examples/fmb3.sco].

**Example 144. Example of the fmb3 opcode.**

```
/* fmb3.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 15000
  kfreq = 440
  kc1 = 5
  kc2 = 5
  kvdepth = 0.005
  kvrate = 6
  ifn1 = 1
  ifn2 = 1
  ifn3 = 1
  ifn4 = 1
  ivfn = 1

  a1 fmb3 kamp, kfreq, kc1, kc2, kvdepth, kvrate, \
      ifn1, ifn2, ifn3, ifn4, ivfn
  out a1
endin
/* fmb3.orc */

/* fmb3.sco */
; Table #1, a sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* fmb3.sco */
```

## See Also

*fmbell, fmmetal, fmpercfl, fmrhode, fmwurlie*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

# fmbell

fmbell -- Uses FM synthesis to create a tublar bell sound.

fmbell

## Description

Uses FM synthesis to create a tublar bell sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

ares **fmbell** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivf

## 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.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- sine wave

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Mod index 1
- *kc2* -- Crossfade of two outputs
- *Algorithm* -- 5

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmbell opcode. It uses the files *fmbell.orc* [examples/fmbell.orc] and *fmbell.sco* [examples/fmbell.sco].

**Example 145. Example of the fmbell opcode.**

```
/* fmbell.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 10000
  kfreq = 880
  kc1 = 5
  kc2 = 5
  kvdepth = 0.005
  kbrate = 6
  ifn1 = 1
  ifn2 = 1
  ifn3 = 1
  ifn4 = 1
  ivfn = 1

  a1 fmbell kamp, kfreq, kc1, kc2, kvdepth, kbrate, ifn1, ifn2, ifn3, ifn4, ivfn
  out a1
endin
/* fmbell.orc */

/* fmbell.sco */
; Table #1, a sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
e
/* fmbell.sco */
```

## See Also

*fmb3, fmmetal, fmpercfl, fmrhode, fmwurlie*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

# fmmetal

fmmetal -- Uses FM synthesis to create a “Heavy Metal” sound.

fmmetal

## Description

Uses FM synthesis to create a “Heavy Metal” sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

ares **fmmetal** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, iv

## 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.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- *twopeaks.aiff* [examples/twopeaks.aiff]
- *ifn3* -- *twopeaks.aiff* [examples/twopeaks.aiff]
- *ifn4* -- sine wave



### Note

The file “twopeaks.aiff” is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Total mod index
- *kc2* -- Crossfade of two modulators
- *Algorithm* -- 3

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate



## Examples

Here is an example of the `fmmetal` opcode. It uses the files *fmmetal.orc* [examples/fmmetal.orc], *fmmetal.sco* [examples/fmmetal.sco], and *twopeaks.aiff* [examples/twopeaks.aiff].

### Example 146. Example of the `fmmetal` opcode.

```
/* fmmetal.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 10000
  kfreq = 440
  kc1 = 6
  kc2 = 5
  kvdepth = 0
  kvrate = 0
  ifn1 = 1
  ifn2 = 2
  ifn3 = 2
  ifn4 = 1
  ivfn = 1

  a1 fmmetal kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
  out a1
endin
/* fmmetal.orc */

/* fmmetal.sco */
; Table #1, a normal sine wave.
f 1 0 32768 10 1
; Table #2, the "twopeaks.aiff" audio file.
f 2 0 256 1 "twopeaks.aiff" 0 0 0

; Play Instrument #1 for one second.
i 1 0 1
e
/* fmmetal.sco */
```

## See Also

*fmb3*, *fmbell*, *fmpercfl*, *fmrhode*, *fmwurlie*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

# fmpercfl

fmpercfl -- Uses FM synthesis to create a percussive flute sound.

fmpercfl

## Description

Uses FM synthesis to create a percussive flute sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

ares **fmpercfl** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, i

## 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.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- sine wave

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Total mod index
- *kc2* -- Crossfade of two modulators
- *Algorithm* -- 4

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the fmpercfl opcode. It uses the files *fmpercfl.orc* [examples/fmpercfl.orc] and *fmpercfl.sco* [examples/fmpercfl.sco].

**Example 147. Example of the fmpercfl opcode.**

```
/* fmpercfl.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  kfreq = 220
  kc1 = 5
  kc2 = 5
  kvdepth = 0.005
  kvrate = 6
  ifn1 = 1
  ifn2 = 1
  ifn3 = 1
  ifn4 = 1
  ivfn = 1

  a1 fmpercfl kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, i
  out a1
endin
/* fmpercfl.orc */

/* fmpercfl.sco */
; Table #1, a sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* fmpercfl.sco */
```

## See Also

*fmb3, fmbell, fmmetal, fmrhode, fmwurlie*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

## fmrhode

fmrhode -- Uses FM synthesis to create a Fender Rhodes electric piano sound.

fmrhode

## Description

Uses FM synthesis to create a Fender Rhodes electric piano sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

ares **fmrhode** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, iv

## 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.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- *fwavblnk.aiff* [examples/fwavblnk.aiff]



### Note

The file “fwavblnk.aiff” is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Mod index 1
- *kc2* -- Crossfade of two outputs
- *Algorithm* -- 5

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the `fmrhode` opcode. It uses the files *fmrhode.orc* [examples/fmrhode.orc], *fmrhode.sco* [examples/fmrhode.sco], and *fwavblnk.aiff* [examples/fwavblnk.aiff].

### Example 148. Example of the `fmrhode` opcode.

```
/* fmrhode.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  kfreq = 220
  kc1 = 6
  kc2 = 0
  kvdepth = 0.01
  kvrate = 3
  ifn1 = 1
  ifn2 = 1
  ifn3 = 1
  ifn4 = 2
  ivfn = 1

  a1 fmrhode kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn
  out a1
endin
/* fmrhode.orc */

/* fmrhode.sco */
; Table #1, a sine wave.
f 1 0 32768 10 1
; Table #2, the "fwavblnk.aiff" audio file.
f 2 0 256 1 "fwavblnk.aiff" 0 0 0

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* fmrhode.sco */
```

## See Also

*fmb3*, *fnbell*, *fnmetal*, *fnpercfl*, *fnwurlie*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

# fmvoice

fmvoice -- FM Singing Voice Synthesis

fmvoice

## Description

FM Singing Voice Synthesis

## Syntax

ares **fmvoice** *kamp*, *kfreq*, *kvowel*, *ktilt*, *kvibamt*, *kvibrate*, *ifn1*, *ifn2*, *ifn3*, *i*

## 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

## Examples

Here is an example of the fmvoice opcode. It uses the files *fmvoice.orc* [examples/fmvoice.orc] and *fmvoice.sco* [examples/fmvoice.sco].

### Example 149. Example of the fmvoice opcode.

```
/* fmvoice.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  kfreq = 110
  ; Use the fourth p-field for the vowel.
  kvowel = p4
  ktilt = 0
  kvibamt = 0.005
  kvibrate = 6
  ifn1 = 1
```



```
    ifn2 = 1
    ifn3 = 1
    ifn4 = 1
    ivibfn = 1

    a1 fmvoice kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate, ifn1, ifn2, ifn3, i
    out a1
  endin
/* fmvoice.orc */

/* fmvoice.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; p4 = vowel (a value from 0 to 64)
; Play Instrument #1 for one second, vowel=1.
i 1 0 1 1
; Play Instrument #1 for one second, vowel=2.
i 1 1 1 2
; Play Instrument #1 for one second, vowel=3.
i 1 2 1 3
; Play Instrument #1 for one second, vowel=4.
i 1 3 1 4
; Play Instrument #1 for one second, vowel=5.
i 1 4 1 5
e
/* fmvoice.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# fmwurlie

fmwurlie -- Uses FM synthesis to create a Wurlitzer electric piano sound.

fmwurlie

## Description

Uses FM synthesis to create a Wurlitzer electric piano sound. It comes from a family of FM sounds, all using 4 basic oscillators and various architectures, as used in the TX81Z synthesizer.

## Syntax

ares **fmwurlie** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, i

## 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.

The initial waves should be:

- *ifn1* -- sine wave
- *ifn2* -- sine wave
- *ifn3* -- sine wave
- *ifn4* -- *fwavblnk.aiff* [examples/fwavblnk.aiff]



### Note

The file “fwavblnk.aiff” is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## Performance

*kamp* -- Amplitude of note.

*kfreq* -- Frequency of note played.

*kc1*, *kc2* -- Controls for the synthesizer:

- *kc1* -- Mod index 1
- *kc2* -- Crossfade of two outputs
- *Algorithm* -- 5

*kvdepth* -- Vibrator depth

*kvrate* -- Vibrator rate

## Examples

Here is an example of the `fmwurlie` opcode. It uses the files *fmwurlie.orc* [examples/fmwurlie.orc], *fmwurlie.sco* [examples/fmwurlie.sco], and *fwavblnk.aiff* [examples/fwavblnk.aiff].

### Example 150. Example of the `fmwurlie` opcode.

```
/* fmwurlie.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  kfreq = 440
  kc1 = 6
  kc2 = 1
  kvdepth = 0.005
  kbrate = 6
  ifn1 = 1
  ifn2 = 1
  ifn3 = 1
  ifn4 = 2
  ivfn = 1

  a1 fmwurlie kamp, kfreq, kc1, kc2, kvdepth, kbrate, ifn1, ifn2, ifn3, ifn4, i
  out a1
endin
/* fmwurlie.orc */

/* fmwurlie.sco */
; Table #1, a sine wave.
f 1 0 32768 10 1
; Table #2, the "fwavblnk.aiff" audio file.
f 2 0 256 1 "fwavblnk.aiff" 0 0 0

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* fmwurlie.sco */
```

## See Also

*fmb3*, *fnbell*, *fnmetal*, *fnpercfl*, *fnrhode*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

# fof

fof -- Produces sinusoid bursts useful for formant and granular synthesis.

fof

## 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.

## Syntax

ares **fof** xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb,

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

*ifmode* (optional, default=0) -- 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.

*iskip* (optional, default=0) -- If non-zero, skip initialisation (allows legato use).

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

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

*kris*, *kdur*, *kdec* -- rise, overall duration, and decay times (in seconds) of the sinusoid burst. These values apply an enveloped duration to each burst, in similar fashion to a Csound *linen* generator but with rise and decay shapes derived from the *ifnb* input. *kris* inversely determines the skirtwidth (at -

40 dB) of the induced formant region. *kdur* affects the density of sineburst overlaps, and thus the speed of computation. Typical values for vocal imitation are .003,.02,.007.

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

## Examples

Here is an example of the *fof* opcode. It uses the files *fof.orc* [examples/fof.orc] and *fof.sco* [examples/fof.sco].

### Example 151. Example of the *fof* opcode.

```
/* fof.orc */
/* Adapted from 1401.orc by Michael Clarke */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Combine five formants together to create
  ; an alto-"a" sound.

  ; Values common to all of the formants.
  kfund init 261.659
  koct init 0
  kris init 0.003
  kdur init 0.02
  kdec init 0.007
  iolaps = 14850
  ifna = 1
  ifnb = 2
  itotdur = p3

  ; First formant.
  klamp = ampdb(0)
  klform init 800
  klband init 80

  ; Second formant.
  k2amp = ampdb(-4)
  k2form init 1150
  k2band init 90

  ; Third formant.
  k3amp = ampdb(-20)
  k3form init 2800
  k3band init 120

  ; Fourth formant.
  k4amp = ampdb(-36)
  k4form init 3500
  k4band init 130

  ; Fifth formant.
  k5amp = ampdb(-60)
  k5form init 4950
  k5band init 140
```

```
a1 fof klamp, kfund, klform, koct, klband, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a2 fof k2amp, kfund, k2form, koct, k2band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a3 fof k3amp, kfund, k3form, koct, k3band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a4 fof k4amp, kfund, k4form, koct, k4band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur
a5 fof k5amp, kfund, k5form, koct, k5band, kris, \
    kdur, kdec, iolaps, ifna, ifnb, itotdur

; Combine all of the formants together.
out (a1+a2+a3+a4+a5) * 16384
endin
/* fof.orc */
```

```
/* fof.sco */
/* Adapted from 1401.sco by Michael Clarke */
; Table #1, a sine wave.
f 1 0 4096 10 1
; Table #2.
f 2 0 1024 19 0.5 0.5 270 0.5

; Play Instrument #1 for three seconds.
i 1 0 3
e
/* fof.sco */
```

The formant values for the alto-"a" sound were taken from the *Formant Values Appendix*.

## See Also

*fof2*, *Formant Values Appendix*

# fof2

fof2 -- Produces sinusoid bursts including k-rate incremental indexing with each successive burst.

fof2

## 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.

## Syntax

ares **fof2** xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb

## 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*.

*iskip* (optional, default=0) -- If non-zero, skip initialization (allows legato use).

## 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* -- octavation 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.

*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.

## See Also

*fof*

## Credits

Author: Rasmus Ekman

*fof2* is a modification of *fof* by Rasmus Ekman

New in Csound 3.45

# fofilter

fofilter -- Formant filter.

fofilter

## Description

Fofilter generates a stream of overlapping sinewave grains, when fed with a pulse train. Each grain is the impulse response of a combination of two BP filters. The grains are defined by their attack time (determining the skirtwidth of the formant region at -60dB) and decay time (-6dB bandwidth). Overlapping will occur when  $1/\text{freq} < \text{decay}$ , but, unlike FOF, there is no upper limit on the number of overlaps. The original idea for this opcode came from J McCartney's formlet class in SuperCollider, but this is possibly implemented differently(?).

## Syntax

```
asig fofilter ain, kcf, kris, kdec[, istor]
```

## Initialization

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

## Performance

*asig* -- input signal.

*kcf* -- filter centre frequency

*kris* -- impulse response attack time (secs).

*kdec* -- impulse response decay time (secs).

## Examples

### Example 152. Example

```
kfe      expseg 10, p3*0.9, 180, p3*0.1, 175
kenv     linen 1000, 0.05, p3, 0.05
asig     buzz  kenv, kfe, sr/(2*kfe), 1
afil     fofilter asig, 900, 0.007, 0.04

        out afil
```

## Credits

Author: Victor Lazzarini;

January 2005

New plugin in version 5

January 2005.

# fog

`fog` -- Audio output is a succession of grains derived from data in a stored function table

`fog`

## 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.

## Syntax

ares **fog** *xamp*, *xdens*, *xtrans*, *aspd*, *koct*, *kband*, *kris*, *kdur*, *kdec*, *iolaps*, *ifna*

## 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 (*GEN05*).

*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.

*iskip* (optional, default=0) -- If non-zero, skip initialization (allows legato use).

## 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.

*aspd* -- speed. The rate at which successive grains advance through the stored function table *ifna*. *aspd* 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* -- octavation 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.

## Examples

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

## Credits

Author: Michael Clark  
Huddersfield  
May 1997

New in version 3.46

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

Added notes by Rasmus Ekman on September 2002.

# fold

fold -- Adds artificial foldover to an audio signal.

fold

## Description

Adds artificial foldover to an audio signal.

## Syntax

ares **fold** asig, kincr

## Performance

*asig* -- input signal

*kincr* -- amount of foldover expressed in multiple of sampling rate. Must be  $\geq 1$

*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.

## Examples

Here is an example of the fold opcode. It uses the files *fold.orc* [examples/fold.orc] and *fold.sco* [examples/fold.sco].

### Example 153. Example of the fold opcode.

```
/* fold.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use an ordinary sine wave.
  asig oscils 30000, 100, 1

  ; Vary the fold-over amount from 1 to 200.
  kincr line 1, p3, 200
  a1 fold asig, kincr

  out a1
endin
/* fold.orc */
```

```
/* fold.sco */
```

```
; Play Instrument #1 for four seconds.  
i 1 0 4  
e  
/* fold.sco */
```

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# follow

follow -- Envelope follower unit generator.

follow

## Description

Envelope follower unit generator.

## Syntax

ares **follow** asig, idt

## 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.

## Examples

Here is an example of the follow opcode. It uses the files *follow.orc* [examples/follow.orc], *follow.sco* [examples/follow.sco], and *beats.wav* [examples/beats.wav].

### Example 154. Example of the follow opcode.

```
/* follow.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - play a WAV file.
instr 1
  al soundin "beats.wav"
  out al
endin

; Instrument #2 - have another waveform follow the WAV file.
instr 2
  ; Follow the WAV file.
  as soundin "beats.wav"
  af follow as, 0.01

  ; Use a sine waveform.
  as oscil 4000, 440, 1
  ; Have it use the amplitude of the followed WAV file.
  al balance as, af

  out al
endin
/* follow.orc */
```



```
/* follow.sco */  
; Just generate a nice, ordinary sine wave.  
f 1 0 32768 10 1  
  
; Play Instrument #1 for two seconds.  
i 1 0 2  
; Play Instrument #2 for two seconds.  
i 2 2 2  
e  
/* follow.sco */
```

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

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# follow2

follow2 -- Another controllable envelope extractor.

follow2

## Description

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

## Syntax

ares **follow2** asig, katt, krel

## 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*.

## Examples

Here is an example of the follow2 opcode. It uses the files *follow2.orc* [examples/follow2.orc], *follow2.sco* [examples/follow2.sco], and *beats.wav* [examples/beats.wav].

### Example 155. Example of the follow2 opcode.

```
/* follow2.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - play a WAV file.
instr 1
  al soundin "beats.wav"
  out al
endin

; Instrument #2 - have another waveform follow the WAV file.
instr 2
  ; Follow the WAV file.
  as soundin "beats.wav"
  af follow2 as, 0.01, 0.1

  ; Use a noise waveform.
  ar rand 44100
  ; Have it use the amplitude of the followed WAV file.
  al balance ar, af
```

```
    out a1
endin
/* follow2.orc */

/* follow2.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* follow2.sco */
```

## Credits

Author: John ffitch  
The algorithm for the *follow2* is attributed to Jean-Marc Jot.  
University of Bath, Codemist Ltd.  
Bath, UK  
February 2000

Example written by Kevin Conder.

New in Csound version 4.03

Added notes by Rasmus Ekman on September 2002.

# foscil

foscil -- A basic frequency modulated oscillator.

foscil

## Description

A basic frequency modulated oscillator.

## Syntax

ares **foscil** xamp, kcps, xcar, xmod, kndx, ifn [, iphs]

## Initialization

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

*iphs* (optional, default=0) -- 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.

## Performance

*xamp* -- the amplitude of the output signal.

*kcps* -- a common denominator, in cycles per second, for the carrier and modulating frequencies.

*xcar* -- a factor that, when multiplied by the *kcps* parameter, gives the carrier frequency.

*xmod* -- a factor that, when multiplied by the *kcps* parameter, gives the modulating frequency.

*kndx* -- the modulation index.

*foscil* is a composite unit that effectively banks two *oscil* opcodes 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 =  $kcps * xcar$ , and modulating frequency =  $kcps * xmod$ . For integral values of *xcar* and *xmod*, the perceived fundamental will be the minimum positive value of  $kcps * (xcar - n * xmod)$ ,  $n = 1, 1, 2, \dots$ . 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  $n = 0, 1, 2, \dots$ , etc. *ifn* should point to a stored sine wave. Previous to version 3.50, *xcar* and *xmod* could be k-rate only.

## Examples

Here is an example of the foscil opcode. It uses the files *foscil.orc* [examples/foscil.orc] and *foscil.sco* [examples/foscil.sco].

### Example 156. Example of the foscil opcode.

```
/* foscil.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1 - a basic FM waveform.
instr 1
  kamp = 10000
  kcps = 440
  kcar = 600
  kmod = 210
  kndx = 2
  ifn = 1

  al foscil kamp, kcps, kcar, kmod, kndx, ifn
  out al
endin
/* foscil.orc */

/* foscil.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* foscil.sco */
```

## Credits

Example written by Kevin Conder.

# foscili

foscili -- Basic frequency modulated oscillator with linear interpolation.

foscili

## Description

Basic frequency modulated oscillator with linear interpolation.

## Syntax

ares **foscili** xamp, kcps, xcar, xmod, kndx, ifn [, iphs]

## Initialization

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

*iphs* (optional, default=0) -- 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.

## Performance

*xamp* -- the amplitude of the output signal.

*kcps* -- the frequency of the output signal measured in cycles per second.

*xcar* -- the carrier frequency.

*xmod* -- the modulating frequency.

*kndx* -- the modulation index.

*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.

## Examples

Here is an example of the foscili opcode. It uses the files *foscili.orc* [examples/foscili.orc] and *foscili.sco* [examples/foscili.sco].

### Example 157. Example of the foscili opcode.

```
/* foscili.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic FM waveform.
instr 1
```

```
kamp = 10000
kcps = 440
kcar = 600
kmod = 210
kndx = 2
ifn = 1

a1 foscil kamp, kcps, kcar, kmod, kndx, ifn
out a1
endin

; Instrument #2 - the basic FM waveform with extra interpolation.
instr 2
  kamp = 10000
  kcps = 440
  kcar = 600
  kmod = 210
  kndx = 2
  ifn = 1

  a1 foscili kamp, kcps, kcar, kmod, kndx, ifn
  out a1
endin
/* foscili.orc */


/* foscili.sco */
; Table #1, a sine wave table with a small amount of data.
f 1 0 4096 10 1

; Play Instrument #1, the basic FM instrument, for
; two seconds. This should sound relatively rough.
i 1 0 2

; Play Instrument #2, the interpolated FM instrument, for
; two seconds. This should sound relatively smooth.
i 2 2 2
e
/* foscili.sco */
```

## Credits

Example written by Kevin Conder.

# fout

**fout** -- Outputs a-rate signals to an arbitrary number of channels.

fout

## Description

*fout* outputs *N* a-rate signals to a specified file of *N* channels.

## Syntax

**fout** ifilename, iformat, aout1 [, aout2, aout3,...,aoutN]

## Initialization

*ifilename* -- the output file's name (in double-quotes).

*iformat* -- a flag to choose output file format (note: Csound versions older than 5.0 may only support formats 0, 1, and 2):

- 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 a header. The header type depends on the render (-o) format. For example, if the user chooses the AIFF format (using the *-A flag*), the header format will be AIFF type.
- 3 - u-law samples with a header (see iformat=2).
- 4 - 16-bit integers with a header (see iformat=2).
- 5 - 32-bit integers with a header (see iformat=2).
- 6 - 32-bit floats with a header (see iformat=2).
- 7 - 8-bit unsigned integers with a header (see iformat=2).
- 8 - 24-bit integers with a header (see iformat=2).
- 9 - 64-bit floats with a header (see iformat=2).

In addition, Csound versions 5.0 and later allow for explicitly selecting a particular header type by specifying the format as 10 \* fileType + sampleFormat, where fileType may be 1 for WAV, 2 for AIFF, 3 for raw (headerless) files, and 4 for IRCAM; sampleFormat is one of the above values in the range 0 to 9, except sample format 0 is taken from the command line (-o), format 1 is 8-bit signed integers, and format 2 is a-law. So, for example, iformat=25 means 32-bit integers with AIFF header.

## Performance

*aout1,... aoutN* -- signals to be written to the file. In the case of raw files, the expected range of audio signals is determined by the selected sample format; for sound files with a header like WAV and AIFF, the audio signals should be in the range -0dbfs to 0dbfs.

*fout* (file output) writes samples of audio signals to a file with any number of channels. Channel



number depends by the number of *aboutN* 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 it. If polyphony is to be used, you can use *vincr* and *clear* opcodes for this task.

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.

## Examples

Here is a simple example of the *fout* opcode. It uses the files *fout.orc* [examples/fout.orc] and *fout.sco* [examples/fout.sco].

### Example 158. Example of the *fout* opcode.

```
/* fout.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iamp = 10000
  icps = 440
  iphs = 0

  ; Create an audio signal.
  asig oscils iamp, icps, iphs

  ; Write the audio signal to a headerless audio file
  ; called "fout.raw".
  fout "fout.raw", 1, asig
endin
/* fout.orc */

/* fout.sco */
; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* fout.sco */
```

Here is an example of the *fout* opcode with a polyphonic score. It uses the files *fout\_poly.orc* [examples/fout\_poly.orc], *fout\_poly.sco* [examples/fout\_poly.sco] and *beats.wav* [examples/beats.wav].

### Example 159. Example of the *fout* opcode with a polyphonic score.

```
/* fout_poly.orc */
```

```
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Initialize the global audio signal.
gaudio init 0

; Instrument #1 - Play an audio file.
instr 1
  ; Generate an audio signal using
  ; the audio file "beats.wav".
  asig soundin "beats.wav"

  ; Add this audio signal to the global one.
  vincr gaudio, asig
endin

; Instrument #2 - Create a basic tone.
instr 2
  iamp = 5000
  icps = 440
  iphs = 0

  ; Create an audio signal.
  asig oscils iamp, icps, iphs

  ; Add this audio signal to the global one.
  vincr gaudio, asig
endin

; Instrument #99 - Save the global signal to a file.
instr 99
  ; Write the global audio signal to a headerless
  ; audio file called "fout_poly.raw".
  fout "fout_poly.raw", 1, gaudio

  ; Clear the global audio signal, preparing it
  ; for the next round.
  clear gaudio
endin
/* fout_poly.orc */

/* fout_poly.sco */
; Play Instrument #1 for two seconds.
i 1 0 2

; Play Instrument #2 every quarter-second.
i 2 0.00 0.1
i 2 0.25 0.1
i 2 0.50 0.1
i 2 0.75 0.1
i 2 1.00 0.1
i 2 1.25 0.1
i 2 1.50 0.1
i 2 1.75 0.1

; Make sure the global instrument, #99, is running
; during the entire performance (2 seconds).
i 99 0 2
e
/* fout_poly.sco */
```

## See Also

*flopen, fouti, foutir, foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

The simple example was written by Kevin Conder.

New in Csound version 3.56

October 2002. Added a note from Richard Dobson.

# fouti

**fouti** -- Outputs i-rate signals of an arbitrary number of channels to a specified file.

fouti

## Description

*fouti* output *N* i-rate signals to a specified file of *N* channels.

## Syntax

**fouti** *ihandle*, *iformat*, *iflag*, *iout1* [, *iout2*, *iout3*, ..., *ioutN*]

## Initialization

*ihandle* -- a number which specifies this file.

*iformat* -- a flag to choose output file format:

- 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

## Performance

*fouti* and *foutir* write i-rate values to a file. The main use of these opcodes is to generate a score file during a realtime 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.

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.

## See Also

*fiopen*, *fout*, *foutir*, *foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# foutir

**foutir** -- Outputs i-rate signals from an arbitrary number of channels to a specified file.

foutir

## Description

*foutir* output *N* i-rate signals to a specified file of *N* channels.

## Syntax

**foutir** *ihandle*, *iformat*, *iflag*, *iout1* [, *iout2*, *iout3*, ..., *ioutN*]

## Initialization

*ihandle* -- a number which specifies this file.

*iformat* -- a flag to choose output file format:

- 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

## Performance

*fouti* and *foutir* write i-rate values to a file. The main use of these opcodes is to generate a score file during a realtime 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. So *fouti* and *foutir* can be used to generate a Csound score while playing a realtime session.

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.

## See Also

*fiopen, fout, fouti, foutk*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# foutk

**foutk** -- Outputs k-rate signals of an arbitrary number of channels to a specified file, in raw (headerless) format.

foutk

## Description

*foutk* outputs *N* k-rate signals to a specified file of *N* channels.

## Syntax

**foutk** ifilename, iformat, kout1 [, kout2, kout3, ..., koutN]

## Initialization

*ifilename* -- the output file's name (in double-quotes).

*iformat* -- a flag to choose output file format (note: Csound versions older than 5.0 may only support formats 0 and 1):

- 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 without header (binary PCM multichannel file)
- 3 - u-law samples without header
- 4 - 16-bit integers without header
- 5 - 32-bit integers without header
- 6 - 32-bit floats without header
- 7 - 8-bit unsigned integers without header
- 8 - 24-bit integers without header
- 9 - 64-bit floats without header

## Performance

*kout1,...koutN* -- control-rate signals to be written to the file. The expected range of the signals is determined by the selected sample format.

*foutk* operates in the same way as *fout*, but with k-rate signals. *iformat* can be set only in the range 0 to 9, or 0 to 1 with an old version of Csound.

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.

## See Also



*fiopen, fout, fouti, foutir*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# fprintks

fprintks -- Similar to printks but prints to a file.

fprintks

## Description

Similar to *printks* but prints to a file.

## Syntax

**fprintks** "filename", "string", [, kval1] [, kval2] [...]

## Initialization

"filename" -- name of the output file.

"string" -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

## Performance

kval1, kval2, ... (optional) -- The k-rate values to be printed. These are specified in "string" with the standard C value specifier (%f, %d, etc.) in the order given.

*fprintks* is similar to the *printks* opcode except it outputs to a file and doesn't have a *itime* parameter. For more information about output formatting, please look at *printks's* [documentation](#).

## Examples

Here is an example of the fprintks opcode. It uses the files *fprintks.orc* [examples/fprintks.orc] and *fprintks.sco* [examples/fprintks.sco].

### Example 160. Example of the fprintks opcode.

```
/* fprintks.orc */
/* Written by Matt Ingalls, edited by Kevin Conder. */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a score generator example.
instr 1
  ; K-rate stuff.
  kstart init 0
  kdur linrand 10
  kpitch linrand 8

  ; Printing to to a file called "my.sco".
  fprintks "my.sco", "i1\\t%2.2f\\t%2.2f\\t%2.2f\\n", kstart, kdur, 4+kpitch

  knext linrand 1
  kstart = kstart + knext
endin
```

```
/* fprintks.orc */
```

```
/* fprintks.sco */  
/* Written by Matt Ingalls, edited by Kevin Conder. */  
; Play Instrument #1.  
i 1 0 0.001  
/* fprintks.sco */
```

This example will generate a file called “my.sco”. It should contain lines like this:

```
i1      0.00    3.94    10.26  
i1      0.20    3.35     6.22  
i1      0.67    3.65    11.33  
i1      1.31    1.42     4.13
```

## See Also

*printks*

## Credits

Author: Matt Ingalls  
January 2003

# fprints

fprints -- Similar to prints but prints to a file.

fprints

## Description

Similar to *prints* but prints to a file.

## Syntax

**fprints** "filename", "string" [, ival1] [, ival2] [...]

## Initialization

"filename" -- name of the output file.

"string" -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

ival1, ival2, ... (optional) -- The i-rate values to be printed. These are specified in "string" with the standard C value specifier (%f, %d, etc.) in the order given.

## Performance

*fprints* is similar to the *prints* opcode except it outputs to a file. For more information about output formatting, please look at *prints's documentation*.

## Examples

Here is an example of the fprints opcode. It uses the files *fprints.orc* [examples/fprints.orc] and *fprints.sco* [examples/fprints.sco].

### Example 161. Example of the fprints opcode.

```
/* fprints.orc */
/* Written by Matt Ingalls, edited by Kevin Conder. */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a score generator example.
instr 1
  ; Print to the file "my.sco".
  fprints "my.sco", "%!Generated score by ma++\\n \\n"
endin
/* fprints.orc */
```

```
/* fprints.sco */
/* Written by Matt Ingalls, edited by Kevin Conder. */
```

```
; Play Instrument #1.  
i 1 0 0.001  
/* fprints.sco */
```

This example will generate a file called “my.sco”. It should contain a line like this:

```
;Generated score by ma++
```

## See Also

*prints*

## Credits

Author: Matt Ingalls  
January 2003

# frac

frac -- Returns the fractional part of a decimal number.

frac

## Description

Returns the fractional part of  $x$ .

## Syntax

**frac**( $x$ ) (init-rate or control-rate args; also works at audio rate in Csound5)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the frac opcode. It uses the files *frac.orc* [examples/frac.orc] and *frac.sco* [examples/frac.sco].

### Example 162. Example of the frac opcode.

```
/* frac.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = 16 / 5
  i2 = frac(i1)

  print i2
endin
/* frac.orc */

/* frac.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* frac.sco */
```

Its output should include a line like this:

```
instr 1:  i2 = 0.200
```

## See Also

*abs, exp, int, log, log10, i, sqrt*

## Credits

Example written by Kevin Conder.

# freeverb

freeverb -- Opcode version of Jezar's Freeverb

freeverb

## Description

freeverb is a stereo reverb unit based on Jezar's public domain C++ sources, composed of eight parallel comb filters on both channels, followed by four allpass units in series. The filters on the right channel are slightly detuned compared to the left channel in order to create a stereo effect.

## Syntax

```
aoutL, aoutR freeverb ainL, ainR, kRoomSize, kHFDamp[, iSRate[, iSkip]]
```

## Initialization

*iSRate* (optional, defaults to 44100): adjusts the reverb parameters for use with the specified sample rate (this will affect the length of the delay lines in samples, and, as of the latest CVS version, the high frequency attenuation). Only integer multiples of 44100 will reproduce the original character of the reverb exactly, so it may be useful to set this to 44100 or 88200 for an orchestra sample rate of 48000 or 96000 Hz, respectively. While *iSRate* is normally expected to be close to the orchestra sample rate, different settings may be useful for special effects.

*iSkip* (optional, defaults to zero): if non-zero, initialization of the opcode will be skipped, whenever possible.

## Performance

*ainL*, *ainR* -- input signals; usually both are the same, but different inputs can be used for special effect



### Note

It is recommended to process the input signal(s) with the denorm opcode in order to avoid denormalized numbers which could significantly increase CPU usage in some cases

*aoutL*, *aoutR* -- output signals for left and right channel

*kRoomSize* (range: 0 to 1) -- controls the length of the reverb, a higher value means longer reverb. Settings above 1 may make the opcode unstable.

*kHFDamp* (range: 0 to 1): high frequency attenuation; a value of zero means all frequencies decay at the same rate, while higher settings will result in a faster decay of the high frequency range.

## Examples

```
sr      = 44100
ksmps   = 32
nchnls  = 2
0dbfs   = 1

instr 1
a1      vco2 0.75, 440, 10
```



```
kfrq    port 100, 0.008, 20000
a1      butterlp a1, kfrq
a2      linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1      = a1 * a2
        denorm a1
aL, aR  freeverb a1, a1, 0.9, 0.35, sr, 0
        outs a1 + aL, a1 + aR
        endin
```

```
i 1 0 5
e
```

## Credits

Author: Istvan Varga  
2005

# ftchnls

ftchnls -- Returns the number of channels in a stored function table.

ftchnls

## Description

Returns the number of channels in a stored function table.

## Syntax

**ftchnls**(x) (init-rate args only)

## Performance

Returns the number of channels of a *GEN01* table, determined from the header of the original file. If the original file has no header or the table was not created by these GEN01, *ftchnls* returns -1.

## Examples

Here is an example of the *ftchnls* opcode. It uses the files *ftchnls.orc* [examples/ftchnls.orc], *ftchnls.sco* [examples/ftchnls.sco], and *mary.wav* [examples/mary.wav].

### Example 163. Example of the ftchnls opcode.

```
/* ftchnls.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the number of channels in Table #1.
  ichnls = ftchnls(1)
  print ichnls
endin
/* ftchnls.orc */

/* ftchnls.sco */
; Table #1: Use an audio file, Csound will determine its size.
f 1 0 0 1 "mary.wav" 0 0 0

; Play Instrument #1 for 1 second.
i 1 0 1
e
/* ftchnls.sco */
```

Since the audio file “mary.wav” is monophonic (1 channel), its output should include a line like

this:

```
instr 1:  ichnls = 1.000
```

## See Also

*ftlen, ftlptim, ftsr, nsamp*

## Credits

Author: Chris McCormick  
Perth, Australia  
December 2001

Example written by Kevin Conder.

## ftconv

ftconv -- Low latency multichannel convolution, using a function table as impulse response source.

ftconv

## Description

Low latency multichannel convolution, using a function table as impulse response source. The algorithm is to split the impulse response to partitions of length determined by the 'iplen' parameter, and delay and mix partitions so that the original, full length impulse response is reconstructed without gaps. The output delay (latency) is 'iplen' samples, and does not depend on the control rate, unlike in the case of other convolve opcodes.

## Syntax

```
a1[, a2[, a3[, ... a8]]] ftconv ain, ift, iplen[, iskip samples [, iirlen[, iskipinit
```

## Initialization

*ift* -- source ftable number. The table is expected to contain interleaved multichannel audio data, with the number of channels equal to the number of output variables (a1, a2, etc.). An interleaved table can be created from a set of mono tables with GEN52.

*iplen* -- length of impulse response partitions, in sample frames; must be an integer power of two. Lower settings allow for shorter output delay, but will increase CPU usage.

*iskipsamples* (optional, defaults to zero) -- number of sample frames to skip at the beginning of the table. Useful for reverb responses that have some amount of initial delay. If this delay is not less than 'iplen' samples, then setting iskip samples to the same value as iplen will eliminate any additional latency by ftconv.

*iirlen* (optional) -- total length of impulse response, in sample frames. The default is to use all table data (not including the guard point).

*iskipinit* (optional, defaults to zero) -- if set to any non-zero value, skip initialization whenever possible without causing an error.

## Performance

*ain* -- input signal.

*a1 ... a8* -- output signal(s).

## Example

```
<CsoundSynthesizer>
<CsInstruments>
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

garvb    init 0
gaW      init 0
gaX      init 0
gaY      init 0
```

```
itmp      ftgen  1, 0, 64, -2, 2, 40, -1, -1, -1, 123,      \
          1, 13.000, 0.05, 0.85, 20000.0, 0.0, 0.50, 2,    \
          1,  2.000, 0.05, 0.85, 20000.0, 0.0, 0.25, 2,    \
          1, 16.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,    \
          1,  9.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,    \
          1, 12.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2,    \
          1,  8.000, 0.05, 0.85, 20000.0, 0.0, 0.35, 2

itmp      ftgen 2, 0, 262144, -2, 0
          spat3dt 2, -0.2, 1, 0, 1, 1, 2, 0.005

itmp      ftgen 3, 0, 262144, -52, 3, 2, 0, 4, 2, 1, 4, 2, 2, 4

          instr 1

a1        vco2 1, 440, 10
kfrq      port 100, 0.008, 20000
a1        butterlp a1, kfrq
a2        linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1        = a1 * a2 * 2
          denorm a1
          vincr garvb, a1
aw, ax, ay, az spat3di a1, p4, p5, p6, 1, 1, 2
          vincr gaW, aw
          vincr gaX, ax
          vincr gaY, ay

          endin

          instr 2

          denorm garvb
; skip as many samples as possible without truncating the IR
arW, arX, arY ftconv garvb, 3, 2048, 2048, (65536 - 2048)
aW        = gaW + arW
aX        = gaX + arX
aY        = gaY + arY
garvb     = 0
gaW       = 0
gaX       = 0
gaY       = 0

aWre, aWim hilbert aW
aXre, aXim hilbert aX
aYre, aYim hilbert aY
aWXr      = 0.0928*aXre + 0.4699*aWre
aWXiYr    = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aL        = aWXr + aWXiYr
aR        = aWXr - aWXiYr

          outs aL, aR

          endin

</CsInstruments>
<CsScore>

i 1 0 0.5 0.0 2.0 -0.8
i 1 1 0.5 1.4 1.4 -0.6
i 1 2 0.5 2.0 0.0 -0.4
i 1 3 0.5 1.4 -1.4 -0.2
i 1 4 0.5 0.0 -2.0 0.0
i 1 5 0.5 -1.4 -1.4 0.2
i 1 6 0.5 -2.0 0.0 0.4
i 1 7 0.5 -1.4 1.4 0.6
i 1 8 0.5 0.0 2.0 0.8
i 2 0 10
e
```

---

```
</CsScore>  
</CsoundSynthesizer>
```

## Credits

Author: Istvan Varga  
2005

# ftfree

ftfree -- Deletes function table.

ftfree

## Description

Deletes function table.

## Syntax

```
ftfree ifno, iwhen
```

## Initialization

*ifno* -- the number of the table to be deleted

*iwhen* -- if zero the table is deleted at init time; otherwise the table number is registered for being deleted at note deactivation.

## Performance

## Credits

Authors: Steven Yi, Istvan Varga  
2005

# ftgen

ftgen -- Generate a score function table from within the orchestra.

ftgen

## Description

Generate a score function table from within the orchestra.

## Syntax

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

## Initialization

*gir* -- either a requested or automatically assigned table number above 100.

*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, iargb, ...* -- function table arguments. Correspond to p5 through *pn* of the score *f statement*.

## Performance

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



### Warning

Although Csound will not protest if ftgen is used inside instr-endin statements, this is not the intended or supported use, and must be handled with care as it has global effects. (In particular, a different size usually leads to relocation of the table, which may cause a crash or otherwise erratic behaviour.

## Examples

Here is an example of the ftgen opcode. It uses the files *ftgen.orc* [examples/ftgen.orc] and *ftgen.sco* [examples/ftgen.sco].

### Example 164. Example of the ftgen opcode.

```
/* ftgen.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1
```



```
; Table #1, a sine wave using the GEN10 routine.
gitemp ftgen 1, 0, 16384, 10, 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 440
  ; Use Table #1.
  ifn = 1

  al oscil kamp, kcps, ifn
  out al
endin
/* ftgen.orc */

/* ftgen.sco */
; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* ftgen.sco */
```

## See also

*GEN routine overview*

## Credits

Author: Barry L. Vercoe  
M.I.T., Cambridge, Mass  
1997

Example written by Kevin Conder.

Added warning April 2002 by Rasmus Ekman

# ftgentmp

ftgentmp -- Generate a score function table from within the orchestra, which is deleted at the end of the note.

ftgentmp

## Description

Generate a score function table from within the orchestra, which is optionally deleted at the end of the note.

## Syntax

```
ifno ftgentmp ip1, ip2dummy, isize,  
            igen, iarga, iargb, ...
```

## Initialization

*ifno* -- either a requested or automatically assigned table number above 100.

*ip1* -- the number of the table to be generated or 0 if the number is to be assigned, in which case the table is deleted at the end of the note activation.

*ip2dummy* -- ignored.

*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, iargb, ...* -- function table arguments. Correspond to p5 through *pn* of the score *f statement*.

## Performance

## Credits

Authors: Istvan Varga  
2005

# ftlen

ftlen -- Returns the size of a stored function table.

ftlen

## Description

Returns the size of a stored function table.

## Syntax

**ftlen**(*x*) (init-rate args only)

## Performance

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*).

*ftlen* differs from *nsamp* in that *nsamp* gives the number of sample frames loaded, while *ftlen* gives the total number of samples without the guard point. For example, with a stereo sound file of 10000 samples, *ftlen*() would return 19999 (i.e. a total of 20000 mono samples, not including a guard point), but *nsamp*() returns 10000.

## Examples

Here is an example of the *ftlen* opcode. It uses the files *ftlen.orc* [examples/ftlen.orc], *ftlen.sco* [examples/ftlen.sco], and *mary.wav* [examples/mary.wav].

### Example 165. Example of the *ftlen* opcode.

```
/* ftlen.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the size of Table #1.
  ; The size will be the number of points excluding the guard point.
  ilen = ftlen(1)
  print ilen
endin
/* ftlen.orc */

/* ftlen.sco */
; Table #1: Use an audio file, Csound will determine its size.
f 1 0 0 1 "mary.wav" 0 0 0
```

```
; Play Instrument #1 for 1 second.  
i 1 0 1  
e  
/* ftlen.sco */
```

The audio file “mary.wav” is 154390 samples long. The ftlen opcode reports it as 154389 samples long because it reserves 1 point for the guard point. Its output should include a line like this:

```
instr 1:  ilen = 154389.000
```

## See Also

*ftchnls, ftlptim, ftsr, nsamp*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachusetts  
1997

Example written by Kevin Conder.

# ftload

ftload -- Load a set of previously-allocated tables from a file.

ftload

## Description

Load a set of previously-allocated tables from a file.

## Syntax

```
ftload "filename", iflag, ifn1 [, ifn2] [...]
```

## Initialization

*"filename"* -- A quoted string containing the name of the file to load.

*iflag* -- Type of the file to load/save. (0 = binary file, Non-zero = text file)

*ifn1, ifn2, ...* -- Numbers of tables to load.

## Performance

*ftload* loads a list of tables from a file. (The tables have to be already allocated though.) The file's format can be binary or text.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## Examples

See the example for *ftsave*.

## See Also

*ftloadk, ftsavek, ftsave*

## Credits

Author: Gabriel Maldonado

New in version 4.21

# ftloadk

ftloadk -- Load a set of previously-allocated tables from a file.

ftloadk

## Description

Load a set of previously-allocated tables from a file.

## Syntax

**ftloadk** "filename", ktrig, iflag, ifn1 [, ifn2] [...]

## Initialization

*"filename"* -- A quoted string containing the name of the file to load.

*iflag* -- Type of the file to load/save. (0 = binary file, Non-zero = text file)

*ifn1*, *ifn2*, ... -- Numbers of tables to load.

## Performance

*ktrig* -- The trigger signal. Load the file each time it is non-zero.

*ftloadk* loads a list of tables from a file. (The tables have to be already allocated though.) The file's format can be binary or text. Unlike *ftload*, the loading operation can be repeated numerous times within the same note by using a trigger signal.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## See Also

*ftload*, *ftsavek*, *ftsave*

## Credits

Author: Gabriel Maldonado

New in version 4.21

# ftlptim

ftlptim -- Returns the loop segment start-time of a stored function table number.

ftlptim

## Description

Returns the loop segment start-time of a stored function table number.

## Syntax

**ftlptim**(x) (init-rate args only)

## Performance

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.

## Examples

Here is an example of the ftlptim opcode. It uses the files *ftlptim.orc* [examples/ftlptim.orc], *ftlptim.sco* [examples/ftlptim.sco], and *mary.wav* [examples/mary.wav].

### Example 166. Example of the ftlptim opcode.

```
/* ftlptim.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
    ; Print out the loop-segment start time in Table #1.
    itim = ftlptim(1)
    print itim
endin
/* ftlptim.orc */

/* ftlptim.sco */
; Table #1: Use an audio file, Csound will determine its size.
f 1 0 0 1 "mary.wav" 0 0 0

; Play Instrument #1 for 1 second.
i 1 0 1
e
/* ftlptim.sco */
```

Since the audio file “mary.wav” is non-looping, its output should include lines like this:

```
WARNING: non-looping sample  
instr 1: itim = 0.000
```

## See Also

*fchnls, filen, ftsr, nsamp*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachusetts  
1997

Example written by Kevin Conder.



# ftmorf

ftmorf -- Morphs between multiple ftables as specified in a list.

ftmorf

## Description

Uses an index into a table of ftable numbers to morph between adjacent tables in the list. This morphed function is written into the table referenced by *iresfn* on every k-cycle.

## Syntax

**ftmorf** kftndx, iftn, iresfn

## Initialization

*iftn* -- The ftable function. The list of values are expected to be pre-existing ftable numbers.

*iresfn* -- Table number of the morphed function

The length of all the tables in *iftn* must equal the length of *iresfn*.

## Performance

*kftndx* -- the index into the *iftn* table.

If *iftn* contains (6, 4, 6, 8, 7, 4):

- *kftndx*=4 will write the contents of f7 into *iresfn*.
- *kftndx*=4.5 will write the average of the contents of f7 and f4 into *iresfn*.

## Examples

Here is an example of the ftmorf opcode. It uses the files *ftmorf.orc* [examples/ftmorf.orc] and *ftmorf.sco* [examples/ftmorf.sco].

### Example 167. Example of the ftmorf opcode.

```
/* ftmorf.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
kndx    line    0, p3, 7
         ftmorf  kndx, 1, 2
asig    oscili  30000, 440, 2
         out     asig
endin
/* ftmorf.orc */
```

```
/* ftmorf.sco */
f1 0 8 -2 3 4 5 6 7 8 9 10
f2 0 1024 10 1 /*contents of f2 dont matter */
f3 0 1024 10 1
f4 0 1024 10 0 1
f5 0 1024 10 0 0 1
f6 0 1024 10 0 0 0 1
f7 0 1024 10 0 0 0 0 1
f8 0 1024 10 0 0 0 0 0 1
f9 0 1024 10 0 0 0 0 0 0 1
f10 0 1024 10 1 1 1 1 1 1 1 1

i1 0 10
e
/* ftmorf.sco */
```

## Credits

Author: William “Pete” Moss  
University of Texas at Austin  
Austin, Texas USA  
Jan. 2002

New in version 4.18

# ftsave

ftsave -- Save a set of previously-allocated tables to a file.

ftsave

## Description

Save a set of previously-allocated tables to a file.

## Syntax

**ftsave** "filename", iflag, ifn1 [, ifn2] [...]

## Initialization

*"filename"* -- A quoted string containing the name of the file to save.

*iflag* -- Type of the file to save. (0 = binary file, Non-zero = text file)

*ifn1*, *ifn2*, ... -- Numbers of tables to save.

## Performance

*ftsave* saves a list of tables to a file. The file's format can be binary or text.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## Examples

Here is an example of the ftsave opcode. It uses the files *ftsave.orc* [examples/ftsave.orc] and *ftsave.sco* [examples/ftsave.sco].

### Example 168. Example of the ftsave opcode.

```
/* ftsave.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Table #1, make a sine wave using the GEN10 routine.
gitmp1 ftgen 1, 0, 32768, 10, 1
; Table #2, create an empty table.
gitmp2 ftgen 2, 0, 32768, 7, 0, 32768, 0

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 20000
  kcps = 440
  ; Use Table #1.
  ifn = 1
```

```
    a1 oscil kamp, kcps, ifn
    out a1
endin

; Instrument #2 - Load Table #1 into Table #2.
instr 2
    ; Save Table #1 to a file called "table1.ftsav".
    ftsave "table1.ftsav", 0, 1

    ; Load the "table1.ftsav" file into Table #2.
    ftload "table1.ftsav", 0, 2

    kamp = 20000
    kcps = 440
    ; Use Table #2, it should contain Table #1's sine wave now.
    ifn = 2

    a1 oscil kamp, kcps, ifn
    out a1
endin
/* ftsave.orc */

/* ftsave.sco */
; Play Instrument #1 for 1 second.
i 1 0 1
; Play Instrument #2 for 1 second.
i 2 2 1
e
/* ftsave.sco */
```

## See Also

*ftloadk, ftload, ftsavek*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in version 4.21

# ftsavek

ftsavek -- Save a set of previously-allocated tables to a file.

ftsavek

## Description

Save a set of previously-allocated tables to a file.

## Syntax

**ftsavek** "filename", ktrig, iflag, ifn1 [, ifn2] [...]

## Initialization

*"filename"* -- A quoted string containing the name of the file to save.

*iflag* -- Type of the file to save. (0 = binary file, Non-zero = text file)

*ifn1*, *ifn2*, ... -- Numbers of tables to save.

## Performance

*ktrig* -- The trigger signal. Save the file each time it is non-zero.

*ftsavek* saves a list of tables to a file. The file's format can be binary or text. Unlike *ftsave*, the saving operation can be repeated numerous times within the same note by using a trigger signal.



### Warning

The file's format is not compatible with a WAV-file and is not endian-safe.

## See Also

*flloadk*, *ftload*, *ftsave*

## Credits

Author: Gabriel Maldonado

New in version 4.21

## ftsr

ftsr -- Returns the sampling-rate of a stored function table.

ftsr

## Description

Returns the sampling-rate of a stored function table.

## Syntax

**ftsr**(x) (init-rate args only)

## Performance

Returns the sampling-rate of a *GEN01* 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 GEN01, *ftsr* returns 0. New in Csound version 3.49.

## Examples

Here is an example of the *ftsr* opcode. It uses the files *ftsr.orc* [examples/ftsr.orc], *ftsr.sco* [examples/ftsr.sco], and *mary.wav* [examples/mary.wav].

### Example 169. Example of the ftsr opcode.

```
/* ftsr.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the sampling rate of Table #1.
  isr = ftsr(1)
  print isr
endin
/* ftsr.orc */
```

```
/* ftsr.sco */
; Table #1: Use an audio file.
f 1 0 262144 1 "mary.wav" 0 0 0

; Play Instrument #1 for 1 second.
i 1 0 1
e
/* ftsr.sco */
```

Since the audio file “mary.wav” uses a 44.1 Khz sampling rate, its output should a line like this:

```
instr 1:  isr = 44100.000
```

## See Also

*ftchnls, filen, filptim, nsamp*

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

Example written by Kevin Conder.

# gain

gain -- Adjusts the amplitude audio signal according to a root-mean-square value.

gain

## Description

Adjusts the amplitude audio signal according to a root-mean-square value.

## Syntax

```
ares gain asig, krms [, ihp] [, iskip]
```

## Initialization

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

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

## Performance

*asig* -- input audio signal

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

## Examples

```
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
```

## See Also

*balance*, *rms*



# gauss

gauss -- Gaussian distribution random number generator.

gauss

## Description

Gaussian distribution random number generator. This is an x-class noise generator.

## Syntax

ares **gauss** krange

ires **gauss** krange

kres **gauss** krange

## Performance

*krange* -- the range of the random numbers (*-krange* to *+krange*). Outputs both positive and negative numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the gauss opcode. It uses the files *gauss.orc* [examples/gauss.orc] and *gauss.sco* [examples/gauss.sco].

### Example 170. Example of the gauss opcode.

```
/* gauss.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number between -1 and 1.
  ; krange = 1

  il gauss 1

  print il
endin
```

```
/* gauss.orc */

/* gauss.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* gauss.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 0.252
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, linrand, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# gbuzz

gbuzz -- Output is a set of harmonically related cosine partials.

gbuzz

## Description

Output is a set of harmonically related cosine partials.

## Syntax

ares **gbuzz** xamp, xcps, knh, klh, kmul, ifn [, iphs]

## Initialization

*ifn* -- table number of a stored function containing a cosine wave. A large table of at least 8192 points is recommended.

*iphs* (optional, default=0) -- 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. If *knh* is negative, the absolute value is used. If *knh* is zero, a value of 1 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.

*kmul* -- 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 \* (*kmul* \*\* n), i.e. strength values trace an exponential curve. *kmul* 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* = *kmul* = 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. *kmul*, however, can be varied during performance to good effect. *gbuzz* can be amplitude- and/or frequency-modulated by either control or audio signals.

N.B. This unit has its analog in *GENII*, 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.

## Examples

Here is an example of the `gbuzz` opcode. It uses the files *gbuzz.orc* [examples/gbuzz.orc] and *gbuzz.sco* [examples/gbuzz.sco].

### Example 171. Example of the `gbuzz` opcode.

```
/* gbuzz.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 20000
  kcps = 440
  knh = 3
  klh = 2
  kmul = 0.7
  ifn = 1

  a1 gbuzz kamp, kcps, knh, klh, kmul, ifn
  out a1
endin
/* gbuzz.orc */

/* gbuzz.sco */
; Table #1, a simple cosine waveform.
f 1 0 16384 11 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* gbuzz.sco */
```

## See Also

*buzz*

## Credits

Example written by Kevin Conder.

September 2003. Thanks to Kanata Motohashi for correcting the mentions of the *kmul* parameter.

# getcfg

getcfg -- Return Csound settings

getcfg

## Description

Return various configuration settings in Svalue as a string at init time.

## Syntax

Svalue **getcfg** iopt

## Initialization

*iopt* -- The parameter to be returned, can be one of:

- 1: the maximum length of string variables in characters; this is at least the value of the -+max\_str\_len command line option - 1
- 2: the input sound file name (-i), or empty if there is no input file
- 3: the output sound file name (-o), or empty if there is no output file
- 4: return "1" if real time audio input or output is being used, and "0" otherwise
- 5: return "1" if running in beat mode (-t command line option), and "0" otherwise
- 6: the host operating system name
- 7: return "1" if a callback function for the chnrecv and chnsend opcodes has been set, and "0" otherwise (which means these opcodes do nothing)

## Credits

Author: Istvan Varga  
2006

# gogobel

`gogobel` -- Audio output is a tone related to the striking of a cow bell or similar.

`gogobel`

## 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.

## Syntax

`ares gogobel kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivfn`

## 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* [examples/marmstk1.wav] is a suitable function from measurements and can be loaded with a *GENO1* table. It is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

*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

## Examples

Here is an example of the `gogobel` opcode. It uses the files *gogobel.orc* [examples/gogobel.orc], *gogobel.sco* [examples/gogobel.sco], and *marmstk1.wav* [examples/marmstk1.wav],

### Example 172. Example of the `gogobel` opcode.

```
/* gogobel.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
    ; kamp = 31129.60
```

```
; kfreq = 440
; ihrd = 0.5
; ipos = 0.561
; imp = 1
; kvibf = 6.0
; kvamp = 0.3
; ivfn = 2

a1 gogobel 31129.60, 440, 0.5, 0.561, 1, 6.0, 0.3, 2
out a1
endin
/* gogobel.orc */

/* gogobel.sco */
; Table #1, the "marmstkl.wav" audio file.
f 1 0 256 1 "marmstkl.wav" 0 0 0
; Table #2, a sine wave for the vibrato.
f 2 0 128 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* gogobel.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# goto

goto -- Transfer control on every pass.

goto

## Description

Transfer control to *label* on every pass. (Combination of *igoto* and *kgoto*)

## Syntax

**goto** label

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

## Examples

Here is an example of the goto opcode. It uses the files *goto.orc* [examples/goto.orc] and *goto.sco* [examples/goto.sco].

### Example 173. Example of the goto opcode.

```
/* goto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  al oscil 10000, 440, 1
  goto playit

  ; The goto will go to the playit label.
  ; It will skip any code in between like this comment.

playit:
  out al
endin
/* goto.orc */

/* goto.sco */
; Table #1: a simple sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* goto.sco */
```



## See Also

*cggoto, cigoto, ckgoto, if, igoto, kgoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

Added a note by Jim Aikin.

# grain

grain -- Generates granular synthesis textures.

grain

## Description

Generates granular synthesis textures.

## Syntax

ares **grain** xamp, xpitch, xdens, kampoﬀ, kpitchoﬀ, kgdur, igfn, iwfn, imgdur [

## 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*.

*igrnd* (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:

$$\text{sndsr} / \text{flen}(\text{igfn})$$

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.

## Examples

This example generates a texture with gradually shorter grains and wider amp and pitch spread. It uses the files *grain.orc* [examples/grain.orc], *grain.sco* [examples/grain.sco], and *mary.wav*

[examples/mary.wav].

### Example 174. Example of the grain opcode.

```
/* grain.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1
  insnd = 10
  ibasfrq = 44100 / ftlen(insnd)    ; Use original sample rate of insnd file

  kamp   expseg 220, p3/2, 600, p3/2, 220
  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
/* grain.orc */

/* grain.sco */
f5 0 512 20 2 ; Hanning window
f10 0 262144 1 "mary.wav" 0 0 0
i1 0 6
e
/* grain.sco */
```

## Credits

Author: Paris Smaragdis  
MIT  
May 1997

# grain2

grain2 -- Easy-to-use granular synthesis texture generator.

grain2

## Description

Generate granular synthesis textures. *grain2* is simpler to use, but *grain3* offers more control.

## Syntax

ares **grain2** kcps, kfmd, kgdur, iovrlp, kfn, iwfn [, irpow] [, iseed] [, imode]

## Initialization

*iovrlp* -- (fixed) number of overlapping grains.

*iwfn* -- function table containing window waveform (Use GEN20 to calculate iwfn).

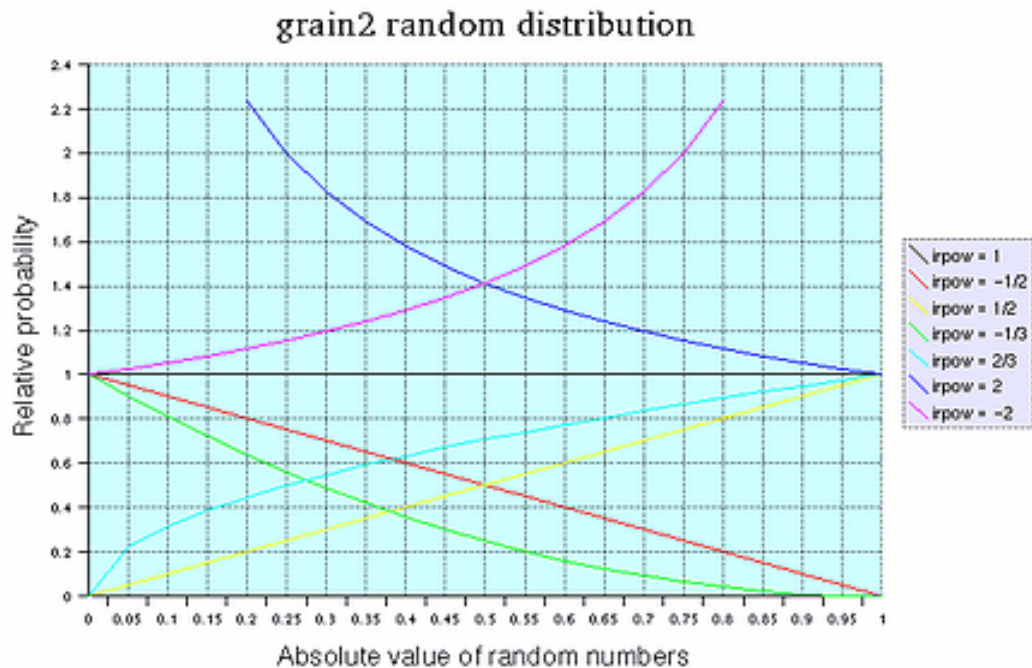
*irpow* (optional, default=0) -- this value controls the distribution of grain frequency variation. If *irpow* is positive, the random distribution (x is in the range -1 to 1) is

$$\text{abs}(x) ^ ((1 / \text{irpow}) - 1)$$

; for negative *irpow* values, it is

$$(1 - \text{abs}(x)) ^ ((-1 / \text{irpow}) - 1)$$

. Setting *irpow* to -1, 0, or 1 will result in uniform distribution (this is also faster to calculate). The image below shows some examples for *irpow*. The default value of *irpow* is 0.

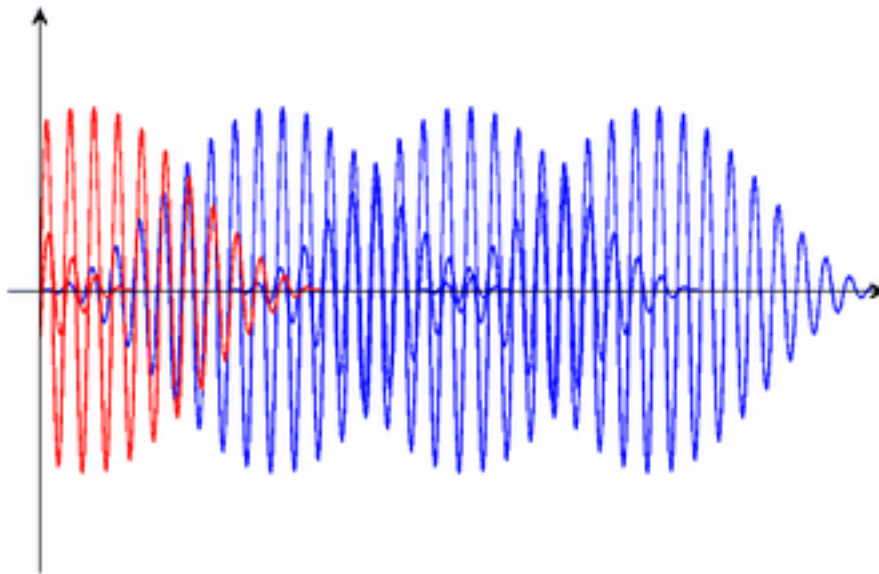


A graph of distributions for different values of *irpow*.

*iseed* (optional, default=0) -- seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). Zero or negative value seeds from current time (this is also the default).

*imode* (optional default=0) -- sum of the following values:

- 8: interpolate window waveform (slower).
- 4: do not interpolate grain waveform (fast, but lower quality).
- 2: grain frequency is continuously modified by *kcps* and *kfmd* (by default, each grain keeps the frequency it was launched with). This may be slower at high control rates.
- 1: skip initialization.



A diagram showing grains with a start time less than zero in red.

## Performance

*ares* -- output signal.

*kcps* -- grain frequency in Hz.

*kfmd* -- random variation (bipolar) in grain frequency in Hz.

*kgdur* -- grain duration in seconds. *kgdur* also controls the duration of already active grains (actually the speed at which the window function is read). This behavior does not depend on the *imode* flags.

*kfn* -- function table containing grain waveform. Table number can be changed at k-rate (this is useful to select from a set of band-limited tables generated by GEN30, to avoid aliasing).



### Note

*grain2* internally uses the same random number generator as *rnd31*. So reading *its documentation* is also recommended.

## Examples

Here is an example of the grain2 opcode. It uses the files *grain2.orc* [examples/grain2.orc] and *grain2.sco* [examples/grain2.sco].

### Example 175. Example of the grain2 opcode.

```
/* grain2.orc */
sr = 48000
kr = 750
ksmps = 64
nchnls = 2

/* square wave */
i_ ftgen 1, 0, 4096, 7, 1, 2048, 1, 0, -1, 2048, -1
/* window */
i_ ftgen 2, 0, 16384, 7, 0, 4096, 1, 4096, 0.3333, 8192, 0
/* sine wave */
i_ ftgen 3, 0, 1024, 10, 1
/* room parameters */
i_ ftgen 7, 0, 64, -2, 4, 50, -1, -1, -1, 11, \
1, 26.833, 0.05, 0.85, 10000, 0.8, 0.5, 2, \
1, 1.753, 0.05, 0.85, 5000, 0.8, 0.5, 2, \
1, 39.451, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
1, 33.503, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
1, 36.151, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
1, 29.633, 0.05, 0.85, 7000, 0.8, 0.5, 2

ga01 init 0

/* generate bandlimited square waves */

i0 = 0
loop1:
imaxh = sr / (2 * 440.0 * exp (log(2.0) * (i0 - 69) / 12))
i_ ftgen i0 + 256, 0, 4096, -30, 1, 1, imaxh
i0 = i0 + 1
if (i0 < 127.5) igoto loop1

instr 1

p3 = p3 + 0.2

/* note velocity */
iamp = 0.0039 + p5 * p5 / 16192
/* vibrato */
kcps oscili 1, 8, 3
kenv linseg 0, 0.05, 0, 0.1, 1, 1, 1
/* frequency */
kcps = (kcps * kenv * 0.01 + 1) * 440 * exp(log(2) * (p4 - 69) / 12)
/* grain ftable */
kfn = int(256 + 69 + 0.5 + 12 * log(kcps / 440) / log(2))
/* grain duration */
kgdur port 100, 0.1, 20
kgdur = kgdur / kcps

a1 grain2 kcps, kcps * 0.02, kgdur, 50, kfn, 2, -0.5, 22, 2
a1 butterlp a1, 3000
a2 grain2 kcps, kcps * 0.02, 4 / kcps, 50, kfn, 2, -0.5, 23, 2
a2 butterbp a2, 12000, 8000
a2 butterbp a2, 12000, 8000
aenv1 linseg 0, 0.01, 1, 1, 1
aenv2 linseg 3, 0.05, 1, 1, 1
aenv3 linseg 1, p3 - 0.2, 1, 0.07, 0, 1, 0

a1 = aenv1 * aenv3 * (a1 + a2 * 0.7 * aenv2)
```

```
ga01 = ga01 + a1 * 10000 * iamp
      endin

/* output instr */

      instr 81

i1 = 0.000001
aLl, aLh, aRl, aRh spat3di ga01 + i1*i1*i1*i1, 3.0, 4.0, 0.0, 0.5, 7, 4
ga01 = 0
aLl butterlp aLl, 800.0
aRl butterlp aRl, 800.0

      outs aLl + aLh, aRl + aRh

      endin
/* grain2.orc */


/* grain2.sco */
t 0 60

i 1 0.0 1.3 60 127
i 1 2.0 1.3 67 127
i 1 4.0 1.3 64 112
i 1 4.0 1.3 72 112

i 81 0 6.4

e
/* grain2.sco */
```

## See Also

*grain3*

## Credits

Author: Istvan Varga

New in version 4.15

Updated April 2002 by Istvan Varga

# grain3

grain3 -- Generate granular synthesis textures with more user control.

grain3

## Description

Generate granular synthesis textures. *grain2* is simpler to use but *grain3* offers more control.

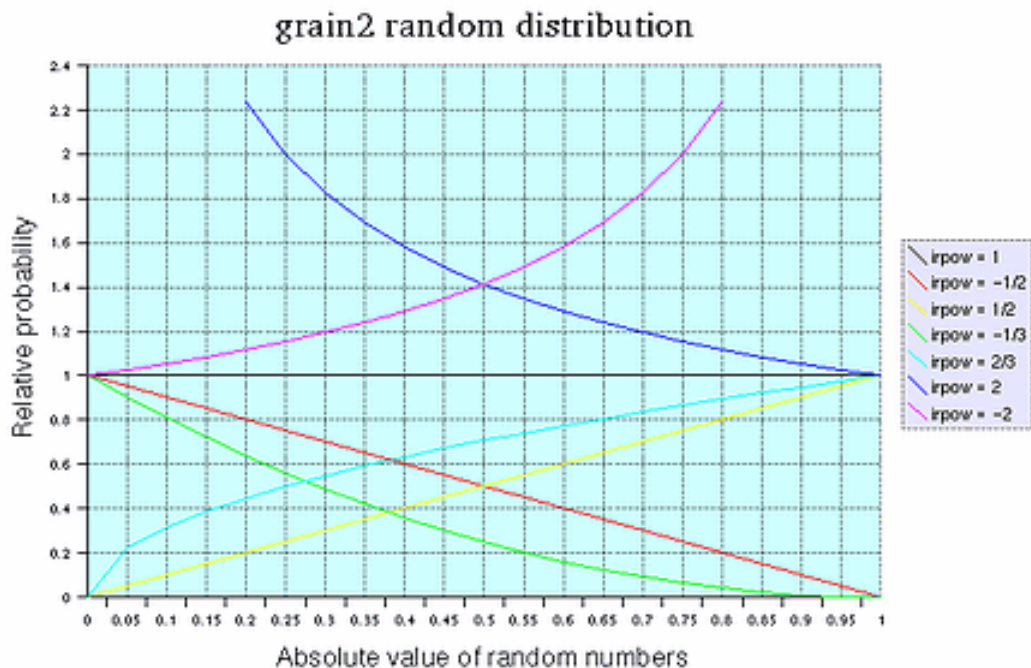
## Syntax

ares **grain3** kcps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, iwfn, kfrpow, k

## Initialization

*imaxovr* -- maximum number of overlapping grains. The number of overlaps can be calculated by  $(\text{kdens} * \text{kgdur})$ ; however, it can be overestimated at no cost in rendering time, and a single overlap uses (depending on system) 16 to 32 bytes of memory.

*iwfn* -- function table containing window waveform (Use GEN20 to calculate iwfn).



A graph of distributions for different values of *irpow*.

*iseed* (optional, default=0) -- seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). Zero or negative value seeds from current time (this is also the default).

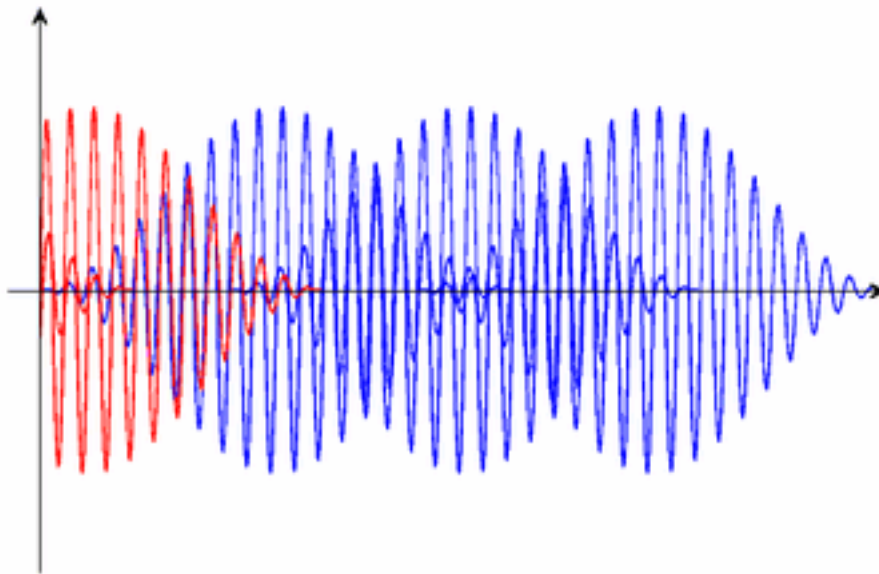
*imode* (optional, default=0) -- sum of the following values:

- 64: synchronize start phase of grains to kcps.
- 32: start all grains at integer sample location. This may be faster in some cases, however it also



makes the timing of grain envelopes less accurate.

- *16*: do not render grains with start time less than zero. (see the image below; this option turns off grains marked with red on the image).
- *8*: interpolate window waveform (slower).
- *4*: do not interpolate grain waveform (fast, but lower quality).
- *2*: grain frequency is continuously modified by *kcps* and *kfmd* (by default, each grain keeps the frequency it was launched with). This may be slower at high control rates. It also controls phase modulation (*kphs*).
- *1*: skip initialization.



A diagram showing grains with a start time less than zero in red.

## Performance

*ares* -- output signal.

*kcps* -- grain frequency in Hz.

*kphs* -- grain phase. This is the location in the grain waveform table, expressed as a fraction (between 0 to 1) of the table length.

*kfmd* -- random variation (bipolar) in grain frequency in Hz.

*kpmd* -- random variation (bipolar) in start phase.

*kgdur* -- grain duration in seconds. *kgdur* also controls the duration of already active grains (actually the speed at which the window function is read). This behavior does not depend on the *imode* flags.

*kdens* -- number of grains per second.

*kfrpow* -- distribution of random frequency variation (see *irpow*).

*kprpow* -- distribution of random phase variation (see *irpow*). Setting *kphs* and *kpmd* to 0.5, and *kprpow* to 0 will emulate *grain2*.

*kfn* -- function table containing grain waveform. Table number can be changed at k-rate (this is useful to select from a set of band-limited tables generated by GEN30, to avoid aliasing).



### Note

*grain3* internally uses the same random number generator as *rnd31*. So reading *its documentation* is also recommended.

## Examples

Here is an example of the *grain3* opcode. It uses the files *grain3.orc* [examples/grain3.orc] and *grain3.sco* [examples/grain3.sco].

### Example 176. Example of the grain3 opcode.

```
/* grain3.orc */
sr = 48000
kr = 1000
ksmps = 48
nchnls = 1

/* Bartlett window */
itmp ftgen 1, 0, 16384, 20, 3, 1
/* sawtooth wave */
itmp ftgen 2, 0, 16384, 7, 1, 16384, -1
/* sine */
itmp ftgen 4, 0, 1024, 10, 1
/* window for "soft sync" with 1/32 overlap */
itmp ftgen 5, 0, 16384, 7, 0, 256, 1, 7936, 1, 256, 0, 7936, 0
/* generate bandlimited sawtooth waves */
itmp ftgen 3, 0, 4096, -30, 2, 1, 2048
icnt = 0
loop01:
; 100 tables for 8 octaves from 30 Hz
ifrq = 30 * exp(log(2) * 8 * icnt / 100)
itmp ftgen icnt + 100, 0, 4096, -30, 3, 1, sr / (2 * ifrq)
icnt = icnt + 1
if (icnt < 99.5) igoto loop01
/* convert frequency to table number */
#define FRQ2FNUM(xout'xcps'xbsfn) #

$xout = int(($xbsfn) + 0.5 + (100 / 8) * log(($xcps) / 30) / log(2))
$xout limit $xout, $xbsfn, $xbsfn + 99

#

/* instr 1: pulse width modulated grains */

instr 1

kfrq = 523.25 ; frequency
$FRQ2FNUM(kfnum'kfrq'100) ; table number
kfmd = kfrq * 0.02 ; random variation in frequency
kgdur = 0.2 ; grain duration
kdens = 200 ; density
iseed = 1 ; random seed

kphs oscili 0.45, 1, 4 ; phase

a1 grain3 kfrq, 0, kfmd, 0.5, kgdur, kdens, 100, \
kfnum, 1, -0.5, 0, iseed, 2
a2 grain3 kfrq, 0.5 + kphs, kfmd, 0.5, kgdur, kdens, 100, \
```

```

        kfnum, 1, -0.5, 0, iseed, 2

; de-click
aenv linseg 0, 0.01, 1, p3 - 0.05, 1, 0.04, 0, 1, 0

        out aenv * 2250 * (a1 - a2)

        endin

/* instr 2: phase variation */

        instr 2

kfqr = 220                ; frequency
$FRQ2FNUM(kfnum'kfqr'100) ; table number
kgdur = 0.2              ; grain duration
kdens = 200              ; density
iseed = 2                ; random seed

kprdst expon 0.5, p3, 0.02 ; distribution

a1 grain3 kfqr, 0.5, 0, 0.5, kgdur, kdens, 100, \
        kfnum, 1, 0, -kprdst, iseed, 64

; de-click
aenv linseg 0, 0.01, 1, p3 - 0.05, 1, 0.04, 0, 1, 0

        out aenv * 1500 * a1

        endin

/* instr 3: "soft sync" */

        instr 3

kdens = 130.8            ; base frequency
kgdur = 2 / kdens        ; grain duration

kfqr expon 880, p3, 220 ; oscillator frequency
$FRQ2FNUM(kfnum'kfqr'100) ; table number

a1 grain3 kfqr, 0, 0, 0, kgdur, kdens, 3, kfnum, 5, 0, 0, 0, 2
a2 grain3 kfqr, 0.667, 0, 0, kgdur, kdens, 3, kfnum, 5, 0, 0, 0, 2

; de-click
aenv linseg 0, 0.01, 1, p3 - 0.05, 1, 0.04, 0, 1, 0

        out aenv * 10000 * (a1 - a2)

        endin
/* grain3.orc */


/* grain3.sco */
t 0 60
i 1 0 3
i 2 4 3
i 3 8 3
e
/* grain3.sco */
```

## See Also

*grain2*

## Credits

Author: Istvan Varga

New in version 4.15

Updated April 2002 by Istvan Varga

# granule

*granule* -- A more complex granular synthesis texture generator.

*granule*

## 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.

## Syntax

ares **granule** xamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip, igskip\_os

## Performance

*xamp* -- amplitude.

*ivoice* -- number of voices.

*iratio* -- ratio of the speed of the gskip pointer relative to output audio sample rate. eg. 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.

*kgsiz*e -- 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* -- decay of the grain envelope in % of grain size.

*iseed* (optional, default=0.5) -- seed for the random number generator.

*ipitch1*, *ipitch2*, *ipitch3*, *ipitch4* (optional, default=1) -- 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.

*ifnenv* (optional, default=0) -- function table number to be used to generate the shape of the envelope.

## Examples

Here is an example of the *granule* opcode. It uses the files *granule.orc* [examples/granule.orc], *granule.sco* [examples/granule.sco], and *mary.wav* [examples/mary.wav].

### Example 177. Example of the *granule* opcode.

```
/* granule.orc */
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
/* granule.orc */

/* granule.sco */
; f statement read sound file sine.aiff in the SFDIR
; directory into f-table 1
f1      0 262144 1 "mary.wav" 0 0 0
i1      0 10 2000 64 0.5 0 0 1 4 0 0.005 5 0.01 50 0.02 50 30 30 0.39 \
        1 1.42 0.29 2
e
/* granule.sco */
```

The above example reads a sound file called *mary.wav* into wavetable number 1 with 262,144 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:

Parameter	Interpreted As
p5 ( <i>ivoice</i> )	the number of voices is set to 64
p6 ( <i>iratio</i> )	set to 0.5, it scans the wavetable at half of the speed of the audio output rate
p7 ( <i>imode</i> )	set to 0, the grain pointer only move forward
p8 ( <i>ithd</i> )	set to 0, skipping the thresholding process
p9 ( <i>ifn</i> )	set to 1, function table number 1 is used
p10 ( <i>ipshift</i> )	set to 4, four different pitches are going to be generated
p11 ( <i>igskip</i> )	set to 0 and p12 ( <i>igskip_os</i> ) is set to 0.005, no skipping into the wavetable and a 5 mSec random offset is used
p13 ( <i>ilength</i> )	set to 5, 5 seconds of the wavetable is to be used
p14 ( <i>kgap</i> )	set to 0.01 and p15 ( <i>igap_os</i> ) is set to 50, 10 mSec gap with 50% random offset is to be used
p16 ( <i>kgsz</i> )	set to 0.02 and p17 ( <i>igsize_os</i> ) is set to 50, 20 mSec grain with 50% random offset is used
p18 ( <i>iatt</i> ) and p19 ( <i>idec</i> )	set to 30, 30% of linear attack and decade is applied to the grain
p20 ( <i>iseed</i> )	seed for the random number generator is set to 0.39
p21 - p24	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.

## Credits

Author: Allan Lee  
Belfast  
1996

# guiro

guiro -- Semi-physical model of a guiro sound.

guiro

## Description

*guiro* is a semi-physical model of a guiro sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **guiro** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1]

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 128.

*idamp* (optional) -- the damping factor of the instrument. *Not used*.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2500.

*ifreq1* (optional) -- the first resonant frequency.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the guiro opcode. It uses the files *guiro.orc* [examples/guiro.orc] and *guiro.sco* [examples/guiro.sco].

### Example 178. Example of the guiro opcode.

```
/* guiro.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 01 ;example of a guiro
a1 guiro p4, 0.01
out a1
endin
/* guiro.orc */
```



```
/* guiro.sco */  
il 0 1 20000  
e  
/* guiro.sco */
```

## See Also

*bamboo, dripwater, sleighbells, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# harmon

harmon -- Analyze an audio input and generate harmonizing voices in synchrony.

harmon

## Description

Analyze an audio input and generate harmonizing voices in synchrony.

## Syntax

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

## Initialization

*imode* -- interpreting mode for the generating frequency inputs *kgenfreq1*, *kgenfreq2*. 0: input values are ratios with respect to the audio signal analyzed frequency. 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

*kestfrq* -- estimated frequency of the input.

*kmaxvar* -- the maximum variance.

*kgenfreq1* -- the first generated frequency.

*kgenfreq2* -- the second generated frequency.

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.

## Examples

Here is an example of the harmon opcode. It uses the files *harmon.orc* [examples/harmon.orc] and *harmon.sco* [examples/harmon.sco].

### Example 179. Example of the harmon opcode.

```
/* harmon.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; The frequency of the base note.
  inote = 440

  ; Generate the base note.
  avco vco 20000, inote, 1

  kestfrq = inote
  kmaxvar = 200

  ; Calculate frequencies 3 semitones above and
  ; below the base note.
  kgenfreq1 = inote * semitone(3)
  kgenfreq2 = inote * semitone(-3)

  imode = 1
  iminfrq = inote - 200
  iprd = 0.1

  ; Generate the harmony notes.
  a1 harmon avco, kestfrq, kmaxvar, kgenfreq1, kgenfreq2, \
    imode, iminfrq, iprd

  out a1
endin
/* harmon.orc */

/* harmon.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* harmon.sco */
```

## Credits

Author: Barry L. Vercoe  
M.I.T., Cambridge, Mass  
1997

Example written by Kevin Conder.

# hilbert

hilbert -- A Hilbert transformer.

hilbert

## Description

An IIR implementation of a Hilbert transformer.

## Syntax

```
ar1, ar2 hilbert asig
```

## 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.

Here is the first example of the *hilbert* opcode. It uses the files *hilbert.orc* [examples/hilbert.orc], *hilbert.sco* [examples/hilbert.sco], and *mary.wav* [examples/mary.wav].

**Example 180. Example of the *hilbert* opcode implementing frequency shifting.**

```
/* hilbert.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  idur = p3
  ; Initial amount of frequency shift.
  ; It can be positive or negative.
  ibegshift = p4
  ; Final amount of frequency shift.
  ; It can be positive or negative.
  iendshift = p5

  ; A simple envelope for determining the
  ; amount of frequency shift.
  kfreq linseg ibegshift, idur, iendshift

  ; Use the sound of your choice.
  ain soundin "mary.wav"

  ; Phase quadrature output derived from input signal.
  areal, aimag hilbert ain

  ; Quadrature oscillator.
  asin oscili 1, kfreq, 1
  acos oscili 1, kfreq, 1, .25

  ; Use a trigonometric identity.
  ; See the references for further details.
  amod1 = areal * acos
  amod2 = aimag * asin

  ; Both sum and difference frequencies can be
  ; output at once.
  ; aupshift corresponds to the sum frequencies.
  aupshift = (amod1 + amod2) * 0.7
  ; adownshift corresponds to the difference frequencies.
  adownshift = (amod1 - amod2) * 0.7

  ; Notice that the adding of the two together is
  ; identical to the output of ring modulation.

  out aupshift
endin
/* hilbert.orc */

/* hilbert.sco */
; Sine table for quadrature oscillator.
f 1 0 16384 10 1

; Starting with no shift, ending with all
; frequencies shifted up by 200 Hz.
i 1 0 2 0 200

; Starting with no shift, ending with all
; frequencies shifted down by 200 Hz.
i 1 2 2 0 -200
e
/* hilbert.sco */
```

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”.

Here is the second example of the *hilbert* opcode. It uses the files *hilbert\_barberpole.orc* [examples/hilbert\_barberpole.orc], *hilbert\_barberpole.sco* [examples/hilbert\_barberpole.sco], and *mary.wav* [examples/mary.wav].

**Example 181. Example of the *hilbert* opcode sounding like a “barberpole phaser”.**

```
/* hilbert_barberpole.orc */
; Initialize the global variables.
sr = 44100
; kr must equal sr for the barberpole effect to work.
kr = 44100
ksmps = 1
nchnls = 2

; Instrument #1
instr 1
  idur = p3
  ibegshift = p4
  iendshift = p5

  ; sawtooth wave, not bandlimited
  asaw phasor 100
  ; add offset to center phasor amplitude between -.5 and .5
  asaw = asaw - .5
  ; sawtooth wave, with amplitude of 10000
  ain = asaw * 20000

  ; The envelope of the frequency shift.
  kfreq linseg ibegshift, idur, iendshift

  ; Phase quadrature output derived from input signal.
  areal, aimag hilbert ain

  ; The quadrature oscillator.
  asin oscili 1, kfreq, 1
  acos oscili 1, kfreq, 1, .25

  ; Based on trigonometric identities.
  amod1 = areal * acos
  amod2 = aimag * asin

  ; Calculate the up-shift and down-shift.
  aupshift = (amod1 + amod2) * 0.7
  adownshift = (amod1 - amod2) * 0.7

  ; Mix in the original signal to achieve the barberpole effect.
  amix1 = aupshift + ain
  amix2 = adownshift + ain

  ; Make sure the output doesn't get louder than the original signal.
  aout1 balance amix1, ain
  aout2 balance amix2, ain

  outs aout1, aout2
endin
/* hilbert_barberpole.orc */
```

```
/* hilbert_barberpole.sco */  
; Table 1: A sine wave for the quadrature oscillator.  
f 1 0 16384 10 1  
  
; The score.  
; p4 = frequency shifter, starting frequency.  
; p5 = frequency shifter, ending frequency.  
i 1 0 6 -10 10  
e  
/* hilbert_barberpole.sco */
```

## Technical History

The use of phase-difference networks in frequency shifters was pioneered by Harald Bode.<sup>1</sup> Bode and Bob Moog provide an excellent description of the implementation and use of a frequency shifter in the analog realm in;<sup>2</sup> 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.<sup>3</sup> A recent paper by Scott Wardle<sup>4</sup> 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 Shifter 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.iaa.upf.es/dafx98/papers/>.

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55

The examples were updated April 2002. Thanks go to Sean Costello for fixing the barberpole example.



## hrtfer

`hrtfer` -- Creates 3D audio for two speakers.

`hrtfer`

## Description

Output is binaural (headphone) 3D audio.

## Syntax

`aleft, aright hrtfer asig, kaz, kelev, "HRTFcompact"`

## 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 below horizontal.

At present, the only file which can be used with *hrtfer* is *HRTFcompact* [examples/HRTFcompact]. It must be passed to the opcode as the last argument within quotes as shown above.

HRTFcompact may also be obtained via anonymous ftp from:  
<ftp://ftp.cs.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.

## Examples

Here is an example of the *hrtfer* opcode. It uses the files *hrtfer.orc* [examples/hrtfer.orc], *hrtfer.sco* [examples/hrtfer.sco], *HRTFcompact* [examples/HRTFcompact], and *beats.wav* [examples/beats.wav].

### Example 182. Example of the *hrtfer* opcode.

```
/* hrtfer.orc */
```

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

instr 1
  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 "beats.wav"
  aleft,aright hrtfer asrc, kaz, kel, "HRTFcompact"
  aleftscale   = aleft * 200
  arightscale  = aright * 200

  outs        aleftscale, arightscale
endin
/* hrtfer.orc */

/* hrtfer.sco */
i 1 0 2
e
/* hrtfer.sco */
```

## Credits

Authors: Eli Breder and David MacIntyre  
Montreal  
1996

Fixed the example thanks to a message from Istvan Varga.

# hsboscil

hsboscil -- An oscillator which takes tonality and brightness as arguments.

hsboscil

## Description

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

## Syntax

ares **hsboscil** kamp, ktone, kbrite, ibasfreq, iwfn, ioctfn [, ioctcnt] [, iphs]

## Initialization

*ibasfreq* -- base frequency to which tonality and brightness 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* (optional) -- number of octaves used for brightness blending. Must be in the range 2 to 10. Default is 3.

*iphs* (optional, default=0) -- 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 original 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

Here is an example of the hsboscil opcode. It uses the files *hsboscil.orc* [examples/hsboscil.orc] and *hsboscil.sco* [examples/hsboscil.sco].

**Example 183. Example of the hsboscil opcode.**

```
/* hsboscil.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

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

; Instrument #1 - produces Risset's glissando.
instr 1
  kamp = 10000
  kbrite = 0.5
  ibasfreq = 200
  ioctcnt = 5

  ; Change ktone linearly from 0 to 1,
  ; over the period defined by p3.
  ktone line 0, p3, 1

  a1 hsboscil kamp, ktone, kbrite, ibasfreq, giwave, giblend, ioctcnt
  out a1
endin
/* hsboscil.orc */

/* hsboscil.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* hsboscil.sco */
```

Here is an example of the hsboscil opcode in a MIDI instrument. It uses the files *hsboscil\_midi.orc* [examples/hsboscil\_midi.orc] and *hsboscil\_midi.sco* [examples/hsboscil\_midi.sco].

**Example 184. Example of the hsboscil opcode in a MIDI instrument.**

```
/* hsboscil_midi.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

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

; Instrument #1 - use hsboscil in a MIDI instrument.
```

```
instr 1
  ibase = cpsoct(6)
  ioctcnt = 5

  ; all octaves sound alike.
  itona octmidi
  ; velocity is mapped to brightness
  ibrite ampmidi 3

  ; Map an exponential envelope for the amplitude.
  kenv expon 20000, 1, 100

  asig hsboscil kenv, itona, ibrite, ibase, giwave, giblend, ioctcnt
  out asig
endin
/* hsboscil_midi.orc */

/* hsboscil_midi.sco */
; Play Instrument #1 for ten minutes
i 1 0 6000
e
/* hsboscil_midi.sco */
```

## Credits

Author: Peter Neubäcker  
Munich, Germany  
August, 1999

New in Csound version 3.58

# i

`i` -- Returns an init-type equivalent of a k-rate argument.

`i`

## Description

Returns an init-type equivalent of a k-rate argument.

## Syntax

`i(x)` (control-rate args only)

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



### Note

Using `i()` with a k-rate expression argument is not recommended, and can produce unexpected results.

## See Also

*a, k, abs, exp, frac, int, log, log10, sqrt*

# ibetarand

ibetarand -- Deprecated.

ibetarand

## Description

Deprecated as of version 3.49. Use the *betarand* opcode instead.

## ibexprnd

ibexprnd -- Deprecated.

ibexprnd

## Description

Deprecated as of version 3.49. Use the *bexprnd* opcode instead.



# icauchy

icauchy -- Deprecated.

icauchy

## Description

Deprecated as of version 3.49. Use the *cauchy* opcode instead.

# ictrl14

ictrl14 -- Deprecated.

ictrl14

## Description

Deprecated as of version 3.52. Use the *ctrl14* opcode instead.

## ictrl21

ictrl21 -- Deprecated.

ictrl21

### Description

Deprecated as of version 3.52. Use the *ctrl21* opcode instead.

## ictrl7

ictrl7 -- Deprecated.

ictrl7

## Description

Deprecated as of version 3.52. Use the *ctrl7* opcode instead.

# iexprand

iexprand -- Deprecated.

iexprand

## Description

Deprecated as of version 3.49. Use the *exprand* opcode instead.

## if

if -- Branches conditionally at initialization or during performance time.

if

## Description

*if...igoto* -- conditional branch at initialization 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 performance 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.

*if...then* -- allows the ability to specify conditional *if/else/endif* blocks. All *if...then* blocks must end with an *endif* statement. *elseif* and *else* statements are optional. Any number of *elseif* statements are allowed. Only one *else* statement may occur and it must be the last conditional statement before the *endif* statement. Nested *if...then* blocks are allowed.



### Note

Note that if the condition uses a k-rate variable (for instance, “if kval > 0”), the *if...goto* or *if...then* statement will be ignored during the i-time pass. This allows for opcode initialization, even if the k-rate variable has already been assigned an appropriate value by an earlier init statement.

## Syntax

```
if ia R ib igoto label
```

```
if ka R kb kgoto label
```

```
if ia R ib goto label
```

```
if xa R xb then
```

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*).

## Examples

Here is an example of the *if...igoto* combination. It uses the files *igoto.orc* [examples/igoto.orc] and *igoto.sco* [examples/igoto.sco].

### Example 185. Example of the if...igoto combination.

```
/* igoto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
```

```
nchnls = 1

; Instrument #1.
instr 1
  ; Get the value of the 4th p-field from the score.
  iparam = p4

  ; If iparam is 1 then play the high note.
  ; If not then play the low note.
  if (iparam == 1) igoto highnote
  igoto lownote

highnote:
  ifreq = 880
  goto playit

lownote:
  ifreq = 440
  goto playit

playit:
  ; Print the values of iparam and ifreq.
  print iparam
  print ifreq

  a1 oscil 10000, ifreq, 1
  out a1
endin
/* igoto.orc */

/* igoto.sco */
; Table #1: a simple sine wave.
f 1 0 32768 10 1

; p4: 1 = high note, anything else = low note
; Play Instrument #1 for one second, a low note.
i 1 0 1 0
; Play a Instrument #1 for one second, a high note.
i 1 1 1 1
e
/* igoto.sco */
```

Its output should include lines like this:

```
instr 1:  iparam = 0.000
instr 1:  ifreq = 440.000
instr 1:  iparam = 1.000
instr 1:  ifreq = 880.000
```

Here is an example of the if...kgoto combination. It uses the files *kgoto.orc* [examples/kgoto.orc] and *kgoto.sco* [examples/kgoto.sco].

### **Example 186. Example of the if...kgoto combination.**

```
/* kgoto.orc */
```

---

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Change kval linearly from 0 to 2 over
  ; the period set by the third p-field.
  kval line 0, p3, 2

  ; If kval is greater than or equal to 1 then play the high note.
  ; If not then play the low note.
  if (kval >= 1) kgoto highnote
  kgoto lownote

highnote:
  kfreq = 880
  goto playit

lownote:
  kfreq = 440
  goto playit

playit:
  ; Print the values of kval and kfreq.
  printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

  a1 oscil 10000, kfreq, 1
  out a1
endin
/* kgoto.orc */

/* kgoto.sco */
; Table #1: a simple sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* kgoto.sco */
```

Its output should include lines like this:

```
kval = 0.000000, kfreq = 440.000000
kval = 0.999732, kfreq = 440.000000
kval = 1.999639, kfreq = 880.000000
```

## Examples

Here is an example of the if...then combo. It uses the files *ifthen.orc* [examples/ifthen.orc] and *ifthen.sco* [examples/ifthen.sco].

### Example 187. Example of the if...then combo.



```
/* ifthen.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Get the note value from the fourth p-field.
  knote = p4

  ; Does the user want a low note?
  if (knote == 0) then
    kcps = 220
  ; Does the user want a middle note?
  elseif (knote == 1) then
    kcps = 440
  ; Does the user want a high note?
  elseif (knote == 2) then
    kcps = 880
  endif

  ; Create the note.
  kamp init 25000
  ifn = 1
  al oscili kamp, kcps, ifn

  out al
endin
/* ifthen.orc */

/* ifthen.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; p4: 0=low note, 1=middle note, 2=high note.
; Play Instrument #1 for one second, low note.
i 1 0 1 0
; Play Instrument #1 for one second, middle note.
i 1 1 1 1
; Play Instrument #1 for one second, high note.
i 1 2 1 2
e
/* ifthen.sco */
```

## See Also

*elseif, else, endif, goto, igoto, kgoto, tigoto, timeout*

## Credits

Examples written by Kevin Conder.

Added a note by Jim Aikin.

February 2004. Added a note by Matt Ingalls.

# igauss

igauss -- Deprecated.

igauss

## Description

Deprecated as of version 3.49. Use the *gauss* opcode instead.

# igoto

igoto -- Transfer control during the i-time pass.

igoto

## Description

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

## Syntax

**igoto** label

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

## Examples

Here is an example of the igoto opcode. It uses the files *igoto.orc* [examples/igoto.orc] and *igoto.sco* [examples/igoto.sco].

### Example 188. Example of the igoto opcode.

```
/* igoto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Get the value of the 4th p-field from the score.
iparam = p4

; If iparam is 1 then play the high note.
; If not then play the low note.
if (iparam == 1) igoto highnote
igoto lownote

highnote:
ifreq = 880
goto playit

lownote:
ifreq = 440
goto playit

playit:
; Print the values of iparam and ifreq.
print iparam
print ifreq

a1 oscil 10000, ifreq, 1
out a1
endin
/* igoto.orc */
```

```
/* igoto.sco */
; Table #1: a simple sine wave.
f 1 0 32768 10 1

; p4: 1 = high note, anything else = low note
; Play Instrument #1 for one second, a low note.
i 1 0 1 0
; Play a Instrument #1 for one second, a high note.
i 1 1 1 1
e
/* igoto.sco */
```

Its output should include lines like this:

```
instr 1:  iparam = 0.000
instr 1:  ifreq = 440.000
instr 1:  iparam = 1.000
instr 1:  ifreq = 880.000
```

## See Also

*cggoto, cigoto, ckgoto, goto, if, kgoto, rigoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

Added a note by Jim Aikin.

# ihold

ihold -- Creates a held note.

ihold

## Description

Causes a finite-duration note to become a “held” note

## Syntax

ihold

## Performance

*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.

## Examples

Here is an example of the *ihold* opcode. It uses the files *ihold.orc* [examples/ihold.orc] and *ihold.sco* [examples/ihold.sco].

### Example 189. Example of the *ihold* opcode.

```
/* ihold.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; A simple oscillator with its note held indefinitely.
a1 oscil 10000, 440, 1
ihold

; If p4 equals 0, turn the note off.
if (p4 == 0) kgoto offnow
kgoto playit

offnow:
; Turn the note off now.
turnoff

playit:
; Play the note.
out a1
endin
/* ihold.orc */
```

```
/* ihold.sco */
; Table #1: an ordinary sine wave.
f 1 0 32768 10 1

; p4 = turn the note off (if it is equal to 0).
; Start playing Instrument #1.
i 1 0 1 1
; Turn Instrument #1 off after 3 seconds.
i 1 3 1 0
e
/* ihold.sco */
```

## See Also

*i Statement, turnoff*

## Credits

Example written by Kevin Conder.

# ilinrand

ilinrand -- Deprecated.

ilinrand

## Description

Deprecated as of version 3.49. Use the *linrand* opcode instead.

# imidic14

imidic14 -- Deprecated.

imidic14

## Description

Deprecated as of version 3.52. Use the *midic14* opcode instead.



## imidic21

imidic21 -- Deprecated.

imidic21

## Description

Deprecated as of version 3.52. Use the *midic21* opcode instead.

## imidic7

imidic7 -- Deprecated.

imidic7

## Description

Deprecated as of version 3.52. Use the *midic7* opcode instead.

# in

in -- Reads mono audio data from an external device or stream.

in

## Description

Reads mono audio data from an external device or stream.

## Syntax

ar1 **in**

## Performance

Reads mono audio data from an external device or stream. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## See Also

*diskin, inh, ino, inq, ins, soundin*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# in32

in32 -- Reads a 32-channel audio signal from an external device or stream.

in32

## Description

Reads a 32-channel audio signal from an external device or stream.

## Syntax

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, ar13, ar14, ar15

## Performance

*in32* reads a 32-channel audio signal from an external device or stream. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer.

## Credits

*inch*, *inx*, *inz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# inch

*inch* -- Reads from a numbered channel in an external audio signal or stream.

*inch*

## Description

Reads from a numbered channel in an external audio signal or stream.

## Syntax

```
ar1 inch ksig1
```

## Performance

*inch* reads from a numbered channel determined by *ksig1* into *a1*. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer.

## Credits

*in32*, *inx*, *inz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# inh

inh -- Reads six-channel audio data from an external device or stream.

inh

## Description

Reads six-channel audio data from an external device or stream.

## Syntax

ar1, ar2, ar3, ar4, ar5, ar6 **inh**

## Performance

Reads six-channel audio data from an external device or stream. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## See Also

*diskin, in, ino, inq, ins, soundin*

## Credits

Author: John ffitch

# init

init -- Puts the value of the i-time expression into a k- or a-rate variable.

init

## Syntax

ares **init** iarg

ires **init** iarg

kres **init** iarg

## Description

Put the value of the i-time expression into a k- or a-rate variable.

## Initialization

Puts 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.

## See Also

=, *divz*, *tival*

# initc14

*initc14* -- Initializes the controllers used to create a 14-bit MIDI value.

*initc14*

## Description

Initializes the controllers used to create a 14-bit MIDI value.

## Syntax

**initc14** *ichan*, *ictlno1*, *ictlno2*, *ivalue*

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlno1* -- most significant byte controller number (0-127)

*ictlno2* -- least significant byte controller number (0-127)

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

## Performance

*initc14* can be used together with both *midic14* and *ctrl14* 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 *midic14* and *ctrl14* min and max range:

$$ivalue = (initial\_value - min) / (max - min)$$

## See Also

*ctrl7*, *ctrl14*, *ctrl21*, *ctrlinit*, *initc7*, *initc21*, *midic7*, *midic14*, *midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.



# initc21

*initc21* -- Initializes the controllers used to create a 21-bit MIDI value.

*initc21*

## Description

Initializes MIDI controller *ictlno* with *ivalue*

## Syntax

**initc21** *ichan*, *ictlno1*, *ictlno2*, *ictlno3*, *ivalue*

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlno1* -- most significant byte controller number (0-127)

*ictlno2* -- medium significant byte controller number (0-127)

*ictlno3* -- least significant byte controller number (0-127)

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

## Performance

*initc21* can be used together with both *midic21* and *ctrl21* 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 *midic21* and *ctrl21* min and max range:

$$\text{ivalue} = (\text{initial\_value} - \text{min}) / (\text{max} - \text{min})$$

## See Also

*ctrl7*, *ctrl14*, *ctrl21*, *ctrlinit*, *initc7*, *initc14*, *midic7*, *midic14*, *midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# initc7

*initc7* -- Initializes the controller used to create a 7-bit MIDI value.

*initc7*

## Description

Initializes MIDI controller *ictlno* with *ivalue*

## Syntax

**initc7** *ichan*, *ictlno*, *ivalue*

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlno* -- controller number (0-127)

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

## Performance

*initc7* can be used together with both *midic7* and *ctrl7* 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 *midic7* and *ctrl7* min and max range:

$$\text{ivalue} = (\text{initial\_value} - \text{min}) / (\text{max} - \text{min})$$

## See Also

*ctrl7*, *ctrl14*, *ctrl21*, *ctrlinit*, *initc14*, *initc21*, *midic7*, *midic14*, *midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# ino

ino -- Reads eight-channel audio data from an external device or stream.

ino

## Description

Reads eight-channel audio data from an external device or stream.

## Syntax

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8 **ino**

## Performance

Reads eight-channel audio data from an external device or stream. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## See Also

*diskin, in, inh, inq, ins, soundin*

## Credits

Author: John ffitch

# inq

inq -- Reads quad audio data from an external device or stream.

inq

## Description

Reads quad audio data from an external device or stream.

## Syntax

ar1, ar2, ar3, a4 **inq**

## Performance

Reads quad audio data from an external device or stream. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## See Also

*diskin, in, inh, ino, ins, soundin*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# ins

ins -- Reads stereo audio data from an external device or stream.

ins

## Description

Reads stereo audio data from an external device or stream.

## Syntax

ar1, ar2 **ins**

## Performance

Reads stereo audio data from an external device or stream. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer. Any number of these opcodes can read freely from this buffer.

## See Also

*diskin, in, inh, ino, inq, soundin*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# instimek

instimek -- Deprecated.

instimek

## Description

Deprecated as of version 3.62. Use the *timeinstk* opcode instead.

## Credits

David M. Boothe originally pointed out this deprecated name.

# instimes

instimes -- Deprecated.

instimes

## Description

Deprecated as of version 3.62. Use the *timeinsts* opcode instead.

## Credits

David M. Boothe originally pointed out this deprecated name.

# instr

instr -- Starts an instrument block.

instr

## Description

Starts an instrument block.

## Syntax

```
instr i, j, ...
```

## Initialization

Starts 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.



### 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, with the exception of notes triggered by real time events that are initialized in the order of being received but still performed in ascending instrument number order). Instrument blocks cannot be nested (i.e. one block cannot contain another).

## Performance

### Calling an Instrument within an Instrument

You can call an instrument within an instrument as if it were an opcode either with the *subinstr* opcode or by specifying an instrument with a text name:

```
instr MyOscil  
...  
endin
```

If an instrument is defined with a name, you simply call it directly like an opcode:

```
asig MyOscil iamp, ipitch, iftable
```

By default, all output parameters correspond to the called instrument's output with the *signal output* opcodes. All input parameters are mapped to the called instrument's p-fields starting with the fourth one, p4. The values of the called instrument's second and third p-fields, p2 and p3, are automatically



set to those of the calling instrument's.

A named instrument must be defined before any instrument that calls it.



## Hint

If you use the *outc* opcode, you can create an instrument that will compile and function in any orchestra of any number of channels greater than or equal to the output channels of the instrument.

A nice feature to use with named instruments is the *#include* feature. You can then define your named instruments in separate files, using *#include* when you need to use one.

## Examples

Here is an example of the *instr* opcode. It uses the files *instr.orc* [examples/instr.orc] and *instr.sco* [examples/instr.sco].

### Example 190. Example of the *instr* opcode.

```
/* instr.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iamp = 10000
  icps = 440
  iphs = 0

  al oscils iamp, icps, iphs
  out al
endin
/* instr.orc */

/* instr.sco */
; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* instr.sco */
```

## See Also

*endin*, *in*, *out*, *opcode*, *endop*, *setksmps*, *xin*, *xout*, *subinstr*, *subinstrinit*

## Credits

Example written by Kevin Conder.

# int

int -- Extracts an integer from a decimal number.

int

## Description

Returns the integer part of  $x$ .

## Syntax

**int**( $x$ ) (init-rate or control-rate; also works at audio rate in Csound5)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the int opcode. It uses the files *int.orc* [examples/int.orc] and *int.sco* [examples/int.sco].

### Example 191. Example of the int opcode.

```
/* int.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = 16 / 5
  i2 = int(i1)

  print i2
endin
/* int.orc */

/* int.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* int.sco */
```

Its output should include a line like this:

```
instr 1:  i2 = 3.000
```

## See Also

*abs, exp, frac, log, log10, i, sqrt*

## Credits

Example written by Kevin Conder.

# integ

integ -- Modify a signal by integration.

integ

## Description

Modify a signal by integration.

## Syntax

```
ares integ asig [, iskip]
```

```
kres integ ksig [, iskip]
```

## Initialization

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

## Performance

*integ* and *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.

## Examples

Here is an example of the integ opcode. It uses the files *integ.orc* [examples/integ.orc] and *integ.sco* [examples/integ.sco].

### Example 192. Example of the integ opcode.

```
/* integ.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 -- a differentiated signal.
instr 1
  ; Generate a band-limited pulse train.
  asrc buzz 20000, 440, 20, 1

  ; Differentiate the signal.
  adiff diff asrc

  out adiff
endin

; Instrument #2 -- a re-integrated signal.
```

```
instr 2
; Generate a band-limited pulse train.
asrc buzz 20000, 440, 20, 1

; Differentiate the signal.
adiff diff asrc

; Re-integrate the previously differentiated signal.
al integ adiff

out al
endin
/* integ.orc */

/* integ.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 1 1
e
/* integ.sco */
```

## See Also

*diff, downsamp, interp, samphold, upsamp*

## Credits

Example written by Kevin Conder.

# interp

interp -- Converts a control signal to an audio signal using linear interpolation.

interp

## Description

Converts a control signal to an audio signal using linear interpolation.

## Syntax

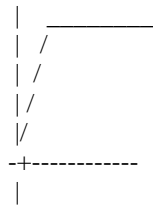
```
ares interp ksig [, iskip] [, imode]
```

## Initialization

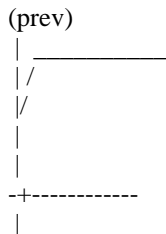
*iskip* (optional, default=0) -- initial disposition of internal save space (see *reson*). The default value is 0.

*imode* (optional, default=0) -- sets the initial output value to the first k-rate input instead of zero. The following graphs show the output of interp with a constant input value, in the original, when skipping init, and in the new mode:

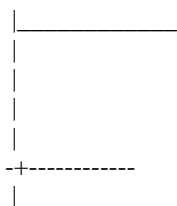
### Example 193. iskip=0, imode=0



### Example 194. iskip=1, imode=0



### Example 195. iskip=0, imode=1



## Performance

*ksig* -- input k-rate signal.

*interp* converts a control signal to an audio signal. It uses linear interpolation between successive kvals.

## Examples

Here is an example of the *interp* opcode. It uses the files *interp.orc* [examples/interp.orc] and *interp.sco* [examples/interp.sco].

### Example 196. Example of the *interp* opcode.

```
/* interp.orc */
; Initialize the global variables.
sr = 8000
kr = 8
ksmps = 1000
nchnls = 1

; Instrument #1 - a simple instrument.
instr 1
; Create an amplitude envelope.
kamp linseg 0, p3/2, 20000, p3/2, 0

; The amplitude envelope will sound rough because it
; jumps every ksmps period, 1000.
a1 oscil kamp, 440, 1
out a1
endin

; Instrument #2 - a smoother sounding instrument.
instr 2
; Create an amplitude envelope.
kamp linseg 0, p3/2, 25000, p3/2, 0
aamp interp kamp

; The amplitude envelope will sound smoother due to
; linear interpolation at the higher a-rate, 8000.
a1 oscil aamp, 440, 1
out a1
endin
/* interp.orc */

/* interp.sco */
```

```
; Table #1, a sine wave.
f 1 0 256 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 2 2
e
/* interp.sco */
```

## See Also

*diff, downsamp, integ, samphold, upsamp*

## Credits

Example written by Kevin Conder.

Updated November 2002, thanks to a note from both Rasmus Ekman and Istvan Varga.



# invalue

*invalue* -- Reads a k-rate signal from a user-defined channel.

*invalue*

## Description

Reads a k-rate signal or string from a user-defined channel.

## Syntax

*kvalue* **invalue** "channel name"

*Sname* **invalue** "channel name"

## Performance

*kvalue* -- The k-rate value that is read from the channel.

*Sname* -- The string variable that is read from the channel.

*"channel name"* -- An integer, string (in double-quotes), or string variable identifying the channel.

## See Also

*outvalue*

## Credits

Author: Matt Ingalls

# inx

*inx* -- Reads a 16-channel audio signal from an external device or stream.

*inx*

## Description

Reads a 16-channel audio signal from an external device or stream.

## Syntax

*ar1*, *ar2*, *ar3*, *ar4*, *ar5*, *ar6*, *ar7*, *ar8*, *ar9*, *ar10*, *ar11*, *ar12*, *ar13*, *ar14*, *ar15*

## Performance

*inx* reads a 16-channel audio signal from an external device or stream. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer.

## Credits

*in32*, *inch*, *inz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# inz

`inz --` Reads multi-channel audio samples into a ZAK array from an external device or stream.

`inz`

## Description

Reads multi-channel audio samples into a ZAK array from an external device or stream.

## Syntax

`inz` *ksigl*

## Performance

`inz` reads audio samples in *nchnls* into a ZAK array starting at *ksigl*. If the command-line *-i* flag is set, sound is read continuously from the audio input stream (e.g. *stdin* or a soundfile) into an internal buffer.

## Credits

*in32*, *inch*, *inx*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# **ioff**

ioff -- Deprecated.

ioff

## **Description**

Deprecated as of version 3.52. Use the *noteoff* opcode instead.

# ion

ion -- Deprecated.

ion

## Description

Deprecated as of version 3.52. Use the *noteon* opcode instead.

# iondur

iondur -- Deprecated.

iondur

## Description

Deprecated as of version 3.52. Use the *noteondur* opcode instead.

## iondur2

iondur2 -- Deprecated.

iondur2

### Description

Deprecated as of version 3.52. Use the *noteondur2* opcode instead.

# ioutat

ioutat -- Deprecated.

ioutat

## Description

Deprecated as of version 3.52. Use the *outiat* opcode instead.



# ioutc

ioutc -- Deprecated.

ioutc

## Description

Deprecated as of version 3.52. Use the *outic* opcode instead.

# ioutc14

ioutc14 -- Deprecated.

ioutc14

## Description

Deprecated as of version 3.52. Use the *outic14* opcode instead.

# ioutpat

ioutpat -- Deprecated.

ioutpat

## Description

Deprecated as of version 3.52. Use the *outipat* opcode instead.

# ioutpb

ioutpb -- Deprecated.

ioutpb

## Description

Deprecated as of version 3.52. Use the *outipb* opcode instead.

# ioutpc

ioutpc -- Deprecated.

ioutpc

## Description

Deprecated as of version 3.52. Use the *outipc* opcode instead.

## ipcauchy

ipcauchy -- Deprecated.

ipcauchy

## Description

Deprecated as of version 3.49. Use the *pcauchy* opcode instead.

# ipoisson

ipoisson -- Deprecated.

ipoisson

## Description

Deprecated as of version 3.49. Use the *poisson* opcode instead.

# ipow

ipow -- Deprecated.

ipow

## Description

Deprecated as of version 3.48. Use the *pow* opcode instead.



## is16b14

is16b14 -- Deprecated.

is16b14

### Description

Deprecated as of version 3.52. Use the *s16b14* opcode instead.

## is32b14

is32b14 -- Deprecated.

is32b14

### Description

Deprecated as of version 3.52. Use the *s32b14* opcode instead.

# islider16

islider16 -- Deprecated.

islider16

## Description

Deprecated as of version 3.52. Use the *slider16* opcode instead.

## islider32

islider32 -- Deprecated.

islider32

## Description

Deprecated as of version 3.52. Use the *slider32* opcode instead.

## islider64

islider64 -- Deprecated.

islider64

## Description

Deprecated as of version 3.52. Use the *slider64* opcode instead.

## islider8

islider8 -- Deprecated.

islider8

## Description

Deprecated as of version 3.52. Use the *slider8* opcode instead.

## itablecopy

itablecopy -- Deprecated.

itablecopy

### Description

Deprecated as of version 3.52. Use the *tablecopy* opcode instead.

# itablegpw

itablegpw -- Deprecated.

itablegpw

## Description

Deprecated as of version 3.52. Use the *tableigpw* opcode instead.



## itablemix

itablemix -- Deprecated.

itablemix

## Description

Deprecated as of version 3.52. Use the *tableimix* opcode instead.

# itablew

itablew -- Deprecated.

itablew

## Description

Deprecated as of version 3.52. Use the *tableiw* opcode instead.

# itrirand

itrirand -- Deprecated.

itrirand

## Description

Deprecated as of version 3.49. Use the *trirand* opcode instead.

# iunirand

iunirand -- Deprecated.

iunirand

## Description

Deprecated as of version 3.49. Use the *unirand* opcode instead.

# iweibull

iweibull -- Deprecated.

iweibull

## Description

Deprecated as of version 3.49. Use the *weibull* opcode instead.

# jitter

`jitter` -- Generates a segmented line whose segments are randomly generated.

`jitter`

## Description

Generates a segmented line whose segments are randomly generated.

## Syntax

kout **jitter** kamp, kcpsMin, kcpsMax

## Performance

*kamp* -- Amplitude of jitter deviation

*kcpsMin* -- Minimum speed of random frequency variations (expressed in cps)

*kcpsMax* -- Maximum speed of random frequency variations (expressed in cps)

*jitter* generates a segmented line whose segments are randomly generated inside the +kamp and -kamp interval. Duration of each segment is a random value generated according to kcpsmin and kcpsmax values.

*jitter* can be used to make more natural and “analog-sounding” some static, dull sound. For best results, it is suggested to keep its amplitude moderate.

## Examples

Here is an example of the jitter opcode. It uses the files *jitter.orc* [examples/jitter.orc] and *jitter.sco* [examples/jitter.sco].

### Example 197. Example of the jitter opcode.

```
/* jitter.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Instrument #1 -- plain instrument.
instr 1
  aplain vco 20000, 220, 2, 0.83

  outs aplain, aplain
endin

; Instrument #2 -- instrument with jitter.
instr 2
  ; Create a signal modulated the jitter opcode.
  kamp init 2
  kcpsmin init 4
  kcpsmax init 6
  kj jitter kamp, kcpsmin, kcpsmax
```

```
    aplain vco 20000, 220, 2, 0.83
    ajitter vco 20000, 220+kj, 2, 0.83

    outs aplain, ajitter
endin
/* jitter.orc */

/* jitter.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 3 seconds.
i 1 0 3
; Play Instrument #2 for 3 seconds.
i 2 3 3
e
/* jitter.sco */
```

## See Also

*jitter2, vibr, vibrato*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Version 4.15

# jitter2

`jitter2` -- Generates a segmented line with user-controllable random segments.

`jitter2`

## Description

Generates a segmented line with user-controllable random segments.

## Syntax

kout **jitter2** ktotamp, kamp1, kcps1, kamp2, kcps2, kamp3, kcps3

## Performance

*ktotamp* -- Resulting amplitude of jitter2

*kamp1* -- Amplitude of the first jitter component

*kcps1* -- Speed of random variation of the first jitter component (expressed in cps)

*kamp2* -- Amplitude of the second jitter component

*kcps2* -- Speed of random variation of the second jitter component (expressed in cps)

*kamp3* -- Amplitude of the third jitter component

*kcps3* -- Speed of random variation of the third jitter component (expressed in cps)

*jitter2* also generates a segmented line such as *jitter*, but in this case the result is similar to the sum of three *randi* opcodes, each one with a different amplitude and frequency value (see *randi* for more details), that can be varied at k-rate. Different effects can be obtained by varying the input arguments.

*jitter2* can be used to make more natural and “analog-sounding” some static, dull sound. For best results, it is suggested to keep its amplitude moderate.

## Examples

Here is an example of the jitter2 opcode. It uses the files *jitter2.orc* [examples/jitter2.orc] and *jitter2.sco* [examples/jitter2.sco].

### Example 198. Example of the jitter2 opcode.

```
/* jitter2.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Instrument #1 -- plain instrument.
instr 1
  aplain vco 20000, 220, 2, 0.83
```



```
    outs aplain, aplain
endin

; Instrument #2 -- instrument with jitter.
instr 2
    ; Create a signal modulated with the jitter2 opcode.
    ktotamp init 2
    kamp1 init 0.66
    kcps1 init 3
    kamp2 init 0.66
    kcps2 init 3
    kamp3 init 0.66
    kcps3 init 3
    kj jitter2 ktotamp, kamp1, kcps1, kamp2, kcps2, \
        kamp3, kcps3

    aplain vco 20000, 220, 2, 0.83
    ajitter vco 20000, 220+kj, 2, 0.83

    outs aplain, ajitter
endin
/* jitter2.orc */

/* jitter2.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 3 seconds.
i 1 0 3
; Play Instrument #2 for 3 seconds.
i 2 3 3
e
/* jitter2.sco */
```

## See Also

*jitter, vibr, vibrato*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Version 4.15

# jspline

jspline -- A jitter-spline generator.

jspline

## Description

A jitter-spline generator.

## Syntax

ares **jspline** xamp, kcpsMin, kcpsMax

kres **jspline** kamp, kcpsMin, kcpsMax

## Performance

*kres, ares* -- Output signal

*xamp* -- Amplitude factor

*kcpsMin, kcpsMax* -- Range of point-generation rate. Min and max limits are expressed in cps.

*jspline* (jitter-spline generator) generates a smooth curve based on random points generated at [cpsMin, cpsMax] rate. This opcode is similar to *randomi* or *randi* or *jitter*, but segments are not straight lines, but cubic spline curves. Output value range is approximately  $> -xamp$  and  $< xamp$ . Actually, real range could be a bit greater, because of interpolating curves between each pair of random-points.

At present time generated curves are quite smooth when cpsMin is not too different from cpsMax. When cpsMin-cpsMax interval is big, some little discontinuity could occur, but it should not be a problem, in most cases. Maybe the algorithm will be improved in next versions.

These opcodes are often better than *jitter* when user wants to “naturalize” or “analogize” digital sounds. They could be used also in algorithmic composition, to generate smooth random melodic lines when used together with *samphold* opcode.

Note that the result is quite different from the one obtained by filtering white noise, and they allow the user to obtain a much more precise control.

## Credits

Author: Gabriel Maldonado

New in Version 4.15

# k

k -- Converts a i-rate parameter to an k-rate value.

k

## Description

Converts an i-rate value to control rate, for example to be used with `rnd()` and `birnd()` to generate random numbers at k-rate.

## Syntax

**k**(x) (i-rate args only)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## See Also

*i a*

## Credits

Author: Istvan Varga

New in version Csound 5.00

# kbetarand

kbetarand -- Deprecated.

kbetarand

## Description

Deprecated as of version 3.49. Use the *betarand* opcode instead.

# kbexprnd

kbexprnd -- Deprecated.

kbexprnd

## Description

Deprecated as of version 3.49. Use the *bexprnd* opcode instead.

# kcauchy

kcauchy -- Deprecated.

kcauchy

## Description

Deprecated as of version 3.49. Use the *cauchy* opcode instead.

# kdump

kdump -- Deprecated.

kdump

## Description

Deprecated as of version 3.49. Use the *dumpk* opcode instead.

## kdump2

kdump2 -- Deprecated.

kdump2

## Description

Deprecated as of version 3.49. Use the *dumpk2* opcode instead.



## **kdump3**

kdump3 -- Deprecated.

kdump3

### **Description**

Deprecated as of version 3.49. Use the *dumpk3* opcode instead.

## kdump4

kdump4 -- Deprecated.

kdump4

## Description

Deprecated as of version 3.49. Use the *dumpk4* opcode instead.

# kexprand

kexprand -- Deprecated.

kexprand

## Description

Deprecated as of version 3.49. Use the *exprand* opcode instead.

## kfilter2

kfilter2 -- Deprecated.

kfilter2

## Description

Deprecated as of version 3.49. Use the *filter2* opcode instead.

# kgauss

kgauss -- Deprecated.

kgauss

## Description

Deprecated as of version 3.49. Use the *gauss* opcode instead.

# kgoto

kgoto -- Transfer control during the p-time passes.

kgoto

## Description

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

## Syntax

**kgoto** label

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

## Examples

Here is an example of the kgoto opcode. It uses the files *kgoto.orc* [examples/kgoto.orc] and *kgoto.sco* [examples/kgoto.sco].

### Example 199. Example of the kgoto opcode.

```
/* kgoto.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Change kval linearly from 0 to 2 over
; the period set by the third p-field.
kval line 0, p3, 2

; If kval is greater than or equal to 1 then play the high note.
; If not then play the low note.
if (kval >= 1) kgoto highnote
kgoto lownote

highnote:
kfreq = 880
goto playit

lownote:
kfreq = 440
goto playit

playit:
; Print the values of kval and kfreq.
printks "kval = %f, kfreq = %f\\n", 1, kval, kfreq

a1 oscil 10000, kfreq, 1
out a1
endin
/* kgoto.orc */
```

```
/* kgoto.sco */  
; Table #1: a simple sine wave.  
f 1 0 32768 10 1  
  
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* kgoto.sco */
```

Its output should include lines like this:

```
kval = 0.000000, kfreq = 440.000000  
kval = 0.999732, kfreq = 440.000000  
kval = 1.999639, kfreq = 880.000000
```

## See Also

*cggoto, cigoto, ckgoto, goto, if, igoto, tigoto, timeout*

## Credits

Example written by Kevin Conder.

Added a note by Jim Aikin.

# klinrand

klinrand -- Deprecated.

klinrand

## Description

Deprecated as of version 3.49. Use the *linrand* opcode instead.



# kon

kon -- Deprecated.

kon

## Description

Deprecated as of version 3.49. Use the *midion* opcode instead.

# koutat

koutat -- Deprecated.

koutat

## Description

Deprecated as of version 3.52. Use the *outkat* opcode instead.

# koutc

koutc -- Deprecated.

koutc

## Description

Deprecated as of version 3.52. Use the *outkc* opcode instead.

# koutc14

koutc14 -- Deprecated.

koutc14

## Description

Deprecated as of version 3.52. Use the *outkc14* opcode instead.

# koutpat

koutpat -- Deprecated.

koutpat

## Description

Deprecated as of version 3.52. Use the *outkpat* opcode instead.

# koutpb

koutpb -- Deprecated.

koutpb

## Description

Deprecated as of version 3.52. Use the *outkpb* opcode instead.

# koutpc

koutpc -- Deprecated.

koutpc

## Description

Deprecated as of version 3.52. Use the *outkpc* opcode instead.

# kpcauchy

kpcauchy -- Deprecated.

kpcauchy

## Description

Deprecated as of version 3.49. Use the *pcauchy* opcode instead.



# kpoisson

kpoisson -- Deprecated.

kpoisson

## Description

Deprecated as of version 3.49. Use the *poisson* opcode instead.

# **kpow**

kpow -- Deprecated.

kpow

## **Description**

Deprecated as of version 3.48. Use the *pow* opcode instead.

# kr

kr -- Sets the control rate.

kr

## 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.

## Syntax

```
kr = iarg
```

## Initialization

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

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, *kr* can be omitted. Csound will attempt to calculate the omitted value from the specified values, but it should evaluate to an integer.

## Examples

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

## See Also

*ksmps, nchnls, sr*

# kread

kread -- Deprecated.

kread

## Description

Deprecated as of version 3.52. Use the *readk* opcode instead.

# kread2

kread2 -- Deprecated.

kread2

## Description

Deprecated as of version 3.52. Use the *readk2* opcode instead.

# kread3

kread3 -- Deprecated.

kread3

## Description

Deprecated as of version 3.52. Use the *readk3* opcode instead.

# kread4

kread4 -- Deprecated.

kread4

## Description

Deprecated as of version 3.52. Use the *readk4* opcode instead.

# ksmps

ksmps -- Sets the number of samples in a control period.

ksmps

## 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.

## Syntax

**ksmps** = iarg

## Initialization

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

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 *ksmps* may be omitted. Csound will attempt to calculate the omitted value from the specified values, but it should evaluate to an integer.



### Warning

ksmps must be an integer value.

## Examples

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

## See Also

*kr*, *nchnls*, *sr*

## Credits

Thanks to a note from Gabriel Maldonado, added a warning about integer values.



## ktableseg

ktableseg -- Same as the tableseg opcode.

ktableseg

## Description

Same as the *tableseg* opcode.

## Syntax

**ktableseg** ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]

# ktrirand

ktrirand -- Deprecated.

ktrirand

## Description

Deprecated as of version 3.49. Use the *trirand* opcode instead.

# kunirand

kunirand -- Deprecated.

kunirand

## Description

Deprecated as of version 3.49. Use the *unirand* opcode instead.

# kweibull

kweibull -- Deprecated.

kweibull

## Description

Deprecated as of version 3.49. Use the *weibull* opcode instead.

# lfo

lfo -- A low frequency oscillator of various shapes.

lfo

## Description

A low frequency oscillator of various shapes.

## Syntax

```
kres lfo kamp, kcps [, itype]
```

```
ares lfo kamp, kcps [, itype]
```

## Initialization

*itype* (optional, default=0) -- determine the waveform of the oscillator. Default is 0.

- *itype* = 0 - sine
- *itype* = 1 - triangles
- *itype* = 2 - square (bipolar)
- *itype* = 3 - square (unipolar)
- *itype* = 4 - saw-tooth
- *itype* = 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

## Examples

Here is an example of the lfo opcode. It uses the files *lfo.orc* [examples/lfo.orc] and *lfo.sco* [examples/lfo.sco].

### Example 200. Example of the lfo opcode.

```
/* lfo.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1
```

```
; Instrument #1.
instr 1
  kamp = 10
  kcps = 5
  itype = 4

  k1 lfo kamp, kcps, itype
  ar oscil p4, p5+k1, 1
  out ar
endin
/* lfo.orc */

/* lfo.sco */
; Table #1: an ordinary sine wave.
f 1 0 32768 10 1

; p4 = amplitude of the output signal.
; p5 = frequency (in cycles per second) of the output signal.
; Play Instrument #1 for two seconds.
i 1 0 2 10000 220
e
/* lfo.sco */
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
November 1998

New in Csound version 3.491

# limit

`limit` -- Sets the lower and upper limits of the value it processes.

`limit`

## Description

Sets the lower and upper limits of the value it processes.

## Syntax

`ares limit asig, klow, khigh`

`ires limit isig, ilow, ihigh`

`kres limit ksig, klow, khigh`

## Initialization

*isig* -- input signal

*ilow* -- low threshold

*ihigh* -- high threshold

## Performance

*xsig* -- input signal

*klow* -- low threshold

*khigh* -- high threshold

*limit* sets the lower and upper limits on the *xsig* value it processes. If *xhigh* is lower than *xlow*, then the output will be the average of the two - it will not be affected by *xsig*.

This opcode is useful in several situations, such as table indexing or for clipping and modeling a-rate, i-rate or k-rate signals.

## See Also

*mirror*, *wrap*

## Credits

Author: Robin Whittle  
Australia

New in Csound version 3.46

# line

line -- Trace a straight line between specified points.

line

## Description

Trace a straight line between specified points.

## Syntax

ares **line** ia, idur1, ib

kres **line** ia, idur1, ib

## 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.

## 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.

## Examples

Here is an example of the line opcode. It uses the files *line.orc* [examples/line.orc] and *line.sco* [examples/line.sco].

### Example 201. Example of the line opcode.

```
/* line.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Define kcps as a frequency value that linearly declines
  ; from 880 to 220. It declines over the period set by p3.
  kcps line 880, p3, 220

  a1 oscil 20000, kcps, 1
  out a1
```



```
endin
/* line.orc */

/* line.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* line.sco */
```

## See Also

*expon, expseg, expsegr, linseg, linsegr*

## Credits

Example written by Kevin Conder.

# linen

*linen* -- Applies a straight line rise and decay pattern to an input amp signal.

*linen*

## Description

*linen* -- apply a straight line rise and decay pattern to an input amp signal.

## Syntax

*ares linen* xamp, irise, idur, idec

*kres linen* kamp, irise, idur, idec

## 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.

## Performance

*kamp*, *xamp* -- input amplitude signal.

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. If *linen* rise and decay periods overlap then both modifications will be in effect for that time. If the overall duration *idur* is exceeded in performance, the final decay will continue on in the same direction, going negative.

## See Also

*envlpx*, *envlpxr*, *linenr*

# linenr

linenr -- The linen opcode extended with a final release segment.

linenr

## Description

*linenr* -- same as *linen* except that the final segment is entered only on sensing a MIDI note release. The note is then extended by the decay time.

## Syntax

```
ares linenr xamp, irise, idec, iatdec
```

```
kres linenr kamp, irise, idec, iatdec
```

## 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.

*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.

## Performance

*kamp*, *xamp* -- input amplitude signal.

*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* is an example of the special Csound “r” units that contain a note-off sensor and release time extender. 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 unless made independent by *irind*. Then it will begin a decay 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*.

## See Also

*envlpx*, *envlpxr*, *linen*

# lineto

lineto -- Generate glissandos starting from a control signal.

lineto

## Description

Generate glissandos starting from a control signal.

## Syntax

```
kres lineto ksig, ktime
```

## Performance

*kres* -- Output signal.

*ksig* -- Input signal.

*ktime* -- Time length of glissando in seconds.

*lineto* adds glissando (i.e. straight lines) to a stepped input signal (for example, produced by *randh* or *lpshold*). It generates a straight line starting from previous step value, reaching the new step value in *ktime* seconds. When the new step value is reached, such value is held until a new step occurs. Be sure that *ktime* argument value is smaller than the time elapsed between two consecutive steps of the original signal, otherwise discontinuities will occur in output signal.

When used together with the output of *lpshold* it emulates the glissando effect of old analog sequencers.

## See Also

*tlinto*

## Credits

Author: Gabriel Maldonado

New in Version 4.13

# linrand

linrand -- Linear distribution random number generator (positive values only).

linrand

## Description

Linear distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

ares **linrand** krange

ires **linrand** krange

kres **linrand** krange

## Performance

*krange* -- the range of the random numbers (0 - *krange*). Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the linrand opcode. It uses the files *linrand.orc* [examples/linrand.orc] and *linrand.sco* [examples/linrand.sco].

### Example 202. Example of the linrand opcode.

```
/* linrand.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number between 0 and 1.
  ; krange = 1

  il linrand 1

  print il
endin
```

```
/* linrand.orc */  
  
/* linrand.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* linrand.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 0.394
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, pcauchy, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# linseg

linseg -- Trace a series of line segments between specified points.

linseg

## Description

Trace a series of line segments between specified points.

## Syntax

```
ares linseg ia, idur1, ib [, idur2] [, ic] [...]
```

```
kres linseg ia, idur1, ib [, idur2] [, ic] [...]
```

## Initialization

*ia* -- starting value. Zero is illegal for exponentials.

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

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

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

## Performance

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

## Examples

Here is an example of the *linseg* opcode. It uses the files *linseg.orc* [examples/linseg.orc] and *linseg.sco* [examples/linseg.sco].

### Example 203. Example of the *linseg* opcode.

```
/* linseg.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; p4 = frequency in pitch-class notation.
  kcps = cpspch(p4)
```

```
; Create an amplitude envelope.
kenv linseg 0, p3*0.25, 1, p3*0.75, 0
kamp = kenv * 30000

a1 oscil kamp, kcps, 1
out a1
endin
/* linseg.orc */

/* linseg.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for a half-second, p4=8.00
i 1 0 0.5 8.00
; Play Instrument #1 for a half-second, p4=8.01
i 1 1 0.5 8.01
; Play Instrument #1 for a half-second, p4=8.02
i 1 2 0.5 8.02
; Play Instrument #1 for a half-second, p4=8.03
i 1 3 0.5 8.03
e
/* linseg.sco */
```

## See Also

*expon, expseg, expsegr, line, linsegr transeg*

## Credits

Example written by Kevin Conder.



# linsegr

linsegr -- Trace a series of line segments between specified points including a release segment.

linsegr

## Description

Trace a series of line segments between specified points including a release segment.

## Syntax

ares **linsegr** *ia*, *idur1*, *ib* [, *idur2*] [, *ic*] [...], *irel*, *iz*

kres **linsegr** *ia*, *idur1*, *ib* [, *idur2*] [, *ic*] [...], *irel*, *iz*

## 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.

Please note that the release time cannot be longer than 32767/*kr* seconds.

## 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* is 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. For two or more extenders in an instrument, extension is by the greatest period.

## Examples

Here is an example of the *linsegr* opcode. It uses the files *linsegr.orc* [examples/*linsegr.orc*] and *linsegr.sco* [examples/*linsegr.sco*].

### Example 204. Example of the *linsegr* opcode.

```
/* linsegr.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; p4 = frequency in pitch-class notation.
  kcps = cpspch(p4)

  ; Use an amplitude envelope with second-long release.
  kenv linsegr 1, p3, 0.25, 1, 0
  kamp = kenv * 30000

  al oscil kamp, kcps, 1
  out al
endin
/* linsegr.orc */
```

```
/* linsegr.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Make sure the score lasts for four seconds.
f 0 4

; p4 = frequency (in pitch-class notation).
; Play Instrument #1 for a half-second, p4=8.00
i 1 0 0.5 8.00
; Play Instrument #1 for a half-second, p4=8.01
i 1 1 0.5 8.01
; Play Instrument #1 for a half-second, p4=8.02
i 1 2 0.5 8.02
; Play Instrument #1 for a half-second, p4=8.03
i 1 3 0.5 8.03
e
/* linsegr.sco */
```

## See Also

*expon, expseg, expsegr, line, linseg*

## Credits

Author: Barry L. Vercoe

Example written by Kevin Conder.

December 2002. Thanks to Istvan Varga, added documentation about the maximum release time.

New in Csound 3.47

# locsend

locsend -- Distributes the audio signals of a previous *locsig* opcode.

locsend

## Description

*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!).

## Syntax

a1, a2 **locsend**

a1, a2, a3, a4 **locsend**

## Examples

```
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, a
endin
instr 99
  ; reverb....
endin
```

A few notes:

```
;place the sound in the left speaker and near:
il 0 1 0 1

;place the sound in the right speaker and far:
il 1 1 90 25

;place the sound equally between left and right and in the middle ground dist:
il 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
```

## See Also

*locsig*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# locsig

locsig -- Takes an input signal and distributes between 2 or 4 channels.

locsig

## 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.

## Syntax

a1, a2 **locsig** asig, kdegree, kdistance, kreverbsend

a1, a2, a3, a4 **locsig** asig, kdegree, kdistance, kreverbsend

## 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 (ie.: 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.

## Examples

```
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 placement 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, a
endin
instr 99
  ; reverb....
endin
```

A few notes:

```
;place the sound in the left speaker and near:
il 0 1 0 1

;place the sound in the right speaker and far:
il 1 1 90 25

;place the sound equally between left and right and in the middle ground dist.
il 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
```

## See Also

*locsend*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# log

log -- Returns a natural log.

log

## Description

Returns the natural log of  $x$  ( $x$  positive only).

The argument value is restricted for *log*, *log10*, and *sqrt*.

## Syntax

**log**( $x$ ) (no rate restriction)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the log opcode. It uses the files *log.orc* [examples/log.orc] and *log.sco* [examples/log.sco].

### Example 205. Example of the log opcode.

```
/* log.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = log(8)
  print i1
endin
/* log.orc */

/* log.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* log.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 2.079
```



## See Also

*abs, exp, frac, int, log10, i, sqrt*

## Credits

Example written by Kevin Conder.

# log10

log10 -- Returns a base 10 log.

log10

## Description

Returns the base 10 log of  $x$  ( $x$  positive only).

The argument value is restricted for *log*, *log10*, and *sqrt*.

## Syntax

**log10**(*x*) (no rate restriction)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the log10 opcode. It uses the files *log10.orc* [examples/log10.orc] and *log10.sco* [examples/log10.sco].

### Example 206. Example of the log10 opcode.

```
/* log10.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = log10(8)
  print i1
endin
/* log10.orc */

/* log10.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* log10.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 0.903
```

## See Also

*abs, exp, frac, int, log, i, sqrt*

## Credits

Example written by Kevin Conder.

# logbtwo

logbtwo -- Performs a logarithmic base two calculation.

logbtwo

## Description

Performs a logarithmic base two calculation.

## Syntax

**logbtwo**(x) (init-rate or control-rate args only)

## Performance

*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.

## Examples

Here is an example of the logbtwo opcode. It uses the files *logbtwo.orc* [examples/logbtwo.orc] and *logbtwo.sco* [examples/logbtwo.sco].

### Example 207. Example of the logbtwo opcode.

```
/* logbtwo.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il = logbtwo(3)
  print il
endin
/* logbtwo.orc */

/* logbtwo.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* logbtwo.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 1.585
```

## See Also

*powoftwo*

## Credits

Author: Gabriel Maldonado  
Italy  
June 1998

Author: John ffitch  
University of Bath, Codemist, Ltd.  
Bath, UK  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# loop

loop -- Four looping constructions.

loop

## Description

Construction of looping operations.

## Syntax

**loop\_lt**    *indx*, *incr*, *imax*, *label*

**loop\_lt**    *kndx*, *kncr*, *kmax*, *label*

**loop\_le**    *indx*, *incr*, *imax*, *label*

**loop\_le**    *kndx*, *kncr*, *kmax*, *label*

**loop\_gt**    *indx*, *idecr*, *imin*, *label*

**loop\_gt**    *kndx*, *kdecr*, *kmin*, *label*

**loop\_ge**    *indx*, *idecr*, *imin*, *label*

**loop\_ge**    *kndx*, *kdecr*, *kmin*, *label*

## Initialization

*indx* -- i-rate variable to count loop.

*incr* -- value to increment the loop (loop\_lt, loop\_le)

*idecr* -- value to decrement the loop (loop\_gt, loop\_ge)

*imax* -- maximum value of loop index (loop\_lt, loop\_le)

*imin* -- minimum value of loop index (loop\_gt, loop\_ge)

## Performance

The actions of **loop\_lt** is equivalent to the code

```
indx = indx + incr
if (indx < imax) igoto label
```

or

```
kndx = kndx + kncr
```

```
if (kndx < kmax) kgoto label
```

The actions of **loop\_le** is equivalent to the code

```
indx = indx + incr  
if (indx <= imax) igoto label
```

or

```
kndx = kndx + kncr  
if (kndx <= kmax) kgoto label
```

The actions of **loop\_gt** is equivalent to the code

```
indx = indx - idecr  
if (indx > imin) igoto label
```

or

```
kndx = kndx - kdecr  
if (kndx > kmin) kgoto label
```

The actions of **loop\_ge** is equivalent to the code

```
indx = indx - idecr  
if (indx >= imin) igoto label
```

or

```
kndx = kndx - kdecr  
if (kndx >= kmin) kgoto label
```

## Credits

Istvan Varga.

# loopseg

`loopseg` -- Generate control signal consisting of linear segments delimited by two or more specified points.

`loopseg`

## Description

Generate control signal consisting of linear segments delimited by two or more specified points. The entire envelope is looped at `kfreq` rate. Each parameter can be varied at `k-rate`.

## Syntax

```
ksig loopseg kfreq, ktrig, ktime0, kvalue0 [, ktime1] [, kvalue1] [, ktime2] [,
```

## Performance

`ksig` -- Output signal

`kfreq` -- Repeat rate in Hz or fraction of Hz

`ktrig` -- If non-zero, retriggers the envelope from start (see *trigger opcode*), before the envelope cycle is completed.

`ktime0...ktimeN` -- Times of points; expressed in fraction of a cycle.

`kvalue0...kvalueN` -- Values of points

`loopseg` opcode is similar to `linseg`, but the entire envelope is looped at `kfreq` rate. Notice that times are not expressed in seconds but in fraction of a cycle. Actually each duration represent is proportional to the other, and the entire cycle duration is proportional to the sum of all duration values.

The sum of all duration is then rescaled according to `kfreq` argument. For example, considering an envelope made up of 3 segments, each segment having 100 as duration value, their sum will be 300. This value represents the total duration of the envelope, and is actually divided into 3 equal parts, a part for each segment.

Actually, the real envelope duration in seconds is determined by `kfreq`. Again, if the envelope is made up of 3 segments, but this time the first and last segments have a duration of 50, whereas the central segment has a duration of 100 again, their sum will be 200. This time 200 represent the total duration of the 3 segments, so the central segment will be twice as long as the other segments.

All parameters can be varied at `k-rate`. Negative frequency values are allowed, reading the envelope backward. `ktime0` should always be set to 0, except if the user wants some special effect.

## Examples

Here is an example of the `loopseg` opcode. It uses the files `loopseg.orc` [examples/loopseg.orc] and `loopseg.sco` [examples/loopseg.sco].

### Example 208. Example of the loopseg opcode.

```
/* loopseg.orc */
; Initialize the global variables.
sr = 44100
```



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

; Instrument #1
instr 1
  kfreq line 1, p3, 20

  klp loopseg kfreq, 0, 0, 0, 0.5, 30000, 1, 0

  al oscil klp, 440, 1
  out al
endin
/* loopseg.orc */

/* loopseg.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for five seconds.
i 1 0 5
e
/* loopseg.sco */
```

## See Also

*lpshold*

## Credits

Author: Gabriel Maldonado

New in Version 4.13

# loopsegp

loopsegp -- Control signals based on linear segments.

loopsegp

## Description

Generate control signal consisting of linear segments delimited by two or more specified points. The entire envelope can be looped at time-variant rate. Each segment coordinate can also be varied at k-rate.

## Syntax

`ksig loopsegp kphase, kvalue0, ktime0, kvalue1, ktime1 [, ... , kvalueN, ktimeN]`

## Initialization

*initphase* - initial phase value (in the 0 to 1 range)

## Performance

*ksig* - output signal

*kphase* - NO INFORMATION

*kvalue0* ...*kvalueN* - values of points

*ktime0* ...*ktimeN* - times of points expressed in fraction of a cycle

*loopsegp* opcode is similar to *loopseg*; the only difference is that, instead of frequency, a time-variant phase is required. If you use a phasor to get the phase value, you will have a behaviour identical to *loopseg*, but interesting results can be achieved when using phases having non-linear motions, making *loopsegp* more powerful and general than *loopseg*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# lorenz

lorenz -- Implements the Lorenz system of equations.

lorenz

## 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.

## Syntax

```
ax, ay, az lorenz ksv, krv, kbv, kh,  
            ix, iy, iz, iskip [, iskipinit]
```

## 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.

*iskipinit* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## 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
```

## Examples

Here is an example of the lorenz opcode. It uses the files *lorenz.orc* [examples/lorenz.orc] and *lorenz.sco* [examples/lorenz.sco].

### Example 209. Example of the lorenz opcode.

```
/* lorenz.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 2

; Instrument #1 - a lorenz system in 3D space.
instr 1
  ; Create a basic tone.
  kamp init 25000
  kcps init 220
  ifn = 1
  asnd oscil kamp, kcps, ifn

  ; Figure out its X, Y, Z coordinates.
  ksv init 10
  krv init 28
  kbv init 2.667
  kh init 0.0003
  ix = 0.6
  iy = 0.6
  iz = 0.6
  iskip = 1
  ax1, ay1, az1 lorenz ksv, krv, kbv, kh, ix, iy, iz, iskip

  ; Place the basic tone within 3D space.
  kx downsamp ax1
  ky downsamp ay1
  kz downsamp az1
  idist = 1
  ift = 0
  imode = 1
  imdel = 1.018853416
  iovr = 2
  aw2, ax2, ay2, az2 spat3d asnd, kx, ky, kz, idist, \
    ift, imode, imdel, iovr

  ; Convert the 3D sound to stereo.
  aleft = aw2 + ay2
  aright = aw2 - ay2

  outs aleft, aright
endin
/* lorenz.orc */

/* lorenz.sco */
; Table #1 a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 5 seconds.
i 1 0 5
e
/* lorenz.sco */
```

## Credits

Author: Hans Mikelson  
February 1999

New in Csound version 3.53

# lorisread

`lorisread` -- Imports a set of bandwidth-enhanced partials from a SDIF-format data file, applying control-rate frequency, amplitude, and bandwidth scaling envelopes, and stores the modified partials in memory.

`lorisread`

## Syntax

```
lorisread ktmpnt, ifilcod, istoreidx, kfreqenv, kampenv, kbwenv[, ifadetime]
```

## Description

`lorisread` imports a set of bandwidth-enhanced partials from a SDIF-format data file, applying control-rate frequency, amplitude, and bandwidth scaling envelopes, and stores the modified partials in memory.

## Initialization

*ifilcod* - integer or character-string denoting a control-file derived from reassigned bandwidth-enhanced analysis of an audio signal. An integer denotes the suffix of a file `loris.sdif` (e.g. `loris.sdif.1`); a character-string (in double quotes) gives a filename, optionally a full pathname. If not a full pathname, the file is sought first in the current directory, then in the one given by the environment variable `SADIR` (if defined). The reassigned bandwidth-enhanced data file contains breakpoint frequency, amplitude, noisiness, and phase envelope values organized for bandwidth-enhanced additive resynthesis. The control data must conform to one of the SDIF formats that can be

Loris stores partials in SDIF RBEP frames. Each RBEP frame contains one RBEP matrix, and each row in a RBEP matrix describes one breakpoint in a Loris partial. A RBEL frame containing one RBEL matrix describing the labeling of the partials may precede the first RBEP frame in the SDIF file. The RBEP and RBEL frame and matrix definitions are included in the SDIF file's header. In addition to RBEP frames, Loris can also read and write SDIF ITRC frames. Since ITRC frames do not represent bandwidth-enhancement or the exact timing of Loris breakpoints, their use is not recommended. ITRC capabilities are provided to allow interchange with programs that are unable to handle RBEP frames.

*istoreidx*, *ireadidx*, *isrcidx*, *itgtidx* are labels that identify a stored set of bandwidth-enhanced partials. `lorisread` imports partials from a SDIF file and stores them with the integer label *istoreidx*. `lorismorph` morphs sets of partials labeled *isrcidx* and *itgtidx*, and stores the resulting partials with the integer label *istoreidx*. `lorisplay` renders the partials stored with the label *ireadidx*. The labels are used only at initialization time, and may be reused without any cost or benefit in efficiency, and without introducing any interaction between instruments or instances.

*ifadetime* (*optional*) - In general, partials exported from Loris begin and end at non-zero amplitude. In order to prevent artifacts, it is very often necessary to fade the partials in and out, instead of turning them abruptly on and off. Specification of a non-zero *ifadetime* causes partials to fade in at their onsets and to fade out at their terminations. This is achieved by adding two more breakpoints to each partial: one *ifadetime* seconds before the start time and another *ifadetime* seconds after the end time. (However, no breakpoint will be introduced at a time less than zero. If necessary, the onset fade time will be shortened.) The additional breakpoints at the partial onset and termination will have the same frequency and bandwidth as the first and last breakpoints in the partial, respectively, but their amplitudes will be zero. The phase of the new breakpoints will be extrapolated to preserve phase correctness. If no value is specified, *ifadetime* defaults to zero. Note that the *fadetime* may not be exact, since the partial parameter envelopes are sampled at the control rate (*krate*) and indexed by *ktmpnt* (see below), and not by real time.

## Performance

`lorisread` reads pre-computed Reassigned Bandwidth-Enhanced analysis data from a file stored in SDIF format, as described above. 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. `kfreqenv` 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. `kampenv` is a control-rate scale factor that is applied to all partial amplitude envelopes. `kbwenv` is a control-rate scale factor that is applied to all partial bandwidth or noisiness envelopes. The bandwidth-enhanced partial data is stored in memory with a specified label for future access by another generator.

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz ([loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org) [<mailto:loris@cerlsoundgroup.org>]). It is patterned after a prototype implementation of the *lorisplay* unit generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael Gogins.

# lorismorph

**lorismorph** -- Morphs two stored sets of bandwidth-enhanced partials and stores a new set of partials representing the morphed sound. The morph is performed by linearly interpolating the parameter envelopes (frequency, amplitude, and bandwidth, or noisiness) of the bandwidth-enhanced partials according to control-rate frequency, amplitude, and bandwidth morphing functions.

lorismorph

## Syntax

**lorismorph** isrcidx, itgtidx, istoreidx, kfreqmorphenv, kampmorphenv, kbwmorphenv

## Description

*lorismorph* morphs two stored sets of bandwidth-enhanced partials and stores a new set of partials representing the morphed sound. The morph is performed by linearly interpolating the parameter envelopes (frequency, amplitude, and bandwidth, or noisiness) of the bandwidth-enhanced partials according to control-rate frequency, amplitude, and bandwidth morphing functions.

## Initialization

*istoreidx*, *ireadidx*, *isrcidx*, *itgtidx* are labels that identify a stored set of bandwidth-enhanced partials. *lorisread* imports partials from a SDIF file and stores them with the integer label *istoreidx*. *lorismorph* morphs sets of partials labeled *isrcidx* and *itgtidx*, and stores the resulting partials with the integer label *istoreidx*. *lorisplay* renders the partials stored with the label *ireadidx*. The labels are used only at initialization time, and may be reused without any cost or benefit in efficiency, and without introducing any interaction between instruments or instances.

## Performance

*lorismorph* generates a set of bandwidth-enhanced partials by morphing two stored sets of partials, the source and target partials, which may have been imported using *lorisread*, or generated by another unit generator, including another instance of *lorismorph*. The morph is performed by interpolating the parameters of corresponding (labeled) partials in the two source sounds. The sound morph is described by three control-rate morphing envelopes. *kfreqmorphenv* describes the interpolation of partial frequency values in the two source sounds. When *kfreqmorphenv* is 0, partial frequencies are obtained from the partials stored at *isrcidx*. When *kfreqmorphenv* is 1, partial frequencies are obtained from the partials at *itgtidx*. When *kfreqmorphenv* is between 0 and 1, the partial frequencies are interpolated between corresponding source and target partials. Interpolation of partial amplitudes and bandwidth (noisiness) coefficients are similarly described by *kampmorphenv* and *kbwmorphenv*.

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz ([loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org) [<mailto:loris@cerlsoundgroup.org>]). It is patterned after a prototype implementation of the *lorisplay* unit generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael gogins.



# lorisplay

*lorisplay* -- renders a stored set of bandwidth-enhanced partials using the method of Bandwidth-Enhanced Additive Synthesis implemented in the Loris software, applying control-rate frequency, amplitude, and bandwidth scaling envelopes.

*lorisplay*

## Syntax

```
ar lorisplay ireadidx, kfreqenv, kampenv, kbwenv
```

## Description

*lorisplay* renders a stored set of bandwidth-enhanced partials using the method of Bandwidth-Enhanced Additive Synthesis implemented in the Loris software, applying control-rate frequency, amplitude, and bandwidth scaling envelopes.

## Initialization

*istoreidx*, *ireadidx*, *isrcidx*, *itgtidx* are labels that identify a stored set of bandwidth-enhanced partials. *lorisread* imports partials from a SDIF file and stores them with the integer label *istoreidx*. *lorismorph* morphs sets of partials labeled *isrcidx* and *itgtidx*, and stores the resulting partials with the integer label *istoreidx*. *lorisplay* renders the partials stored with the label *ireadidx*. The labels are used only at initialization time, and may be reused without any cost or benefit in efficiency, and without introducing any interaction between instruments or instances.

## Performance

*lorisplay* implements signal reconstruction using Bandwidth-Enhanced Additive Synthesis. The control data is obtained from a stored set of bandwidth-enhanced partials imported from an SDIF file using *lorisread* or constructed by another unit generator such as *lorismorph*. *kfreqenv* 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. *kampenv* is a control-rate scale factor that is applied to all partial amplitude envelopes. *kbwenv* is a control-rate scale factor that is applied to all partial bandwidth or noisiness envelopes. The bandwidth-enhanced partial data is stored in memory with a specified label for future access by another generator.

## Credits

This implementation of the Loris unit generators was written by Kelly Fitz ([loris@cerlsoundgroup.org](mailto:loris@cerlsoundgroup.org) [<mailto:loris@cerlsoundgroup.org>]). It is patterned after a prototype implementation of the *lorisplay* unit generator written by Corbin Champion, and based on the method of Bandwidth-Enhanced Additive Synthesis and on the sound morphing algorithms implemented in the Loris library for sound modeling and manipulation. The opcodes were further adapted as a plugin for Csound 5 by Michael Gogins.

# loscil

loscil -- Read sampled sound from a table.

loscil

## Description

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

## Syntax

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

## Initialization

*ifn* -- function table number, typically denoting an 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 audio 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). If this value is not known or not present, use 1 here and in *kcps*.

*imod1*, *imod2* (optional, default=-1) -- 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. Make sure you select an appropriate mode if the file does not contain this information.

*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. If no loop points are specified, and a looping mode (*imod1*, *imod2*) is given, the file will be looped for the whole length.

## Performance

*ar1*, *ar2* -- the output at audio-rate. There is just *ar1* for mono output. However, there is both *ar1* and *ar2* for stereo output.

*xamp* -- the amplitude of the output signal.

*kcps* -- the frequency of the output signal in cycles per second.

*loscil* samples the *fable* 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, *fable* 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.

If you want to loop the whole file, specify a looping mode in *imod1* and do not enter any values for *ibeg* and *iend*.



## Note

This is mono loscil:

```
a1 loscil 10000, 1, 1, 1 ,1
```

...and this is stereo loscil:

```
a1, a2 loscil 10000, 1, 1, 1 ,1
```

## Examples

Here is an example of the *loscil* opcode. It uses the files *loscil.orc* [examples/loscil.orc], *loscil.sco* [examples/loscil.sco], and *beats.aiff* [examples/beats.aiff].

### Example 210. Example of the *loscil* opcode.

```
/* loscil.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  ; If you don't know the frequency of your audio file,
  ; set both the kcps and ibas parameters equal to 1.
  kcps = 1
  ifn = 1
  ibas = 1

  a1 loscil kamp, kcps, ifn, ibas
  out a1
endin
/* loscil.orc */

/* loscil.sco */
; Table #1: an audio file.
; Its table size is deferred,
; and format taken from the soundfile header.
f 1 0 0 1 "beats.aiff" 0 0 0
```

```
; Play Instrument #1 for 6 seconds.  
; This will loop the audio file several times.  
i 1 0 6  
e  
/* loscil.sco */
```

## See Also

*loscil3*

## Credits

Note about the mono/stereo difference was contributed by Rasmus Ekman.

Example written by Kevin Conder.

# loscil3

loscil3 -- Read sampled sound from a table using cubic interpolation.

loscil3

## Description

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

## Syntax

```
ar1 [,ar2] loscil3 xamp, kcps, ifn [, ibas] [, imod1] [, ibeg1] [, iend1] [, imod2] [, ibeg2] [, iend2]
```

## Initialization

*ifn* -- function table number, typically denoting a 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 audio 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). If this value is not known or not present, use 1 here and in *kcps*.

*imod1*, *imod2* (optional, default=-1) -- 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. Make sure you select an appropriate mode if the file does not contain this information.

*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. If no loop points are specified, and a looping mode (*imod1*, *imod2*) is given, the file will be looped for the whole length.

## Performance

*ar1*, *ar2* -- the output at audio-rate. There is just *ar1* for mono output. However, there is both *ar1* and *ar2* for stereo output.

*xamp* -- the amplitude of the output signal.

*kcps* -- the frequency of the output signal in cycles per second.

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



### Note

This is mono loscil3:

```
a1 loscil3 10000, 1, 1, 1, 1
```

...and this is stereo loscil3:

```
a1, a2 loscil3 10000, 1, 1, 1, 1
```

## Examples

Here is an example of the `loscil3` opcode. It uses the files *loscil3.orc* [examples/loscil3.orc], *loscil3.sco* [examples/loscil3.sco], and *beats.aiff* [examples/beats.aiff].

### Example 211. Example of the `loscil3` opcode.

```
/* loscil3.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  ; If you don't know the frequency of your audio file,
  ; set both the kcps and ibas parameters equal to 1.
  kcps = 1
  ifn = 1
  ibas = 1

  a1 loscil3 kamp, kcps, ifn, ibas
  out a1
endin
/* loscil3.orc */

/* loscil3.sco */
; Table #1: an audio file.
; Its table size is deferred,
; and format taken from the soundfile header.
f 1 0 0 1 "beats.aiff" 0 0 0

; Play Instrument #1 for 6 seconds.
; This will loop the drum pattern several times.
i 1 0 6
e
/* loscil3.sco */
```

## See Also

*loscil*

## Credits

Note about the mono/stereo difference was contributed by Rasmus Ekman.

Example written by Kevin Conder.

# lowpass2

lowpass2 -- A resonant lowpass filter.

lowpass2

## Description

Implementation of a resonant second-order lowpass filter.

## Syntax

ares **lowpass2** asig, kcf, kq [, iskip]

## 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 low-pass 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.

## Examples

Here is an example of the *lowpass2* opcode. It uses the files *lowpass2.orc* [examples/lowpass2.orc] and *lowpass2.sco* [examples/lowpass2.sco].

### Example 212. Example of the lowpass2 opcode.

```
/* lowpass.orc */
/* Written by Sean Costello */
; 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
/* lowpass.orc */

/* lowpass2.sco */
/* Written by Sean Costello */
f1 0 8192 9 1 1 .25

i1 0 5 100 1000
i1 5 5 200 1000
e
/* lowpass2.sco */
```

## Credits

Author: Sean Costello  
Seattle, Washington  
August 1999

New in Csound version 4.0

# lowres

lowres -- Another resonant lowpass filter.

lowres

## Description

*lowres* is a resonant lowpass filter.

## Syntax

ares **lowres** asig, kcutoff, kresonance [, iskip]

## 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

*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.

## Examples

Here is an example of the lowres opcode. It uses the files *lowres.orc* [examples/lowres.orc], *lowres.sco* [examples/lowres.sco] and *beats.wav* [examples/beats.wav].

### Example 213. Example of the lowres opcode.

```
/* lowres.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a nice sawtooth waveform.
  asig vco 5000, 440, 1

  ; Vary the cutoff frequency from 30 to 300 Hz.
  kcutoff line 30, p3, 300
  kresonance = 10

  ; Apply the filter.
  al lowres asig, kcutoff, kresonance
```

```
    out a1
endin
/* lowres.orc */

/* lowres.sco */
; Table #1, a sine wave for the vco opcode.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* lowres.sco */
```

## See Also

*lowresx*

## Credits

Author: Gabriel Maldonado (adapted by John fitch)  
Italy

Example written by Kevin Conder.

New in Csound version 3.49

# lowresx

lowresx -- Simulates layers of serially connected resonant lowpass filters.

lowresx

## Description

*lowresx* is equivalent to more layers of *lowres* with the same arguments serially connected.

## Syntax

ares **lowresx** asig, kcutoff, kresonance [, inumlayer] [, iskip]

## 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

*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

## Examples

Here is an example of the lowresx opcode. It uses the files *lowresx.orc* [examples/lowresx.orc], *lowresx.sco* [examples/lowresx.sco], and *beats.wav* [examples/beats.wav].

### Example 214. Example of the lowresx opcode.

```
/* lowresx.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - play the sawtooth waveform through a
; stack of filters.
instr 1
  ; Use a nice sawtooth waveform.
  asig vco 5000, 440, 1

  ; Vary the cutoff frequency from 30 to 300 Hz.
```

```
kcutoff line 30, p3, 300
kresonance = 3
inumlayer = 2

alr lowresx asig, kcutoff, kresonance, inumlayer

; It gets loud, so clip the output amplitude to 30,000.
a1 clip alr, 1, 30000
out a1
endin
/* lowresx.orc */
```

```
/* lowresx.sco */
; Table #1, a sine wave for the vco opcode.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* lowresx.sco */
```

## See Also

*lowres*

## Credits

Author: Gabriel Maldonado (adapted by John ffitch)  
Italy

New in Csound version 3.49

# lpf18

lpf18 -- A 3-pole sweepable resonant lowpass filter.

lpf18

## Description

Implementation of a 3 pole sweepable resonant lowpass filter.

## Syntax

ares **lpf18** asig, kfco, kres, kdist

## Performance

*kfco* -- the filter cutoff 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.

## Examples

Here is an example of the lpf18 opcode. It uses the files *lpf18.orc* [examples/lpf18.orc] and *lpf18.sco* [examples/lpf18.sco].

### Example 215. Example of the lpf18 opcode.

```
/* lpf18.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a sine waveform.
  ; Note that its amplitude (kamp) ranges from 0 to 1.
  kamp init 1
```

```
kcps init 440
knh init 3
ifn = 1
asine buzz kamp, kcps, knh, ifn

; Filter the sine waveform.
; Vary the cutoff frequency (kfco) from 300 to 3,000 Hz.
kfco line 300, p3, 3000
kres init 0.8
kdist init 0.3
aout lpf18 asine, kfco, kres, kdist

out aout * 30000
endin
/* lpf18.orc */

/* lpf18.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for four seconds.
i 1 0 4
e
/* lpf18.sco */
```

## Credits

Author: Josep M Comajuncosas  
Spain  
December 2000

Example written by Kevin Conder with help from Iain Duncan. Thanks goes to Iain for helping with the example.

New in Csound version 4.10

# lpfreson

lpfreson -- Resynthesises a signal from the data passed internally by a previous lpread, applying formant shifting.

lpfreson

## Description

Resynthesises a signal from the data passed internally by a previous lpread, applying formant shifting.

## Syntax

ares **lpfreson** asig, kfrqratio

## Performance

*asig* -- an audio driving function for resynthesis.

*kfrqratio* -- frequency ratio. Must be greater than 0.

*lpfreson* receives values internally produced by a leading *lpread*. *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 .3. During synthesis, the error signal value can be used to determine the nature of the *lpreson* driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a mix of the two. In normal speech resynthesis, the pitched input to *lpreson* is a wideband periodic signal or pulse train derived from a unit such as *buzz*, and the nonpitched source is usually derived from *rand*. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

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

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

## See Also

*lpread*, *lpreson*



# lphasor

lphasor -- Generates a table index for sample playback

lphasor

## Description

This opcode can be used to generate table index for sample playback (e.g. tablexkt).

## Syntax

```
ares lphasor xtrns [, ilps] [, ilpe] [, imode] [, istr] [, istor]
```

## Initialization

*ilps* -- loop start.

*ilpe* -- loop end (must be greater than *ilps* to enable looping). The default value of *ilps* and *ilpe* is zero.

*imode* (optional: default = 0) -- loop mode. Allowed values are:

- 0: no loop
- 1: forward loop
- 2: backward loop
- 3: forward-backward loop

*istr* (optional: default = 0) -- The initial output value (phase). It must be less than *ilpe* if looping is enabled, but is allowed to be greater than *ilps* (i.e. you can start playback in the middle of the loop).

*istor* (optional: default = 0) -- skip initialization if set to any non-zero value.

## Performance

*ares* -- a raw table index in samples (same unit for loop points). Can be used as index with the table opcodes.

*xtrns* -- transpose factor, expressed as a playback ratio. *ares* is incremented by this value, and wraps around loop points. For example, 1.5 means a fifth above, 0.75 means fourth below. It is not allowed to be negative.

## Credits

Author: Istvan Varga  
January 2002

New in version 4.18

Updated April 2002 and November 2002 by Istvan Varga

# lpinterp

lpinterp -- Computes a new set of poles from the interpolation between two analysis.

lpslot, lpinterp

## Description

Computes a new set of poles from the interpolation between two analysis.

## Syntax

**lpinterp** islot1, islot2, kmix

## Initialization

*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.

## Examples

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,krms0,kerr,kcps lpread  ktime,"square.pol"
        lpslot 1              ; Read triangle data poles
krmsr,krms0,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
```

## See Also

*lpslot*

## Credits

Author: Gabriel Maldonado

# lposcil

`lposcil` -- Read sampled sound from a table with optional looping and high precision.

`lposcil`, `lposcil3`

## Description

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

## Syntax

ares **lposcil** *kamp*, *kfregratio*, *kloop*, *kend*, *ifn* [, *iphs*]

## Initialization

*ifn* -- function table number

## Performance

*kamp* -- amplitude

*kfregratio* -- 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.

## See Also

*lposcil3*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.52

# lposcil3

`lposcil3` -- Read sampled sound from a table with high precision and cubic interpolation.

`lposcil3`

## Description

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

## Syntax

```
ares lposcil3 kamp, kfrequatio, kloop, kend, ifn [, iphs]
```

## Initialization

*ifn* -- function table number

## Performance

*kamp* -- amplitude

*kfrequatio* -- 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.

## See Also

*lposcil*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.52

# lpread

`lpread` -- Reads a control file of time-ordered information frames.

`lpread`

## Description

Reads a control file of time-ordered information frames.

## Syntax

`krmsr, krms0, kerr, kcps lpread ktmpnt, ifilcod [, inpoles] [, ifrmrate]`

## 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* (optional, default=0) -- number of poles in the lpc analysis. It is required only when the control file does not have a header; it is ignored when a header is detected.

*ifrmrate* (optional, default=0) -- frame rate per second in the lpc analysis. It is required only when the control file does not have a header; it is ignored when a header is detected.

## 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

*krms0* -- rms of the original signal

*kerr* -- the normalized error signal

*kcps* -- pitch in Hz

*ktmpnt* -- The passage of time, in seconds, through the analysis file. *ktmpnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

*lpread* gets its values from the control file according to the input value *ktmpnt* (in seconds). If *ktmpnt* 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 .3. During synthesis, the error signal value can be used to determine the nature of the *lpreson* driving function: for example, by arbitrating between pitched and non-pitched input,

or even by determining a mix of the two. In normal speech resynthesis, the pitched input to *lpreson* is a wideband periodic signal or pulse train derived from a unit such as *buzz*, and the nonpitched source is usually derived from *rand*. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

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

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

## See Also

*lpfreson*, *lpreson*

# lpreson

*lpreson* -- Resynthesises a signal from the data passed internally by a previous *lpread*.

*lpreson*

## Description

Resynthesises a signal from the data passed internally by a previous *lpread*.

## Syntax

ares **lpreson** asig

## Performance

*asig* -- an audio driving function for resynthesis.

*lpreson* receives values internally produced by a leading *lpread*. *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 .3. During synthesis, the error signal value can be used to determine the nature of the *lpreson* driving function: for example, by arbitrating between pitched and non-pitched input, or even by determining a mix of the two. In normal speech resynthesis, the pitched input to *lpreson* is a wideband periodic signal or pulse train derived from a unit such as *buzz*, and the nonpitched source is usually derived from *rand*. However, any audio signal can be used as the driving function, the only assumption of the analysis being that it has a flat response.

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

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

## See Also

*lpfreson*, *lpread*

# lpshold

lpshold -- Generate control signal consisting of held segments.

lpshold

## Description

Generate control signal consisting of held segments delimited by two or more specified points. The entire envelope is looped at *kfreq* rate. Each parameter can be varied at *k-rate*.

## Syntax

*ksig* **lpshold** *kfreq*, *ktrig*, *ctime0*, *kvalue0* [, *ctime1*] [, *kvalue1*] [, *ctime2*] [

## Performance

*ksig* -- Output signal

*kfreq* -- Repeat rate in Hz or fraction of Hz

*ktrig* -- If non-zero, retriggers the envelope from start (see *trigger opcode*), before the envelope cycle is completed.

*ctime0...ctimeN* -- Times of points; expressed in fraction of a cycle

*kvalue0...kvalueN* -- Values of points

*lpshold* is similar to *loopseg*, but can generate only horizontal segments, i.e. holds values for each time interval placed between *ctimeN* and *ctimeN+1*. It can be useful, among other things, for melodic control, like old analog sequencers.

## Examples

Here is an example of the *lpshold* opcode. It uses the files *lpshold.orc* [examples/lpshold.orc] and *lpshold.sco* [examples/lpshold.sco].

### Example 216. Example of the lpshold opcode.

```
/* lpshold.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1
instr 1
  kfreq line 1, p3, 20

  klp lpshold kfreq, 0, 0, 0, p3*0.25, 20000, p3*0.75, 0

  a1 oscil klp, 440, 1
  out a1
endin
/* lpshold.orc */
```



```
/* lpshold.sco */  
; Table #1, a sine wave.  
f 1 0 16384 10 1  
  
; Play Instrument #1 for five seconds.  
i 1 0 5  
e  
/* lpshold.sco */
```

## See Also

*loopseg*

## Credits

Author: Gabriel Maldonado

New in Version 4.13

# lpsholdp

lpsholdp -- Control signals based on held segments.

lpsholdp

## Description

Generate control signal consisting of held segments delimited by two or more specified points. The entire envelope can be looped at time-variant rate. Each segment coordinate can also be varied at k-rate.

## Syntax

```
ksig lpsholdp kphase, ktrig, ktime0, kvalue0 [, ktime1] [, kvalue1] [, ktime2
```

## Performance

*ksig* - output signal

*kphase* -

*kvalue0 ...kvalueN* - values of points

*ktime0 ...ktimeN* - times of points expressed in fraction of a cycle

*lpsholdp* opcode is similar to *lpshold*; the only difference is that, instead of frequency, a time-variant phase is required. If you use a phasor to get the phase value, you will have a behaviour identical to *lpshold*, but interesting results can be achieved when using phases having non-linear motions, making *lpsholdp* more powerful and general than *lpshold*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# lpslot

*lpslot* -- Selects the slot to be use by further lp opcodes.

*lpslot*

## Description

Selects the slot to be use by further lp opcodes.

## Syntax

**lpslot** *islot*

## Initialization

*islot* -- number of slot to be selected.

## Performance

*lpslot* selects the slot to be use by further lp opcodes. This is the way to load and reference several analyses at the same time.

## Examples

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,krms0,kerr,kcps lpread ktime,"square.pol"
       lpslot 1              ; Read triangle data poles
krmsr,krms0,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
```

## See Also

*lpinterp*

## Credits

Author: Mark Resibois  
Brussels  
1996

# mac

mac -- Multiplies and accumulates a- and k-rate signals.

mac

## Description

Multiplies and accumulates a- and k-rate signals.

## Syntax

ares **mac** asig1, ksig1 [, asig2] [, ksig2] [, asig3] [, ksig3] [...]

## 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:

$$\text{ares} = \text{asig1} * \text{ksig1} + \text{asig2} * \text{ksig2} + \text{asig3} * \text{ksig3} + \dots$$

## See Also

*maca*

## Credits

Author: John ffitch  
University of Bath, Codemist, Ltd.  
Bath, UK  
May 1999

New in Csound version 3.54

# maca

maca -- Multiply and accumulate a-rate signals only.

maca

## Description

Multiply and accumulate a-rate signals only.

## Syntax

ares **maca** asig1 , asig2 [, asig3] [, asig4] [, asig5] [...]

## Performance

*asig1, asig2, ...* -- a-rate input signals

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

$$\text{ares} = \text{asig1} * \text{asig2} + \text{asig3} * \text{asig4} + \text{asig5} * \text{asig6} + \dots$$

## See Also

*mac*

## Credits

Author: John fitch  
University of Bath, Codemist, Ltd.  
Bath, UK  
May 1999

New in Csound version 3.54

## madsr

madsr -- Calculates the classical ADSR envelope using the linsegr mechanism.

madsr

## Description

Calculates the classical ADSR envelope using the linsegr mechanism.

## Syntax

```
ares madsr iatt, idec, islev, irel [, idel] [, ireltim]
```

```
kres madsr iatt, idec, islev, irel [, idel] [, ireltim]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

*islev* -- level for sustain phase

*irel* -- duration of release phase.

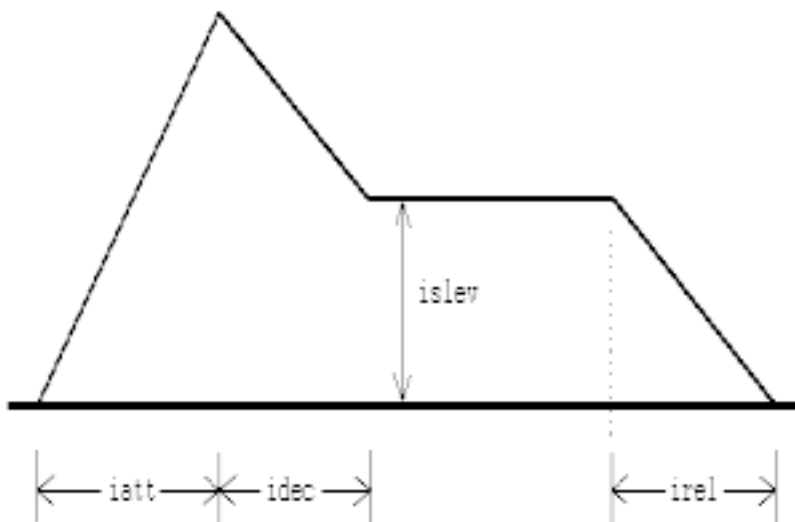
*idel* -- period of zero before the envelope starts

*ireltim* (optional, default=-1) -- Control release time after receiving a MIDI noteoff event. If less than zero, the longest release time given in the current instrument is used. If zero or more, the given value will be used for release time. Its default value is -1. (New in Csound 3.59 - not yet properly tested)

Please note that the release time cannot be longer than  $32767/kr$  seconds.

## Performance

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



Picture of an ADSR envelope.

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.

## Examples

Here is an example of the *madsr* opcode. It uses the files *madsr.orc* [examples/madsr.orc] and *madsr.sco* [examples/madsr.sco].

### Example 217. Example of the *madsr* opcode.

```
/* madsr.orc */
/* Written by Iain McCurdy */
; Initialize the global variables.
sr = 44100
kr = 441
ksmps = 100
nchnls = 1

; Instrument #1.
instr 1
  ; Attack time.
  iattack = 0.5
  ; Decay time.
  idecay = 0
  ; Sustain level.
  isustain = 1
  ; Release time.
  irelease = 0.5
  aenv madsr iattack, idecay, isustain, irelease

  al oscili 10000, 440, 1
  out al*aenv
endin
/* madsr.orc */
```

```
/* madsr.sco */
/* Written by Iain McCurdy */
; Table #1, a sine wave.
f 1 0 1024 10 1

; Leave the score running for 6 seconds.
f 0 6

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* madsr.sco */
```

## See Also

*adsr*, *mxadsr*, *xadsr*

## Credits

Author: John ffitch

November 2002. Thanks to Rasmus Ekman, added documentation for the *ireltim* parameter.

December 2002. Thanks to Iain McCurdy, added an example.

December 2002. Thanks to Istvan Varga, added documentation about the maximum release time.

New in Csound version 3.49.



# mandel

mandel -- Mandelbrot set

mandel

## Description

Returns the number of iterations corresponding to a given point of complex plane by applying the Mandelbrot set formula.

## Syntax

*kiter*, *koutrig* **mandel** *ktrig*, *kx*, *ky*, *kmaxIter*

## Performance

*kiter* - number of iterations

*koutrig* - output trigger signal

*ktrig* - input trigger signal

*kx*, *ky* - coordinates of a given point belonging to the complex plane

*kmaxIter* - maximum iterations allowed

*mandel* is an opcode that allows the use of the Mandelbrot set formula to generate an output that can be applied to any musical (or non-musical) parameter. It has two output arguments: *kiter*, that contains the iteration number of a given point, and *koutrig*, that generates a trigger 'bang' each time *kiter* changes. A new number of iterations is evaluated only when *ktrig* is set to a non-zero value. The coordinates of the complex plane are set in *kx* and *ky*, while *kmaxIter* contains the maximum number of iterations. Output values, which are integer numbers, can be mapped in any sorts of ways by the composer.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# mandol

mandol -- An emulation of a mandolin.

mandol

## Description

An emulation of a mandolin.

## Syntax

ares **mandol** *kamp*, *kfreq*, *kpluck*, *kdetune*, *kgain*, *ksize*, *ifn* [, *iminfreq*]

## Initialization

*ifn* -- table number containing the pluck wave form. The file *mandpluck.aiff* [examples/mandpluck.aiff] is suitable for this. It is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

*iminfreq* (optional, default=0) -- 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.

*kdetune* -- The proportional detuning between the two strings. Suggested range 0.9 to 1.

*kgain* -- the loop gain of the model, in the range 0.97 to 1.

*ksize* -- The size of the body of the mandolin. Range 0 to 2.

## Examples

Here is an example of the mandol opcode. It uses the files *mandol.orc* [examples/mandol.orc], *mandol.sco* [examples/mandol.sco], and *mandpluck.aiff* [examples/mandpluck.aiff].

### Example 218. Example of the mandol opcode.

```
/* mandol.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; kamp = 30000
  ; kfreq = 880
  ; kpluck = 0.4
```

```
; kdetune = 0.99
; kgain = 0.99
; ksize = 2
; ifn = 1
; ifreq = 220

a1 mandol 30000, 880, 0.4, 0.99, 0.99, 2, 1, 220

    out a1
endin
/* mandol.orc */


/* mandol.sco */
; Table #1: the "mandpluk.aiff" audio file
f 1 0 8192 1 "mandpluk.aiff" 0 0 0

; Play Instrument #1 for one second.
i 1 0 1
e
/* mandol.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

# marimba

marimba -- Physical model related to the striking of a wooden block.

marimba

## Description

Audio output is a tone related to the striking of a wooden block as found in a marimba. The method is a physical model developed from Perry Cook but re-coded for Csound.

## Syntax

ares **marimba** kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec [, idoubles

## 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* [examples/marmstk1.wav] is a suitable function from measurements and can be loaded with a *GENO1* table. It is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

*ivfn* -- shape of vibrato, usually a sine table, created by a function

*idec* -- time before end of note when damping is introduced

*idoubles* (optional) -- percentage of double strikes. Default is 40%.

*itriples* (optional) -- 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

## Examples

Here is an example of the marimba opcode. It uses the files *marimba.orc* [examples/marimba.orc], *marimba.sco* [examples/marimba.sco], and *marmstk1.wav* [examples/marmstk1.wav].

### Example 219. Example of the marimba opcode.

```
/* marimba.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
```

```
nchnls = 1

; Instrument #1.
instr 1
  ; kamp = 31129.60
  ; kfreq = 440
  ; ihrd = 0.5
  ; ipos = 0.561
  ; imp = 1
  ; kvibf = 6.0
  ; kvamp = 0.05
  ; ivibfn = 2
  ; idec = 0.1

  a1 marimba 31129.60, 440, 0.5, 0.561, 1, 6.0, 0.05, 2, 0.1

  out a1
endin
/* marimba.orc */


/* marimba.sco */
; Table #1, the "marmstk1.wav" audio file.
f 1 0 256 1 "marmstk1.wav" 0 0 0
; Table #2, a sine wave for the vibrato.
f 2 0 128 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* marimba.sco */
```

## See Also

*vibes*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# massign

`massign` -- Assigns a MIDI channel number to a Csound instrument.

`massign`

## Description

Assigns a MIDI channel number to a Csound instrument.

## Syntax

```
massign ichnl, insnum[, ireset]
```

```
massign ichnl, "insname"[, ireset]
```

## Initialization

*ichnl* -- MIDI channel number (1-16).

*insnum* -- Csound orchestra instrument number. If zero or negative, the channel is muted (i.e. it doesn't trigger a csound instrument, though information will still be received by opcodes like *midiiin*).

*"insname"* -- A string (in double-quotes) representing a named instrument.

*ireset* -- If non-zero resets the controllers; default is to reset.

## Performance

Assigns a MIDI channel number to a Csound instrument. Also useful to make sure a certain instrument (if its number is from 1 to 16) will not be triggered by midi noteon messages (if using something *midiiin* to interpret midi information). In this case set *insnum* to 0 or a negative number.

You can disable the turning on of any instruments by using the following in the header:

```
massign 0, 0  
pgmassign 0, 0
```

## See Also

*ctrlinit* and *pgmassign*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT, Cambridge, Mass.

New in Csound version 3.47

*ireset* parameter new in Csound5

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number

ranges.

# max

max -- Produces a signal that is the maximum of any number of input signals.

max

## Description

The *max* opcode takes any number of a-rate or k-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the maximum of all of the inputs. For a-rate signals, the inputs are compared one sample at a time (i.e. *max* does not scan an entire ksmps period of a signal for its local maximum as the *max\_k* opcode does).

## Syntax

```
amax max ain1 [, ain2] [, ain3] [, ain4] [...]
```

```
kmax max kin1 [, kin2] [, kin3] [, kin4] [...]
```

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

## See Also

*min, maxabs, minabs, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01



# maxabs

`maxabs` -- Produces a signal that is the maximum of the absolute values of any number of input signals.

`maxabs`

## Description

The *maxabs* opcode takes any number of a-rate or k-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the maximum of all of the inputs. It is identical to the *max* opcode except that it takes the absolute value of each input before comparing them. Therefore, the output is always non-negative. For a-rate signals, the inputs are compared one sample at a time (i.e. *maxabs* does not scan an entire ksmps period of a signal for its local maximum as the *max\_k* opcode does).

## Syntax

`amax maxabs ain1 [, ain2] [, ain3] [, ain4] [...]`

`kmax maxabs kin1 [, kin2] [, kin3] [, kin4] [...]`

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

## See Also

*minabs, max, min, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# maxabsaccum

**maxabsaccum** -- Accumulates the maximum of the absolute values of audio signals.

**maxabsaccum**

## Description

*maxabsaccum* compares two audio-rate variables and stores the maximum of their absolute values into the first.

## Syntax

**maxabsaccum** aAccumulator, aInput

## Performance

*aAccumulator* -- audio variable to store the maximum value

*aInput* -- signal that *aAccumulator* is compared to

The *maxabsaccum* opcode is designed to accumulate the maximum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the *maxabs* opcode. *maxabsaccum* is identical to *maxaccum* except that it takes the absolute value of *aInput* before the comparison. Its semantics are similar to *vincr* since *aAccumulator* is used as both an input and an output variable, except that *maxabsaccum* keeps the maximum absolute value instead of adding the signals together. *maxabsaccum* performs the following operation on each pair of samples:

$$\text{if } (\text{abs}(\text{aInput}) > \text{aAccumulator}) \text{ aAccumulator} = \text{abs}(\text{aInput})$$

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (k-period), after its value is read and used in some way, the accumulator variable should usually be reset to zero (perhaps by using the *clear* opcode). Clearing to zero is sufficient for *maxabsaccum*, unlike the *maxaccum* opcode.

## See Also

*minabsaccum*, *maxaccum*, *minaccum*, *max*, *min*, *maxabs*, *minabs*, *vincr* *clear*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# maxaccum

maxaccum -- Accumulates the maximum value of audio signals.

maxaccum

## Description

*maxaccum* compares two audio-rate variables and stores the maximum value between them into the first.

## Syntax

**maxaccum** aAccumulator, aInput

## Performance

*aAccumulator* -- audio variable to store the maximum value

*aInput* -- signal that *aAccumulator* is compared to

The *maxaccum* opcode is designed to accumulate the maximum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the *max* opcode. Its semantics are similar to *vincr* since *aAccumulator* is used as both an input and an output variable, except that *maxaccum* keeps the maximum value instead of adding the signals together. *maxaccum* performs the following operation on each pair of samples:

$$\text{if } (aInput > aAccumulator) \ aAccumulator = aInput$$

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (k-period), after its value is read and used in some way, the accumulator variable should usually be reset to zero (perhaps by using the *clear* opcode). Care must be taken however if *aInput* is negative at any point, in which case the accumulator should be initialized and reset to some large enough negative value that will always be less than the input signals to which it is compared.

## See Also

*minaccum, maxabsaccum, minabsaccum, max, min, maxabs, minabs, vincr clear*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# maxalloc

maxalloc -- Limits the number of allocations of an instrument.

maxalloc

## Description

Limits the number of allocations of an instrument.

## Syntax

**maxalloc** insnum, icount

## Initialization

*insnum* -- instrument number

*icount* -- number of instrument allocations

## Performance

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

## Examples

Here is an example of the maxalloc opcode. It uses the files *maxalloc.orc* [examples/maxalloc.orc] and *maxalloc.sco* [examples/maxalloc.sco].

### Example 220. Example of the maxalloc opcode.

```
/* maxalloc.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Limit Instrument #1 to three instances.
maxalloc 1, 3

; Instrument #1
instr 1
  ; Generate a waveform, get the cycles per second from the 4th p-field.
  al oscil 6500, p4, 1
  out al
endin
/* maxalloc.orc */

/* maxalloc.sco */
; Just generate a nice, ordinary sine wave.
f 1 0 32768 10 1
```

```
; Play five instances of Instrument #1 for one second.  
; Note that 4th p-field contains cycles per second.  
i 1 0 1 220  
i 1 0 1 440  
i 1 0 1 880  
i 1 0 1 1320  
i 1 0 1 1760  
e  
/* maxalloc.sco */
```

Its output should contain a message like this:

```
WARNING: cannot allocate last note because it exceeds instr maxalloc
```

## See Also

*cpuprc, prealloc*

## Credits

Author: Gabriel Maldonado  
Italy  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# max\_k

max\_k -- Local maximum (or minimum) value of an incoming asig signal

max\_k

## Description

*max\_k* outputs the local maximum (or minimum) value of the incoming *asig* signal, checked in the time interval between *ktrig* has become true twice.

## Syntax

knumkout **max\_k** asig, ktrig, itype

## Initialization

*itype* - itype determinates the behaviour of max\_k (see below)

## Performance

*asig* - incoming (input) signal

*ktrig* - trigger signal

max\_k outputs the local maximum (or minimum) value of the incoming *asig* signal, checked in the time interval between *ktrig* has become true twice. *itype* determinates the behaviour of max\_k:

- 1 - absolute maximum (sign of negative values is changed to positive before evaluation)
- 2 - actual maximum
- 3 - actual minimum
- 4 - calculate average value of *asig* in the time interval

This opcode can be useful in several situations, for example to implement a vu-meter.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# mclock

mclock -- Sends a MIDI CLOCK message.

mclock

## Description

Sends a MIDI CLOCK message.

## Syntax

`mclock ifreq`

## Initialization

*ifreq* -- clock message frequency rate in Hz

## Performance

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

## See Also

*mrtmsg*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

# mdelay

`mdelay` -- A MIDI delay opcode.

`mdelay`

## Description

A MIDI delay opcode.

## Syntax

**mdelay** *kstatus*, *kchan*, *kd1*, *kd2*, *kdelay*

## 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.

## Credits

Author: Gabriel Maldonado  
Italy  
November 1998

New in Csound version 3.492



# metro

metro -- Trigger Metronome

metro

## Description

Generate a metronomic signal to be used in any circumstance an isochronous trigger is needed.

## Syntax

```
ktrig metro kfreq [, initphase]
```

## Initialization

*initphase* - initial phase value (in the 0 to 1 range)

## Performance

*ktrig* - output trigger signal

*kfreq* - frequency of trigger bangs in cps

*metro* is a simple opcode that outputs a sequence of isochronous bangs (that is 1 values) each  $1/kfreq$  seconds. Trigger signals can be used in any circumstance, mainly to temporize realtime algorithmic compositional structures.

## Examples

Here is an example of the metro opcode. It uses the file *metro.csd* [examples/metro.csd]

### Example 221. Example of the metro opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr      =      44100
kr      =      100
ksmps   =      441
nchnls  =      2

        instr    1
ktrig metro 0.2
printk2 ktrig
        endin

</CsInstruments>
<CsScore>
i 1 0 20

</CsScore>
```

</CsoundSynthesizer>

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# midic14

`midic14` -- Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

`midic14`

## Description

Allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

`idest midic14 ictrlno1, ictrlno2, imin, imax [, ifn]`

`kdest midic14 ictrlno1, ictrlno2, kmin, kmax [, ifn]`

## Initialization

*idest* -- output signal

*ictrlno* -- most-significant byte controller number (0-127)

*ictrlno2* -- least-significant byte controller number (0-127)

*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 *imin* and *imax* values.

## Performance

*kdest* -- output signal

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

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

*midic14* (i- and k-rate 14 bit MIDI control) allows a floating-point 14-bit MIDI signal scaled with a minimum and a maximum range. The minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires two MIDI controllers as input.



### Note

Please note that the *midic* family of opcodes are designed for MIDI triggered events, and don't require a channel number since they will respond to the same channel as the one that triggered the instrument (see *massign*). However they will crash if called from a score driven event.

## See Also

*ctrl7, ctrl14, ctrl21, initc7, initc14, initc21, midic7, midic21*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# midic21

**midic21** -- Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

**midic21**

## Description

Allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

**idest midic21** *ictlno1*, *ictlno2*, *ictlno3*, *imin*, *imax* [, *ifn*]

**kdest midic21** *ictlno1*, *ictlno2*, *ictlno3*, *kmin*, *kmax* [, *ifn*]

## Initialization

*idest* -- output signal

*ictlno1* -- most-significant byte controller number (0-127)

*ictlno2* -- mid-significant byte controller number (0-127)

*ictlno3* -- least-significant byte controller number (0-127)

*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 the *imin* and *imax* values.

## Performance

*kdest* -- output signal

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

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

*midic21* (i- and k-rate 21 bit MIDI control) allows a floating-point 21-bit MIDI signal scaled with a minimum and a maximum range. Minimum and maximum values can be varied at k-rate. It can use optional interpolated table indexing. It requires three MIDI controllers as input.



### Note

Please note that the *midic* family of opcodes are designed for MIDI triggered events, and don't require a channel number since they will respond to the same channel as the one that triggered the instrument (see *massign*). However they will crash if called from a score driven event.

## See Also

*ctrl7*, *ctrl14*, *ctrl21*, *initc7*, *initc14*, *initc21*, *midic7*, *midic14*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# midic7

`midic7` -- Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

`midic7`

## Description

Allows a floating-point 7-bit MIDI signal scaled with a minimum and a maximum range.

## Syntax

`idest midic7 ictlno, imin, imax [, ifn]`

`kdest midic7 ictlno, kmin, kmax [, ifn]`

## Initialization

*idest* -- output signal

*ictlno* -- MIDI controller number (0-127)

*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 the *imin* and *imax* values.

## 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 a 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.



### Note

Please note that the *midic* family of opcodes are designed for MIDI triggered events, and don't require a channel number since they will respond to the same channel as the one that triggered the instrument (see *massign*). However they will crash if called from a score driven event.

## See Also

*ctrl7, ctrl14, ctrl21, initc7, initc14, initc21, midic14, midic21*

## Credits

Author: Gabriel Maldonado

Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.



# midichannelaftertouch

`midichannelaftertouch` -- Gets a MIDI channel's aftertouch value.

`midichannelaftertouch`

## Description

*midichannelaftertouch* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

`midichannelaftertouch` *xchannelaftertouch* [, *ilow*] [, *ihigh*]

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xchannelaftertouch* -- returns the MIDI channel aftertouch during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the value of *xchannelaftertouch* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the value of *xchannelaftertouch* remains unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
```

```
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## Examples

Here is an example of the *midichannelaftertouch* opcode. It uses the files *midichannelaftertouch.orc* [examples/midichannelaftertouch.orc] and *midichannelaftertouch.sco* [examples/midichannelaftertouch.sco].

### Example 222. Example of the *midichannelaftertouch* opcode.

```
/* midichannelaftertouch.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kaft init 0
  midichannelaftertouch kaft

  ; Display the aftertouch value when it changes.
  printk2 kaft
endin
/* midichannelaftertouch.orc */

/* midichannelaftertouch.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* midichannelaftertouch.sco */
```

Its output should include lines like:

```
i1 127.00000
i1 20.00000
i1 44.00000
```

## See Also

*midicontrolchange, mididefault, midinoteoff, midinoteoncps, midinoteonkey, midinoteonoct, midinoteonpch, midipitchbend, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

Example written by Kevin Conder.

New in version 4.20

# midichn

`midichn` -- Returns the MIDI channel number from which the note was activated.

`midichn`

## Description

*midichn* returns the MIDI channel number (1 - 16) from which the note was activated. In the case of score notes, it returns 0.

## Syntax

`ichn midichn`

## Initialization

*ichn* -- channel number. If the current note was activated from score, it is set to zero.

## Examples

Here is a simple example of the `midichn` opcode. It uses the files *midichn.orc* [examples/midichn.orc] and *midichn.sco* [examples/midichn.sco].

### Example 223. Example of the `midichn` opcode.

```
/* midichn.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il midichn

  print il
endin
/* midichn.orc */

/* midichn.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* midichn.sco */
```

Here is an advanced example of the `midichn` opcode. It uses the files *midichn\_advanced.mid* [examples/midichn\_advanced.mid], *midichn\_advanced.orc* [examples/midichn\_advanced.orc], and *midichn\_advanced.sco* [examples/midichn\_advanced.sco].

Don't forget that you must include the *-F flag* when using an external MIDI file like “midichn\_advanced.mid”.

**Example 224. An advanced example of the midichn opcode.**

```
/* midichn_advanced.orc - written by Istvan Varga */
sr = 44100
ksmps = 10
nchnls = 1

    massign 1, 1          ; all channels use instr 1
    massign 2, 1
    massign 3, 1
    massign 4, 1
    massign 5, 1
    massign 6, 1
    massign 7, 1
    massign 8, 1
    massign 9, 1
    massign 10, 1
    massign 11, 1
    massign 12, 1
    massign 13, 1
    massign 14, 1
    massign 15, 1
    massign 16, 1

gicnt = 0                ; note counter

    instr 1

gicnt = gicnt + 1 ; update note counter
kcnt init gicnt ; copy to local variable
ichn midichn      ; get channel number
istime times      ; note-on time

    if (ichn > 0.5) goto l2      ; MIDI note
    printks "note %.0f (time = %.2f) was activated from the score\\n", \
        3600, kcmt, istime
    goto l1
l2:
    printks "note %.0f (time = %.2f) was activated from channel %.0f\\n", \
        3600, kcmt, istime, ichn
l1:
    endin
/* midichn_advanced.orc - written by Istvan Varga */

/* midichn_advanced.sco - written by Istvan Varga */
t 0 60
f 0 6 2 -2 0
i 1 1 0.5
i 1 4 0.5
e
/* midichn_advanced.sco - written by Istvan Varga */
```

Its output should include lines like:

```
note 7 (time = 0.00) was activated from channel 4  
note 8 (time = 0.00) was activated from channel 2
```

## See Also

*pgmassign*

## Credits

Author: Istvan Varga  
May 2002

The simple example was written by Kevin Conder.

New in version 4.20

# midicontrolchange

`midicontrolchange` -- Gets a MIDI control change value.

`midicontrolchange`

## Description

*midicontrolchange* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

**midicontrolchange** *xcontroller*, *xcontrollervalue* [, *ilow*] [, *ihigh*]

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xcontroller* -- specifies a MIDI controller number (0-127).

*xcontrollervalue* -- returns the value of the MIDI controller during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of the *xcontroller* and *xcontrollervalue* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xcontroller* and *xcontrollervalue* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument  
; will have a positive p3 field.  
mididefault 60, p3  
; Puts MIDI key translated to cycles per  
; second into p4, and MIDI velocity into p5  
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## See Also

*midichannelaftertouch*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.



# midictrl

`midictrl` -- Get the current value (0-127) of a specified MIDI controller.

`midictrl`

## Description

Get the current value (0-127) of a specified MIDI controller.

## Syntax

`ival midictrl inum [ , imin] [ , imax]`

`kval midictrl inum [ , imin] [ , imax]`

## Initialization

*inum* -- MIDI controller number (0-127)

*imin, imax* -- set minimum and maximum limits on values obtained.

## Performance

Get the current value (0-127) of a specified MIDI controller.

## Warning

*midictrl* should only be used in notes that were triggered from MIDI, so that an associated channel number is available. For notes activated from the score, line events, or orchestra, the *ctrl7* opcode that takes an explicit channel number should be used instead.

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# mididefault

`mididefault` -- Changes values, depending on MIDI activation.

`mididefault`

## Description

*mididefault* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

`mididefault` *xdefault*, *xvalue*

## Performance

*xdefault* -- specifies a default value that will be used during MIDI activation.

*xvalue* -- overwritten by *xdefault* during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode will overwrite the value of *xvalue* with the value of *xdefault*. If the instrument was *NOT* activated by MIDI input, *xvalue* will remain unchanged.

This enables score pfields to receive a default value during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## See Also

*midichannelaftertouch*, *midicontrolchange*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midiin

midiin -- Returns a generic MIDI message received by the MIDI IN port.

midiin

## Description

Returns a generic MIDI message received by the MIDI IN port

## Syntax

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

## 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 message 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 MIDI IN buffer, is *kstatus* set to the type of the relevant messages.

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

# midinoteoff

midinoteoff -- Gets a MIDI noteoff value.

midinoteoff

## Description

*midinoteoff* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

**midinoteoff** *xkey*, *xvelocity*

## Performance

*xkey* -- returns MIDI key during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of the *xkey* and *xvelocity* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xkey* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## Examples

Here is an example of the *midinoteoff* opcode. It uses the files *midinoteoff.orc* [examples/midinoteoff.orc] and *midinoteoff.sco* [examples/midinoteoff.sco].

### Example 225. Example of the *midinoteoff* opcode.

```
/* midinoteoff.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kkey init 0
  kvelocity init 0

  midinoteoff kkey, kvelocity

  ; Display the key value when it changes.
  printk2 kkey
endin
/* midinoteoff.orc */

/* midinoteoff.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* midinoteoff.sco */
```

Its output should include lines like:

```
i1      60.00000
i1      76.00000
```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

Example written by Kevin Conder.

New in version 4.20

# midinoteoncps

`midinoteoncps` -- Gets a MIDI note number as a cycles-per-second frequency.

`midinoteoncps`

## Description

*midinoteoncps* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

`midinoteoncps xcps, xvelocity`

## Performance

*xcps* -- returns MIDI key translated to cycles per second during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xcps* and *xvelocity* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xcps* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```



Obviously, *midinoteoncps* could be changed to *midinoteonct* or any of the other options, and the choice of p-fields is arbitrary.

## Examples

Here is an example of the *midinoteoncps* opcode. It uses the files *midinoteoncps.orc* [examples/midinoteoncps.orc] and *midinoteoncps.sco* [examples/midinoteoncps.sco].

### Example 226. Example of the *midinoteoncps* opcode.

```
/* midinoteoncps.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kcps init 0
  kvelocity init 0

  midinoteoncps kcps, kvelocity

  ; Display the cycles-per-second value when it changes.
  printk2 kcps
endin
/* midinoteoncps.orc */

/* midinoteoncps.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* midinoteoncps.sco */
```

Its output should include lines like:

```
i1    261.62561
i1    440.00006
```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteonkey*, *midinoteonct*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

Example written by Kevin Conder.

New in version 4.20

# midinoteonkey

`midinoteonkey` -- Gets a MIDI note number value.

`midinoteonkey`

## Description

*midinoteonkey* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

`midinoteonkey` *xkey*, *xvelocity*

## Performance

*xkey* -- returns MIDI key during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xkey* and *xvelocity* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xkey* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonct* or any of the other options, and the choice of p-fields is arbitrary.

## Examples

Here is an example of the *midinoteonkey* opcode. It uses the files *midinoteonkey.orc* [examples/midinoteonkey.orc] and *midinoteonkey.sco* [examples/midinoteonkey.sco].

### Example 227. Example of the *midinoteonkey* opcode.

```
/* midinoteonkey.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kkey init 0
  kvelocity init 0

  midinoteonkey kkey, kvelocity

  ; Display the key value when it changes.
  printk2 kkey
endin
/* midinoteonkey.orc */

/* midinoteonkey.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* midinoteonkey.sco */
```

Its output should include lines like:

```
i1      60.00000
i1      69.00000
```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonct*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

Example written by Kevin Conder.

New in version 4.20

# midinoteonoct

`midinoteonoct` -- Gets a MIDI note number value as octave-point-decimal value.

`midinoteonoct`

## Description

*midinoteonoct* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

`midinoteonoct` *xoct*, *xvelocity*

## Performance

*xoct* -- returns MIDI key translated to linear octaves during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xoct* and *xvelocity* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xoct* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## Examples

Here is an example of the *midinoteonoct* opcode. It uses the files *midinoteonoct.orc* [examples/midinoteonoct.orc] and *midinoteonoct.sco* [examples/midinoteonoct.sco].

### Example 228. Example of the *midinoteonoct* opcode.

```
/* midinoteonoct.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  koct init 0
  kvelocity init 0

  midinoteonoct koct, kvelocity

  ; Display the octave-point-decimal value when it changes.
  printk2 koct
endin
/* midinoteonoct.orc */

/* midinoteonoct.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* midinoteonoct.sco */
```

Its output should include lines like:

```
i1      8.00000
i1      9.33333
```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

Example written by Kevin Conder.

New in version 4.20



# midinoteonpch

midinoteonpch -- Gets a MIDI note number as a pitch-class value.

midinoteonpch

## Description

*midinoteonpch* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

**midinoteonpch** *xpch*, *xvelocity*

## Performance

*xpch* -- returns MIDI key translated to octave.pch during MIDI activation, remains unchanged otherwise.

*xvelocity* -- returns MIDI velocity during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xpch* and *xvelocity* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xpch* and *xvelocity* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## Examples

Here is an example of the *midinoteonpch* opcode. It uses the files *midinoteonpch.orc* [examples/midinoteonpch.orc] and *midinoteonpch.sco* [examples/midinoteonpch.sco].

### Example 229. Example of the *midinoteonpch* opcode.

```
/* midinoteonpch.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kpch init 0
  kvelocity init 0

  midinoteonpch kpch, kvelocity

  ; Display the pitch-class value when it changes.
  printk2 kpch
endin
/* midinoteonpch.orc */

/* midinoteonpch.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* midinoteonpch.sco */
```

Its output should include lines like:

```
i1      8.09000
i1      9.05000
```

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midipitchbend*, *midipolyaftertouch*, *midiprogramchange*

## Credits

Author: Michael Gogins

Example written by Kevin Conder.

New in version 4.20

# midion

midion -- Plays MIDI notes.

midion

## Description

Plays MIDI notes.

## Syntax

**midion** *kchn*, *knum*, *kvel*

## Performance

*kchn* -- MIDI channel number (1-16)

*knum* -- note number (0-127)

*kvel* -- velocity (0-127)

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

## See Also

*moscil*

## Credits

Author: Gabriel Maldonado  
Italy  
May 1997

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# midion2

midion2 -- Sends noteon and noteoff messages to the MIDI OUT port.

midion2

## Description

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

## Syntax

**midion2** *kchn*, *knum*, *kvel*, *ktrig*

## Performance

*kchn* -- MIDI channel (1-16)

*knum* -- MIDI note number (0-127)

*kvel* -- note velocity (0-127)

*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.

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# midiout

midiout -- Sends a generic MIDI message to the MIDI OUT port.

midiout

## Description

Sends a generic MIDI message to the MIDI OUT port.

## Syntax

**midiout** *kstatus*, *kchan*, *kdata1*, *kdata2*

## 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

*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.

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

# midipitchbend

midipitchbend -- Gets a MIDI pitchbend value.

midipitchbend

## Description

*midipitchbend* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

**midipitchbend** *xpitchbend* [, *ilow*] [, *ihigh*]

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xpitchbend* -- returns the MIDI pitch bend during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the value of *xpitchbend* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the value of *xpitchbend* remains unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument  
; will have a positive p3 field.
```

```
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## Examples

Here is an example of the *midipitchbend* opcode. It uses the files *midipitchbend.orc* [examples/midipitchbend.orc] and *midipitchbend.sco* [examples/midipitchbend.sco].

### Example 230. Example of the *midipitchbend* opcode.

```
/* midipitchbend.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kpb init 0

  midipitchbend kpb

  ; Display the pitch-bend value when it changes.
  printk2 kpb
endin
/* midipitchbend.orc */

/* midipitchbend.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* midipitchbend.sco */
```

Its output should include lines like:

```
i1      0.12695
i1      0.00000
i1     -0.01562
```

## See Also



*midichannelaftertouch, midicontrolchange, mididefault, midinoteoff, midinoteoncps, midinoteonkey, midinoteonoct, midinoteonpch, midipolyaftertouch, midiprogramchange*

## Credits

Author: Michael Gogins

Example written by Kevin Conder.

New in version 4.20

# midipolyaftertouch

`midipolyaftertouch` -- Gets a MIDI polyphonic aftertouch value.

`midipolyaftertouch`

## Description

*midipolyaftertouch* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

`midipolyaftertouch` *xpolyaftertouch*, *xcontrollervalue* [, *ilow*] [, *ihigh*]

## Initialization

*ilow* (optional) -- optional low value after rescaling, defaults to 0.

*ihigh* (optional) -- optional high value after rescaling, defaults to 127.

## Performance

*xpolyaftertouch* -- returns MIDI polyphonic aftertouch during MIDI activation, remains unchanged otherwise.

*xcontrollervalue* -- returns the value of the MIDI controller during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the values of *xpolyaftertouch* and *xcontrollervalue* with the corresponding values from MIDI input. If the instrument was *NOT* activated by MIDI input, the values of *xpolyaftertouch* and *xcontrollervalue* remain unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument  
; will have a positive p3 field.  
mididefault 60, p3  
; Puts MIDI key translated to cycles per  
; second into p4, and MIDI velocity into p5  
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipitchbend*, *midiprogramchange*

## Credits

Author: Michael Gogins

New in version 4.20

# midiprogramchange

midiprogramchange -- Gets a MIDI program change value.

midiprogramchange

## Description

*midiprogramchange* is designed to simplify writing instruments that can be used interchangeably for either score or MIDI input, and to make it easier to adapt instruments originally written for score input to work with MIDI input.

In general, it should be possible to write instrument definitions that work identically with both scores and MIDI, including both MIDI files and real-time MIDI input, without using any conditional statements, and that take full advantage of MIDI voice messages.

Note that correlating Csound instruments with MIDI channel numbers is done using the *massign* opcode for real-time performance,. For file-driven performance, instrument numbers default to MIDI channel number + 1, but the defaults are overridden by any MIDI program change messages in the file.

## Syntax

**midiprogramchange** xprogram

## Performance

*xprogram* -- returns the MIDI program change value during MIDI activation, remains unchanged otherwise.

If the instrument was activated by MIDI input, the opcode overwrites the value of *xprogram* with the corresponding value from MIDI input. If the instrument was *NOT* activated by MIDI input, the value of *xprogram* remains unchanged.

This enables score p-fields to receive MIDI input data during MIDI activation, and score values otherwise.



### Adapting a score-activated Csound instrument.

To adapt an ordinary Csound instrument designed for score activation for score/MIDI interoperability:

- Change all *linen*, *linseg*, and *expseg* opcodes to *linenr*, *linsegr*, and *expsegr*, respectively, except for a de-clicking or damping envelope. This will not materially change score-driven performance.
- Add the following lines at the beginning of the instrument definition:

```
; Ensures that a MIDI-activated instrument
; will have a positive p3 field.
mididefault 60, p3
; Puts MIDI key translated to cycles per
; second into p4, and MIDI velocity into p5
midinoteoncps p4, p5
```

Obviously, *midinoteoncps* could be changed to *midinoteonoct* or any of the other options, and the choice of p-fields is arbitrary.

## See Also

*midichannelaftertouch*, *midicontrolchange*, *mididefault*, *midinoteoff*, *midinoteoncps*, *midinoteonkey*, *midinoteonoct*, *midinoteonpch*, *midipitchbend*, *midipolyaftertouch*

## Credits

Author: Michael Gogins

New in version 4.20

# miditempo

miditempo -- Returns the current tempo at k-rate, of either the MIDI file (if available) or the score

miditempo

## Description

Returns the current tempo at k-rate, of either the MIDI file (if available) or the score

## Syntax

```
ksig miditempo
```

## Credits

Author: Istvan Varga  
March 2005  
New in Csound5

# min

min -- Produces a signal that is the minimum of any number of input signals.

min

## Description

The *min* opcode takes any number of a-rate or k-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the minimum of all of the inputs. For a-rate signals, the inputs are compared one sample at a time (i.e. *min* does not scan an entire ksmps period of a signal for its local minimum as the *max\_k* opcode does).

## Syntax

```
amin min ain1 [, ain2] [, ain3] [, ain4] [...]
```

```
kmin min kin1 [, kin2] [, kin3] [, kin4] [...]
```

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

## See Also

*max, maxabs, minabs, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# minabs

`minabs` -- Produces a signal that is the minimum of the absolute values of any number of input signals.

`minabs`

## Description

The *minabs* opcode takes any number of a-rate or k-rate signals as input (all of the same rate), and outputs a signal at the same rate that is the minimum of all of the inputs. It is identical to the *min* opcode except that it takes the absolute value of each input before comparing them. Therefore, the output is always non-negative. For a-rate signals, the inputs are compared one sample at a time (i.e. *minabs* does not scan an entire ksmps period of a signal for its local minimum as the *max\_k* opcode does).

## Syntax

```
amin minabs ain1 [, ain2] [, ain3] [, ain4] [...]
```

```
kmin minabs kin1 [, kin2] [, kin3] [, kin4] [...]
```

## Performance

*ain1, ain2, ...* -- a-rate signals to be compared.

*kin1, kin2, ...* -- k-rate signals to be compared.

## See Also

*maxabs, max, min, maxaccum, minaccum, maxabsaccum, minabsaccum, max\_k*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01



# minabsaccum

`minabsaccum` -- Accumulates the minimum of the absolute values of audio signals.

`minabsaccum`

## Description

*minabsaccum* compares two audio-rate variables and stores the minimum of their absolute values into the first.

## Syntax

**minabsaccum** *aAccumulator*, *aInput*

## Performance

*aAccumulator* -- audio variable to store the minimum value

*aInput* -- signal that *aAccumulator* is compared to

The *minabsaccum* opcode is designed to accumulate the minimum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the *minabs* opcode. *minabsaccum* is identical to *minaccum* except that it takes the absolute value of *aInput* before the comparison. Its semantics are similar to *vincr* since *aAccumulator* is used as both an input and an output variable, except that *minabsaccum* keeps the minimum absolute value instead of adding the signals together. *minabsaccum* performs the following operation on each pair of samples:

$$\text{if } (\text{abs}(\text{aInput}) < \text{aAccumulator}) \text{ aAccumulator} = \text{abs}(\text{aInput})$$

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (k-period), after its value is read and used in some way, the accumulator variable should usually be reset to some large enough positive value that will always be greater than the input signals to which it is compared.

## See Also

*maxabsaccum*, *maxaccum*, *minaccum*, *max*, *min*, *maxabs*, *minabs*, *vincr*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# minaccum

**minaccum** -- Accumulates the minimum value of audio signals.

**minaccum**

## Description

*minaccum* compares two audio-rate variables and stores the minimum value between them into the first.

## Syntax

**minaccum** aAccumulator, aInput

## Performance

*aAccumulator* -- audio variable to store the minimum value

*aInput* -- signal that aAccumulator is compared to

The *minaccum* opcode is designed to accumulate the minimum value from among many audio signals that may be in different note instances, different channels, or otherwise cannot all be compared at once using the *min* opcode. Its semantics are similar to *vincr* since *aAccumulator* is used as both an input and an output variable, except that *minaccum* keeps the minimum value instead of adding the signals together. *minaccum* performs the following operation on each pair of samples:

```
if (aInput < aAccumulator) aAccumulator = aInput
```

*aAccumulator* will usually be a global audio variable. At the end of any given computation cycle (k-period), after its value is read and used in some way, the accumulator variable should usually be reset to some large enough positive value that will always be greater than the input signals to which it is compared.

## See Also

*maxaccum*, *maxabsaccum*, *minabsaccum*, *max*, *min*, *maxabs*, *minabs*, *vincr*

## Credits

Author: Anthony Kozar  
March 2006

New in Csound version 5.01

# mirror

`mirror` -- Reflects the signal that exceeds the low and high thresholds.

`mirror`

## Description

Reflects the signal that exceeds the low and high thresholds.

## Syntax

`ares mirror asig, klow, khigh`

`ires mirror isig, ilow, ihigh`

`kres mirror ksig, klow, khigh`

## Initialization

*isig* -- input signal

*ilow* -- low threshold

*ihigh* -- high threshold

## Performance

*xsig* -- input signal

*klow* -- low threshold

*khigh* -- high threshold

*mirror* “reflects” the signal that exceeds the low and high thresholds.

This opcode is useful in several situations, such as table indexing or for clipping and modeling a-rate, i-rate or k-rate signals.

## See Also

*limit*, *wrap*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.49

# MixerSetLevel

MixerSetLevel -- Sets the level of a send to a buss.

MixerSetLevel

## Syntax

**MixerSetLevel** isend, ibuss, kgain

## Description

Sets the level at which signals from the send are added to the buss. The actual sending of the signal to the buss is performed by the *MixerSend* opcode.

## Initialization

*isend* -- The number of the send, for example the number of the instrument sending the signal (but any integer can be used).

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal (but any integer can be used).

Setting the gain for a buss also creates the buss.

## Performance

*kgain* -- The level (any real number) at which the signal from the send will be mixed onto the buss. The default is 0.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses before the next kperiod.

## Examples

In the orchestra, define an instrument to control mixer levels:

```
instr 1
    MixerSetLevel      p4, p5, p6
endin
```

In the score, use that instrument to set mixer levels:

```
; SoundFonts
; to Chorus
i 1 0 0 100 200 0.9
; to Reverb
i 1 0 0 100 210 0.7
; to Output
i 1 0 0 100 220 0.3

; Kelley Harpsichord
; to Chorus
i 1 0 0 3 200 0.30
```

```
; to Reverb
i 1 0 0 3 210 0.9
; to Output
i 1 0 0 3 220 0.1

; Chorus to Reverb
i 1 0 0 200 210 0.5
; Chorus to Output
i 1 0 0 200 220 0.5
; Reverb to Output
i 1 0 0 210 220 0.2
```

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerGetLevel

MixerGetLevel -- Gets the level of a send to a buss.

MixerGetLevel

## Syntax

*kgain* **MixerGetLevel** *isend*, *ibuss*

## Description

Gets the level at which signals from the send are being added to the buss. The actual sending of the signal to the buss is performed by the *MixerSend* opcode.

## Initialization

*isend* -- The number of the send, for example the number of the instrument sending the signal.

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal.

## Performance

*kgain* -- The level (any real number) at which the signal from the send will be mixed onto the buss.

This opcode reports the level set by *MixerSetLevel* for a send and buss pair.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next kperiod.

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerSend

MixerSend -- Mixes an arate signal into a channel of a buss.

MixerSend

## Syntax

**MixerSend** asignal, isend, ibuss, ichannel

## Description

Mixes an arate signal into a channel of a buss.

## Initialization

*isend* -- The number of the send, for example the number of the instrument sending the signal. The gain of the send is controlled by the *MixerSetLevel* opcode. The reason that the sends are numbered is to enable different levels for different sends to be set independently of the actual level of the signals.

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal.

*ichannel* -- The number of the channel. Each buss has *nchnls* channels.

## Performance

*asignal* -- The signal that will be mixed into the indicated channel of the buss.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next kperiod.

## Examples

```
instr 100 ; Fluidsynth output
; INITIALIZATION
; Normalize so iamplitude for p5 of 80 == ampdb(80).
iamplitude = ampdb(p5) * 2.0
; AUDIO
aleft, aright fluidAllOut giFluidsynth
asig1 = aleft * iamplitude
asig2 = aright * iamplitude
; To the chorus.
MixerSend asig1, 100, 200, 0
MixerSend asig2, 100, 200, 1
; To the reverb.
MixerSend asig1, 100, 210, 0
MixerSend asig2, 100, 210, 1
; To the output.
MixerSend asig1, 100, 220, 0
MixerSend asig2, 100, 220, 1
endin
```

## Credits

Michael Gogins (gogins at pipeline dot com).



# MixerReceive

MixerReceive -- Receives an arate signal from a channel of a buss.

MixerReceive

## Syntax

```
asignal MixerReceive ibuss, ichannel
```

## Description

Receives an arate signal that has been mixed onto a channel of a buss.

## Initialization

*ibuss* -- The number of the buss, for example the number of the instrument receiving the signal.

*ichannel* -- The number of the channel. Each buss has `nchnls` channels.

## Performance

*asignal* -- The signal that has been mixed onto the indicated channel of the buss.

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next kperiod.

## Examples

```
instr 220 ; Master output
    ; It applies a bass enhancement, compression and fadeout
    ; to the whole piece, outputs signals, and clears the mixer.
a1 MixerReceive 220, 0
a2 MixerReceive 220, 1
    ; Bass enhancement
a11 butterlp a1, 100
a12 butterlp a2, 100
a1 = a11*1.5 +a1
a2 = a12*1.5 +a2

    ; Global amplitude shape
kenv linseg 0., p5 / 2.0, p4, p3 - p5, p4, p5 / 2.0, 0.
a1=a1*kenv
a2=a2*kenv

    ; Compression
a1 dam a1, 5000, 0.5, 1, 0.2, 0.1
a2 dam a2, 5000, 0.5, 1, 0.2, 0.1

    ; Remove DC bias
a1blocked dcblock a1
a2blocked dcblock a2

    ; Output signals
outs a1blocked, a2blocked
MixerClear
endin
```

## Credits

Michael Gogins (gogins at pipeline dot com).

# MixerClear

MixerClear -- Resets all channels of a buss to 0.

MixerClear

## Syntax

**MixerClear**

## Description

Resets all channels of a buss to 0.

## Performance

Use of the mixer requires that instruments setting gains have smaller numbers than instruments sending signals, and that instruments sending signals have smaller numbers than instruments receiving those signals. However, an instrument may have any number of sends or receives. After the final signal is received, *MixerClear* must be invoked to reset the busses to 0 before the next kperiod.

## Examples

```
instr 220 ; Master output
    ; It applies a bass enhancement, compression and fadeout
    ; to the whole piece, outputs signals, and clears the mixer.
a1 MixerReceive 220, 0
a2 MixerReceive 220, 1
    ; Bass enhancement
a11 butterlp a1, 100
a12 butterlp a2, 100
a1 = a11*1.5 +a1
a2 = a12*1.5 +a2

    ; Global amplitude shape
kenv linseg 0., p5 / 2.0, p4, p3 - p5, p4, p5 / 2.0, 0.
a1=a1*kenv
a2=a2*kenv

    ; Compression
a1 dam a1, 5000, 0.5, 1, 0.2, 0.1
a2 dam a2, 5000, 0.5, 1, 0.2, 0.1

    ; Remove DC bias
a1blocked dcblock a1
a2blocked dcblock a2

    ; Output signals
outs a1blocked, a2blocked
MixerClear
endin
```

## Credits

Michael Gogins (gogins at pipeline dot com).

# monitor

monitor -- Returns the audio spout frame.

monitor

## Description

Returns the audio spout frame (if active), otherwise it returns zero.

## Syntax

aout1 [,aout2 ... aoutX] **monitor**

## Performance

This opcode can be used for monitoring the output signal from csound. It should not be used for processing the signal further.

## See also

The *Mixer opcodes* and the *Zak Patching System*.

## Credits

Istvan Varga 2006

# moog

**moog** -- An emulation of a mini-Moog synthesizer.

**moog**

## Description

An emulation of a mini-Moog synthesizer.

## Syntax

ares **moog** *kamp*, *kfreq*, *kfiltq*, *kfiltrate*, *kvibf*, *kvamp*, *iafn*, *iwfn*, *ivfn*

## 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* [examples/mandpluk.aiff] and *impuls20.aiff* [examples/impuls20.aiff] are suitable for the first two, and a sine wave for the last.



### Note

The files “mandpluk.aiff” and “impuls20.aiff” are also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

## 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

## Examples

Here is an example of the moog opcode. It uses the files *moog.orc* [examples/moog.orc], *moog.sco* [examples/moog.sco], *mandpluk.aiff* [examples/mandpluk.aiff], and *impuls20.aiff* [examples/impuls20.aiff].

### Example 231. Example of the moog opcode.

```
/* moog.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
  kamp = 30000
  kfreq = 220
  kfiltq = 0.81
  kfiltrate = 0
  kvibf = 1.4
  kvamp = 2.22
  iafn = 1
  iwfn = 2
  ivfn = 3

  am moog kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn, iwfn, ivfn

  ; It tends to get loud, so clip moog's amplitude at 30,000.
  al clip am, 2, 30000
  out al
endin
/* moog.orc */


/* moog.sco */
; Table #1: the "mandpluk.aiff" audio file
f 1 0 8192 1 "mandpluk.aiff" 0 0 0
; Table #2: the "impuls20.aiff" audio file
f 2 0 256 1 "impuls20.aiff" 0 0 0
; Table #3: a sine wave
f 3 0 256 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
e
/* moog.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47

# moogladder

moogladder -- Moog ladder lowpass filter.

moogladder

## Description

Moogladder is a new digital implementation of the Moog ladder filter based on the work of Antti Huovilainen, described in the paper "Non-Linear Digital Implementation of the Moog Ladder Filter" (Proceedings of DaFX04, Univ of Napoli). This implementation is probably a more accurate digital representation of the original analogue filter.

## Syntax

asig **moogladder** ain, kcf, kres[, istor]

## Initialization

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

## Performance

*asig* -- input signal.

*kcf* -- filter cutoff frequency

*kres* -- resonance, generally < 1, but not limited to it. Higher than 1 resonance values might cause aliasing, analogue synths generally allow resonances to be above 1.

## Examples

### Example 232. Example

```
kfe      expseg 500, p3*0.9, 1800, p3*0.1, 3000
kenv     linen 10000, 0.05, p3, 0.05
asig     buzz  kenv, 100, sr/(200), 1
afil     moogladder asig, kfe, 1

        out afil
```

## Credits

Author: Victor Lazzarini;  
January 2005

New plugin in version 5

January 2005.



# moogvcf

moogvcf -- A digital emulation of the Moog diode ladder filter configuration.

moogvcf

## Description

A digital emulation of the Moog diode ladder filter configuration.

## Syntax

```
ares moogvcf asig, xfco, xres [,  
    iscale, iskip]
```

## Initialization

*iscale* (optional, default=1) -- 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)  
*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## 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. Self-oscillation occurs when *xres* is approximately one. As of version 3.50, may a-rate, i-rate, or k-rate.

*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

Here is an example of the moogvcf opcode. It uses the files *moogvcf.orc* [examples/moogvcf.orc] and *moogvcf.sco* [examples/moogvcf.sco].

### Example 233. Example of the moogvcf opcode.

```
/* moogvcf.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Instrument #1.  
instr 1  
    ; Use a nice sawtooth waveform.  
    asig vco 32000, 220, 1
```

```
; Vary the filter-cutoff frequency from .2 to 2 KHz.
kfco line 200, p3, 2000

; Set the resonance amount to one.
krez init 1

; Scale the amplitude to 32768.
iscale = 32768

al moogvcf asig, kfco, krez, iscale

out al
endin
/* moogvcf.orc */

/* moogvcf.sco */
; Table #1, a sine wave for the vco opcode.
f 1 0 16384 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
e
/* moogvcf.sco */
```

## See Also

*biquad, rezzv*

## Credits

Author: Hans Mikelson  
October 1998

Example written by Kevin Conder.

New in Csound version 3.49

# moscil

moscil -- Sends a stream of the MIDI notes.

moscil

## Description

Sends a stream of the MIDI notes.

## Syntax

**moscil** *kchn*, *knum*, *kvel*, *kdur*, *kpause*

## Performance

*kchn* -- MIDI channel number (1-16)

*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.

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

## See Also

*midion*

## Credits

Author: Gabriel Maldonado  
Italy  
May 1997

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# mpulse

mpulse -- Generates a set of impulses.

mpulse

## Description

Generates 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.

## Syntax

ares **mpulse** kamp, kfreq [, ioffset]

## Initialization

*ioffset* (optional, default=0) -- 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.

## Examples

Here is an example of the mpulse opcode. It uses the files *mpulse.orc* [examples/mpulse.orc] and *mpulse.sco* [examples/mpulse.sco].

### Example 234. Example of the mpulse opcode.

```
/* mpulse.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate an impulse every 1/10th of a second.
  kamp = 30000
  kfreq = 0.1

  a1 mpulse kamp, kfreq
  out a1
endin
/* mpulse.orc */
```

```
/* mpulse.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* mpulse.sco */
```

## Credits

Example written by Kevin Conder.

# mrtmsg

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

mrtmsg

## Description

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

## Syntax

**mrtmsg** *imsgtype*

## Initialization

*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

Sends a real-time message once, in init stage of current instrument. *imsgtype* parameter is a flag to indicate the message type.

## See Also

*mclock*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

# multitap

multitap -- Multitap delay line implementation.

multitap

## Description

Multitap delay line implementation.

## Syntax

```
ares multitap asig [, itime1] [, igain1] [, itime2] [, igain2] [...]
```

## Initialization

The arguments *itime* and *igain* set the position and gain of each tap.

The delay line is fed by *asig*.

## Examples

```
a1      oscil      1000, 100, 1  
a2      multitap   a1, 1.2, .5, 1.4, .2  
out     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.

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1996

# mute

mute -- Mutes/unmutes new instances of a given instrument.

mute

## Description

Mutes/unmutes new instances of a given instrument.

## Syntax

```
mute insnum [, iswitch]
```

```
mute "insname" [, iswitch]
```

## Initialization

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

*"insname"* -- A string (in double-quotes) representing a named instrument.

*iswitch* (optional, default=0) -- represents a switch to mute/unmute an instrument. A value of 0 will mute new instances of an instrument, other values will unmute them. The default value is 0.

## Performance

All new instances of instrument *inst* will be muted (*iswitch* = 0) or unmuted (*iswitch* not equal to 0). There is no difficulty with muting muted instruments or unmuting unmuted instruments. The mechanism is the same as used by the score *q statement*. For example, it is possible to mute in the score and unmute in some instrument.

Muting/Unmuting is indicated by a message (depending on message level).

## Examples

Here is an example of the mute opcode. It uses the files *mute.orc* [examples/mute.orc] and *mute.sco* [examples/mute.sco].

### Example 235. Example of the mute opcode.

```
/* mute.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Mute Instrument #2.
mute 2

; Instrument #1.
instr 1
  al oscils 10000, 440, 0
  out al
```



```
endin

; Instrument #2.
instr 2
  al oscils 10000, 880, 0
  out al
endin
/* mute.orc */

/* mute.sco */
; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e
/* mute.sco */
```

## Credits

Example written by Kevin Conder.

New in version 4.22

## mxadsr

`mxadsr` -- Calculates the classical ADSR envelope using the expsegr mechanism.

`mxadsr`

## Description

Calculates the classical ADSR envelope using the expsegr mechanism.

## Syntax

```
ares mxadsr iatt, idec, islev, irel [, idel] [, ireltim]
```

```
kres mxadsr iatt, idec, islev, irel [, idel] [, ireltim]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

*islev* -- level for sustain phase

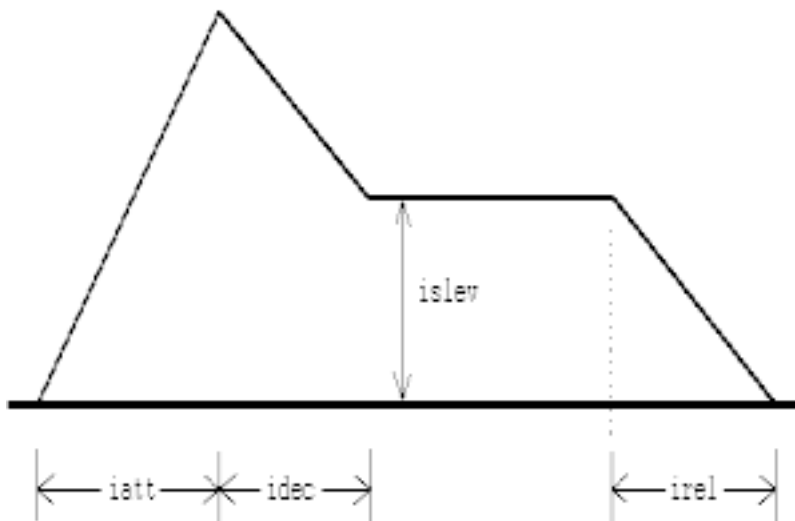
*irel* -- duration of release phase

*idel* (optional, default=0) -- period of zero before the envelope starts

*ireltim* (optional, default=-1) -- Control release time after receiving a MIDI noteoff event. If less than zero, the longest release time given in the current instrument is used. If zero or more, the given value will be used for release time. Its default value is -1. (New in Csound 3.59 - not yet properly tested)

## Performance

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



Picture of an ADSR envelope.

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 opcode *mxadsr* is identical to *madsr* except it uses exponential, rather than linear, line segments.

*mxadsr* is new in Csound version 3.51.

## See Also

*adsr*, *madsr*, *xadsr*

## Credits

Author: John ffitch

November 2002. Thanks to Rasmus Ekman, added documentation for the *ireltim* parameter.

November 2003. Thanks to Kanata Motohashi, fixed the link to the *linsegr* opcode.

# nchnls

nchnls -- Sets the number of channels of audio output.

nchnls

## 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.

## Syntax

**nchnls** = iarg

## Initialization

*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.

## See Also

*kr, ksmpr, sr*

# nestedap

nestedap -- Three different nested all-pass filters.

nestedap

## Description

Three different nested all-pass filters, useful for implementing reverbs.

## Syntax

ares **nestedap** asig, imode, imaxdel, idel1, igain1 [, idel2] [, igain2] [, idel3

## 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.

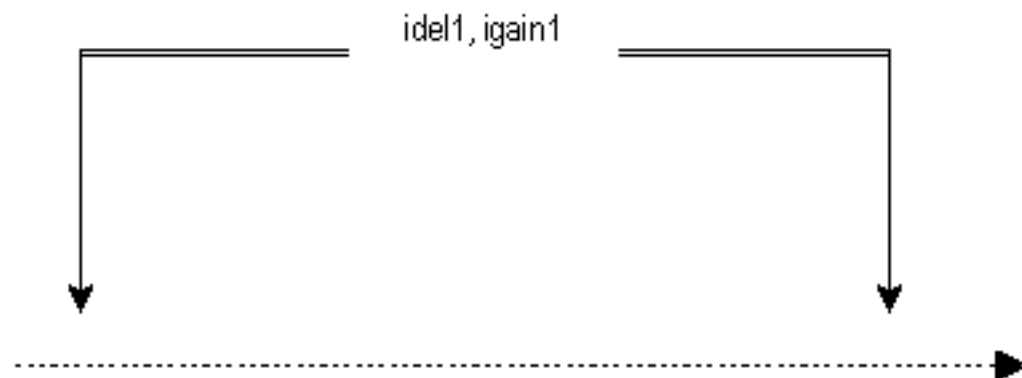
*imaxdel* -- will be necessary if k-rate delays are implemented. Not currently used.

*istor* -- Skip initialization if non-zero (default: 0).

## Performance

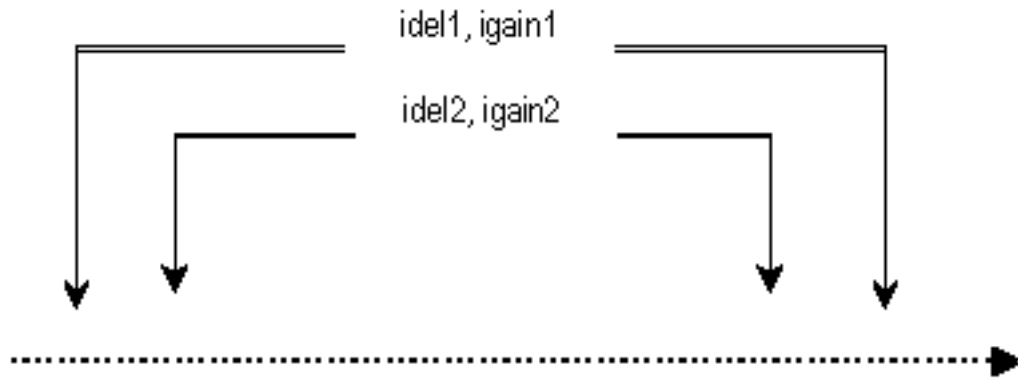
*asig* -- input signal

If *imode* = 1, the filter takes the form:



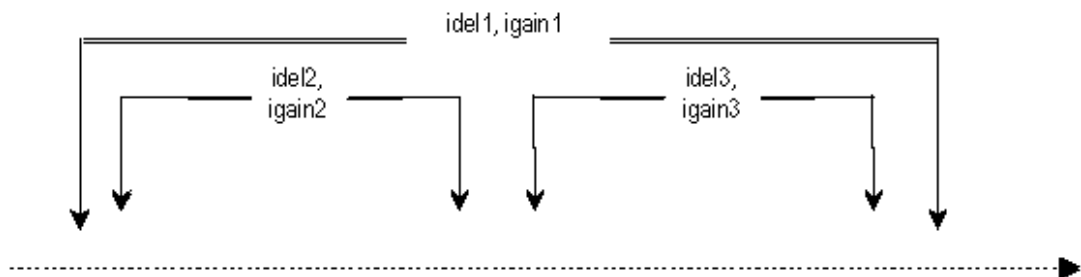
Picture of imode 1 filter.

If *imode* = 2, the filter takes the form:



Picture of `imode 2` filter.

If `imode = 3`, the filter takes the form:



Picture of `imode 3` filter.

## Examples

Here is an example of the `nestedap` opcode. It uses the files *nestedap.orc* [examples/nestedap.orc], *nestedap.sco* [examples/nestedap.sco], and *beats.wav* [examples/beats.wav].

### Example 236. Example of the `nestedap` opcode.

```
/* nestedap.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

instr 5
  insnd      =          p4
  gasig      diskln insnd, 1
endin

instr 10
  imax       =          1
  idel1      =          p4/1000
  igain1     =          p5
  idel2      =          p6/1000
  igain2     =          p7
  idel3      =          p8/1000
  igain3     =          p9
  idel4      =          p10/1000
  igain4     =          p11
  idel5      =          p12/1000
  igain5     =          p13
  idel6      =          p14/1000
  igain6     =          p15
```

```
afdbk      init 0

aout1      nestedap gasig+afdbk*.4, 3, imax, idel1, igain1, idel2, igain2, ide
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
/* nestedap.orc */

/* nestedap.sco */
f1 0 8192 10 1

; Diskin
;   Sta   Dur   Soundin
i5 0     3     "beats.wav"

; 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
/* nestedap.sco */
```

## Credits

Author: Hans Mikelson  
February 1999

New in Csound version 3.53

The example was updated May 2002, thanks to Hans Mikelson

# nlfilt

nlfilt -- A filter with a non-linear effect.

nlfilt

## 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)

## Syntax

ares **nlfilt** ain, ka, kb, kd, kC, kL

## Performance

1. Non-linear effect. The range of parameters are:

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.

2. Low Pass with non-linear. The range of parameters are:

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.

3. 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



4. 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.

## Credits

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

# noise

noise -- A white noise generator with an IIR lowpass filter.

noise

## Description

A white noise generator with an IIR lowpass filter.

## Syntax

ares **noise** xamp, kbeta

## 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

*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.

## Examples

Here is an example of the noise opcode. It uses the files *noise.orc* [examples/noise.orc] and *noise.sco* [examples/noise.sco].

### Example 237. Example of the noise opcode.

```
/* noise.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000

  ; Change the beta value linearly from 0 to 1.
  kbeta line 0, p3, 1

  a1 noise kamp, kbeta
  out a1
```

```
endin
/* noise.orc */

/* noise.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* noise.sco */
```

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
December 2000

Example written by Kevin Conder.

New in Csound version 4.10

# noteoff

noteoff -- Send a noteoff message to the MIDI OUT port.

noteoff

## Description

Send a noteoff message to the MIDI OUT port.

## Syntax

**noteoff** *ichn*, *inum*, *ivel*

## Initialization

*ichn* -- MIDI channel number (1-16)

*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* opcodes are useful only when introducing a *timeout* statement to play a non-zero duration MIDI note. For most purposes, it is better to use *noteondur* and *noteondur2*.

## See Also

*noteon*, *noteondur*, *noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# noteon

noteon -- Send a noteon message to the MIDI OUT port.

noteon

## Description

Send a noteon message to the MIDI OUT port.

## Syntax

**noteon** *ichn*, *inum*, *ivel*

## Initialization

*ichn* -- MIDI channel number (1-16)

*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* opcodes are useful only when introducing a *timeout* statement to play a non-zero duration MIDI note. For most purposes, it is better to use *noteondur* and *noteondur2*.

## See Also

*noteoff*, *noteondur*, *noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# noteondur

*noteondur* -- Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

*noteondur*

## Description

Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

## Syntax

**noteondur** *ichn*, *inum*, *ivel*, *idur*

## Initialization

*ichn* -- MIDI channel number (1-16)

*inum* -- note number (0-127)

*ivel* -- velocity (0-127)

*idur* -- how long, in seconds, this note should last.

## Performance

*noteondur* (i-rate note on with duration) sends 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* opcodes can appear in the same Csound instrument, allowing chords to be played by a single instrument.

## See Also

*noteoff*, *noteon*, *noteondur2*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# noteondur2

*noteondur2* -- Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

*noteondur2*

## Description

Sends a noteon and a noteoff MIDI message both with the same channel, number and velocity.

## Syntax

**noteondur2** *ichn*, *inum*, *ivel*, *idur*

## Initialization

*ichn* -- MIDI channel number (1-16)

*inum* -- note number (0-127)

*ivel* -- velocity (0-127)

*idur* -- how long, in seconds, this note should last.

## Performance

*noteondur2* (i-rate note on with duration) sends 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 *noteondur2* 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 *noteondur2* opcodes can appear in the same Csound instrument, allowing chords to be played by a single instrument.

## See Also

*noteoff*, *noteon*, *noteondur*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# notnum

notnum -- Get a note number from a MIDI event.

notnum

## Description

Get a note number from a MIDI event.

## Syntax

ival **notnum**

## Performance

Get the MIDI byte value (0 - 127) denoting the note number of the current event.

## Examples

Here is an example of the notnum opcode. It uses the files *notnum.orc* [examples/notnum.orc] and *notnum.sco* [examples/notnum.sco].

### Example 238. Example of the notnum opcode.

```
/* notnum.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il notnum

  print il
endin
/* notnum.orc */

/* notnum.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* notnum.sco */
```

## See Also

*aftouch*, *ampmidi*, *cpsmidi*, *cpsmidib*, *midictrl*, *octmidi*, *octmidib*, *pchbend*, *pchmidi*, *pchmidib*, *ve-*



*loc*

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# nreverb

nreverb -- A reverberator consisting of 6 parallel comb-lowpass filters.

nreverb

## Description

This is a reverberator consisting of 6 parallel comb-lowpass filters being fed into a series of 5 all-pass filters. *nreverb* replaces *reverb2* (version 3.48) and so both opcodes are identical.

## Syntax

ares **nreverb** asig, ktime, khdif [, iskip] [, inumCombs] [, ifnCombs] [, inumAlpas

## Initialization

*iskip* (optional, default=0) -- Skip initialization if present and non-zero.

*inumCombs* (optional) -- 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 the 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* (optional) -- same as *inumCombs/ifnCombs*, for allpass filter. New in Csound 4.09.

## Performance

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

Here is a simple example of the nreverb opcode. It uses the files *nreverb.orc* [examples/nreverb.orc] and *nreverb.sco* [examples/nreverb.sco].

### Example 239. Simple example of the nreverb opcode.

```
/* nreverb.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
instr 1
  a1 oscil 10000, 440, 1
  a2 nreverb a1, 2.5, .3
  out a1 + a2 * .2
endin
/* nreverb.orc */

/* nreverb.sco */
; Table 1: an ordinary sine wave.
f 1 0 32768 10 1

i 1 0.0 0.5
i 1 1.0 0.5
i 1 2.0 0.5
i 1 3.0 0.5
i 1 4.0 0.5
e
/* nreverb.sco */
```

Here is an example of the `nreverb` opcode using an `f`table for filter constants. It uses the files *nreverb\_f*table.orc [examples/nreverb\_ftable.orc], *nreverb\_f*table.sco [examples/nreverb\_ftable.sco], and *beats*.wav [examples/beats.wav].

**Example 240. An example of the `nreverb` opcode using an `f`table for filter constants.**

```
/* nreverb_ftable.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
  a1 soundin "beats.wav"
  a2 nreverb a1, 1.5, .75, 0, 8, 71, 4, 72
  out a1 + a2 * .4
endin
/* nreverb_ftable.orc */

/* nreverb_ftable.sco */
; 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

f72 0 16   -2  -556 -441 -341 -225  0.7  0.72  0.74  0.76

i1 0 3
e
/* nreverb_ftable.sco */
```

## Credits

Authors: Paris Smaragdis (*reverb2*)  
MIT, Cambridge  
1995

Author: Richard Karpen (*nreverb*)  
Seattle, Wash  
1998

## nrpn

**nrpn** -- Sends a Non-Registered Parameter Number to the MIDI OUT port.

**nrpn**

## Description

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

## Syntax

**nrpn** *kchan*, *kparmnum*, *kparmvalue*

## Performance

*kchan* -- MIDI channel (1-16)

*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)

## Credits

Author: Gabriel Maldonado  
Italy  
1998

New in Csound version 3.492

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# nsamp

nsamp -- Returns the number of samples loaded into a stored function table number.

nsamp

## Description

Returns the number of samples loaded into a stored function table number.

## Syntax

**nsamp**(*x*) (init-rate args only)

## Performance

Returns the number of samples loaded into stored function table number *x* by GEN01. This is useful when a sample is shorter than the power-of-two function table that holds it. New in Csound version 3.49.

As of Csound version 5.02, *ftlen* works with deferred-length function tables (see GEN01).

*nsamp* differs from *ftlen* in that *nsamp* gives the number of sample frames loaded, while *ftlen* gives the total number of samples. For example, with a stereo sound file of 10000 samples, *ftlen*() would return 19999 (i.e. a total of 20000 mono samples, not including a guard point), but *nsamp*() returns 10000.

## Examples

Here is an example of the *nsamp* opcode. It uses the files *nsamp.orc* [examples/nsamp.orc], *nsamp.sco* [examples/nsamp.sco], and *mary.wav* [examples/mary.wav].

### Example 241. Example of the nsamp opcode.

```
/* nsamp.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the size (in samples) of Table #1.
  isz = nsamp(1)
  print isz
endin
/* nsamp.orc */

/* nsamp.sco */
; Table #1: Use an audio file.
f 1 0 262144 1 "mary.wav" 0 0 0
```

```
; Play Instrument #1 for 1 second.  
i 1 0 1  
e  
/* nsamp.sco */
```

Since the audio file “mary.wav” has 154390 samples, its output should include a line like this:

```
instr 1:  isz = 154390.000
```

## See Also

*ftchnls, filen, filptim, ftsr*

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

Example written by Kevin Conder.

# nstrnum

nstrnum -- Returns the number of a named instrument.

nstrnum

## Description

Returns the number of a named instrument.

## Syntax

`insno nstrnum "name"`

## Initialization

*insno* -- the instrument number of the named instrument.

## Performance

*"name"* -- the named instrument's name.

If an instrument with the specified name does not exist, an init error occurs, and -1 is returned.

## Credits

Author: Istvan Varga  
New in version 4.23  
Written in the year 2002.



# ntrpol

ntrpol -- Calculates the weighted mean value of two input signals.

ntrpol

## Description

Calculates the weighted mean value (i.e. linear interpolation) of two input signals

## Syntax

ares **ntrpol** asig1, asig2, kpoint [, imin] [, imax]

ires **ntrpol** isig1, isig2, ipoint [, imin] [, imax]

kres **ntrpol** ksig1, ksig2, kpoint [, imin] [, imax]

## 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.

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

New in Csound version 3.49

# octave

octave -- Calculates a factor to raise/lower a frequency by a given amount of octaves.

octave

## Description

Calculates a factor to raise/lower a frequency by a given amount of octaves.

## Syntax

**octave** (*x*)

This function works at a-rate, i-rate, and k-rate.

## Initialization

*x* -- a value expressed in octaves.

## Performance

The value returned by the *octave* function is a factor. You can multiply a frequency by this factor to raise/lower it by the given amount of octaves.

## Examples

Here is an example of the octave opcode. It uses the files *octave.orc* [examples/octave.orc] and *octave.sco* [examples/octave.sco].

### Example 242. Example of the octave opcode.

```
/* octave.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; The root note is A above middle-C (440 Hz)
  iroot = 440

  ; Raise the root note by two octaves.
  ioctaves = 2

  ; Calculate the new note.
  ifactor = octave(ioctaves)
  inew = iroot * ifactor

  ; Print out of all of the values.
  print iroot
  print ifactor
  print inew
endin
/* octave.orc */
```

```
/* octave.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* octave.sco */
```

Its output should include lines like:

```
instr 1:  iroot = 440.000  
instr 1:  ifactor = 4.000  
instr 1:  inew = 1760.149
```

## See Also

*cent, db, semitone*

## Credits

Example written by Kevin Conder.

New in version 4.16

# octcps

octcps -- Converts a cycles-per-second value to octave-point-decimal.

octcps

## Description

Converts a cycles-per-second value to octave-point-decimal.

## Syntax

**octcps** (cps) (init- or control-rate args only)

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

## Performance

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:

**Table 3. Pitch and Frequency Values**

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

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

## Examples

Here is an example of the `octcps` opcode. It uses the files *octcps.orc* [examples/octcps.orc] and *octcps.sco* [examples/octcps.sco].

### Example 243. Example of the `octcps` opcode.

```
/* octcps.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Convert a cycles-per-second value into an
  ; octave value.
  icps = 440
  ioct = octcps(icps)

  print ioct
endin
/* octcps.orc */

/* octcps.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* octcps.sco */
```

Its output should include a line like this:

```
instr 1:  ioct = 8.750
```

## See Also

*cpsoct*, *cpspch*, *octpch*, *pchoct*

## Credits

Example written by Kevin Conder.

# octmidi

octmidi -- Get the note number, in octave-point-decimal units, of the current MIDI event.

octmidi

## Description

Get the note number, in octave-point-decimal units, of the current MIDI event.

## Syntax

ioct **octmidi**

## Performance

Get the note number of the current MIDI event, expressed in octave-point-decimal units, for local processing.

## Examples

Here is an example of the octmidi opcode. It uses the files *octmidi.orc* [examples/octmidi.orc] and *octmidi.sco* [examples/octmidi.sco].

### Example 244. Example of the octmidi opcode.

```
/* octmidi.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il octmidi

  print il
endin
/* octmidi.orc */

/* octmidi.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* octmidi.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# octmidib

octmidib -- Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in octave-point-decimal.

octmidib

## Description

Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in octave-point-decimal.

## Syntax

```
ioct octmidib [irange]
```

```
koct octmidib [irange]
```

## Initialization

*irange* (optional) -- the pitch bend range in semitones

## Performance

Get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in octave-point-decimal units. Available as an i-time value or as a continuous k-rate value.

## Examples

Here is an example of the octmidib opcode. It uses the files *octmidib.orc* [examples/octmidib.orc] and *octmidib.sco* [examples/octmidib.sco].

### Example 245. Example of the octmidib opcode.

```
/* octmidib.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il octmidib

  print il
endin
/* octmidib.orc */
```

```
/* octmidib.sco */
```



```
; Play Instrument #1 for 12 seconds.  
i 1 0 12  
e  
/* octmidib.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# octpch

octpch -- Converts a pitch-class value to octave-point-decimal.

octpch

## Description

Converts a pitch-class value to octave-point-decimal.

## Syntax

**octpch** (pch) (init- or control-rate args only)

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

## Performance

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:

**Table 4. Pitch and Frequency Values**

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

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

## Examples

Here is an example of the `octpch` opcode. It uses the files *octpch.orc* [examples/octpch.orc] and *octpch.sco* [examples/octpch.sco].

### Example 246. Example of the `octpch` opcode.

```
/* octpch.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Convert a pitch-class value into an
  ; octave-point-decimal value.
  ipch = 8.09
  ioct = octpch(ipch)

  print ioct
endin
/* octpch.orc */

/* octpch.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* octpch.sco */
```

Its output should include a line like this:

```
instr 1:  ioct = 8.750
```

## See Also

*cpsoct*, *cpspch*, *octcps*, *pchoct*

## Credits

Example written by Kevin Conder.

# opcode

opcode -- Defines the start of user-defined opcode block.

opcode

The *opcode* and *endop* statements allow defining a new opcode that can be used the same way as any of the built-in Csound opcodes. These opcode blocks are very similar to instruments (and are, in fact, implemented as special instruments), but cannot be called as a normal instrument e.g. with the *i* statements.

A user-defined opcode block must precede the instrument (or other opcode) from which it is used. But it is possible to call the opcode from itself. This allows recursion of any depth that is limited only by available memory. Additionally, there is an experimental feature that allows running the opcode definition at a higher control rate than the *kr* value specified in the orchestra header.

Similarly to instruments, the variables and labels of a user-defined opcode block are local and cannot be accessed from the caller instrument (and the opcode cannot access variables of the caller, either).

Some parameters are automatically copied at initialization, however:

- all p-fields (including *p1*)
- extra time (see also *xtratim*, *linsegr*, and related opcodes). This may affect the operation of *linsegr/expsegr/linenr/envlpxr* in the user-defined opcode block.
- MIDI parameters, if there are any.

Also, the release flag (see the *release* opcode) is copied at performance time.

Modifying the note duration in the opcode definition by assigning to *p3*, or using *ihold*, *turnoff*, *xtratim*, *linsegr*, or similar opcodes will also affect the caller instrument. Changes to MIDI controllers (for example with *ctrlinit*) will also apply to the instrument from which the opcode was called.

Use the *setksmps* opcode to set the local *ksmps* value.

The *xin* and *xout* opcodes copy variables to and from the opcode definition, allowing communication with the calling instrument.

The types of input and output variables are defined by the parameters *intypes* and *outtypes*.



## Notes

- *xin* and *xout* should be called only once, and *xin* should precede *xout*, otherwise an init error and deactivation of the current instrument may occur.
- These opcodes actually run only at i-time. Performance time copying is done by the user opcode call. This means that skipping *xin* or *xout* with *kgoto* has no effect, while skipping with *igoto* affects both init and performance time operation.

## Syntax

**opcode** name, outtypes, intypes

## Initialization

*name* -- name of the opcode. It may consist of any combination of letters, digits, and underscore but should not begin with a digit. If an opcode with the specified name already exists, it is redefined (a warning is printed in such cases). Some reserved words (like *instr* and *endin*) cannot be redefined.

*intypes* -- list of input types, any combination of the characters: a, k, K, i, o, p, and j. A single 0 character can be used if there are no input arguments. Double quotes and delimiter characters (e.g. comma) are *not* needed.

The meaning of the various *intypes* is shown in the following table:

Type	Description	Variable Types Allowed	Updated At
a	a-rate variable	a-rate	a-rate
i	i-rate variable	i-rate	i-time
j	optional i-time, defaults to -1	i-rate, constant	i-time
k	k-rate variable	k- and i-rate, constant	k-rate
K	k-rate with initialization	k- and i-rate, constant	i-time and k-rate
o	optional i-time, defaults to 0	i-rate, constant	i-time
p	optional i-time, defaults to 1	i-rate, constant	i-time

The maximum allowed number of input arguments is 256.

*outtypes* -- list of output types. The format is the same as in the case of *intypes*.

Here are the available *outtypes*:

Type	Description	Variable Types Allowed	Updated At
a	a-rate variable	a-rate	a-rate
i	i-rate variable	i-rate	i-time
k	k-rate variable	k-rate	k-rate
K	k-rate with initialization	k-rate	i-time and k-rate

The maximum allowed number of output arguments is 256.

*ksmps* (optional, default=0) -- sets the local *ksmps* value. Must be a positive integer, and also the *ksmps* of the calling instrument or opcode must be an integer multiple of this value. For example, if *ksmps* is 10 in the instrument from which the opcode was called, the allowed values for *ksmps* are 1, 2, 5, and 10.

If *ksmps* is set to zero, the *ksmps* of the caller instrument or opcode is used (this is the default behavior).



### Note

The local *ksmps* is implemented by splitting up a control period into smaller sub-periods and temporarily modifying internal Csound global variables. This also requires converting the rate of k-rate input and output arguments (input variables receive the same value in all sub-kperiods, while outputs are written only in the last one).



## Warning about local *ksmps*

When the local *ksmps* is not the same as the orchestra level *ksmps* value (as specified in the orchestra header), global a-rate operations must not be used in the user-defined opcode block.

These include:

- any access to “ga” variables
- a-rate zak opcodes (*zar*, *zaw*, etc.)
- *tablera* and *tablewa* (these two opcodes may in fact work, but caution is needed)
- The *in* and *out* opcode family (these read from, and write to global a-rate buffers)

In general, the local *ksmps* should be used with care as it is an experimental feature, although it works correctly in most cases.

The *setksmps* statement can be used to set the local *ksmps* value of the user-defined opcode block. It has one i-time parameter specifying the new *ksmps* value (which is left unchanged if zero is used, see also the notes about *iksmpls* above). *setksmps* should be used before any other opcodes (but allowed after *xin*), otherwise unpredictable results may occur.

The input parameters can be read with *xin*, and the output is written by *xout* opcode. Only one instance of these units should be used, as *xout* overwrites and does not accumulate the output. The number and type of arguments for *xin* and *xout* must be the same as in the declaration of the user-defined opcode block (see tables above).

The input and output arguments must agree with the definition both in number (except if the optional i-time input is used) and type. An optional i-time input parameter (*iksmpls*) is automatically added to the *intypes* list, and (similarly to *setksmps*) sets the local *ksmps* value.

## Performance

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmpls]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmpls]
```



## Note

The opcode call is always executed both at initialization and performance time, even if there are no a- or k-rate arguments. If there are many user opcode calls that are known to have no effect at performance time in an instrument, then it may save some CPU time to jump over groups of such opcodes with *kgoto*.

## Examples

Here is an example of a user-defined opcode. It uses the files *opcode\_example.orc* [examples/opcode\_example.orc] and *opcode\_example.sco* [examples/opcode\_example.sco].

### Example 247. Example of a user-defined opcode.

```
/* ---- opcode_example.orc ---- */
sr      = 44100
ksmps   = 50
nchnls  = 1

/* example opcode 1: simple oscillator */

opcode Oscillator, a, kk

kamp, kcps      xin      ; read input parameters
a1      vco2 kamp, kcps   ; sawtooth oscillator
xout a1          ; write output

endop

/* example opcode 2: lowpass filter with local ksmps */

opcode Lowpass, a, akk

setksmps 1          ; need sr=kr
ain, ka1, ka2      xin      ; read input parameters
aout      init 0      ; initialize output
aout      = ain*ka1 + aout*ka2 ; simple tone-like filter
xout aout          ; write output

endop

/* example opcode 3: recursive call */

opcode RecursiveLowpass, a, akkpp

ain, ka1, ka2, idep, icnt      xin      ; read input parameters
if (icnt >= idep) goto skip1    ; check if max depth reached
ain      RecursiveLowpass ain, ka1, ka2, idep, icnt + 1
skip1:
aout      Lowpass ain, ka1, ka2      ; call filter
xout aout          ; write output

endop

/* example opcode 4: de-click envelope */

opcode DeClick, a, a

ain      xin
aenv      linseg 0, 0.02, 1, p3 - 0.05, 1, 0.02, 0, 0.01, 0
xout ain * aenv          ; apply envelope and write output

endop

/* instr 1 uses the example opcodes */

instr 1

kamp      = 20000          ; amplitude
kcps      expon 50, p3, 500 ; pitch
a1      Oscillator kamp, kcps      ; call oscillator
kflt      linseg 0.4, 1.5, 0.4, 1, 0.8, 1.5, 0.8 ; filter envelope
a1      RecursiveLowpass a1, kflt, 1 - kflt, 10 ; 10th order lowpass
```

```
a1      DeClick a1
        out a1

        endin
/* ---- opcode_example.orc ---- */

/* ---- opcode_example.sco ---- */
i 1 0 4
e
/* ---- opcode_example.sco ---- */
```

## See Also

*endop, setksmps, xin, xout*

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22



# OSCsend

OSCsend -- Sends data to other processes using the OSC protocol

OSCsend

## Description

Uses the OSC protocol to send message to other OSC listening processes.

## Syntax

```
OSCsend kwhen, ihost, iport,  
        idestination, itype [, kdata1, kdata2, ...]
```

## Initialization

*ihost* -- a string that is the intended host computer domain name. An empty string is interpreted as the current computer.

*iport* -- the number of the port that is used for the communication.

*idest* -- a string that is the destination address. This takes the form of a file name with directories. Csound just passes this string to the raw sending code and makes no interpretation.

*itype* -- a string that indicates the types of the optional arguments that are read at k-rate. The string can contain the characters "bcdfilmst" which stand for Boolean, character, double, float, 32-bit integer, 64-bit integer, MIDI, string and timestamp.

## Performance

*kwhen* -- a message is sent whenever this value changes. A message will always be sent on the first call.

The data is taken from the k-values that follow the format string. In a similar way to a printf format, the characters in order determine how the argument is interpreted. Note that a time stamp takes two arguments.

## Example

The example shows a simple string of messages being sent just once to a computer called xenakis, on port 7770 to be read by a process that recognises /foo/bar as its address.

```
sr = 44100  
ksmps = 100  
nchnls = 2  
  
instr 1  
    OSCsend 1, "xenakis.cs.bath.ac.uk", 7770, "/foo/bar", "sis", "FOO"  
endin
```

## Credits

Author: John ffitch  
2005

# OSCinit

OSCinit -- Start a listening process for OSC messages to a particular port.

OSCinit

## Description

Starts a listening process, which can be used by OSClisten.

## Syntax

```
ihandle OSCinit iport
```

## Initialization

*ihandle* -- handle returned that can be passed to any number of OSClisten opcodes to receive messages on this port.

*iport* -- the port on which to listen.

## Performance

The listener runs in the background. See OSClisten for details.

## Example

The example shows a pair of floating point numbers being received on port 7770.

```
sr = 44100
ksmps = 100
nchnls = 2

gihandle OSCinit 7770

instr 1
  kf1 init 0
  kf2 init 0
nxtmsg:
  kk OSClisten gihandle, "/foo/bar", "ff", kf1, kf2
if (kk == 0) goto ex
  printk 0,kf1
  printk 0,kf2
  kgoto nxtmsg
ex:
  endin
```

## Credits

Author: John fitch  
2005

# OSClisten

OSClisten -- Listen for OSC messages to a particular path.

OSClisten

## Description

On each k-cycle looks to see if an OSC message has been send to a given path of a given type.

## Syntax

```
kans OSClisten ihandle, idest, itype [, xdata1, xdata2, ...]
```

## Initialization

*ihandle* -- a handle returned by an earlier call to OSCinit, to associate OSClisten with a particular port number.

*idest* -- a string that is the destination address. This takes the form of a file name with directories. Csound uses this address to decide if messages are meant for csound.

*itype* -- a string that indicates the types of the optional arguments that are to be read. The string can contain the characters "cdfhis" which stand for character, double, float, 64-bit integer, 32-bit integer, and string. All types other than 's' require a k-rate variable, while 's' requires a string variable.

A handler is inserted into the listener (see OSCinit) to intercept messages of this pattern.

## Performance

*kans* -- set to 1 if a new message was received, or zero if not. If multiple messages are received in a single control period, the messages are buffered, and OSClisten can be called again until zero is returned.

If there was a message the *xdata* variables are set to the incoming values, as interpreted by the *itype* parameter. Note that although the *xdata* variables are on the right of an operation they are actually outputs, and so must be variables of type k, gk, S, or gS, and may need to be declared with init, or = in the case of string variables, before calling OSClisten.

## Example

The example shows a pair of floating point numbers being received on port 7770.

```
sr = 44100
ksmps = 100
nchnls = 2

gihandle OSCinit 7770

instr 1
  kf1 init 0
  kf2 init 0
nxtmsg:
  kk OSClisten gihandle, "/foo/bar", "ff", kf1, kf2
  if (kk == 0) goto ex
  printk 0,kf1
  printk 0,kf2
```

```
      kgoto nxtmsg  
ex:     
      endin
```

## Credits

Author: John ffitch  
2005

# oscbnk

oscbnk -- Mixes the output of any number of oscillators.

oscbnk

## Description

This unit generator mixes the output of any number of oscillators. The frequency, phase, and amplitude of each oscillator can be modulated by two LFOs (all oscillators have a separate set of LFOs, with different phase and frequency); additionally, the output of each oscillator can be filtered through an optional parametric equalizer (also controlled by the LFOs). This opcode is most useful for rendering ensemble (strings, choir, etc.) instruments.

Although the LFOs run at k-rate, amplitude, phase and filter modulation are interpolated internally, so it is possible (and recommended in most cases) to use this unit at low (~1000 Hz) control rates without audible quality degradation.

The start phase and frequency of all oscillators and LFOs can be set by a built-in seedable 31-bit random number generator, or specified manually in a function table (GEN2).

## Syntax

ares **oscbnk** kcps, kamd, kfmd, kpmd, iovrlap, iseed, kllminf, kllmaxf, kl2minf,

## Initialization

*iovrlap* -- Number of oscillator units.

*iseed* -- Seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). *iseed* <= seeds 0 from the current time.

*ieqmode* -- Parametric equalizer mode

- -1: disable EQ (faster)
- 0: peak
- 1: low shelf
- 2: high shelf
- 3: peak (filter interpolation disabled)
- 4: low shelf (interpolation disabled)
- 5: high shelf (interpolation disabled)

The non-interpolated modes are faster, and in some cases (e.g. high shelf filter at low cutoff frequencies) also more stable; however, interpolation is useful for avoiding “zipper noise” at low control rates.

*ilfomode* -- LFO modulation mode, sum of:

- 128: LFO1 to frequency
- 64: LFO1 to amplitude

- 32: LFO1 to phase
- 16: LFO1 to EQ
- 8: LFO2 to frequency
- 4: LFO2 to amplitude
- 2: LFO2 to phase
- 1: LFO2 to EQ

If an LFO does not modulate anything, it is not calculated, and the ftable number (il1fn or il2fn) can be omitted.

*il1fn* (optional: default=0) -- LFO1 function table number. The waveform in this table has to be normalized (absolute value  $\leq 1$ ), and is read with linear interpolation.

*il2fn* (optional: default=0) -- LFO2 function table number. The waveform in this table has to be normalized, and is read with linear interpolation.

*ieqffn*, *ieqlfn*, *ieqqfn* (optional: default=0) -- Lookup tables for EQ frequency, level, and Q (optional if EQ is disabled). Table read position is 0 if the modulator signal is less than, or equal to -1, (table length / 2) if the modulator signal is zero, and the guard point if the modulator signal is greater than, or equal to 1. These tables have to be normalized to the range 0 - 1, and have an extended guard point (table length = power of two + 1). All tables are read with linear interpolation.

*itabl* (optional: default=0) -- Function table storing phase and frequency values for all oscillators (optional). The values in this table are in the following order (5 for each oscillator unit):

oscillator phase, lfo1 phase, lfo1 frequency, lfo2 phase, lfo2 frequency, ...

All values are in the range 0 to 1; if the specified number is greater than 1, it is wrapped (phase) or limited (frequency) to the allowed range. A negative value (or end of table) will use the output of the random number generator. The random seed is always updated (even if no random number was used), so switching one value between random and fixed will not change others.

*ioutfn* (optional: default=0) -- Function table to write phase and frequency values (optional). The format is the same as in the case of *itabl*. This table is useful when experimenting with random numbers to record the best values.

The two optional tables (*itabl* and *ioutfn*) are accessed only at i-time. This is useful to know, as the tables can be safely overwritten after opcode initialization, which allows precalculating parameters at i-time and storing in a temporary table before oscbnk initialization.

## Performance

*ares* -- Output signal.

*kcps* -- Oscillator frequency in Hz.

*kamd* -- AM depth (0 - 1).

(AM output) = (AM input) \* ((1 - (AM depth)) + (AM depth) \* (modulator))

If *ilfomode* isn't set to modulate the amplitude, then (AM output) = (AM input) regardless of the value of *kamd*. That means that *kamd* will have no effect.

Note: Amplitude modulation is applied before the parametric equalizer.

*kfmd* -- FM depth (in Hz).

*kpmd* -- Phase modulation depth.

*kl1minf*, *kl1maxf* -- LFO1 minimum and maximum frequency in Hz.

*kl2minf*, *kl2maxf* -- LFO2 minimum and maximum frequency in Hz. (Note: oscillator and LFO frequencies are allowed to be zero or negative.)

*keqminf*, *keqmaxf* -- Parametric equalizer minimum and maximum frequency in Hz.

*keqminl*, *keqmaxl* -- Parametric equalizer minimum and maximum level.

*keqminq*, *keqmaxq* -- Parametric equalizer minimum and maximum Q.

*kfn* -- Oscillator waveform table. Table number can be changed at k-rate (this is useful to select from a set of band-limited tables generated by GEN30, to avoid aliasing). The table is read with linear interpolation.



### Note

*oscblk* uses the same random number generator as *rnd31*. So reading *its documentation* is also recommended.

## Examples

Here is an example of *oscblk* opcode. It uses the files *oscblk.orc* [examples/oscblk.orc] and *oscblk.sco* [examples/oscblk.sco].

### Example 248. Example of the *oscblk* opcode.

```
/* oscblk.orc */
/* Written by Istvan Varga */
sr = 48000
kr = 750
ksmps = 64
nchnls = 2

ga01 init 0
ga02 init 0

/* sawtooth wave */
i_ ftgen 1, 0, 16384, 7, 1, 16384, -1
/* FM waveform */
i_ ftgen 3, 0, 4096, 7, 0, 512, 0.25, 512, 1, 512, 0.25, 512, \
    0, 512, -0.25, 512, -1, 512, -0.25, 512, 0
/* AM waveform */
i_ ftgen 4, 0, 4096, 5, 1, 4096, 0.01
/* FM to EQ */
i_ ftgen 5, 0, 1024, 5, 1, 512, 32, 512, 1
/* sine wave */
i_ ftgen 6, 0, 1024, 10, 1
/* room parameters */
i_ ftgen 7, 0, 64, -2, 4, 50, -1, -1, -1, 11, \
    1, 26.833, 0.05, 0.85, 10000, 0.8, 0.5, 2, \
    1, 1.753, 0.05, 0.85, 5000, 0.8, 0.5, 2, \
    1, 39.451, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 33.503, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
    1, 36.151, 0.05, 0.85, 7000, 0.8, 0.5, 2, \
```



```
1, 29.633, 0.05, 0.85, 7000, 0.8, 0.5, 2

/* generate bandlimited sawtooth waves */

i0 = 0
loop1:
imaxh = sr / (2 * 440.0 * exp (log(2.0) * (i0 - 69) / 12))
i_ ftgen i0 + 256, 0, 4096, -30, 1, 1, imaxh
i0 = i0 + 1
    if (i0 < 127.5) igoto loop1

instr 1

p3 = p3 + 0.4

; note frequency
kcps = 440.0 * exp (log(2.0) * (p4 - 69) / 12)
; lowpass max. frequency
klpmaxf limit 64 * kcps, 1000.0, 12000.0
; FM depth in Hz
kfmd1 = 0.02 * kcps
; AM frequency
kamfr = kcps * 0.02
kamfr2 = kcps * 0.1
; table number
kfnum = (256 + 69 + 0.5 + 12 * log(kcps / 440.0) / log(2.0))
; amp. envelope
aenv linseg 0, 0.1, 1.0, p3 - 0.5, 1.0, 0.1, 0.5, 0.2, 0, 1.0, 0

/* oscillator / left */

a1 oscbnk kcps, 0.0, kfmd1, 0.0, 40, 200, 0.1, 0.2, 0, 0, 144, \
    0.0, klpmaxf, 0.0, 0.0, 1.5, 1.5, 2, \
    kfnum, 3, 0, 5, 5, 5
a2 oscbnk kcps, 1.0, kfmd1, 0.0, 40, 201, 0.1, 0.2, kamfr, kamfr2, 148, \
    0, 0, 0, 0, 0, 0, -1, \
    kfnum, 3, 4
a2 pareq a2, kcps * 8, 0.0, 0.7071, 2
a0 = a1 + a2 * 0.12
/* delay */
adel = 0.001
a01 vdelayx a0, adel, 0.01, 16
a_ oscili 1.0, 0.25, 6, 0.0
adel = adel + 1.0 / (exp(log(2.0) * a_) * 8000)
a02 vdelayx a0, adel, 0.01, 16
a0 = a01 + a02

ga01 = ga01 + a0 * aenv * 2500

/* oscillator / right */

; lowpass max. frequency

a1 oscbnk kcps, 0.0, kfmd1, 0.0, 40, 202, 0.1, 0.2, 0, 0, 144, \
    0.0, klpmaxf, 0.0, 0.0, 1.0, 1.0, 2, \
    kfnum, 3, 0, 5, 5, 5
a2 oscbnk kcps, 1.0, kfmd1, 0.0, 40, 203, 0.1, 0.2, kamfr, kamfr2, 148, \
    0, 0, 0, 0, 0, 0, -1, \
    kfnum, 3, 4
a2 pareq a2, kcps * 8, 0.0, 0.7071, 2
a0 = a1 + a2 * 0.12
/* delay */
adel = 0.001
a01 vdelayx a0, adel, 0.01, 16
a_ oscili 1.0, 0.25, 6, 0.25
adel = adel + 1.0 / (exp(log(2.0) * a_) * 8000)
a02 vdelayx a0, adel, 0.01, 16
a0 = a01 + a02
```

---

```
ga02 = ga02 + a0 * aenv * 2500

        endin

/* output / left */

        instr 81

i1 = 0.000001
aLl, aLh, aRl, aRh spat3di ga01 + i1*i1*i1*i1, -8.0, 4.0, 0.0, 0.3, 7, 4
ga01 = 0
aLl butterlp aLl, 800.0
aRl butterlp aRl, 800.0

        outs aLl + aLh, aRl + aRh

        endin

/* output / right */

        instr 82

i1 = 0.000001
aLl, aLh, aRl, aRh spat3di ga02 + i1*i1*i1*i1, 8.0, 4.0, 0.0, 0.3, 7, 4
ga02 = 0
aLl butterlp aLl, 800.0
aRl butterlp aRl, 800.0

        outs aLl + aLh, aRl + aRh

        endin
/* oscbnk.orc */


/* oscbnk.sco */
/* Written by Istvan Varga */
t 0 60

i 1 0 4 41
i 1 0 4 60
i 1 0 4 65
i 1 0 4 69

i 81 0 5.5
i 82 0 5.5
e
/* oscbnk.sco */
```

## Credits

Author: Istvan Varga  
2001

New in version 4.15

Updated April 2002 by Istvan Varga

# oscil

oscil -- A simple oscillator.

oscil

## Description

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

## Syntax

ares **oscil** xamp, xcps, ifn [, iphs]

kres **oscil** kamp, kcps, ifn [, iphs]

## Initialization

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

*iphs* (optional, default=0) -- 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

*kamp*, *xamp* -- amplitude

*kcps*, *xcps* -- frequency in cycles per second.

The *oscil* opcode generates 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.

## Examples

Here is an example of the *oscil* opcode. It uses the files *oscil.orc* [examples/oscil.orc] and *oscil.sco* [examples/oscil.sco].

### Example 249. Example of the *oscil* opcode.

```
/* oscil.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 440
  ifn = 1

  a1 oscil kamp, kcps, ifn
```

```
    out a1
  endin
/* oscil.orc */

/* oscil.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* oscil.sco */
```

## See Also

*oscili*, *oscil3*

## Credits

Example written by Kevin Conder.

# oscil1

oscil1 -- Accesses table values by incremental sampling.

oscil1

## Description

Accesses table values by incremental sampling.

## Syntax

kres **oscil1** idel, kamp, idur, ifn

## Initialization

*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.

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

## Performance

*kamp* -- amplitude factor.

*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.

## See Also

*table*, *tablei*, *table3*, *oscil1i*, *osciln*

# oscil1i

oscil1i -- Accesses table values by incremental sampling with linear interpolation.

oscil1i

## Description

Accesses table values by incremental sampling with linear interpolation.

## Syntax

kres **oscil1i** idel, kamp, idur, ifn

## Initialization

*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.

*ifn* -- function table number. *oscil1i* requires the extended guard point.

## Performance

*kamp* -- amplitude factor

*oscil1i* is an interpolating unit 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.

## See Also

*table*, *tablei*, *table3*, *oscil1*, *osciln*

# oscil3

oscil3 -- A simple oscillator with cubic interpolation.

oscil3

## Description

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

## Syntax

ares **oscil3** xamp, xcps, ifn [, iphs]

kres **oscil3** kamp, kcps, ifn [, iphs]

## 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

*kamp*, *xamp* -- amplitude

*kcps*, *xcps* -- frequency in cycles per second.

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

## Examples

Here is an example of the oscil3 opcode. It uses the files *oscil3.orc* [examples/oscil3.orc] and *oscil3.sco* [examples/oscil3.sco].

### Example 250. Example of the oscil3 opcode.

```
/* oscil3.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 220
  ifn = 1

  a1 oscil kamp, kcps, ifn
  out a1
```

```
endin

; Instrument #2 - the basic oscillator with cubic interpolation.
instr 2
  kamp = 10000
  kcps = 220
  ifn = 1

  a1 oscil3 kamp, kcps, ifn
  out a1
endin
/* oscil3.orc */


/* oscil3.sco */
; Table #1, a sine wave table with a small amount of data.
f 1 0 32 10 0 1

; Play Instrument #1, the basic oscillator, for
; two seconds. This should sound relatively rough.
i 1 0 2

; Play Instrument #2, the cubic interpolated oscillator, for
; two seconds. This should sound relatively smooth.
i 2 2 2
e
/* oscil3.sco */
```

## See Also

*oscil, oscili*

## Credits

Author: John ffitch

Example written by Kevin Conder.



# oscili

oscili -- A simple oscillator with linear interpolation.

oscili

## Description

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

## Syntax

```
ares oscili xamp, xcps, ifn [, iphs]
```

```
kres oscili kamp, kcps, ifn [, iphs]
```

## 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

*kamp*, *xamp* -- amplitude

*kcps*, *xcps* -- frequency in cycles per second.

*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.

## Examples

Here is an example of the *oscili* opcode. It uses the files *oscili.orc* [examples/oscili.orc] and *oscili.sco* [examples/oscili.sco].

### Example 251. Example of the *oscili* opcode.

```
/* oscili.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 220
  ifn = 1
```

```
    al oscil kamp, kcps, ifn
    out al
endin

; Instrument #2 - the basic oscillator with extra interpolation.
instr 2
    kamp = 10000
    kcps = 220
    ifn = 1

    al oscili kamp, kcps, ifn
    out al
endin
/* oscili.orc */


/* oscili.sco */
; Table #1, a sine wave table with a small amount of data.
f 1 0 32 10 0 1

; Play Instrument #1, the basic oscillator, for
; two seconds. This should sound relatively rough.
i 1 0 2

; Play Instrument #2, the interpolated oscillator, for
; two seconds. This should sound relatively smooth.
i 2 2 2
e
/* oscili.sco */
```

## See Also

*oscil*, *oscil3*

## Credits

Example written by Kevin Conder.

# oscilikt

oscilikt -- A linearly interpolated oscillator that allows changing the table number at k-rate.

oscilikt

## Description

*oscilikt* is very similar to *oscili*, but allows changing the table number at k-rate. It is slightly slower than *oscili* (especially with high control rate), although also more accurate as it uses a 31-bit phase accumulator, as opposed to the 24-bit one used by *oscili*.

## Syntax

ares **oscilikt** xamp, xcps, kfn [, iphs] [, istor]

kres **oscilikt** kamp, kcps, kfn [, iphs] [, istor]

## Initialization

*iphs* (optional, defaults to 0) -- initial phase in the range 0 to 1. Other values are wrapped to the allowed range.

*istor* (optional, defaults to 0) -- skip initialization.

## Performance

*kamp*, *xamp* -- amplitude.

*kcps*, *xcps* -- frequency in Hz. Zero and negative values are allowed. However, the absolute value must be less than *sr* (and recommended to be less than *sr/2*).

*kfn* -- function table number. Can be varied at control rate (useful to “morph” waveforms, or select from a set of band-limited tables generated by *GEN30*).

## Examples

Here is an example of the *oscilikt* opcode. It uses the files *oscilikt.orc* [examples/oscilikt.orc] and *oscilikt.sco* [examples/oscilikt.sco].

### Example 252. Example of the *oscilikt* opcode.

```
/* oscilikt.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Generate a uni-polar (0-1) square wave.
kamp1 init 1
kcps1 init 2
itype = 3
```

```
ksquare lfo kamp1, kcps1, itype

; Use the square wave to switch between Tables #1 and #2.
kamp2 init 20000
kcps2 init 220
kfn = ksquare + 1

a1 oscilikt kamp2, kcps2, kfn
out a1
endin
/* oscilikt.orc */


/* oscilikt.sco */
; Table #1, a sine waveform.
f 1 0 4096 10 0 1
; Table #2: a sawtooth wave
f 2 0 3 -2 1 0 -1

; Play Instrument #1 for two seconds.
i 1 0 2
/* oscilikt.sco */
```

## See Also

*osciliktp* and *oscilikts*.

## Credits

Author: Istvan Varga

Example written by Kevin Conder.

New in version 4.22

# osciliktp

osciliktp -- A linearly interpolated oscillator that allows allows phase modulation.

osciliktp

## Description

*osciliktp* allows phase modulation (which is actually implemented as k-rate frequency modulation, by differentiating phase input). The disadvantage is that there is no amplitude control, and frequency can be varied only at the control-rate. This opcode can be faster or slower than *oscilikt*, depending on the control-rate.

## Syntax

```
ares osciliktp kcps, kfn, kphs [, istor]
```

## Initialization

*istor* (optional, defaults to 0) -- Skips initialization.

## Performance

*ares* -- audio-rate output signal.

*kcps* -- frequency in Hz. Zero and negative values are allowed. However, the absolute value must be less than *sr* (and recommended to be less than *sr/2*).

*kfn* -- function table number. Can be varied at control rate (useful to “morph” waveforms, or select from a set of band-limited tables generated by *GEN30*).

*kphs* -- phase (k-rate), the expected range is 0 to 1. The absolute value of the difference of the current and previous value of *kphs* must be less than *ksmps*.

## Examples

Here is an example of the *osciliktp* opcode. It uses the files *osciliktp.orc* [examples/osciliktp.orc] and *osciliktp.sco* [examples/osciliktp.sco].

### Example 253. Example of the osciliktp opcode.

```
/* osciliktp.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1: osciliktp example
instr 1
  kphs line 0, p3, 4

  alx osciliktp 220.5, 1, 0
  aly osciliktp 220.5, 1, -kphs
  a1 = alx - aly
```

```
    out a1 * 14000
endin
/* osciliktp.orc */

/* osciliktp.sco */
; Table #1: Sawtooth wave
f 1 0 3 -2 1 0 -1

; Play Instrument #1 for four seconds.
i 1 0 4
e
/* osciliktp.sco */
```

## See Also

*oscilikt* and *oscilikts*.

## Credits

Author: Istvan Varga

New in version 4.22

# oscilikts

*oscilikts* -- A linearly interpolated oscillator with sync status that allows changing the table number at k-rate.

*oscilikts*

## Description

*oscilikts* is the same as *oscilikt*. Except it has a sync input that can be used to re-initialize the oscillator to a k-rate phase value. It is slower than *oscilikt* and *osciliktp*.

## Syntax

ares **oscilikts** xamp, xcps, kfn, async, kphs [, istor]

## Initialization

*istor* (optional, defaults to 0) -- skip initialization.

## Performance

*xamp* -- amplitude.

*xcps* -- frequency in Hz. Zero and negative values are allowed. However, the absolute value must be less than *sr* (and recommended to be less than  $sr/2$ ).

*kfn* -- function table number. Can be varied at control rate (useful to “morph” waveforms, or select from a set of band-limited tables generated by *GEN30*).

*async* -- any positive value resets the phase of *oscilikts* to *kphs*. Zero or negative values have no effect.

*kphs* -- sets the phase, initially and when it is re-initialized with *async*.

## Examples

Here is an example of the *oscilikts* opcode. It uses the files *oscilikts.orc* [examples/oscilikts.orc] and *oscilikts.sco* [examples/oscilikts.sco].

### Example 254. Example of the *oscilikts* opcode.

```
/* oscilikts.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1: oscilikts example.
instr 1
; Frequency envelope.
kfrq expon 400, p3, 1200
; Phase.
kphs line 0.1, p3, 0.9
```

```
; Sync 1
atmp1 phasor 100
; Sync 2
atmp2 phasor 150
async diff 1 - (atmp1 + atmp2)

a1 oscilikts 14000, kfrq, 1, async, 0
a2 oscilikts 14000, kfrq, 1, async, -kphs

out a1 - a2
endin
/* oscilikts.orc */

/* oscilikts.sco */
; Table #1: Sawtooth wave
f 1 0 3 -2 1 0 -1

; Play Instrument #1 for four seconds.
i 1 0 4
e
/* oscilikts.sco */
```

## See Also

*oscilikt* and *osciliktp*.

## Credits

Author: Istvan Varga

New in version 4.22



# osciln

osciln -- Accesses table values at a user-defined frequency.

osciln

## Description

Accesses table values at a user-defined frequency. This opcode can also be written as *oscilx*.

## Syntax

ares **osciln** kamp, ifrq, ifn, itimes

## Initialization

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

*ifn* -- function table number.

## Performance

*kamp* -- amplitude factor

*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*.

## See Also

*table, tablei, table3, oscil1, oscilli*

# oscils

oscils -- A simple, fast sine oscillator

oscils

## Description

Simple, fast sine oscillator, that uses only one multiply, and two add operations to generate one sample of output, and does not require a function table.

## Syntax

ares **oscils** iamp, icps, iphs [, iflg]

## Initialization

*iamp* -- output amplitude.

*icps* -- frequency in Hz (may be zero or negative, however the absolute value must be less than  $sr/2$ ).

*iphs* -- start phase between 0 and 1.

*iflg* -- sum of the following values:

- 2: use double precision even if Csound was compiled to use floats. This improves quality (especially in the case of long performance time), but may be up to twice as slow.
- 1: skip initialization.

## Performance

*ares* -- audio output

## Examples

Here is an example of the oscils opcode. It uses the files *oscils.orc* [examples/oscils.orc] and *oscils.sco* [examples/oscils.sco].

### Example 255. Example of the oscils opcode.

```
/* oscils.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a fast sine oscillator.
instr 1
  iamp = 10000
  icps = 440
  iphs = 0
```

```
    al oscils iamp, icps, iphs  
    out al  
endin  
/* oscils.orc */
```

```
/* oscils.sco */  
; Play Instrument #1 for 2 seconds.  
i 1 0 2  
e  
/* oscils.sco */
```

## Credits

Author: Istvan Varga  
January 2002

Example written by Kevin Conder.

New in version 4.18

## oscilx

oscilx -- Same as the osciln opcode.

oscilx

## Description

Same as the *osciln* opcode.

## Syntax

ares **oscilx** kamp, ifrq, ifn, itimes

# out

out -- Writes mono audio data to an external device or stream.

out

## Description

Writes mono audio data to an external device or stream.

## Syntax

`out asig`

## Performance

Sends mono 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, Csound will attempt to change an incorrect opcode to agree with *nchnls* statement.

## See Also

*outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# out32

out32 -- Writes 32-channel audio data to an external device or stream.

out32

## Description

Writes 32-channel audio data to an external device or stream.

## Syntax

**out32** asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, asig10, asig11, a

## Performance

*out32* outputs 32 channels of audio.

## Credits

*outc, outch, outx, outz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# outc

`outc` -- Writes audio data with an arbitrary number of channels to an external device or stream.

`outc`

## Description

Writes audio data with an arbitrary number of channels to an external device or stream.

## Syntax

```
outc asig1 [, asig2] [...]
```

## Performance

*outc* outputs as many channels as provided. Any channels greater than *nchnls* are ignored. Zeros are added as necessary

## Credits

*out32, outch, outx, outz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# outch

`outch` -- Writes multi-channel audio data, with user-controllable channels, to an external device or stream.

`outch`

## Description

Writes multi-channel audio data, with user-controllable channels, to an external device or stream.

## Syntax

**outch** *ksig1*, *asig1* [, *ksig2*] [, *asig2*] [...]

## Performance

*outch* outputs *asig1* on the channel determined by *ksig1*, *asig2* on the channel determined by *ksig2*, etc.

## Credits

*out32*, *outc*, *outx*, *outz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07



# outh

outh -- Writes 6-channel audio data to an external device or stream.

outh

## Description

Writes 6-channel audio data to an external device or stream.

## Syntax

**outh** asig1, asig2, asig3, asig4, asig5, asig6

## Performance

Sends 6-channel 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, Csound will attempt to change an incorrect opcode to agree with *nchnls* statement.

## See Also

*out, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: John fitch

# outiat

outiat -- Sends MIDI aftertouch messages at i-rate.

outiat

## Description

Sends MIDI aftertouch messages at i-rate.

## Syntax

**outiat** ichn, ivalue, imin, imax

## Initialization

*ichn* -- MIDI channel number (1-16)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outiat* (i-rate aftertouch output) sends aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outic

outic -- Sends MIDI controller output at i-rate.

outic

## Description

Sends MIDI controller output at i-rate.

## Syntax

**outic** ichn, inum, ivalue, imin, imax

## Initialization

*ichn* -- MIDI channel number (1-16)

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

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outic* (i-rate MIDI controller output) sends controller messages to the MIDI OUT device. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outic14

outic14 -- Sends 14-bit MIDI controller output at i-rate.

outic14

## Description

Sends 14-bit MIDI controller output at i-rate.

## Syntax

**outic14** *ichn*, *imsb*, *ilsb*, *ivalue*, *imin*, *imax*

## Initialization

*ichn* -- MIDI channel number (1-16)

*imsb* -- most significant byte controller number when using 14-bit parameters (0-127)

*ilsb* -- least significant byte controller number when using 14-bit parameters (0-127)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 16383 (14-bit))

## Performance

*outic14* (i-rate MIDI 14-bit controller output) sends a pair of controller messages. This opcode can drive 14-bit parameters on MIDI instruments that recognize them. The first control message contains the most significant byte of *ivalue* argument while the second message contains the less significant byte. *imsb* and *ilsb* are the number of the most and less significant controller.

This opcode can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outipat

outipat -- Sends polyphonic MIDI aftertouch messages at i-rate.

outipat

## Description

Sends polyphonic MIDI aftertouch messages at i-rate.

## Syntax

**outipat** *ichn*, *inotenum*, *ivalue*, *imin*, *imax*

## Initialization

*ichn* -- MIDI channel number (1-16)

*inotenum* -- MIDI note number (used in polyphonic aftertouch messages)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outipat* (i-rate polyphonic aftertouch output) sends polyphonic aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outipb

outipb -- Sends MIDI pitch-bend messages at i-rate.

outipb

## Description

Sends MIDI pitch-bend messages at i-rate.

## Syntax

**outipb** ichn, ivalue, imin, imax

## Initialization

*ichn* -- MIDI channel number (1-16)

*ivalue* -- floating point value

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outipb* (i-rate pitch bend output) sends pitch bend messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outipc

outipc -- Sends MIDI program change messages at i-rate

outipc

## Description

Sends MIDI program change messages at i-rate

## Syntax

**outipc** *ichn*, *ipro*, *imin*, *imax*

## Initialization

*ichn* -- MIDI channel number (1-16)

*ipro* -- program change number in floating point

*imin* -- minimum floating point value (converted in MIDI integer value 0)

*imax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

## Performance

*outipc* (i-rate program change output) sends program change messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale an i-value floating-point argument according to the *imin* and *imax* values. For example, set *imin* = 1.0 and *imax* = 2.0. When the *ivalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *ivalue* argument receives a 1.0 value, it will send a 0 value. i-rate opcodes send their message once during instrument initialization.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkat

outkat -- Sends MIDI aftertouch messages at k-rate.

outkat

## Description

Sends MIDI aftertouch messages at k-rate.

## Syntax

**outkat** *kchn*, *kvalue*, *kmin*, *kmax*

## Performance

*kchn* -- MIDI channel number (1-16)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127)

*outkat* (k-rate aftertouch output) sends aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.



# outkc

outkc -- Sends MIDI controller messages at k-rate.

outkc

## Description

Sends MIDI controller messages at k-rate.

## Syntax

**outkc** kchn, knum, kvalue, kmin, kmax

## Performance

*kchn* -- MIDI channel number (1-16)

*knun* -- controller number (0-127 for example 1 = ModWheel; 2 = BreathControl etc.)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkc* (k-rate MIDI controller output) sends controller messages to MIDI OUT device. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkc14

outkc14 -- Sends 14-bit MIDI controller output at k-rate.

outkc14

## Description

Sends 14-bit MIDI controller output at k-rate.

## Syntax

**outkc14** *kchn*, *kmsb*, *klsb*, *kvalue*, *kmin*, *kmax*

## Performance

*kchn* -- MIDI channel number (1-16)

*kmsb* -- most significant byte controller number when using 14-bit parameters (0-127)

*klsb* -- least significant byte controller number when using 14-bit parameters (0-127)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 16383 (14-bit))

*outkc14* (k-rate MIDI 14-bit controller output) sends a pair of controller messages. It works only with MIDI instruments which recognize them. These opcodes can drive 14-bit parameters on MIDI instruments that recognize them. The first control message contains the most significant byte of *kvalue* argument while the second message contains the less significant byte. *kmsb* and *klsb* are the number of the most and less significant controller.

It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkpat

outkpat -- Sends polyphonic MIDI aftertouch messages at k-rate.

outkpat

## Description

Sends polyphonic MIDI aftertouch messages at k-rate.

## Syntax

**outkpat** *kchn*, *knotenum*, *kvalue*, *kmin*, *kmax*

## Performance

*kchn* -- MIDI channel number (1-16)

*knotenum* -- MIDI note number (used in polyphonic aftertouch messages)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkpat* (k-rate polyphonic aftertouch output) sends polyphonic aftertouch messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkpb

outkpb -- Sends MIDI pitch-bend messages at k-rate.

outkpb

## Description

Sends MIDI pitch-bend messages at k-rate.

## Syntax

**outkpb** *kchn*, *kvalue*, *kmin*, *kmax*

## Performance

*kchn* -- MIDI channel number (1-16)

*kvalue* -- floating point value

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkpb* (k-rate pitch-bend output) sends pitch-bend messages. It works only with MIDI instruments which recognize them. It can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outkpc

outkpc -- Sends MIDI program change messages at k-rate.

outkpc

## Description

Sends MIDI program change messages at k-rate.

## Syntax

**outkpc** *kchn*, *kprog*, *kmin*, *kmax*

## Performance

*kchn* -- MIDI channel number (1-16)

*kprog* -- program change number in floating point

*kmin* -- minimum floating point value (converted in MIDI integer value 0)

*kmax* -- maximum floating point value (converted in MIDI integer value 127 (7 bit))

*outkpc* (k-rate program change output) sends program change messages. It works only with MIDI instruments which recognize them. These opcodes can drive a different value of a parameter for each note currently active.

It can scale the k-value floating-point argument according to the *kmin* and *kmax* values. For example: set *kmin* = 1.0 and *kmax* = 2.0. When the *kvalue* argument receives a 2.0 value, the opcode will send a 127 value to the MIDI OUT device. When the *kvalue* argument receives a 1.0 value, it will send a 0 value. k-rate opcodes send a message each time the MIDI converted value of argument *kvalue* changes.

## See Also

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

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# outo

outo -- Writes 8-channel audio data to an external device or stream.

outo

## Description

Writes 8-channel audio data to an external device or stream.

## Syntax

**outo** asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8

## Performance

Sends 8-channel 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, Csound will attempt to change an incorrect opcode to agree with *nchnls* statement.

## See Also

*out, outh, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: John ffitch

# outq

outq -- Writes 4-channel audio data to an external device or stream.

outq

## Description

Writes 4-channel audio data to an external device or stream.

## Syntax

**outq** asig1, asig2, asig3, asig4

## Performance

Sends 4-channel 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out*, *outh*, *outo*, *outq1*, *outq2*, *outq3*, *outq4*, *outs*, *outs1*, *outs2*, *soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outq1

outq1 -- Writes samples to quad channel 1 of an external device or stream.

outq1

## Description

Writes samples to quad channel 1 of an external device or stream.

## Syntax

`outq1 asig`

## Performance

Sends 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out, outh, outo, outq, outq2, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997



# outq2

outq2 -- Writes samples to quad channel 2 of an external device or stream.

outq2

## Description

Writes samples to quad channel 2 of an external device or stream.

## Syntax

`outq2 asig`

## Performance

Sends 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out, outh, outh, outq, outq1, outq3, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

## outq3

outq3 -- Writes samples to quad channel 3 of an external device or stream.

outq3

## Description

Writes samples to quad channel 3 of an external device or stream.

## Syntax

**outq3** *asig*

## Performance

Sends 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out, outh, outh, outq, outq1, outq2, outq4, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

## outq4

outq4 -- Writes samples to quad channel 4 of an external device or stream.

outq4

## Description

Writes samples to quad channel 4 of an external device or stream.

## Syntax

**outq4** *asig*

## Performance

Sends 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out, outh, outh, outq, outq1, outq2, outq3, outs, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outs

outs -- Writes stereo audio data to an external device or stream.

outs

## Description

Writes stereo audio data to an external device or stream.

## Syntax

**outs** asig1, asig2

## Performance

Sends stereo 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out, outh, outh, outq, outq1, outq2, outq3, outq4, outs1, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outs1

outs1 -- Writes samples to stereo channel 1 of an external device or stream.

outs1

## Description

Writes samples to stereo channel 1 of an external device or stream.

## Syntax

**outs1** *asig*

## Performance

Sends 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out, outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs2, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

## outs2

outs2 -- Writes samples to stereo channel 2 of an external device or stream.

outs2

## Description

Writes samples to stereo channel 2 of an external device or stream.

## Syntax

**outs2** asig

## Performance

Sends 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, Csound will attempt to change an incorrect opcode to agree with the *nchnls* statement. Opcodes can be chosen to direct sound to any particular channel: *outs1* sends to stereo channel 1, *outq3* to quad channel 3, etc.

## See Also

*out, outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, soundout*

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

# outvalue

outvalue -- Sends a k-rate signal or string to a user-defined channel.

outvalue

## Description

Sends a k-rate signal or string to a user-defined channel.

## Syntax

**outvalue** "channel name", kvalue

**outvalue** "channel name", "string"

## Performance

*"channel name"* -- An integer or string (in double-quotes) representing channel.

*kvalue* -- The k-rate value that is sent to the channel.

*string* -- The string or string variable that is sent to the channel.

## See Also

*invalue*

## Credits

Author: Matt Ingalls

# outx

outx -- Writes 16-channel audio data to an external device or stream.

outx

## Description

Writes 16-channel audio data to an external device or stream.

## Syntax

**outx** asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, asig9, asig10, asi

## Performance

*outx* outputs 32 channels of audio.

## Credits

*out32, outc, outch, outz*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07



# outz

outz -- Writes multi-channel audio data from a ZAK array to an external device or stream.

outz

## Description

Writes multi-channel audio data from a ZAK array to an external device or stream.

## Syntax

**outz** *ksigl*

## Performance

*outz* outputs from a ZAK array for *nchnls* of audio.

## Credits

*out32*, *outc*, *outch*, *outx*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# p

*p* -- Show the value in a given p-field.

*p*

## Description

Show the value in a given p-field.

## Syntax

**p**(*x*)

This function works at i-rate and k-rate.

## Initialization

*x* -- the number of the p-field.

## Performance

The value returned by the *p* function is the value in a p-field.

## Examples

Here is an example of the *p* opcode. It uses the files *p.orc* [examples/p.orc] and *p.sco* [examples/p.sco].

### Example 256. Example of the *p* opcode.

```
/* p.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Get the value in the fourth p-field, p4.
  i1 = p(4)

  print i1
endin
/* p.orc */

/* p.sco */
; p4 = value to be printed.
; Play Instrument #1 for one second, p4 = 50.375.
i 1 0 1 50.375
e
/* p.sco */
```

Its output should include lines like:

```
instr 1:  i1 = 50.375
```

## Credits

Example written by Kevin Conder.

# pan

pan -- Distribute an audio signal amongst four channels.

pan

## Description

Distribute an audio signal amongst four channels with localization control.

## Syntax

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

## 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.

## Examples

```
instr      1
  k1          phasor      1/p3              ; fraction of circle
  k2          tablei      k1, 1, 1          ; sin of angle (sinusoid in f
  k3          tablei      k1, 1, 1, .25, 1   ; cos of angle (sin offset 1/
  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 qu  
    outq a1, a2, a3, a4  
endin
```

## pareq

pareq -- Implementation of Zoelzer's parametric equalizer filters.

pareq

## 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)
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 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
```

## Syntax

```
ares pareq asig, kc, kv, kq [, imode]
      [, iskip]
```

## Initialization

*imode* (optional, default: 0) -- operating mode

- 0 = Peaking
- 1 = Low Shelving
- 2 = High Shelving

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*kc* -- center frequency in peaking mode, corner frequency in shelving mode.

*kv* -- amount of boost or cut. A value less than 1 is a cut. A value greater than 1 is a boost. A value of 1 is a flat response.

*kq* -- Q of the filter (sqrt(.5) is no resonance)

*asig* -- the incoming signal

## Examples

Here is an example of the *pareq* opcode. It uses the files *pareq.orc* [examples/pareq.orc] and *pareq.sco* [examples/pareq.sco].

### Example 257. Example of the *pareq* opcode.

```
/* pareq.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

instr 15
  ifc      =      p4                ; Center / Shelf
  kq       =      p5                ; Quality factor sqrt(.5) is no reso
  kv       =      ampdb(p6)         ; Volume Boost/Cut
  imode    =      p7                ; Mode 0=Peaking EQ, 1=Low Shelf, 2=
  kfc      linseg ifc*2, p3, ifc/2
  asig     rand 5000                ; Random number source for testing
  aout     pareq asig, kfc, kv, kq, imode ; Parmetric equalization
          outs aout, aout          ; Output the results
endin
/* pareq.orc */

/* pareq.sco */
; 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
/* pareq.sco */
```

## Credits

Hans Mikelson  
December 1998

New in Csound version 3.50



# partials

partials -- Partial track spectral analysis.

partials

## Description

The partials opcode takes two input PV streaming signals containing AMP\_FREQ and AMP\_PHASE signals (as generated for instance by pvsifd or in the first case, by pvsanal) and performs partial track analysis, as described in Lazzarini et al, "Time-stretching using the Instantaneous Frequency Distribution and Partial Tracking", Proc.of ICMC05, Barcelona. It generates a TRACKS PV streaming signal, containing amplitude, frequency, phase and track ID for each output track. This type of signal will contain a variable number of output tracks, up to the total number of analysis bins contained in the inputs ( $\text{fftsize}/2 + 1$  bins). The second input (AMP\_PHASE) is optional, as it can take the same signal as the first input. In this case, however, all phase information will be NULL and resynthesis using phase information cannot be performed.

## Syntax

```
ftrks partials ffr, fphs, kthresh, kminpts, kmaxgap, imaxtracks
```

## Performance

*ffr* -- output pv stream in TRACKS format

*ffr* -- input pv stream in AMP\_FREQ format

*fphs* -- input pv stream in AMP\_PHASE format

*kthresh* -- analysis threshold. Tracks below  $\text{kthresh} * \text{max\_magnitude}$  will be discarded ( $1 > \text{kthresh} \geq 0$ ).

*kminpoints* -- minimum number of time points for a detected peak to make a track (1 is the minimum). Since this opcode works with streaming signals, larger numbers will increase the delay between input and output, as we have to wait for the required minimum number of points.

*kmaxgap* -- maximum gap between time-points for track continuation ( $> 0$ ). Tracks that have no continuation after *kmaxgap* will be discarded.

*imaxtracks* -- maximum number of analysis tracks (number of bins  $\geq$  *imaxtracks*)

## Examples

### Example 258. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
    aout resyn fst, 1, 1.5, 500, 1 ; resynthesis (up a 5th)
out aout
```

The example above shows partial tracking of an ifd-analysis signal and cubic-phase additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini;  
June 2005

New plugin in version 5

November 2004.

# pcauchy

pcauchy -- Cauchy distribution random number generator (positive values only).

pcauchy

## Description

Cauchy distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

ares **pcauchy** kalpha

ires **pcauchy** kalpha

kres **pcauchy** kalpha

## Performance

*pcauchy kalpha* -- controls the spread from zero (big kalpha = big spread). Outputs positive numbers only.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the pcauchy opcode. It uses the files *pcauchy.orc* [examples/pcauchy.orc] and *pcauchy.sco* [examples/pcauchy.sco].

### Example 259. Example of the pcauchy opcode.

```
/* pcauchy.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Generate a random number between 0 and 1.
; kalpha = 1

il pcauchy 1

print il
```

```
endin
/* pcauchy.orc */

/* pcauchy.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* pcauchy.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 0.012
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, linrand, poisson, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# pchbend

pchbend -- Get the current pitch-bend value for this channel.

pchbend

## Description

Get the current pitch-bend value for this channel.

## Syntax

ibend **pchbend** [imin] [, imax]

kbend **pchbend** [imin] [, imax]

## Initialization

*imin*, *imax* (optional) -- set minimum and maximum limits on values obtained

## Performance

Get the current pitch-bend value for this channel. 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.

## Examples

Here is an example of the pchbend opcode. It uses the files *pchbend.orc* [examples/pchbend.orc] and *pchbend.sco* [examples/pchbend.sco].

### Example 260. Example of the pchbend opcode.

```
/* pchbend.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il pchbend

  print il
endin
/* pchbend.orc */

/* pchbend.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* pchbend.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# pchmidi

pchmidi -- Get the note number of the current MIDI event, expressed in pitch-class units.

pchmidi

## Description

Get the note number of the current MIDI event, expressed in pitch-class units.

## Syntax

ipch **pchmidi**

## Performance

Get the note number of the current MIDI event, expressed in pitch-class units for local processing.

## Examples

Here is an example of the pchmidi opcode. It uses the files *pchmidi.orc* [examples/pchmidi.orc] and *pchmidi.sco* [examples/pchmidi.sco].

### Example 261. Example of the pchmidi opcode.

```
/* pchmidi.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il pchmidi

  print il
endin
/* pchmidi.orc */

/* pchmidi.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* pchmidi.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.



# pchmidib

pchmidib -- Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in pitch-class units.

pchmidib

## Description

Get the note number of the current MIDI event and modify it by the current pitch-bend value, express it in pitch-class units.

## Syntax

ipch **pchmidib** [irange]

kpch **pchmidib** [irange]

## Initialization

*irange* (optional) -- the pitch bend range in semitones

## Performance

Get the note number of the current MIDI event, modify it by the current pitch-bend value, and express the result in pitch-class units. Available as an i-time value or as a continuous k-rate value.

## Examples

Here is an example of the pchmidib pchmidib. It uses the files *pchmidib.orc* [examples/pchmidib.orc] and *pchmidib.sco* [examples/pchmidib.sco].

### Example 262. Example of the pchmidib pchmidib.

```
/* pchmidib.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il pchmidib

  print il
endin
/* pchmidib.orc */

/* pchmidib.sco */
; Play Instrument #1 for 12 seconds.
```

```
i 1 0 12  
e  
/* pchmidib.sco */
```

## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

# pchoct

pchoct -- Converts an octave-point-decimal value to pitch-class.

pchoct

## Description

Converts an octave-point-decimal value to pitch-class.

## Syntax

**pchoct** (oct) (init- or control-rate args only)

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

## Performance

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:

**Table 5. Pitch and Frequency Values**

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

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

## Examples

Here is an example of the `pchoct` opcode. It uses the files *pchoct.orc* [examples/pchoct.orc] and *pchoct.sco* [examples/pchoct.sco].

### Example 263. Example of the `pchoct` opcode.

```
/* pchoct.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Convert an octave-point-decimal value into a
  ; pitch-class value.
  ioct = 8.75
  ipch = pchoct(ioct)

  print ipch
endin
/* pchoct.orc */

/* pchoct.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* pchoct.sco */
```

Its output should include a line like this:

```
instr 1: ipch = 8.090
```

## See Also

*cpsoct*, *cpspch*, *octcps*, *octpch*

## Credits

Example written by Kevin Conder.

# pconvolve

pconvolve -- Convolution based on a uniformly partitioned overlap-save algorithm

convolve

## Description

Convolution based on a uniformly partitioned overlap-save algorithm. Compared to the *convolve* opcode, 'pconvolve' has these benefits:

- small delay
- possible to run in real-time for shorter impulse files
- no pre-process analysis pass
- can often render faster than convolve

## Syntax

```
ar1 [, ar2] [, ar3] [, ar4] pconvolve ain, ifilcod [, ipartitionsizes, ichannel]
```

## Initialization

*ifilcod* -- integer or character-string denoting an impulse response soundfile. multichannel files are supported, the file must have the same sample-rate as the orc. [Note: cval files cannot be used!] Keep in mind that longer files require more calculation time [and probably larger partition sizes and more latency]. At current processor speeds, files longer than a few seconds may not render in real-time.

*ipartitionsizes* (optional, defaults to the output buffersize [-b]) -- the size in samples of each partition of the impulse file. This is the parameter that needs tweaking for best performance depending on the impulse file size. Generally, a small size means smaller latency but more computation time. If you specify a value that is not a power-of-2 the opcode will find the next power-of-2 greater and use that as the actual partition size.

*ichannel* (optional) -- which channel to use from the impulse response data file.

## Performance

*ain* -- input audio signal.

The overall latency of the opcode can be calculated as such [assuming *ipartitionsizes* is a power of 2]

```
ilateney = (ksmps < ipartitionsizes ? ipartitionsizes + ksmps : ipartitionsizes)
```

## Examples

Instrument 1 shows an example of real-time convolution.

Instrument 2 shows how to do file-based convolution with a 'look ahead' method to remove all delay.

**NOTE**

You will need to download the impulse response files from [noisevault.com](http://noisevault.com) or replace the filenames with your own impulse files

```
sr = 44100
ksmps = 100
nchnls = 2

instr 1
kmix = .5 ; Wet/dry mix. Vary as desired.
kvol = .5*kmix ; Overall volume level of reverb. May need to adjust
              ; when wet/dry mix is changed, to avoid clipping.

; do some safety checking to make sure we the parameters a good
kmix = (kmix < 0 || kmix > 1 ? .5 : kmix)
kvol = (kvol < 0 ? 0 : .5*kvol*kmix)

; size of each convolution partion -- for best performance, this parameter need
ipartitionsize = p4

; calculate latency of pconvolve opcode
idel = (ksmps < ipartitionsize ? ipartitionsize + ksmps : ipartitionsize)/sr
prints "Convoluting with a latency of %f seconds\n", idel

; actual processing
al, ar ins

awetl, awetr pconvolve kvol*(al+ar), "Mercedes-van.wav", ipartitionsize

; Delay dry signal, to align it with the convoled sig
adryl delay (1-kmix)*al, idel
adryr delay (1-kmix)*ar, idel

outs adryl+awetl, adryr+awetr

endin

instr 2
imix = 0.5 ; Wet/dry mix. Vary as desired.
ivol = .5*imix ; Overall volume level of reverb. May need to adjust
              ; when wet/dry mix is changed, to avoid clipping.

ipartitionsize = 32768 ; size of each convolution partion
idel = (ksmps < ipartitionsize ? ipartitionsize + ksmps : ipartitionsize)/sr
kcount init idel*kr

; since we are using a soundin [instead of ins] we can
; do a kind of "look ahead" by looping during one k-pass
; without output, creating zero-latency

loop:
  al, ar soundin "John_Cage_1.aif", 0

  awetl, awetr pconvolve ivol*(al+ar), "FactoryHall.aif", ipartitionsize

  adryl delay (1-imix)*al, idel ; Delay dry signal, to align it with
  adryr delay (1-imix)*ar, idel ;

  kcount = kcount - 1
  if kcount > 0 goto loop
```

```
outs    awetl+adryl, awetr+adryr  
endin
```

## Credits

Author: Matt Ingalls  
2004

# peak

*peak* -- Maintains the output equal to the highest absolute value received.

*peak*

## Description

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

## Syntax

*kres* **peak** *asig*

*kres* **peak** *ksig*

## Performance

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

*ksig* -- k-rate input signal.

*asig* -- a-rate input signal.

## Examples

Here is an example of the *peak* opcode. It uses the files *peak.orc* [examples/peak.orc], *peak.sco* [examples/peak.sco], and *beats.wav* [examples/beats.wav].

### Example 264. Example of the *peak* opcode.

```
/* peak.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1 - play an audio file.
instr 1
  ; Capture the highest amplitude in the "beats.wav" file.
  asig soundin "beats.wav"
  kp peak asig

  ; Print out the peak value once per second.
  printk 1, kp

  out asig
endin
/* peak.orc */
```

```
/* peak.sco */
```



```
; Play Instrument #1, the audio file, for three seconds.  
i 1 0 3  
e  
/* peak.sco */
```

Its output should include lines like this:

```
i   1 time      0.00002:  4835.00000  
i   1 time      1.00002: 29312.00000  
i   1 time      2.00002: 32767.00000
```

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# peakk

peakk -- Deprecated.

peakk

## Description

Deprecated as of version 3.63. Use the *peak* opcode instead.

# pgmassign

pgmassign -- Assigns an instrument number to a specified MIDI program.

pgmassign

## Description

Assigns an instrument number to a specified (or all) MIDI program(s).

By default, the instrument is the same as the program number. If the selected instrument is zero or negative or does not exist, the program change is ignored. This opcode is normally used in the orchestra header. Although, like *massign*, it also works in instruments.

## Syntax

```
pgmassign ipgm, inst[, ichn]
```

```
pgmassign ipgm, "insname"[, ichn]
```

## Initialization

*ipgm* -- MIDI program number (1 to 128). A value of zero selects all programs.

*inst* -- instrument number. If set to zero, or negative, MIDI program changes to *ipgm* are ignored. Currently, assignment to an instrument that does not exist has the same effect. This may be changed in a later release to print an error message.

*"insname"* -- A string (in double-quotes) representing a named instrument.

*"ichn"* (optional, defaults to zero) -- channel number. If zero, program changes are assigned on all channels.

You can disable the turning on of any instruments by using the following in the header:

```
massign 0, 0  
pgmassign 0, 0
```

## Examples

Here is an example of the *pgmassign* opcode. It uses the files *pgmassign.orc* [examples/pgmassign.orc] and *pgmassign.sco* [examples/pgmassign.sco].

### Example 265. Example of the *pgmassign* opcode.

```
/* pgmassign.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1  
  
; Program 55 (synth vox) uses Instrument #10.
```

```
pgmassign 55, 10

; Instrument #10.
instr 10
    ; Just an example, no working code in here!
endin
/* pgmassign.orc */

/* pgmassign.sco */
; Play Instrument #10 for one second.
i 10 0 1
e
/* pgmassign.sco */
```

Here is an example of the `pgmassign` opcode that will ignore program change events. It uses the files `pgmassign_ignore.orc` [examples/pgmassign\_ignore.orc] and `pgmassign_ignore.sco` [examples/pgmassign\_ignore.sco].

**Example 266. Example of the `pgmassign` opcode that will ignore program change events.**

```
/* pgmassign_ignore.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Ignore all program change events.
pgmassign 0, -1

; Instrument #1.
instr 1
    ; Just an example, no working code in here!
endin
/* pgmassign_ignore.orc */

/* pgmassign_ignore.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* pgmassign_ignore.sco */
```

Here is an advanced example of the `pgmassign` opcode. It uses the files `pgmassign_advanced.mid` [examples/pgmassign\_advanced.mid], `pgmassign_advanced.orc` [examples/pgmassign\_advanced.orc], and `pgmassign_advanced.sco` [examples/pgmassign\_advanced.sco].

Don't forget that you must include the *-F flag* when using an external MIDI file like "pgmassign\_advanced.mid".

**Example 267. An advanced example of the pmassign opcode.**

```
/* pmassign_advanced.orc - written by Istvan Varga */
sr = 44100
ksmps = 10
nchnls = 1

    massign 1, 1 ; channels 1 to 4 use instr 1 by default
    massign 2, 1
    massign 3, 1
    massign 4, 1

; pmassign.mid has 4 notes with these parameters:
;
;   Start time Channel Program
;
; note 1 0.5      1      10
; note 2 1.5      2      11
; note 3 2.5      3      12
; note 4 3.5      4      13

    pmassign 0, 0          ; disable program changes
    pmassign 11, 3         ; program 11 uses instr 3
    pmassign 12, 2         ; program 12 uses instr 2

; waveforms for instruments
itmp ftgen 1, 0, 1024, 10, 1
itmp ftgen 2, 0, 1024, 10, 1, 0.5, 0.3333, 0.25, 0.2, 0.1667, 0.1429, 0.125
itmp ftgen 3, 0, 1024, 10, 1, 0, 0.3333, 0, 0.2, 0, 0.1429, 0, 0.10101

    instr 1                /* sine */

kcps cpsmidib 2 ; note frequency
asnd oscili 30000, kcps, 1
    out asnd

    endin

    instr 2                /* band-limited sawtooth */

kcps cpsmidib 2 ; note frequency
asnd oscili 30000, kcps, 2
    out asnd

    endin

    instr 3                /* band-limited square */

kcps cpsmidib 2 ; note frequency
asnd oscili 30000, kcps, 3
    out asnd

    endin
/* pmassign_advanced.orc - written by Istvan Varga */

/* pmassign_advanced.sco - written by Istvan Varga */
t 0 120
```

```
f 0 8.5 2 -2 0  
e  
/* pgmassign_advanced.sco - written by Istvan Varga */
```

## See Also

*midichn* and *massign*

## Credits

Author: Istvan Varga  
May 2002

New in version 4.20

# phaser1

phaser1 -- First-order allpass filters arranged in a series.

phaser1

## Description

An implementation of *iord* number of first-order allpass filters in series.

## Syntax

ares **phaser1** asig, kfreq, kord, kfeedback [, iskip]

## Initialization

*iskip* (optional, default=0) -- 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.

## Performance

*kfreq* -- frequency (in Hz) of the filter(s). This is the frequency at which each filter in the series shifts its input by 90 degrees.

*kord* -- the number of allpass stages in series. These are first-order filters and can range from 1 to 4999.



### Note

Although *kord* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*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.

*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.

## Examples

Here is an example of the phaser1 opcode. It uses the files *phaser1.orc* [examples/phaser1.orc] and *phaser1.sco* [examples/phaser1.sco].

### Example 268. Example of the phaser1 opcode.

```
/* phaser1.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; demonstration of phase shifting abilities of phaser1.
instr 1
; 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 osci
kfreq   oscili 5500, ifreq, 1
kmod    = kfreq + 5600

aphs    phaser1 asig, kmod, iorder, ifeed

out      (asig + apha) * iamp
endin
/* phaser1.orc */

/* phaser1.sco */
; inverted half-sine, used for modulating phaser1 frequency
f1 0 16384 9 .5 -1 0
; cosine wave for gbuzz
f2 0 8192 9 1 1 .25

; 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
e
/* phaser1.sco */
```

## 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.

## See Also

*phaser2*

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

November 2002. Added a note about the *kord* parameter, thanks to Rasmus Ekman.

New in Csound version 4.0

# phaser2

phaser2 -- Second-order allpass filters arranged in a series.

phaser2

## Description

An implementation of *iord* number of second-order allpass filters in series.

## Syntax

ares **phaser2** asig, kfreq, kq, kord, kmode, ksep, kfeedback

## Initialization

*iskip* (optional, default=0) -- 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.

## Performance

*kfreq* -- frequency (in Hz) of the filter(s). 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.

*kord* -- the number of allpass stages in series. These are second-order filters, and *iord* can range from 1 to 2499. With higher orders, the computation time increases.

*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.

*kmode* -- used in calculation of notch frequencies.



### Note

Although *kord* and *kmode* are listed as k-rate, they are in fact accessed only at init-time. So if you are using k-rate arguments, they must be assigned with *init*.

*ksep* -- scaling factor used, in conjunction with *imode*, to determine the frequencies of the additional notches in the output spectrum.

*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 separated, 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 + aphis
```

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

Here is an example of the phaser2 opcode. It uses the files *phaser2.orc* [examples/phaser2.orc] and *phaser2.sco* [examples/phaser2.sco].

### Example 269. Example of the phaser2 opcode.

```
/* phaser2.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

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

; "Sawtooth" waveform exponentially decaying function, to control notch frequency
asig gbuzz 1, 100, iharms, 1, .95, 2
kline expseg 1, idur, .005
aphs phaser2 asig, kline * 2000, .5, iorder, imode, isep, ifeed

out (asig + aphis) * iamp
endin
/* phaser2.orc */
```

```
/* phaser2.sco */
; cosine wave for gbuzz
f2 0 8192 9 1 1 .25

; phaser2, imode=1
i2 00 10 7000 8 .2 .9 1 .33
i2 11 10 7000 8 .2 .9 1 2

; phaser2, imode=2
i2 22 10 7000 8 .2 .9 2 .33
i2 33 10 7000 8 .2 .9 2 2
e
/* phaser2.sco */
```

## 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.

## See Also

*phaser1*

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

November 2002. Added a note about the *kord* and *kmode* parameters, thanks to Rasmus Ekman.

New in Csound version 4.0

# phasor

phasor -- Produce a normalized moving phase value.

phasor

## Description

Produce a normalized moving phase value.

## Syntax

ares **phasor** xcps [, iphs]

kres **phasor** kcps [, iphs]

## Initialization

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

## Performance

An internal phase is successively accumulated in accordance with the *kcps* or *xcps* frequency to produce a moving phase value, normalized to lie in the range  $0 \leq \text{phs} < 1$ .

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.

## Examples

Here is an example of the phasor opcode. It uses the files *phasor.orc* [examples/phasor.orc] and *phasor.sco* [examples/phasor.sco].

### Example 270. Example of the phasor opcode.

```
/* phasor.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Create an index that repeats once per second.
  kcps init 1
  kndx phasor kcps

  ; Read Table #1 with our index.
  ifn = 1
  ixmode = 1
```

```
kfreq table kndx, ifn, ixmode

; Generate a sine waveform, use our table values
; to vary its frequency.
a1 oscil 20000, kfreq, 2
out a1
endin
/* phasor.orc */

/* phasor.sco */
; Table #1, a line from 200 to 2,000.
f 1 0 1025 -7 200 1024 2000
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* phasor.sco */
```

## See also

The *Table Access* opcodes like: *table*, *tablei*, *table3* and *tab*.

Also: *table*.

## Credits

Example written by Kevin Conder.

# phasorbnk

phasorbnk -- Produce an arbitrary number of normalized moving phase values.

phasorbnk

## Description

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

## Syntax

ares **phasorbnk** xcps, kndx, icnt [, iphs]

kres **phasorbnk** kcps, kndx, icnt [, iphs]

## 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 *iphs*>1 each phasor will be initialized with a random value.

## 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  $0 \leq \text{phs} < 1$ . 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*.

## Examples

Here is an example of the phasorbnk opcode. It uses the files *phasorbnk.orc* [examples/phasorbnk.orc] and *phasorbnk.sco* [examples/phasorbnk.sco].

### Example 271. Example of the phasorbnk opcode.

```
/* phasorbnk.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Generate a sinewave table.
giwave ftgen 1, 0, 1024, 10, 1

; Instrument #1
instr 1
  ; Generate 10 voices.
  icnt = 10
```

```
; Empty the output buffer.
asum = 0
; Reset the loop index.
kindex = 0

; This loop is executed every k-cycle.
loop:
; Generate non-harmonic partials.
kcps = (kindex+1)*100+30
; Get the phase for each voice.
aphas phasorbnk kcps, kindex, icnt
; Read the wave from the table.
asig table aphas, giwave, 1
; Accumulate the audio output.
asum = asum + asig

; Increment the index.
kindex = kindex + 1

; Perform the loop until the index (kindex) reaches
; the counter value (icnt).
if (kindex < icnt) kgoto loop

out asum*3000
endin
/* phasorbnk.orc */

/* phasorbnk.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* phasorbnk.sco */
```

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

## Credits

Author: Peter Neubäcker  
Munich, Germany  
August 1999

New in Csound version 3.58



# pinkish

pinkish -- Generates approximate pink noise.

pinkish

## 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

## Syntax

```
ares pinkish xin [, imethod] [, inumbands] [, iseed] [, iskip]
```

## Initialization

*imethod* (optional, default=0) -- 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, default=0) -- 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, default=0) -- 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 ( $\pm 15\%$ ) 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.

## Examples

Here is an example of the pinkish opcode. It uses the files *pinkish.orc* [examples/pinkish.orc] and *pinkish.sco* [examples/pinkish.sco].

### Example 272. Example of the pinkish opcode.

```
/* pinkish.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  awhite unirand 2.0

  ; Normalize to +/-1.0
  awhite = awhite - 1.0

  apink pinkish awhite, 1, 0, 0, 1

  out apink * 30000
endin
/* pinkish.orc */

/* pinkish.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* pinkish.sco */
```

Kellet-filtered noise for a tied note (*iskip* is non-zero).

## Credits

Authors: Phil Burk and John fitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

Adapted for Csound by Rasmus Ekman

The noise bands method is due to F. R. Moore (or R. F. Voss), and was presented by Martin Gard-

ner in an oft-cited article in Scientific American. The present version was coded by Phil Burk as the result of discussion on the music-dsp mailing list, with significant optimizations suggested by James McCartney.

The filter bank was designed by Paul Kellet, posted to the music-dsp mailing list.

The whole pink noise discussion was collected on a HTML page by Robin Whittle, which is currently available at <http://www.firstpr.com.au/dsp/pink-noise/>.

Added notes by Rasmus Ekman on September 2002.

# pitch

`pitch` -- Tracks the pitch of a signal.

`pitch`

## 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.

## Syntax

```
koct, kamp pitch asig, iupdt, ilo, ihi, idbthresh [, ifrqs] [, iconf] [, istrtr
```

## Initialization

*iupdt* -- 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* (optional) -- number of divisions of an octave. Default is 12 and is limited to 120.

*iconf* (optional) -- the number of conformations needed for an octave jump. Default is 10.

*istrtr* (optional) -- starting pitch for tracker. Default value is  $(ilo + ihi)/2$ .

*iocts* (optional) -- number of octave decimations in spectrum. Default is 6.

*iq* (optional) -- Q of analysis filters. Default is 10.

*inptls* (optional) -- number of harmonics, used in matching. Computation time increases with the number of harmonics. Default is 4.

*irolloff* (optional) -- amplitude rolloff for the set of filters expressed as fraction per octave. Values must be positive. Default is 0.6.

*iskip* (optional) -- if non-zero, skips initialization. Default is 0.

## Performance

*koct* -- The pitch output, given in the octave point decimal format.

*kamp* -- The amplitude output.

*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 *iupdt* 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.

## Examples

Here is an example of the `pitch` opcode. It uses the files *pitch.orc* [examples/pitch.orc], *pitch.sco*

[examples/pitch.sco] and *mary.wav* [examples/mary.wav].

### Example 273. Example of the pitch opcode.

```
/* pitch.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1 - play an audio file without effects.
instr 1
  asig soundin "mary.wav"
  out asig
endin

; Instrument #2 - track the pitch of an audio file.
instr 2
  iupdt = 0.01
  ilo = 7
  ihi = 9
  idbthresh = 10
  ifrqs = 12
  iconf = 10
  istr = 8

  asig soundin "mary.wav"

  ; Follow the audio file, get its pitch and amplitude.
  koct, kamp pitch asig, iupdt, ilo, ihi, idbthresh, ifrqs, iconf, istr

  ; Re-synthesize the audio file with a different sounding waveform.
  kamp2 = kamp * 10
  kcps = cpsoct(koct)
  a1 oscil kamp2, kcps, 1

  out a1
endin
/* pitch.orc */

/* pitch.sco */
; Table #1: A different sounding waveform.
f 1 0 32768 11 7 3 .7

; Play Instrument #1, the audio file, for three seconds.
i 1 0 3
; Play Instrument #2, the "re-synthesized" waveform, for three seconds.
i 2 3 3
e
/* pitch.sco */
```

## Credits

Author: John ffitch  
University of Bath, Codemist Ltd.

Bath, UK  
April 1999

Example written by Kevin Conder.

New in Csound version 3.54

# pitchamdf

pitchamdf -- Follows the pitch of a signal based on the AMDF method.

pitchamdf

## 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.

## Syntax

*kcps*, *krms* **pitchamdf** *asig*, *imincps*, *imaxcps* [, *icps*] [, *imedi*] [, *idowns*] [, *icps*]

## Initialization

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

*imaxcps* -- estimated maximum frequency present in the signal

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

*imedi* (optional, default=1) -- 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* (optional, default=1) -- 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.

*icps* (optional, default=0) -- how frequently pitch analysis is executed, expressed in Hz. If 0, *icps* is set to *imincps*. This is usually reasonable, but experimentation with other values may lead to better results. Default is 0.

*irmsmedi* (optional, default=0) -- size of median filter applied to the output *krms*. The size of the filter will be *irmsmedi*\*2+1. If 0, no median filtering will be applied. The 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 detection 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 performance, especially with complex waveforms.

## Examples

Here is an example of the `pitchamdf` opcode. It uses the files `pitchamdf.orc` [examples/pitchamdf.orc], `pitchamdf.sco` [examples/pitchamdf.sco] and `mary.wav` [examples/mary.wav].

### Example 274. Example of the `pitchamdf` opcode.

```
/* pitchamdf.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; synth waveform
giwave ftgen 2, 0, 1024, 10, 1, 1, 1, 1

; Instrument #1 - play an audio file with no effects.
instr 1
  ; get input signal with original freq.
  asig soundin "mary.wav"

  out asig
endin

; Instrument #2 - play the synth waveform using the
; same pitch and amplitude as the audio file.
instr 2
  ; get input signal with original freq.
  asig soundin "mary.wav"

  ; lowpass-filter
  asig tone asig, 1000
  ; extract pitch and envelope
  kcps, krms pitchamdf asig, 150, 500, 200
  ; "re-synthesize" with the synth waveform, giwave.
  asigl oscil krms, kcps, giwave

  out asigl
endin
/* pitchamdf.orc */

/* pitchamdf.sco */
; Play Instrument #1, the audio file, for three seconds.
i 1 0 3
; Play Instrument #2, the "re-synthesized" waveform, for three seconds.
i 2 3 3
e
/* pitchamdf.sco */
```

## Credits

Author: Peter Neubäcker  
Munich, Germany



August 1999

New in Csound version 3.59

# planet

`planet` -- Simulates a planet orbiting in a binary star system.

`planet`

## 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.

## Syntax

```
ax, ay, az planet kmass1, kmass2,  
ksep, ix, iy, iz, ivx, ivy, ivz, idelta [, ifriction] [, iskip]
```

## 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* (optional, default=0) -- a value for friction, which can used to keep the system from blowing up

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*ax, ay, az* -- the output x, y, and z coordinates of the planet

*kmass1* -- the mass of the first star

*kmass2* -- the mass of the second star

## Examples

Here is an example of the `planet` opcode. It uses the files *planet.orc* [examples/planet.orc] and *planet.sco* [examples/planet.sco].

### Example 275. Example of the planet opcode.

```
/* planet.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 44100  
ksmps = 1  
nchnls = 2  
  
; Instrument #1 - a planet orbiting in 3D space.  
instr 1  
; Create a basic tone.
```

```
kamp init 5000
kcps init 440
ifn = 1
asnd oscil kamp, kcps, ifn

; Figure out its X, Y, Z coordinates.
km1 init 0.5
km2 init 0.35
ksep init 2.2
ix = 0
iy = 0.1
iz = 0
ivx = 0.5
ivy = 0
ivz = 0
ih = 0.0003
ifric = -0.1
ax1, ay1, az1 planet km1, km2, ksep, ix, iy, iz, \
                    ivx, ivy, ivz, ih, ifric

; Place the basic tone within 3D space.
kx downsamp ax1
ky downsamp ay1
kz downsamp az1
idist = 1
ift = 0
imode = 1
imdel = 1.018853416
iovr = 2
aw2, ax2, ay2, az2 spat3d asnd, kx, ky, kz, idist, \
                        ift, imode, imdel, iovr

; Convert the 3D sound to stereo.
aleft = aw2 + ay2
aright = aw2 - ay2

outs aleft, aright
endin
/* planet.orc */

/* planet.sco */
; Table #1 a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 10 seconds.
i 1 0 10
e
/* planet.sco */
```

## Credits

Author: Hans Mikelson  
December 1998

New in Csound version 3.50

# pluck

pluck -- Produces a naturally decaying plucked string or drum sound.

pluck

## Description

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

## Syntax

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

## 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 coeffs .5. Unaffected by parameter values.

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

## Performance

*kamp* -- the output amplitude.

*kcps* -- the resampling frequency in cycles-per-second.

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.

## Examples

Here is an example of the pluck opcode. It uses the files *pluck.orc* [examples/pluck.orc] and *pluck.sco* [examples/pluck.sco].

### Example 276. Example of the pluck opcode.

```
/* pluck.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 20000
  kcps = 440
  icps = 440
  ifn = 0
  imeth = 1

  a1 pluck kamp, kcps, icps, ifn, imeth
  out a1
endin
/* pluck.orc */

/* pluck.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* pluck.sco */
```

## Credits

Example written by Kevin Conder.

# poisson

poisson -- Poisson distribution random number generator (positive values only).

poisson

## Description

Poisson distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

ares **poisson** klambda

ires **poisson** klambda

kres **poisson** klambda

## Performance

*klambda* -- the mean of the distribution. Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the poisson opcode. It uses the files *poisson.orc* [examples/poisson.orc] and *poisson.sco* [examples/poisson.sco].

### Example 277. Example of the poisson opcode.

```
/* poisson.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Generates a random number in a poisson distribution.
; klambda = 1

il poisson 1

print il
endin
```

```
/* poisson.orc */

/* poisson.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* poisson.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 1.000
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, linrand, pcauchy, trirand, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# polyaft

`polyaft` -- Returns the polyphonic after-touch pressure of the selected note number.

`polyaft`

## Description

*polyaft* returns the polyphonic pressure of the selected note number, optionally mapped to an user-specified range.

## Syntax

```
ires polyaft inote [, ilow] [, ihigh]
```

```
kres polyaft inote [, ilow] [, ihigh]
```

## Initialization

*inote* -- note number. Normally set to the value returned by *notnum*

*ilow* (optional, default: 0) -- lowest output value

*ihigh* (optional, default: 127) -- highest output value

## Performance

*kres* -- polyphonic pressure (aftertouch).

## Examples

Here is an example of the `polyaft` opcode. It uses the files *polyaft.mid* [examples/polyaft.mid], *polyaft.orc* [examples/polyaft.orc] and *polyaft.sco* [examples/polyaft.sco].

Don't forget that you must include the *-F flag* when using an external MIDI file like “polyaft.mid”.

### Example 278. Example of the `polyaft` opcode.

```
/* polyaft.orc - written by Istvan Varga */
sr = 44100
ksmps = 10
nchnls = 1

    massign 1, 1
itmp ftgen 1, 0, 1024, 10, 1          ; sine wave

    instr 1

kcps cpsmidib 2          ; note frequency
inote notnum              ; note number
kaft polyaft inote, 0, 127 ; aftertouch
; interpolate aftertouch to eliminate clicks
ktmp phasor 40
ktmp trigger 1 - ktmp, 0.5, 0
```



```
kaft tlineto kaft, 0.025, ktmp
; map to sine curve for crossfade
kaft = sin(kaft * 3.14159 / 254) * 22000

asnd oscili kaft, kcps, 1

      out asnd

      endin
/* polyaft.orc - written by Istvan Varga */

/* polyaft.sco - written by Istvan Varga */
t 0 120
f 0 9 2 -2 0
e
/* polyaft.sco - written by Istvan Varga */
```

## Credits

Added thanks to an email from Istvan Varga

New in version 4.12

# port

**port** -- Applies portamento to a step-valued control signal.

**port**

## Description

Applies portamento to a step-valued control signal.

## Syntax

```
kres port ksig, ihtim [, isig]
```

## Initialization

*ih*tim -- half-time of the function, in seconds.

*isig* (optional, default=0) -- initial (i.e. previous) value for internal feedback. The default value is 0. Negative value will cause initialization to be skipped and last value from previous instance to be used as initial value for note.

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*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 *ih*tim. *ih*tim 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.

## See Also

*areson*, *aresonk*, *atone*, *atonek*, *portk*, *reson*, *resonk*, *tone*, *tonek*

# portk

portk -- Applies portamento to a step-valued control signal.

portk

## Description

Applies portamento to a step-valued control signal.

## Syntax

```
kres portk ksig, khtim [, isig]
```

## Initialization

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

## Performance

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*khtim* -- half-time of the function in seconds.

*portk* is like *port* except the half-time can be varied at the control rate.

## See Also

*areson*, *aresonk*, *atone*, *atonek*, *port*, *reson*, *resonk*, *tone*, *tonek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# poscil

poscil -- High precision oscillator.

poscil

## Description

High precision oscillator.

## Syntax

ares **poscil** aamp, acps, ifn [, iphs]

ares **poscil** aamp, kcps, ifn [, iphs]

ares **poscil** kamp, acps, ifn [, iphs]

ares **poscil** kamp, kcps, ifn [, iphs]

ires **poscil** kamp, kcps, ifn [, iphs]

kres **poscil** kamp, kcps, ifn [, iphs]

## Initialization

*ifn* -- function table number

*iphs* (optional, default=0) -- initial phase (in samples)

## Performance

*ares* -- output signal

*kamp*, *aamp* -- the amplitude of the output signal.

*kcps*, *acps* -- the frequency of the output signal in cycles per second.

*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*.

Since Csound 4.22, *poscil* can accept also negative frequency values and use a-rate values both for amplitude and frequency. So both AM and FM are allowed using this opcode.

## Examples

Here is an example of the *poscil* opcode. It uses the files *poscil.orc* [examples/poscil.orc] and *poscil.sco* [examples/poscil.sco].

**Example 279. Example of the poscil opcode.**

```
/* poscil.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 440
  ifn = 1

  a1 poscil kamp, kcps, ifn
  out a1
endin
/* poscil.orc */

/* poscil.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* poscil.sco */
```

## See Also

*poscil3*

## Credits

Author: Gabriel Maldonado  
Italy  
1998

Example written by Kevin Conder.

November 2002. Added a note about the changes to Csound version 4.22, thanks to Rasmus Ekman.

New in Csound version 3.52

# poscil3

poscil3 -- High precision oscillator with cubic interpolation.

poscil3

## Description

High precision oscillator with cubic interpolation.

## Syntax

ares **poscil3** kamp, kcps, ifn [, iphs]

kres **poscil3** kamp, kcps, ifn [, iphs]

## Initialization

*ifn* -- function table number

*iphs* (optional, default=0) -- initial phase (in samples)

## Performance

*ares* -- output signal

*kamp* -- the amplitude of the output signal.

*kcps* -- the frequency of the output signal in cycles per second.

*poscil3* uses cubic interpolation.

## Examples

Here is an example of the poscil3 opcode. It uses the files *poscil3.orc* [examples/poscil3.orc] and *poscil3.sco* [examples/poscil3.sco].

### Example 280. Example of the poscil3 opcode.

```
/* poscil3.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a basic oscillator.
instr 1
  kamp = 10000
  kcps = 440
  ifn = 1

  a1 poscil3 kamp, kcps, ifn
  out a1
endin
```

```
/* poscil3.orc */

/* poscil3.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* poscil3.sco */
```

## See Also

*poscil*

## Credits

Authors: John ffitch, Gabriel Maldonado  
Italy

Example written by Kevin Conder.

New in Csound version 3.52

# pow

pow -- Computes one argument to the power of another argument.

pow

## Description

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

## Syntax

ares **pow** aarg, kpow [, inorm]

ires **pow** iarg, ipow [, inorm]

kres **pow** karg, kpow [, inorm]

## Initialization

*inorm* (optional, default=1) -- 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!

## Performance

*aarg*, *iarg*, *karg* -- the base.

*ipow*, *kpow* -- the exponent.



### Note

Use ^ with caution in arithmetical statements, as the precedence may not be correct.  
New in Csound version 3.493.

## Examples

Here is an example of the pow opcode. It uses the files *pow.orc* [examples/pow.orc] and *pow.sco* [examples/pow.sco].

### Example 281. Example of the pow opcode.

```
/* pow.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; This could also be expressed as: i1 = 2 ^ 12
  i1 pow 2, 12
```



```
    print i1
  endin
/* pow.orc */
```

```
/* pow.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* pow.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 4096.000
```

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# powoftwo

powoftwo -- Performs a power-of-two calculation.

powoftwo

## Description

Performs a power-of-two calculation.

## Syntax

**powoftwo**(x) (init-rate or control-rate args only)

## 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*.

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.

## Examples

Here is an example of the powoftwo opcode. It uses the files *powoftwo.orc* [examples/powoftwo.orc] and *powoftwo.sco* [examples/powoftwo.sco].

### Example 282. Example of the powoftwo opcode.

```
/* powoftwo.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il = powoftwo(12)
  print il
endin
/* powoftwo.orc */

/* powoftwo.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* powoftwo.sco */
```

Its output should include a line like this:

```
instr 1:  i1 = 4096.000
```

## See Also

*logbtwo, pow*

## Credits

Author: Gabriel Maldonado  
Italy  
June 1998

Author: John ffitch  
University of Bath, Codemist, Ltd.  
Bath, UK  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# prealloc

`prealloc` -- Creates space for instruments but does not run them.

`prealloc`

## Description

Creates space for instruments but does not run them.

## Syntax

`prealloc insnum, icount`

`prealloc "insname", icount`

## Initialization

*insnum* -- instrument number

*icount* -- number of instrument allocations

*"insname"* -- A string (in double-quotes) representing a named instrument.

## Performance

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

## Examples

Here is an example of the *prealloc* opcode. It uses the files *prealloc.orc* [examples/prealloc.orc] and *prealloc.sco* [examples/prealloc.sco].

### Example 283. Example of the prealloc opcode.

```
/* prealloc.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Pre-allocate memory for five instances of Instrument #1.
prealloc 1, 5

; Instrument #1
instr 1
; Generate a waveform, get the cycles per second from the 4th p-field.
  al oscil 6500, p4, 1
  out al
endin
/* prealloc.orc */
```

```
/* prealloc.sco */
; Just generate a nice, ordinary sine wave.
f 1 0 32768 10 1

; Play five instances of Instrument #1 for one second.
; Note that 4th p-field contains cycles per second.
i 1 0 1 220
i 1 0 1 440
i 1 0 1 880
i 1 0 1 1320
i 1 0 1 1760
e
/* prealloc.sco */
```

## See Also

*cpuprc, maxalloc*

## Credits

Author: Gabriel Maldonado  
Italy  
July 1999

Example written by Kevin Conder.

New in Csound version 3.57

# print

`print --` Displays the values `init`, `control`, or audio signals.

`print`

## 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.

## Syntax

```
print iarg [, iarg1] [, iarg2] [...]
```

## Initialization

*iarg*, *iarg2*, ... -- i-rate arguments.

## Performance

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

## Examples

Here is an example of the `print` opcode. It uses the files *print.orc* [examples/print.orc] and *print.sco* [examples/print.sco].

### Example 284. Example of the `print` opcode.

```
/* print.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print the fourth p-field.
  print p4
endin
/* print.orc */

/* print.sco */
; p4 = value to be printed.
; Play Instrument #1 for one second, p4 = 50.375.
i 1 0 1 50.375
; Play Instrument #1 for one second, p4 = 300.
i 1 1 1 300
```

```
; Play Instrument #1 for one second, p4 = -999.  
i 1 2 1 -999  
e  
/* print.sco */
```

Its output should include lines like this:

```
instr 1:  p4 = 50.375  
instr 1:  p4 = 300.000  
instr 1:  p4 = -999.000
```

## See Also

*disppft*, *display*, *printk*, *printk2*, *printks* and *prints*

## Credits

Example written by Kevin Conder.

Comments about the *inprds* parameter contributed by Rasmus Ekman.

# printf

`printf` -- printf-style formatted output

`printf`

## Description

**printf** and **printf\_i** write formatted output, similarly to the C function `printf()`. **printf\_i** runs at i-time only, while **printfk** runs both at initialization and performance time.

## Syntax

```
printf_i Sfmt, itrig, [xarg1[, xarg2[, ... ]]]
```

```
printf Sfmt, itrig, [xarg1[, xarg2[, ... ]]]
```

## Initialization

*Sfmt* -- format string, has the same format as in `printf()` and other similar C functions, except length modifiers (l, ll, h, etc.) are not supported. The following conversion specifiers are allowed:

- d, i, o, u, x, X, e, E, f, F, g, G, c, s

## Performance

*itrig* -- if greater than zero the opcode performs the printing; otherwise it is a null operation.

*ktrig* -- if greater than zero and different from the value on the previous control cycle the opcode performs the requested printing. Initially this previous value is taken as zero.

*xarg1*, *xarg2*, ... -- input arguments (max. 30) for format. Integer formats like `%d` round the input values to the nearest integer.

## Example

## Credits

Author: Istvan Varga  
2005



# printk

printk -- Prints one k-rate value at specified intervals.

printk

## Description

Prints one k-rate value at specified intervals.

## Syntax

**printk** *itime*, *kval* [, *ispace*]

## Initialization

*itime* -- time in seconds between printings.

*ispace* (optional, default=0) -- number of spaces to insert before printing. (default: 0, max: 130)

## Performance

*kval* -- The k-rate values to be printed.

*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.

This opcode can be run on every k-cycle it is 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.

## Examples

Here is an example of the printk opcode. It uses the files *printk.orc* [examples/printk.orc] and *printk.sco* [examples/printk.sco].

### Example 285. Example of the printk opcode.

```
/* printk.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
; Change a value linearly from 0 to 100,
; over the period defined by p3.
kval line 0, p3, 100
```

```
    ; Print the value of kval, once per second.
    printk 1, kval
  endin
/* printk.orc */
```

```
/* printk.sco */
; Play Instrument #1 for 5 seconds.
i 1 0 5
e
/* printk.sco */
```

Its output should include lines like this:

```
i   1 time      0.00002:      0.00000
i   1 time      1.00002:     20.01084
i   1 time      2.00002:     40.02999
i   1 time      3.00002:     60.04914
i   1 time      4.00002:     79.93327
```

## See Also

*printk2* and *printks*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

Thanks goes to Luis Jure for pointing out a mistake wit the *itime* parameter.

# printk2

printk2 -- Prints a new value every time a control variable changes.

printk2

## Description

Prints a new value every time a control variable changes.

## Syntax

**printk2** *kvar* [, *inumspaces*]

## Initialization

*inumspaces* (optional, default=0) -- 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

**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.

## Examples

Here is an example of the printk2 opcode. It uses the files *printk2.orc* [examples/printk2.orc] and *printk2.sco* [examples/printk2.sco].

### Example 286. Example of the printk2 opcode.

```
/* printk2.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
  ; Change a value linearly from 0 to 10,
  ; over the period defined by p3.
  kval1 line 0, p3, 10

  ; If kval1 is greater than or equal to 5,
  ; then kval=2, else kval=1.
  kval2 = (kval1 >= 5 ? 2 : 1)

  ; Print the value of kval2 when it changes.
```

```
    printk2 kval2
endin
/* printk2.orc */

/* printk2.sco */
; Play Instrument #1 for 5 seconds.
i 1 0 5
e
/* printk2.sco */
```

Its output should include a line like this:

```
i1      1.00000
i1      2.00000
```

## See Also

*printk* and *printks*

## Credits

Author: Gabriel Maldonado  
Italy  
1998

Example written by Kevin Conder.

New in Csound version 3.48

# printks

printks -- Prints at k-rate using a printf() style syntax.

printks

## Description

Prints at k-rate using a printf() style syntax.

## Syntax

```
printks "string", itime [, kval1] [, kval2] [...]
```

## Initialization

*"string"* -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

*itime* -- time in seconds between printings.

## Performance

*kval1*, *kval2*, ... (optional) -- The k-rate values to be printed. These are specified in *"string"* with the standard C value specifier (%f, %d, etc.) in the order given.

In Csound version 4.23, you can use as many *kval* variables as you like. In versions prior to 4.23, you must specify 4 and only 4 kvals (using 0 for unused kvals).

*printks* prints numbers and text which can be i-time 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.

This opcode can be run on every k-cycle it is 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

All standard C language printf() control characters may be used. For example, if *kval1* = 153.26789 then some common formatting options are:

1. %f prints with full precision: 153.26789
2. %5.2f prints: 153.26
3. %d prints integers-only: 153
4. %c treats *kval1* as an ascii character code.

In addition to all the `printf()` codes, `printks` supports these useful character codes:

printks Code	Character Code
<code>\\r, \\R, %r, or %R</code>	return character ( <code>\r</code> )
<code>\\n, \\N, %n, %N</code>	newline character ( <code>\n</code> )
<code>\\t, \\T, %t, or %T</code>	tab character ( <code>\t</code> )
<code>%!</code>	semicolon character ( <code>;</code> ) This was needed because a “ <code>;</code> ” is interpreted as an comment.
<code>^</code>	escape character ( <code>0x1B</code> )
<code>^ ^</code>	caret character ( <code>^</code> )
<code>~</code>	ESC[ (escape+[ is the escape sequence for ANSI consoles)
<code>~~</code>	tilde ( <code>~</code> )

For more information about `printf()` formatting, consult any C language documentation.



### Note

Prior to version 4.23, only the `%f` format code was supported.

## Examples

Here is an example of the `printks` opcode. It uses the files *printks.orc* [examples/printks.orc] and *printks.sco* [examples/printks.sco].

### Example 287. Example of the `printks` opcode.

```
/* printks.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
  ; Change a value linearly from 0 to 100,
  ; over the period defined by p3.
  kup line 0, p3, 100
  ; Change a value linearly from 30 to 10,
  ; over the period defined by p3.
  kdown line 30, p3, 10

  ; Print the value of kup and kdown, once per second.
  printks "kup = %f, kdown = %f\\n", 1, kup, kdown
endin
/* printks.orc */

/* printks.sco */
; Play Instrument #1 for 5 seconds.
i 1 0 5
e
```

```
/* printks.sco */
```

Its output should include lines like this:

```
kup = 0.000000, kdown = 30.000000  
kup = 20.010843, kdown = 25.962524  
kup = 40.029991, kdown = 21.925049  
kup = 60.049141, kdown = 17.887573  
kup = 79.933266, kdown = 13.872493
```

## See Also

*printk2* and *printk*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

Thanks goes to Luis Jure for pointing out a mistake with the *itime* parameter.

Thanks to Matt Ingalls, updated the documentation for version 4.23.

# prints

prints -- Prints at init-time using a printf() style syntax.

prints

## Description

Prints at init-time using a printf() style syntax.

## Syntax

```
prints "string" [, kval1] [, kval2] [...]
```

## Initialization

"string" -- the text string to be printed. Can be up to 8192 characters and must be in double quotes.

## Performance

*kval1*, *kval2*, ... (optional) -- The k-rate values to be printed. These are specified in "string" with the standard C value specifier (%f, %d, etc.) in the order given. Use 0 for those which are not used.

*prints* is similar to the *printks* opcode except it operates at init-time instead of k-rate. For more information about output formatting, please look at *printks's documentation*.

## Examples

Here is an example of the prints opcode. It uses the files *prints.orc* [examples/prints.orc] and *prints.sco* [examples/prints.sco].

### Example 288. Example of the prints opcode.

```
/* prints.orc */
/* Written by Matt Ingalls, edited by Kevin Conder. */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Init-time print.
  prints "%2.3f\\t%!%!%!%!%!semicolons!\\n", 1234.56789
endin
/* prints.orc */
```

```
/* prints.sco */
/* Written by Matt Ingalls, edited by Kevin Conder. */
; Play instrument #1.
i 1 0 0.004
```



```
/* prints.sco */
```

Its output should include a line like this:

```
1234.568          ;;;;;;semicolons!
```

## See Also

*prints*

## Credits

Author: Matt Ingalls  
January 2003

# product

product -- Multiplies any number of a-rate signals.

product

## Description

Multiplies any number of a-rate signals.

## Syntax

```
ares product asig1, asig2 [, asig3] [...]
```

## Performance

*asig1, asig2, asig3, ...* -- a-rate signals to be multiplied.

## Credits

Author: Gabriel Maldonado  
Italy  
April 1999

New in Csound version 3.54

# pset

`pset` -- Defines and initializes numeric arrays at orchestra load time.

`pset`

## Description

Defines and initializes numeric arrays at orchestra load time.

## Syntax

```
pset icon1 [, icon2] [...]
```

## Initialization

*icon1, icon2, ...* -- preset values for a MIDI instrument

*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 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
```

## See Also

*strset*

# puts

puts -- Print a string constant or variable

puts

## Description

puts prints a string with an optional newline at the end whenever the trigger signal is positive and changes.

## Syntax

**puts** *Sstr*, *ktrig*[, *inonl*]

## Initialization

*Sstr* -- string to be printed

*inonl* (optional, defaults to 0) -- if non-zero, disables the default printing of a newline character at the end of the string

## Performance

*ktrig* -- trigger signal, should be valid at i-time. The string is printed at initialization time if *ktrig* is positive, and at performance time whenever *ktrig* is both positive and different from the previous value. Use a constant value of 1 to print once at note initialization.

## Credits

Author: Istvan Varga  
2005

# pvadd

`pvadd` -- Reads from a *pvoc* file and uses the data to perform additive synthesis.

`pvadd`

## 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.

## Syntax

ares **pvadd** ktmpnt, kfmod, ifilcod, ifn, ibins [, ibinoffset] [, ibinincr] [, i

## Initialization

*ifilcod* -- integer or character-string denoting a control-file derived from *pvanal* 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 indices 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

*ktmpnt* 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 indices 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 synthesize 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 use 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 (ie. 2000).

Expect to have to scale up the amplitudes by factors of 10-100, by the way.

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48, additional arguments version 3.56

# pvbufread

pvbufread -- Reads from a phase vocoder analysis file and makes the retrieved data available.

pvbufread

## Description

*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.

## Syntax

**pvbufread** ktimepnt, ifile

## Initialization

*ifile* -- the *pvoc* number (n in *pvoc.n*) or the name in quotes of the analysis file made using *pvanal*. (See *pvoc*.)

## Performance

*ktimepnt* -- the passage of time, in seconds, through this file. *ktimepnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

## Examples

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 re-synthesis 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
```

## See Also

*pvcross, pvinterp, pvread, tableseg, tablexseg*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997

## pvcross

`pvcross` -- Applies the amplitudes from one phase vocoder analysis file to the data from a second file.

`pvcross`

## Description

*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*.

## Syntax

ares **pvcross** *ktimpnt*, *kfmod*, *ifile*, *kampscale1*, *kampscale2* [, *ispecwp*]

## Initialization

*ifile* -- the *pvoc* number (n in *pvoc.n*) or the name in quotes of the analysis file made using *pvanal*. (See *pvoc*.)

*ispecwp* (optional, default=0) -- if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by *kfmod*. The default value is zero.

## Performance

*ktimpnt* -- the passage of time, in seconds, through this file. *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* -- a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

*kampscale1*, *kampscale2* -- used to scale the amplitudes stored in each frame of the phase vocoder analysis file. *kampscale1* scale the amplitudes of the data from the file read by the previously called *pvbufread*. *kampscale2* scale the amplitudes of the file named by *ifile*.

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.

## Examples

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
```

## See Also

*pvbufread, pvinterp, pvread, tableseg, tablexseg*

## Credits

Author: Richard Karpen  
Seattle, Wash  
1997

# pvinterp

`pvinterp` -- Interpolates between the amplitudes and frequencies of two phase vocoder analysis files.

`pvinterp`

## Description

*pvinterp* interpolates between the amplitudes and frequencies, on a bin by bin basis, of two phase vocoder analysis files (one from a previously called *pdbufread* 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.

## Syntax

ares **pvinterp** *ktimpnt*, *kfmod*, *ifile*, *kfreqscale1*, *kfreqscale2*, *kampscale1*, *kamp*

## Initialization

*ifile* -- the *pvoc* number (n in *pvoc.n*) or the name in quotes of the analysis file made using *pvanal*. (See *pvoc*.)

## Performance

*ktimpnt* -- the passage of time, in seconds, through this file. *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* -- a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

*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 *pdbufread* (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*, 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 1, the frequency values will be entirely those from the first file (read by the *pdbufread*), post scaling by the *kfreqscale1* argument. When the value of *kfreqinterp* is 0 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).

*kampinterp* works 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.

## Examples

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.

```
ptime1  line      0, p3, 3.5 ; used as index in the "oboe.pvoc" file
ptime2  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 ptime1, "oboe.pvoc"
apv      pvinterp ptime2,1,"clar.pvoc",1,1.065,1,.75,1-kinterp,1-kinterp
```

## See Also

*pvbufread*, *pvcross*, *pvread*, *tableseg*, *tablexseg*

## Credits

Author: Richard Karpen  
Seattle, Wash  
1997

# pvoc

pvoc -- Implements signal reconstruction using an fft-based phase vocoder.

pvoc

## Description

Implements signal reconstruction using an fft-based phase vocoder.

## Syntax

ares **pvoc** ktmpnt, kfmod, ifilcod [, ispecwp] [, iextractmode] [, ifreqlim] [,

## 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 files involved, which are read and held entirely in memory during computation but are shared by multiple calls (see also *lpread*).

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

*extractmode* (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 *extractmode* 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 *extractmode* will cause *pvadd* to synthesize only those components where the frequency difference between frames is less than *ifreqlim*. The default values for *extractmode* and *ifreqlim* are 0, in which case a simple resynthesis will be done. See examples under *pvadd* for how to use spectral extraction.

*igatefn* (optional) -- 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 indices 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.

## Performance

*ktmpnt* -- The passage of time, in seconds, through the analysis file. *ktmpnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

*kfmod* -- a control-rate transposition factor: a value of 1 incurs no transposition, 1.5 transposes up a perfect fifth, and .5 down an octave.

*pvoc* implements signal reconstruction using an fft-based phase vocoder. The control data stems from a precomputed analysis file with a known frame rate.

This implementation of *pvoc* was originally 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.

## See Also

*vpvoc*

## Credits

Authors: Dan Ellis and Richard Karpen  
Seattle, Wash  
1997

# pvread

`pvread` -- Reads from a phase vocoder analysis file and returns the frequency and amplitude from a single analysis channel or bin.

`pvread`

## 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.

## Syntax

`kfreq, kamp` **pvread** `ktimpnt, ifile, ibin`

## 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.

## 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* -- the passage of time, in seconds, through this file. *ktimpnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

## Examples

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
kfreq, kamp  pvread  ktime, "pvoc.file", 7 ; read
                                           ;data from 7th analysis bin.
asig        oscili   kamp, kfreq, 1      ; function 1
                                           ;is a stored sine
```

## See Also



*pvbufread, pvcross, pvinterp, tableseg, tablexseg*

## Credits

Author: Richard Karpen  
Seattle, Wash  
1997

# pvsadsyn

pvsadsyn -- Resynthesize using a fast oscillator-bank.

pvsadsyn

## Description

Resynthesize using a fast oscillator-bank.

## Syntax

ares **pvsadsyn** fsrc, inoscs, kfmod [, ibinoffset] [, ibinincr] [, iinit]

## Initialization

*inoscs* -- The number of analysis bins to synthesise. Cannot be larger than the size of fsrc (see *pvsinfo*), e.g. as created by *pvsanal*. Processing time is directly proportional to inoscs.

*ibinoffset* (optional, default=0) -- The first (lowest) bin to resynthesize, counting from 0 (default = 0).

*ibinincr* (optional) -- Starting from bin ibinoffset, resynthesize bins ibinincr apart.

*iinit* (optional) -- Skip reinitialization. This is not currently implemented for any of these opcodes, and it remains to be seen if it is even practical.

## Performance

*kfmod* -- Scale all frequencies by factor kfmod. 1.0 = no change, 2 = up one octave.

*pvsadsyn* is experimental, and implements the oscillator bank using a fast direct calculation method, rather than a lookup table. This takes advantage of the fact, empirically arrived at, that for the analysis rates generally used, (and presuming analysis using *pvsanal*, where frequencies in a bin change only slightly between frames) it is not necessary to interpolate frequencies between frames, only amplitudes. Accurate resynthesis is often contingent on the use of *pvsanal* with *iwinsize* = *ifftsize*\*2.

This opcode is the most likely to change, or be much extended, according to feedback and advice from users. It is likely that a full interpolating table-based method will be added, via a further optional *iarg*. The parameter list to *pvsadsyn* mimics that for *pvadd*, but excludes spectral extraction.

## Examples

```
; resynth the first 100 odd-numbered bins, with pitch scaling envelope.
kpch linseg 1,p3/3,1,p3/3,1.5,p3/3,1
aout pvsadsyn fsrc, 100,kpch,1,2
```

## See Also

*pvsynth*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsanal

**pvsanal** -- Generate an fsig from a mono audio source ain, using phase vocoder overlap-add analysis.

**pvsanal**

## Description

Generate an fsig from a mono audio source ain, using phase vocoder overlap-add analysis.

## Syntax

fsig **pvsanal** ain, ifftsize, ioverlap, iwinsize, iwintype [, iformat] [, iinit]

## Initialization

*ifftsize* -- The FFT size in samples. Need not be a power of two (though these are especially efficient), but must be even. Odd numbers are rounded up internally. *ifftsize* determines the number of analysis bins in fsig, as  $\text{ifftsize}/2 + 1$ . For example, where *ifftsize* = 1024, fsig will contain 513 analysis bins, ordered linearly from the fundamental to Nyquist. The fundamental of analysis (which in principle gives the lowest resolvable frequency) is determined as  $\text{sr}/\text{ifftsize}$ . Thus, for the example just given and assuming  $\text{sr} = 44100$ , the fundamental of analysis is 43.07Hz. In practice, due to the phase-preserving nature of the phase vocoder, the frequency of any bin can deviate bilaterally, so that DC components are recorded. Given a strongly pitched signal, frequencies in adjacent bins can bunch very closely together, around partials in the source, and the lowest bins may even have negative frequencies.

As a rule, the only reason to use a non power-of-two value for *ifftsize* would be to match the known fundamental frequency of a strongly pitched source. Values with many small factors can be almost as efficient as power-of-two sizes; for example: 384, for a source pitched at around low A=110Hz.

*ioverlap* -- The distance in samples (“hop size”) between overlapping analysis frames. As a rule, this needs to be at least  $\text{ifftsize}/4$ , e.g. 256 for the example above. *ioverlap* determines the underlying analysis rate, as  $\text{sr}/\text{ioverlap}$ . *ioverlap* does not require to be a simple factor of *ifftsize*; for example a value of 160 would be legal. The choice of *ioverlap* may be dictated by the degree of pitch modification applied to the fsig, if any. As a rule of thumb, the more extreme the pitch shift, the higher the analysis rate needs to be, and hence the smaller the value for *ioverlap*. A higher analysis rate can also be advantageous with broadband transient sounds, such as drums (where a small analysis window gives less smearing, but more frequency-related errors).

Note that it is possible, and reasonable, to have distinct fsigs in an orchestra (even in the same instrument), running at different analysis rates. Interactions between such fsigs is currently unsupported, and the fsig assignment opcode does not allow copying between fsigs with different properties, even if the only difference is in *ioverlap*. However, this is not a closed issue, as it is possible in theory to achieve crude rate conversion (especially with regard to in-memory analysis files) in ways analogous to time-domain techniques.

*iwinsize* -- The size in samples of the analysis window filter (as set by *iwintype*). This must be at least *ifftsize*, and can usefully be larger. Though other proportions are permitted, it is recommended that *iwinsize* always be an integral multiple of *ifftsize*, e.g. 2048 for the example above. Internally, the analysis window (Hamming, von Hann) is multiplied by a sinc function, so that amplitudes are zero at the boundaries between frames. The larger analysis window size has been found to be especially important for oscillator bank resynthesis (e.g. using *pvsadsyn*), as it has the effect of increasing the frequency resolution of the analysis, and hence the accuracy of the resynthesis. As noted above, *iwinsize* determines the overall latency of the analysis/resynthesis system. In many cases, and especially in the absence of pitch modifications, it will be found that setting *iwinsize*=*ifftsize* works very well, and offers the lowest latency.

*iwintype* -- The shape of the analysis window. Currently only two choices are implemented:

- 0 = Hamming window
- 1 = von Hann window

Both are also supported by the PVOC-EX file format. The window type is stored as an internal attribute of the fsig, together with the other parameters (see *pvsinfo*). Other types may be implemented later on (e.g. the Kaiser window, also supported by PVOC-EX), though an obvious alternative is to enable windows to be defined via a function table. The main issue here is the constraint of f-tables to power-of-two sizes, so this method does not offer a complete solution. Most users will find the Hamming window meets all normal needs, and can be regarded as the default choice.

*iformat* -- (optional) The analysis format. Currently only one format is implemented by this opcode:

- 0 = amplitude + frequency

This is the classic phase vocoder format; easy to process, and a natural format for oscillator-bank re-synthesis. It would be very easy (tempting, one might say) to treat an fsig frame not purely as a phase vocoder frame but as a generic additive synthesis frame. It is indeed possible to use an fsig this way, but it is important to bear in mind that the two are not, strictly speaking, directly equivalent.

Other important formats (supported by PVOC-EX) are:

- 1 = amplitude + phase
- 2 = complex (real + imaginary)

iformat is provided in case it proves useful later to add support for these other formats. Formats 0 and 1 are very closely related (as the phase is “wrapped” in both cases - it is a trivial matter to convert from one to the other), but the complex format might warrant a second explicit signal type (a “csig”) specifically for convolution-based processes, and other processes where the full complement of arithmetic operators may be useful.

*iinit* -- (optional) Skip reinitialization. This is not currently implemented for any of these opcodes, and it remains to be seen if it is even practical.

## Examples

```
ain    in                                ; live source
ffin   pvsanal   ain,1024,256,2048,0    ; analyse, using Hamming
ffout  pvsmaska  ffin,1,0.75            ; apply eq from f-table
aout   pvsynth   ffout                  ; and resynthesize
```

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsarp

pvsarp -- Arpeggiate the spectral components of a streaming pv signal.

pvsarp

## Description

This opcode arpeggiates spectral components, by amplifying one bin and attenuating all the others around it. Used with an LFO it will provide a spectral arpeggiator similar to Trevor Wishart's CDP program specarp.

## Syntax

fsig **pvsarp** fsigin, kbin, kdepth, kgain

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream

*kbin* -- target bin, normalised 0 - 1 (0Hz - Nyquist).

*kdepth* -- depth of attenuation of surrounding bins

*kgain* -- gain boost applied to target bin

## Examples

### Example 289. Example

```
asig in ; get the signal in
fsig pvsanal asig, 1024, 256, 1024, 1 ; analyse it
kbin oscili 0.1, 0.5, 1 ; ftable 1 in the 0-1 range
ftps pvsarp fsig, kbin+0.01, 0, 2 ; arpeggiate it (range 220.5 - 2425.5)
atps pvsynth ftps ; synthesise it

out atps
```

The example above shows a spectral arpeggiator working in the 220.5 - 2425.5 range (sr=44100). The LFO outputs a positive-only signal, so its ftable will be defined in the 0 - 1 range (a hanning window can be used, for instance).

## Credits

Author: Victor Lazzarini;  
April 2005

New plugin in version 5

April 2005.



## pvcross

pvcross -- Performs cross-synthesis between two source fsigs.

pvcross

## Description

Performs cross-synthesis between two source fsigs.

## Syntax

fsig **pvcross** fsrc, fdest, kamp1, kamp2

## Performance

The operation of this opcode is identical to that of *pvcross* (q.v.), except in using *fsigs* rather than analysis files, and the absence of spectral envelope preservation. The amplitudes from *fsrc* are applied to *fdest*, using scale factors *kamp1* and *kamp2* respectively. *kamp1* and *kamp2* must not exceed the range 0 to 1.

With this opcode, cross-synthesis can be performed on real-time audio input, by using *pvsanal* to generate *fsrc* and *fdest*. These must have the same format.

## Examples

```
kcross  linseg      0,p3/3,0,p3/3,1,p3/3,1 ; progressive cross-synthesis
fcross  pvcross     fsig1,fsig2,1-kcross,kcross
across  pvsynth     fcross
```

## Credits

Author: Richard Dobson  
August 2001

November 2003. Thanks to Kanata Motohashi, fixed the link to the *pvcross* opcode.

New in version 4.13

# pvscent

pvscent -- Calculate the spectral centroid of a signal.

pvscent

## Description

Calculate the spectral centroids of a signal from its discrete Fourier transform.

## Syntax

kcent **pvscent** fsig

## Performance

*kcent* -- the spectral centroid

*fsig* -- an input pv stream

## Examples

### Example 290. Example

```
ifftsize = 1024
iwtype = 1      /* cleaner with hanning window */
ipos = -0.8     /* to the left of the stereo image */
iwidth = 20     /* use peaks of 20 points around it */

a1  soundin "input.wav"

fsig pvsanal    a1, ifftsize, ifftsize/4, ifftsize, iwtype
kcen pvscent    fsig
adm  oscil      32000, kcent, 1

      out      adm
```

## Credits

Author: John ffitch;  
March 2005

New plugin in version 5

March 2005.

# pvsdemix

pvsdemix -- Spectral azimuth-based de-mixing of stereo sources.

pvsdemix

## Description

Spectral azimuth-based de-mixing of stereo sources, with a reverse-panning result. This opcode implements the Azimuth Discrimination and Resynthesis (ADResS) algorithm, developed by Dan Barry (Barry et Al. "Sound Source Separation Azimuth Discrimination and Resynthesis". DAFx'04, Univ. of Napoli). The source separation, or de-mixing, is controlled by two parameters: an azimuth position (kpos) and a subspace width (kwidth). The first one is used to locate the spectral peaks of individual sources on a stereo mix, whereas the second widens the 'search space', including/excluding the peaks around kpos. These two parameters can be used interactively to extract source sounds from a stereo mix. The algorithm is particularly successful with studio recordings where individual instruments occupy individual panning positions; it is, in fact, a reverse-panning algorithm.

## Syntax

*fsig* **pvsdemix** *fleft*, *fright*, *kpos*, *kwidth*, *ipoints*

## Performance

*fsig* -- output pv stream

*fleft* -- left channel input pv stream.

*fright* -- right channel pv stream.

*kpos* -- the azimuth target centre position, which will be de-mixed, from left to right ( $-1 \leq kpos \leq 1$ ). This is the reverse pan-pot control.

*kwidth* -- the azimuth subspace width, which will determine the number of points around kpos which will be used in the de-mixing process. ( $1 \leq kwidth \leq ipoints$ )

*ipoints* -- total number of discrete points, which will divide each pan side of the stereo image. This ultimately affects the resolution of the process.

## Examples

The example below takes a stereo input and passes through a de-mixing process revealing a source located at ipos +/- iwidth points. These parameters can be controlled in realtime (e.g. using FLTK widgets or MIDI) for an interactive search of sound sources.

### Example 291. Example

```
ifftsize = 1024
iwtype = 1      /* cleaner with hanning window */
ipos = -0.8    /* to the left of the stereo image */
iwidth = 20    /* use peaks of 20 points around it */

a1,a2 soundin "sinut.wav"

flc pvsanal a1, ifftsize, ifftsize/4, ifftsize, iwtype
```

```
frc pvsanal ar, ifftsize, ifftsize/4, ifftsize, iwtype
fdm pvsdemix flc, frc, kpos, kwidth, 100
adm pvsynth fdm

      outs adm,adm
```

## Credits

Author: Victor Lazzarini;  
January 2005

New plugin in version 5

January 2005.

# pvsfread

pvsfread -- Read a selected channel from a PVOC-EX analysis file.

pvsfread

## Description

Create an fsig stream by reading a selected channel from a PVOC-EX analysis file loaded into memory, with frame interpolation. Only format 0 files (amplitude+frequency) are currently supported. The operation of this opcode mirrors that of pvoc, but outputs an fsig instead of a resynthesized signal.

## Syntax

```
fsig pvsfread ktimpt, ifn [, ichan]
```

## Initialization

*ifn* -- Name of the analysis file. This must have the .pvx file extension.

A multi-channel PVOC-EX file can be generated using the extended *pvanal* utility.

*ichan* -- (optional) The channel to read (counting from 0). Default is 0.

## Performance

*ktimpt* -- Time pointer into analysis file, in seconds. See the description of the same parameter of *pvoc* for usage.

Note that analysis files can be very large, especially if multi-channel. Reading such files into memory will very likely incur breaks in the audio during real-time performance. As the file is read only once, and is then available to all other interested opcodes, it can be expedient to arrange for a dedicated instrument to preload all such analysis files at startup.

## Examples

```
idur  filelen    "test.pvx"           ; find dur of (stereo) analysis file
kpos  line       0,p3,idur             ; to ensure we process whole file
fsigr pvsfread   kpos,"test.pvx",1     ; create fsig from R channel
```

(NB: as this example shows, the filelen opcode has been extended to accept both old and new analysis file formats).

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsfreeze

pvsfreeze -- Freeze the amplitude and frequency time functions of a pv stream according to a control-rate trigger.

pvsfreeze

## Description

This opcodes 'freezes' the evolution of pvs stream by locking into steady amplitude and/or frequency values for each bin. The freezing is controlled, independently for amplitudes and frequencies, by a control-rate trigger, which switches the freezing 'on' if equal to or above 1 and 'off' if below 1.

## Syntax

fsig **pvsfreeze** fsigin, kfreeza, kfreezf

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*kfreeza* -- freezing switch for amplitudes. Freezing is on if above or equal to 1 and off if below 1.

*kfcf* -- freezing switch for frequencies. Freezing is on if above or equal to 1 and off if below 1.

## Examples

### Example 292. Example

```
asig in                                ; input
ktrig oscil 1.5, 0.25, 1               ; trigger
fin pvsanal asigl,1024,256,1024,0     ; pvoc analysis
fout pvsfreeze fin, abs(ktrig), abs(ktrig) ; regular 'freeze' of spectra
aout pvsynth fsigout                  ; pvoc synthesis
```

In the example above the input signal will be regularly 'frozen' for a short while, as the trigger rises above 1 about every two seconds.

## Credits

Author: Victor Lazzarini;  
May 2006

New plugin in version 5

May 2006.

# pvsftr

pvsftr -- Reads amplitude and/or frequency data from function tables.

pvsftr

## Description

Reads amplitude and/or frequency data from function tables.

## Syntax

**pvsftr** *fsrc*, *ifna* [, *ifnf*]

## Initialization

*ifna* -- A table, at least inbins in size, that stores amplitude data. Ignored if *ifna* = 0

*ifnf* (optional) -- A table, at least inbins in size, that stores frequency data. Ignored if *ifnf* = 0

## Performance

*fsrc* -- a PVOC-EX formatted source.

Enables the contents of *fsrc* to be exchanged with function tables for custom processing. Except when the frame overlap equals *ksmps* (which will generally not be the case), the frame data is not updated each control period. The data in *ifna*, *ifnf* should only be processed when *kflag* is set to 1. To process only frequency data, set *ifna* to zero.

As the function tables are required only to store data from *fsrc*, there is no advantage in defining them in the score, and they should generally be created in the instrument, using *ftgen*.

By exporting amplitude data, say, from one fsig and importing it into another, basic cross-synthesis (as in *pvsftr*) can be performed, with the option to modify the data beforehand using the table manipulation opcodes.

Note that the format data in the source fsig is not written to the tables. This therefore offers a means of transferring amplitude and frequency data between non-identical fsigs. Used this way, these opcodes become potentially pathological, and can be relied upon to produce unexpected results. In such cases, resynthesis using *pvsadsyn* would almost certainly be required.

To perform a straight copy from one fsig to another one of identical format, the conventional assignment syntax can be used:

```
fsig1 = fsig2
```

It is not necessary to use function tables in this case.

## Examples

```
ifn      ftgen      0,0,inbins,10,1      ; make ftable
kflag    pvsftw     fsrc,ifn             ; export amps to table,
kamp     init       0
```

```
if      kflag==0    kgoto contin    ; only proc when frame is ready
; kill lowest bins, for obvious effect
        tablew      kamp,1,ifn
        tablew      kamp,2,ifn
        tablew      kamp,3,ifn
        tablew      kamp,4,ifn
; read modified data back to fsrc
        pvsftr      fsrc,ifn
contin:
; and resynth
aout    pvsynth     fsrc
```

## See Also

*pvsftw*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13



# pvsftw

pvsftw -- Writes amplitude and/or frequency data to function tables.

pvsftw

## Description

Writes amplitude and/or frequency data to function tables.

## Syntax

kflag **pvsftw** fsrc, ifna [, ifnf]

## Initialization

*ifna* -- A table, at least inbins in size, that stores amplitude data. Ignored if ifna = 0

*ifnf* -- A table, at least inbins in size, that stores frequency data. Ignored if ifnf = 0

## Performance

*kflag* -- A flag that has the value of 1 when new data is available, 0 otherwise.

*fsrc* -- a PVOC-EX formatted source.

Enables the contents of *fsrc* to be exchanged with function tables, for custom processing. Except when the frame overlap equals *ksmps* (which will generally not be the case), the frame data is not updated each control period. The data in *ifna*, *ifnf* should only be processed when *kflag* is set to 1. To process only frequency data, set *ifna* to zero.

As the functions tables are required only to store data from *fsrc*, there is no advantage in defining them in the score. They should generally be created in the instrument using *ftgen*.

By exporting amplitude data, say, from one fsig and importing it into another, basic cross-synthesis (as in *pvsccross*) can be performed, with the option to modify the data beforehand using the table manipulation opodes.

Note that the format data in the source fsig is not written to the tables. This therefore offers a means of transferring amplitude and frequency data between non-identical fsigs. Used this way, these opcodes become potentially pathological, and can be relied upon to produce unexpected results. In such cases, resynthesis using *pvsadsyn* would almost certainly be required.

To perform a straight copy from one fsig to another one of identical format, the conventional assignment syntax can be used:

```
fsig1 = fsig2
```

It is not necessary to use function tables in this case.

## Examples

```
ifn      ftgen      0,0,inbins,10,1      ; make ftable
```

```
kflag    pvsftw    fsrc,ifn          ; export amps to table,
kamp     init      0
if       kflag==0  kgoto contin      ; only proc when frame is ready
; kill lowest bins, for obvious effect
        tablew    kamp,1,ifn
        tablew    kamp,2,ifn
        tablew    kamp,3,ifn
        tablew    kamp,4,ifn
; read modified data back to fsrc
        pvsftr     fsrc,ifn
contin:
; and resynth
aout     pvsynth   fsrc
```

## See Also

*pvsftr*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsifd

pvsifd -- Instantaneous Frequency Distribution, magnitude and phase analysis.

pvsifd

## Description

The pvsifd opcode takes an input a-rate signal and performs an Instantaneous Frequency, magnitude and phase analysis, using the STFT and pvsifd (Instantaneous Frequency Distribution), as described in Lazzarini et al, "Time-stretching using the Instantaneous Frequency Distribution and Partial Tracking", Proc.of ICMC05, Barcelona. It generates two PV streaming signals, one containing the amplitudes and frequencies (a similar output to pvsanal) and another containing amplitudes and unwrapped phases.

## Syntax

```
ffr,fphs pvsifd ain, ifftsize, ihopsize, iwintype[,iscal]
```

## Performance

*ffr* -- output pv stream in AMP\_FREQ format

*fphs* -- output pv stream in AMP\_PHASE format

*ifftsize* -- FFT analysis size, must be power-of-two and integer multiple of the hopsize.

*ihopsize* -- hopsize in samples

*iwintype* -- window type (0: Hamming, 1: Hanning)

*iscal* -- amplitude scaling (defaults to 1).

## Examples

### Example 293. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; pvsifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
aout resyn fst, 1, 1.5, 500, 1 ; resynthesis (up a 5th)
out aout
```

The example above shows the pvsifd analysis feeding into partial tracking and cubic-phase additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini;  
June 2005

New plugin in version 5

November 2004.

# pvsinfo

pvsinfo -- Get information from a PVOC-EX formatted source.

pvsinfo

## Description

Get format information about fsrc, whether created by an opcode such as pvsanal, or obtained from a PVOC-EX file by pvsfread. This information is available at init time, and can be used to set parameters for other pvs opcodes, and in particular for creating function tables (e.g. for pvsftw), or setting the number of oscillators for pvsadsyn.

## Syntax

*ioverlap*, *inumbins*, *iwinsize*, *iformat* **pvsinfo** *fsrc*

## Initialization

*ioverlap* -- The stream overlap size.

*inumbins* -- The number of analysis bins (amplitude+frequency) in fsrc. The underlying FFT size is calculated as (inumbins -1) \* 2.

*iwinsize* -- The analysis window size. May be larger than the FFT size.

*iformat* -- The analysis frame format. If fsrc is created by an opcode, iformat will always be 0, signifying amplitude+frequency. If fsrc is defined from a PVOC-EX file, iformat may also have the value 1 or 2 (amplitude+phase, complex).

## Examples

```
fin      pvsfread  "test.pvx"      ; import pvocex file
iovl,inb,iws,ifmt pvsinfo  fin      ; get inumbins info
ifn      ftgen     0,0,inb,10,1    ; and create f-table
```

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

# pvsinit

pvsinit -- Initialise a spectral (f) variable to zero.

pvsinit

## Description

Fermorms the equavent to an init operation on an f-variable.

## Syntax

```
fsig pvsinit isize
```

## Performance

*fsig* -- output pv stream set to zero.

*isize* -- size of the DFT frame.

## Examples

### Example 294. Example

```
fsig  pvsinit    1024
```

## Credits

Author: Victor Lazzarini;  
November 2004

New plugin in version 5

November 2004.

# pvsmaska

pvsmaska -- Modify amplitudes using a function table, with dynamic scaling.

pvsmaska

## Description

Modify amplitudes of fsrc using function table, with dynamic scaling.

## Syntax

fsig **pvsmaska** fsrc, ifn, kdepth

## Initialization

*ifn* -- The f-table to use. Given fsrc has N analysis bins, table ifn must be of size N or larger. The table need not be normalized, but values should lie within the range 0 to 1. It can be supplied from the score in the usual way, or from within the orchestra by using *pvsinfo* to find the size of fsrc, (returned by pvsinfo in inbins), which can then be passed to ftgen to create the f-table.

## Performance

*kdepth* -- Controls the degree of modification applied to fsrc, using simple linear scaling. 0 leaves amplitudes unchanged, 1 applies the full profile of ifn.

Note that power-of-two FFT sizes are particularly convenient when using table-based processing, as the number of analysis bins (inbins) is then a power-of-two plus one, for which an exactly matching f-table can be created. In this case it is important that the f-table be created with a size of inbins, rather than as a power of two, as the latter will copy the first table value to the guard point, which is inappropriate for this opcode.

## Examples

**Example 295. Example (using score-supplied f-table, assuming fsig fftsize = 1024)**

```
; score f-table using cubic spline to define shaped peaks
f1 0 513 8 0 2 1 3 0 4 1 6 0 10 1 12 0 16 1 32 0 1 0 436 0

asig buzz      20000,199,50,3      ; pulsewave source
fsig pvsanal   asig,1024,256,1024,0 ; create fsig
kmod linseg    0,p3/2,1,p3/2,0     ; simple control sig

fsig pvsmaska  fsig,2,kmod          ; apply weird eq to fsig
aout pvsynth   fsig                ; resynthesize,
      dispfft   aout,0.1,1024      ; and view the effect
```

This also illustrates that the usual Csound behaviour applies to fsigs; the same name can be used for both input and output.

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13



# pvsynth

pvsynth -- Resynthesise using a FFT overlap-add.

pvsynth

## Description

Resynthesise using a FFT overlap-add.

## Syntax

ares **pvsynth** fsrc, [iinit]

## Performance

*ares* -- output audio signal

*fsrc* -- input signal

*iinit* -- not yet implemented.

## Examples

**Example 296. Example (using score-supplied f-table, assuming fsig fftsize = 1024)**

```
; score f-table using cubic spline to define shaped peaks
f1 0 513 8 0 2 1 3 0 4 1 6 0 10 1 12 0 16 1 32 0 1 0 436 0

asig  buzz      20000,199,50,3          ; pulsewave source
fsig  pvsanal   asig,1024,256,1024,0    ; create fsig
kmod  linseg    0,p3/2,1,p3/2,0         ; simple control sig

fsig  pvsmaska  fsig,2,kmod             ; apply weird eq to fsig
aout  pvsynth   fsig                   ; resynthesize,
      dispfft   aout,0.1,1024          ; and view the effect
```

This also illustrates that the usual Csound behaviour applies to fsigs; the same name can be used for both input and output.

## See Also

*pvsadsyn*

## Credits

Author: Richard Dobson  
August 2001

New in version 4.13

February 2004. Thanks to a note from Francisco Vila, updated the example.

# pvscale

pvscale -- Scale the frequency components of a pv stream.

pvscale

## Description

Scale the frequency components of a pv stream, resulting in pitch shift. Output amplitudes can be optionally modified in order to attempt formant preservation.

## Syntax

fsig **pvscale** fsigin, kscal[, ikeepform, igain]

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream

*kscal* -- scaling ratio.

*ikeepform* -- attempt to keep input signal -- -- formants; 0: do not keep formants; 1: keep formants by imposing original amps; 2: keep formants by filtering using the original spec envelope (defaults to 0).

*igain* -- amplitude scaling (defaults to 1).

The quality of the pitch shift will be improved with the use of a Hanning window in the pvoc analysis. Formant preservation is only successful with strong-formant sounds, such as voices and certain instrumental sounds, but also can be used for interesting transformation effects.

## Examples

### Example 297. Example

```
asig  in                                ; get the signal in

fsig  pvssanal  asig, 1024, 256, 1024, 1 ; analyse it
ftps  pvsscale  fsig, 1.5, 1, 2          ; transpose it keeping formants
atps  pvsynth  ftps                      ; synthesise it

adp   delayr .1                          ; delay original signal
adel  deltapi 1024                        ; by 1024 samples
      delayw  asig

      out atps+adel                      ; add transposed and original
```

The example above shows a vocal harmoniser. The delay is necessary to time-align the signals, as the analysis-synthesis process will imply a delay of 1024 samples between the analysis input and the synthesis output.

## Credits

Author: Victor Lazzarini;  
November 2004

New plugin in version 5

November 2004.

# pvshift

pvshift -- Shift the frequency components of a pv stream, stretching/compressing its spectrum.

pvshift

## Description

Shift the frequency components of a pv stream, stretching/compressing its spectrum.

## Syntax

*fsig* **pvshift** *fsigin*, *kshift*, *klowest*[, *ikeepform*, *igain*]

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream

*kshift* -- shift amount (in Hz, positive or negative).

*klowest* -- lowest frequency to be shifted.

*ikeepform* -- attempt to keep input signal formants; 0: do not keep formants; 1: keep formants by imposing original amps; 2: keep formants by filtering using the original spec envelope (defaults to 0).

*igain* -- amplitude scaling (defaults to 1).

This opcode will shift the components of a pv stream, from a certain frequency upwards, up or down a fixed amount (in Hz). It can be used to transform a harmonic spectrum into an inharmonic one. The *ikeepform* flag can be used to try and preserve formants for possibly interesting and unusual spectral modifications.

## Examples

### Example 298. Example

```
asig in ; get the signal in
fsig pvsanal asig, 1024, 256, 1024, 1 ; analyse it
ftps pvshift fsig, 100, 0 ; add 100 Hz to each component
atps pvsynth ftps ; synthesise it
```

Depending on the input, this will transform a pitched sound into an inharmonic, bell-like sound.

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

Nivember 2004.

## pvmix

pvmix -- Mix 'seamlessly' two pv signals.

pvmix

## Description

Mix 'seamlessly' two pv signals. This opcode combines the most prominent components of two pvoc streams into a single mixed stream.

## Syntax

fsig **pvmix** fsigin1, fsigin2

## Performance

*fsig* -- output pv stream

*fsigin1* -- input pv stream.

*fsigin2* -- input pv stream, which must have same format as fsigin1.

## Examples

### Example 299. Example

```
fsig1 pvsanal asig1,1024,256,1024,0 ; pvoc analysis
fsig2 pvsanal asig2,1024,256,1024,0
fsigout pvmix fsig1, fsig2 ; mix signals
aout pvsynth fsigout ; pvoc synthesis
```

Depending on the input, this will transform a pitched sound into an inharmonic, bell-like sound.

## Credits

Author: Victor Lazzarini  
November 2004

New plugin in version 5

November 2004.

# pvsmooth

pvsmooth -- Smooth the amplitude and frequency time functions of a pv stream using parallel 1st order lowpass IIR filters with time-varying cutoff frequency.

pvsmooth

## Description

Smooth the amplitude and frequency time functions of a pv stream using a 1st order lowpass IIR with time-varying cutoff frequency. This opcode uses the same filter as the 'tone' opcode, but this time acting separately on the amplitude and frequency time functions that make up a pv stream. The cutoff frequency parameter runs at the control-rate, but unlike tone and tonek, it is not specified in Hz, but as fractions of 1/2 frame-rate (actually the pv stream sampling rate), which is easier to understand. This means that the highest cutoff frequency is 1 and the lowest 0; the lower the frequency the smoother the functions and more pronounced the effect will be. This opcode produces effects that are more or less similar to pvsblur, but with two important differences: 1. smoothing of amplitudes and frequencies use separate sets of filters; and 2. there is no increase in computational cost when higher amounts of 'blurring' (smoothing) are desired.

## Syntax

*fsig* **pvsmooth** *fsigin*, *kacf*, *kfcf*

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*kacf* -- amount of cutoff frequency for amplitude function filtering, between 0 and 1, in fractions of 1/2 frame-rate.

*kfcf* -- amount of cutoff frequency for frequency function filtering, between 0 and 1, in fractions of 1/2 frame-rate.

## Examples

### Example 300. Example

```
asig in                                ; input
fin  pvstanal asigl,1024,256,1024,0 ; pvoc analysis
fout pvsmooth fin, 0.01, 0.01       ; smooth with cf at 1% of 1/2 frame-rate (ca
aout pvsynth fsigout                ; pvoc synthesis
```

In the example above the input signal will be smoothed/blurred by pvsmooth with a cutoff frequency of 1% of 1/2 frame-rate (which is about 172Hz, so the cf is about 8.6Hz) .

## Credits



Author: Victor Lazzarini;  
May 2006

New plugin in version 5

May 2006.

# pvsfilter

pvsfilter -- Multiply amplitudes of a pvoc stream by those of a second pvoc stream, with dynamic scaling.

pvsfilter

## Description

Multiply amplitudes of a pvoc stream by those of a second pvoc stream, with dynamic scaling.

## Syntax

*fsig* **pvsfilter** *fsigin*, *fsigfil*, *kdepth*[, *igain*]

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*fsigfil* -- filtering pvoc stream.

*kdepth* -- controls the depth of filtering of *fsigin* by *fsigfil* .

*igain* -- amplitude scaling (optional, defaults to 1).

Here the input pvoc stream amplitudes are modified by the filtering stream, keeping its frequencies intact. As usual, both signals have to be in the same format.

## Examples

### Example 301. Example

```
kfreq  expon 500, p3, 4000          ; 3-octave sweep
kdepth  linseg 1, p3/2, 0.5, p3/2, 1 ; varying filter depth

asig  in                               ; input
afil  oscili 1, kfreq, 1              ; filter t-domain signal

fin  pvsanal asig1,1024,256,1024,0 ; pvoc analysis
fil  pvsanal asig2,1024,256,1024,0
fout  pvsfilter fin, fout, kdepth    ; filter signal
aout  pvsynth fsigout                ; pvoc synthesis
```

In the example above the filter curve will depend on the spectral envelope of *afil*; in the simple case of a sinusoid, it will be equivalent to a narrowband band-pass filter.

## Credits

Author: Victor Lazzarini;

November 2004

New plugin in version 5

November 2004.

# pvsblur

pvsblur -- Average the amp/freq time functions of each analysis channel for a specified time.

pvsblur

## Description

Average the amp/freq time functions of each analysis channel for a specified time (truncated to number of frames). As a side-effect the input pvoc stream will be delayed by that amount.

## Syntax

fsig **pvsblur** fsigin, kblurtime, imaxdel

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*kblurtime* -- time in secs during which windows will be averaged .

*imaxdel* -- maximum delay time, used for allocating memory used in the averaging operation.

This opcode will blur a pvstream by smoothing the amplitude and frequency time functions (a type of low-pass filtering); the amount of blur will depend on the length of the averaging period, larger blur times will result in a more pronounced effect.

## Examples

### Example 302. Example

```
asig  in                                ; get the signal in
fsig  pvsanal    asig, 1024, 256, 1024, 1 ; analyse it
ftps  pvsblur    fsig, 0.2, 0.2           ; blur it for 200 ms
atps  pvsynth    ftps                     ; synthesise it
```

## Credits

Author: Victor Lazzarini;  
November 2004

New plugin in version 5

November 2004.

# pvstencil

pvstencil -- Transforms a pvoc stream according to a masking function table.

pvstencil

## Description

Transforms a pvoc stream according to a masking function table; if the pvoc stream amplitude falls below the value of the function for a specific pvoc channel, it applies a gain to that channel.

The pvoc stream amplitudes are compared to a masking table, if they fall below the table values, they are scaled by *kgain*. Prior to the operation, table values are scaled by *klevel*, which can be used as masking depth control.

Tables have to be at least  $\text{fftsize}/2$  in size; for most GENS it is important to use an extended-guard point (size power-of-two plus one), however this is not necessary with GEN43.

One of the typical uses of *pvstencil* would be in noise reduction. A noise print can be analysed with *pvanal* into a PVOCEX file and loaded in a table with GEN43. This then can be used as the masking table for *pvstencil* and the amount of reduction would be controlled by *kgain*. Skipping post-normalisation will keep the original noise print average amplitudes. This would provide a good starting point for a successful noise reduction (so that *klevel* can be generally set to close to 1).

Other possible transformation effects are possible, such as filtering and 'inverse-masking'.

## Syntax

*fsig* **pvstencil** *fsigin*, *kgain*, *klevel*, *iftable*

## Performance

*fsig* -- output pv stream

*fsigin* -- input pv stream.

*kgain* -- 'stencil' gain.

*klevel* -- masking function level (scales the ftable prior to 'stenciling').

*iftable* -- masking function table.

## Examples

### Example 303. Example

```
fsig    pvsanal    asig, 1024, 256, 1024, 1
fclean  pvstencil  fsig, 0, 1, 1
aclean  pvsynth   fclean
```

## Credits

Author: Victor Lazzarini;  
November 2004

New plugin in version 5

Nivember 2004.

# pvsvoc

pvsvoc -- Combine the spectral envelope of one fsig with the excitation (frequencies) of another.

pvsvoc

## Description

This opcode provides support for cross-synthesis of amplitudes and frequencies. It takes the amplitudes of one input fsig and combines with frequencies from another. It is a spectral version of the well-known channel vocoder.

## Syntax

fsig **pvsvoc** famp, fexc, kdepth, kgain

## Performance

*fsig* -- output pv stream

*famp* -- input pv stream from which the amplitudes will be extracted

*fexc* -- input pv stream from which the frequencies will be taken

*kdepth* -- depth of effect, affecting how much of the frequencies will be taken from the second fsig: 0, the output is the famp signal, 1 the output is the famp amplitudes and fexc frequencies.

*kgain* -- gain boost/attenuation applied to the output.

## Examples

### Example 304. Example

```
asig  in                                ; get the signal in
asyn  oscili 16000, 150, 1              ; excitation signal

famp  pvsanal  asig, 1024, 256, 1024, 1 ; analyse in signal
fexc  pvsanal  asyn, 1024, 256, 1024, 1 ; analyse excitation signal
ftps  pvsvoc   famp, fexc, 1, 1         ; cross it
atps  pvsynth  ftps                    ; synthesise it

      out atps
```

The example above shows a typical cross-synthesis operation. The input signal (say a vocal sound) is used for its amplitude spectrum. An oscillator with an arbitrary complex waveform produces the excitation signal, giving the vocal sound its pitch.

## Credits

Author: Victor Lazzarini;  
April 2005

New plugin in version 5

April 2005.



## pyassign Opcodes

pyassign Opcodes -- Assign the value of the given Csound variable to a Python variable possibly destroying its previous content.

pyassign

### Syntax

**pyassign** "variable", kvalue

**pyassigni** "variable", ivalue

**pylassign** "variable", kvalue

**pylassigni** "variable", ivalue

**pyassignt** ktrigger, "variable", kvalue

**pylassignt** ktrigger, "variable", kvalue

### Description

Assign the value of the given Csound variable to a Python variable possibly destroying its previous content. The resulting Python object will be a float.

### Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

## pycall Opcodes

pycall Opcodes -- Invoke the specified Python callable at k-time and i-time (i suffix), passing the given arguments. The call is performed in the global environment, and the result (the returning value) is copied into the Csound output variables specified.

pycall

### Syntax

kresult	pycall	"callable", karg1, ...
kresult1, kresult2	pycall1	"callable", karg1, ...
kr1, kr2, kr3	pycall2	"callable", karg1, ...
kr1, kr2, kr3, kr4	pycall3	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5	pycall4	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6	pycall5	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7	pycall6	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8	pycall7	"callable", karg1, ...
	pycall8	"callable", karg1, ...
	pycallt	ktrigger, "callable", karg1, ...
kresult	pycall1t	ktrigger, "callable", karg1, ...
kresult1, kresult2	pycall2t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3	pycall3t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4	pycall4t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5	pycall5t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6	pycall6t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7	pycall7t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8	pycall8t	ktrigger, "callable", karg1, ...
	pycalli	"callable", karg1, ...
iresult	pycall1i	"callable", iarg1, ...
iresult1, iresult2	pycall2i	"callable", iarg1, ...
ir1, ir2, ir3	pycall3i	"callable", iarg1, ...
ir1, ir2, ir3, ir4	pycall4i	"callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5	pycall5i	"callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5, ir6	pycall6i	"callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5, ir6, ir7	pycall7i	"callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5, ir6, ir7, ir8	pycall8i	"callable", iarg1, ...
pycalln "callable", nresults, kresult1, ..., kresultn, karg1, ...		
pycallni "callable", nresults, iresult1, ..., iresultn, iarg1, ...		
	pylcall	"callable", karg1, ...
kresult	pylcall1	"callable", karg1, ...
kresult1, kresult2	pylcall2	"callable", karg1, ...
kr1, kr2, kr3	pylcall3	"callable", karg1, ...
kr1, kr2, kr3, kr4	pylcall4	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5	pylcall5	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6	pylcall6	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7	pylcall7	"callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8	pylcall8	"callable", karg1, ...
	pylcallt	ktrigger, "callable", karg1, ...
kresult	pylcall1t	ktrigger, "callable", karg1, ...
kresult1, kresult2	pylcall2t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3	pylcall3t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4	pylcall4t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5	pylcall5t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6	pylcall6t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7	pylcall7t	ktrigger, "callable", karg1, ...
kr1, kr2, kr3, kr4, kr5, kr6, kr7, kr8	pylcall8t	ktrigger, "callable", karg1, ...

```
iresult          pylcalli    "callable", karg1, ...
iresult1, iresult2 pylcall1i  "callable", iarg1, ...
ir1, ir2, ir3     pylcall2i  "callable", iarg1, ...
ir1, ir2, ir3, ir4 pylcall3i  "callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5 pylcall4i  "callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5, ir6 pylcall5i  "callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5, ir6, ir7 pylcall6i  "callable", iarg1, ...
ir1, ir2, ir3, ir4, ir5, ir6, ir7, ir8 pylcall7i  "callable", iarg1, ...
pylcalln         "callable", nresults, kresult1, ..., kresultn, karg1, ...
pylcallni        "callable", nresults, iresult1, ..., iresultn, iarg1, ...
```

## Description

This family of opcodes call the specified Python callable at k-time and i-time (i suffix), passing the given arguments. The call is performed in the global environment and the result (the returning value) is copied into the Csound output variables specified.

They pass any number of parameters which are cast to float inside the Python interpreter.

The *pycall/pycalli*, *pycall1/pycall1i* ... *pycall8/pycall8i* opcodes can accomodate for a number of results ranging from 0 to 8 according to their numerical prefix (0 is omitted).

The *pycalln/pycallni* opcodes can accomodate for any number of results: the callable name is followed by the number of output arguments, then come the list of Csound output variable and the list of parameters to be passed.

The returning value of the callable must be None for *pycall* or *pycalli*, a float for *pycall1i* or *pycall1i* and a tuple (with proper size) of floats for the *pycall2/pycall2i* ... *pycall8/pycall8i* and *pycalln/pycallni* opcodes.

### Example 305. Calling a C or Python function

Supposing we have previously defined or imported a function named `get_number_from_pool` as:

```
from random import random, choice

# a pool of 100 numbers
pool = [i ** 1.3 for i in range(100)]

def get_number_from_pool(n, p):
    # substitute an old number with the new number?
    if random() < p:
        i = choice(range(len(pool)))
        pool[i] = n

    # return a random number from the pool
    return choice(pool)
```

then the following orchestra code

```
k2    pycall11 "get_number_from_pool", k1, p6
```

would set k2 randomly from a pool of numbers changing in time. You can pass new pools elements and control the change rate from the orchestra.

### Example 306. Calling a Function Object

A more generic implementation of the previous example makes use of a simple function object:

```
from random import random, choice

class GetNumberFromPool:
    def __init__(self, e, begin=0, end=100, step=1):
        self.pool = [i ** e for i in range(begin, end, step)]

    def __call__(self, n, p):
        # substitute an old number with the new number?
        if random() < p:
            i = choice(range(len(pool)))
            pool[i] = n

        # return a random number from the pool
        return choice(pool)

get_number_from_pool1 = GetNumberFromPool(1.3)
get_number_from_pool2 = GetNumberFromPool(1.5, 50, 250, 2)
```

Then the following orchestra code:

```
k2    pycall1 "get_number_from_pool1", k1, p6
k4    pycall1 "get_number_from_pool2", k3, p7
```

would set k2 and k3 randomly from a pool of numbers changing in time. You can pass new pools elements (here k1 and k3) and control the change rate (here p6 and p7) from the orchestra.

As you can see in the first snippet, you can customize the initialization of the pool as well as create several pools.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

## pyeval Opcodes

pyeval Opcodes -- Evaluate a generic Python expression and store the result in a Csound variable at k-time or i-time (i suffix).

pyeval

### Syntax

```
kresult pyeval "expression"
```

```
irestult pyevali "expression"
```

```
kresult pyleval "expression"
```

```
irestult pylevali "expression"
```

```
kresult pyevalt ktrigger, "expression"
```

```
kresult pylevalt ktrigger, "expression"
```

### Description

These opcodes evaluate a generic Python expression and store the result in a Csound variable at k-time or i-time (i suffix).

The expression must evaluate in a float or an object that can be cast to a float.

They can be used effectively to transfer data from a Python object into a Csound variable.

### Example of the pyleval Opcode Group

The code:

```
k1          pyleval      "v1"
```

will copy the content of the Python variable v1 into the Csound variable k1 at each k-time.

### Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

## pyexec Opcodes

pyexec Opcodes -- Execute a script from a file at k-time or i-time (i suffix).

pyexec

### Syntax

```
pyexec "filename"
```

```
pyexeci "filename"
```

```
pylexec "filename"
```

```
pylexeci "filename"
```

```
pyexec ktrigger, "filename"
```

```
plyexec ktrigger, "filename"
```

### Description

Execute a script from a file at k-time or i-time (i suffix).

This is not the same as calling the script with the `system()` call, since the code is executed by the embedded interpreter.

The code contained in the specified file is executed in the global environment for opcodes `pyexec` and `pyexeci` and in the private environment for the opcodes `pylexec` and `pylexeci`.

These opcodes perform no message passing. However, since the statement has access to the main namespace and the private namespace, it can interact with objects previously created in that environment.

The "local" version of the *pyexec* opcodes are useful when the code ran by different instances of an instrument should not interact.

## Example of the pyexec Opcode Group

### Example 307. Orchestra (pyexec.orc)

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

;If you're not running CsoundVST you need the following line
;to initialize the python interpreter
;pyinit

    pyruni "import random"

    pyexeci "pyexec1.py"
```

```
instr 1
    pyexec          "pyexec2.py"
    pylexeci        "pyexec3.py"
    pylexec         "pyexec4.py"
endin
```

### **Example 308. Score (pyexec.sco)**

```
i1 0 0.01
i1 0 0.01
```

### **Example 309. The pyexec1.py Script**

```
import time, os

print
print "Welcome to Csound!"

try:
    s = ', %s?' % os.getenv('USER')
except:
    s = '?'

print 'What sound do you want to hear today%s' % s
answer = raw_input()
```

### **Example 310. The pyexec2.py script**

```
print 'your answer is "%s"' % answer
```

### **Example 311. The pyexec3.py script**

```
message = 'a private random number: %f' % random.random()
```

### **Example 312. The pyexec4.py script**

```
print message
```

If I run this example on my machine I get something like:

```
Using ../../csound.xmg
Csound Version 4.19 (Mar 23 2002)
Embedded Python interpreter version 2.2
orchname: pyexec.orc
scorename: pyexec.sco
sorting score ...
... done
orch compiler:
11 lines read
      instr 1
Csound Version 4.19 (Mar 23 2002)
displays suppressed

Welcome to Csound!
What sound do you want to hear today, maurizio?
```

then I answer

a sound

then Csound continues with the normal performance

```
your answer is "a sound"
a private random number: 0.884006
new alloc for instr 1:
your answer is "a sound"
a private random number: 0.884006
your answer is "a sound"
a private random number: 0.889868
your answer is "a sound"
a private random number: 0.884006
your answer is "a sound"
a private random number: 0.889868
your answer is "a sound"
a private random number: 0.884006
your answer is "a sound"
...
```

In the same instrument a message is created in the private namespace and printed, appearing different for each instance.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.



## pyinit Opcodes

pyinit Opcodes -- Initialize the Python interpreter.

pyinit

## Syntax

**pyinit**

## Description

In the command-line version of Csound, you must first invoke the *pyinit* opcode in the orchestra header to initialize the Python interpreter, before using any of the other Python opcodes.

But if you use the Python opcodes in the CsoundVST version of Csound, you need not invoke *pyinit*, because CsoundVST automatically initializes the Python interpreter for you. In addition, CsoundVST automatically creates a Python interface to the Csound API, in the form a global instance of the `CsoundVST.CppSound` class named `csound`. Therefore, Python code written in the Csound orchestra has access to the global `csound` object.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

# pyrun Opcodes

pyrun Opcodes -- Run a Python statement or block of statements.

pyrun

## Syntax

```
pyrun "statement"
```

```
pyruni "statement"
```

```
pylrun "statement"
```

```
pylruni "statement"
```

```
pyrunt ktrigger, "statement"
```

```
pylrunt ktrigger, "statement"
```

## Description

Execute the specified Python statement at k-time (*pyrun* and *pylrun*) or i-time (*pyruni* and *pylruni*).

The statement is executed in the global environment for *pyrun* and *pyruni* or the local environment for *pylrun* and *pylruni*.

These opcodes perform no message passing. However, since the statement have access to the main namespace and the private namespace, it can interact with objects previously created in that environment.

The "local" version of the *pyrun* opcodes are useful when the code ran by different instances of an instrument should not interact.

## Example of the pyrun Opcode Group

### Example 313. Orchestra

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

```
;If you're not running CsoundVST you need the following line  
;to initialize the python interpreter  
;pyinit
```

```
pyruni "import random"
```

```
instr 1  
    ; This message is stored in the main namespace  
    ; and is the same for every instance  
    pyruni "message = 'a global random number: %f' % random.random()"
```

```
pyrun  "print message"

; This message is stored in the private namespace
; and is different for different instances
pylrni "message = 'a private random number: %f' % random.random()"
pylrn  "print message"

endin
```

### Example 314. Score

```
i1 0 0.1
```

Running this score you should get intermixed pairs of messages from the two instances of instrument 1.

The first message of each pair is stored into the main namespace and so the second instance overwrites the message of the first instance. The result is that first message will be the same for both instances.

The second message is different for the two instances, being stored in the private namespace.

## Credits

Copyright (c) 2002 by Maurizio Umberto Puxeddu. All rights reserved. Portions copyright (c) 2004 and 2005 by Michael Gogins. This document has been updated Sunday 25 July 2004 and 1 February 2005 by Michael Gogins.

# rand

rand -- Generates a controlled random number series.

rand

## Description

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

## Syntax

```
ares rand xamp [, iseed] [, isel] [, ibase]
```

```
kres rand xamp [, iseed] [, isel] [, ibase]
```

## Initialization

*iseed* (optional, default=0.5) -- a seed value for the recursive pseudo-random formula. A value between 0 and 1 will produce an initial output of *kamp \* iseed*. A value greater than 1 will be seeded from the system clock. A negative value will cause seed re-initialization to be skipped. The default seed value is .5.

*isel* (optional, default=0) -- if zero, a 16-bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* (optional, default=0) -- a base value added to the random result. New in Csound version 4.03.

## Performance

*kamp*, *xamp* -- range over which random numbers are distributed.

*kcps*, *xcps* -- the frequency which new random numbers are generated.

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 R.M.S value of *kamp / root 2*.

The remaining units produce band-limited noise: the *kcps* and *xcps* parameters permit the user to specify that new random numbers are to be generated at a rate less than the sampling or control frequencies.

## Examples

Here is an example of the rand opcode. It uses the files *rand.orc* [examples/rand.orc] and *rand.sco* [examples/rand.sco].

### Example 315. Example of the rand opcode.

```
/* rand.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1
```

```
; Instrument #1.
instr 1
; Choose a random frequency between 4,100 and 44,100.
kfreq rand 20000
kcps = kfreq + 24100

a1 oscil 30000, kcps, 1
out a1
endin
/* rand.orc */

/* rand.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* rand.sco */
```

## See Also

*randh, randi*

## Credits

Example written by Kevin Conder.

Thanks to a note from John ffitch, I changed the names of the parameters.

# randh

randh -- Generates random numbers and holds them for a period of time.

randh

## Description

Generates random numbers and holds them for a period of time.

## Syntax

ares **randh** xamp, xcps [, iseed] [, isize] [, ioffset]

kres **randh** kamp, kcps [, iseed] [, isize] [, ioffset]

## Initialization

*iseed* (optional, default=0.5) -- seed value for the recursive pseudo-random formula. A value between 0 and +1 will produce an initial output of  $kamp * iseed$ . A value greater than 1 will be used directly, without scaling. A negative value will cause seed re-initialization to be skipped. The default seed value is .5.

*isize* (optional, default=0) -- if zero, a 16 bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* (optional, default=0) -- a base value added to the random result. New in Csound version 4.03.

## Performance

*kamp*, *xamp* -- range over which random numbers are distributed.

*kcps*, *xcps* -- the frequency which new random numbers are generated.

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 R.M.S value of  $kamp / \text{root } 2$ .

The remaining units produce band-limited noise: the *kcps* and *xcps* 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.

## Examples

Here is an example of the randh opcode. It uses the files *randh.orc* [examples/randh.orc] and *randh.sco* [examples/randh.sco].

### Example 316. Example of the randh opcode.

```
/* randh.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
; Choose a random frequency between 4,100 and 44,100.
; Generate new random numbers at 220 Hz.
; kamp = 40000
; kcps = 220
; iseed = 0.5
; isize = 0
; ioffset = 4100

kcps randh 40000, 220, 0.5, 0, 4100

a1 oscil 30000, kcps, 1
out a1
endin
/* randh.orc */

/* randh.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* randh.sco */
```

## See Also

*rand*, *randi*

## Credits

Example written by Kevin Conder.

# randi

randi -- Generates a controlled random number series with interpolation between each new number.

rand

## Description

Generates a controlled random number series with interpolation between each new number.

## Syntax

```
ares randi xamp, xcps [, iseed] [, isize] [, ioffset]
```

```
kres randi kamp, kcps [, iseed] [, isize] [, ioffset]
```

## Initialization

*iseed* (optional, default=0.5) -- seed value for the recursive pseudo-random formula. A value between 0 and +1 will produce an initial output of *kamp \* iseed*. A value greater than 1 will be used directly, without scaling. A negative value will cause seed re-initialization to be skipped. The default seed value is .5.

*isize* (optional, default=0) -- if zero, a 16 bit number is generated. If non-zero, a 31-bit random number is generated. Default is 0.

*ioffset* (optional, default=0) -- a base value added to the random result. New in Csound version 4.03.

## Performance

*kamp*, *xamp* -- range over which random numbers are distributed.

*kcps*, *xcps* -- the frequency which new random numbers are generated.

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 R.M.S value of *kamp / root 2*.

The remaining units produce band-limited noise: the *kcps* and *xcps* parameters permit the user to specify that new random numbers are to be generated at a rate less than the sampling or control frequencies. *randi* will produce straight-line interpolation between each new number and the next.

## Examples

Here is an example of the randi opcode. It uses the files *randi.orc* [examples/rand\_i.orc] and *randi.sco* [examples/rand\_i.sco].

### Example 317. Example of the randi opcode.

```
/* randi.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1
```



```
; Instrument #1.
instr 1
; Choose a random frequency between 4,100 and 44,100.
; Generate new random numbers at 10 Hz.
; kamp = 40000
; kcps = 10
; iseed = 0.5
; isize = 0
; ioffset = 4100

kcps randi 40000, 10, 0.5, 0, 4100

a1 oscil 30000, kcps, 1
out a1
endin
/* randi.orc */

/* randi.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* randi.sco */
```

## See Also

*rand*, *randh*

## Credits

Example written by Kevin Conder.

# random

`random` -- Generates is a controlled pseudo-random number series between min and max values.

`random`

## Description

Generates is a controlled pseudo-random number series between min and max values.

## Syntax

`ares random kmin, kmax`

`ires random imin, imax`

`kres random kmin, kmax`

## Initialization

*imin* -- minimum range limit

*imax* -- maximum range limit

## Performance

*kmin* -- minimum range limit

*kmax* -- maximum range limit

The *random* opcode is similar to *linrand* and *trirand* but sometimes I [Gabriel Maldonado] find it more convenient because allows the user to set arbitrary minimum and maximum values.

## Examples

Here is an example of the *random* opcode. It uses the files *random.orc* [examples/random.orc] and *random.sco* [examples/random.sco].

### Example 318. Example of the random opcode.

```
/* random.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number between 220 and 440.
  kmin init 220
  kmax init 440
  k1 random kmin, kmax
```

```
    printks "k1 = %f\\n", 0.1, k1
endin
/* random.orc */
```

```
/* random.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* random.sco */
```

Its output should include lines like:

```
k1 = 414.232056
k1 = 419.393402
k1 = 275.376373
```

## See Also

*linrand, randomh, randomi, trirand*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

# randomh

`randomh` -- Generates random numbers with a user-defined limit and holds them for a period of time.

`randomh`

## Description

Generates random numbers with a user-defined limit and holds them for a period of time.

## Syntax

`ares randomh kmin, kmax, acps`

`kres randomh kmin, kmax, kcps`

## Performance

*kmin* -- minimum range limit

*kmax* -- maximum range limit

*kcps, acps* -- rate of random break-point generation

The *randomh* opcode is similar to *randh* but allows the user to set arbitrary minimum and maximum values.

## Examples

Here is an example of the *randomh* opcode. It uses the files *randomh.orc* [examples/randomh.orc] and *randomh.sco* [examples/randomh.sco].

### Example 319. Example of the randomh opcode.

```
/* randomh.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Choose a random frequency between 220 and 440 Hz.
  ; Generate new random numbers at 10 Hz.
  kmin = 220
  kmax = 440
  kcps = 10

  k1 randomh kmin, kmax, kcps

  printks "k1 = %f\\n", 0.1, k1
endin
/* randomh.orc */
```

```
/* randh.sco */  
; Table #1, a sine wave.  
f 1 0 16384 10 1  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* randh.sco */
```

Its output should include lines like:

```
k1 = 220.000000  
k1 = 414.232056  
k1 = 284.095184
```

## See Also

*randh*, *random*, *randomi*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

# randomi

`randomi` -- Generates a user-controlled random number series with interpolation between each new number.

`randomi`

## Description

Generates a user-controlled random number series with interpolation between each new number.

## Syntax

`ares randomi kmin, kmax, acps`

`kres randomi kmin, kmax, kcps`

## Performance

*kmin* -- minimum range limit

*kmax* -- maximum range limit

*kcps, acps* -- rate of random break-point generation

The *randomi* opcode is similar to *randi* but allows the user to set arbitrary minimum and maximum values.

## Examples

Here is an example of the *randomi* opcode. It uses the files *randomi.orc* [examples/randomi.orc] and *randomi.sco* [examples/randomi.sco].

### Example 320. Example of the randomi opcode.

```
/* randomi.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Choose a random frequency between 220 and 440.
  ; Generate new random numbers at 10 Hz.
  kmin init 220
  kmax init 440
  kcps init 10

  k1 randomi kmin, kmax, kcps

  printks "k1 = %f\\n", 0.1, k1
endin
/* randomi.orc */
```

```
/* randomi.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* randomi.sco */
```

Its output should include lines like:

```
k1 = 220.000000  
k1 = 414.226196  
k1 = 284.101074
```

## See Also

*randi, random, randomh*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

# rbjeq

rbjeq -- Parametric equalizer and filter opcode with 7 filter types, based on algorithm by Robert Bristow-Johnson.

rbjeq

## Description

Parametric equalizer and filter opcode with 7 filter types, based on algorithm by Robert Bristow-Johnson.

## Syntax

ar **rbjeq** asig, kfco, klvl, kQ, kS[, imode]

## Initialization

*imode* ( optional, defaults to zero) - sum of:

- 1: skip initialization (should be used in tied, or re-initialized notes only)

and exactly one of the following values to select filter type:

- 0: resonant lowpass filter. kQ controls the resonance: at the cutoff frequency (kfco), the amplitude gain is kQ (e.g. 20 dB for kQ = 10), and higher kQ values result in a narrower resonance peak. If kQ is set to  $\sqrt{0.5}$  (about 0.7071), there is no resonance, and the filter has a response that is very similar to that of butterlp. If kQ is less than  $\sqrt{0.5}$ , there is no resonance, and the filter has a -6 dB / octave response from about  $\text{kfco} * \text{kQ}$  to kfco. Above kfco, there is always a -12 dB / octave cutoff.



### NOTE

The rbjeq lowpass filter is basically the same as ar pareq asig, kfco, 0, kQ, 2 but is faster to calculate.

- 2: resonant highpass filter. The parameters are the same as for the lowpass filter, but the equivalent filter is butterhp if kQ is 0.7071, and "ar pareq asig, kfco, 0, kQ, 1" in other cases.
- 4: bandpass filter. kQ controls the bandwidth, which is  $\text{kfco} / \text{kQ}$ , and must be always less than  $\text{sr} / 2$ . The bandwidth is measured between -3 dB points (i.e. amplitude gain = 0.7071), beyond which there is a +/- 6 dB / octave slope. This filter type is very similar to ar butterbp asig, kfco, kfco / kQ.
- 6: band-reject filter, with the same parameters as the bandpass filter, and a response similar to that of butterbr.
- 8: peaking EQ. It has an amplitude gain of 1 (0 dB) at 0 Hz and  $\text{sr} / 2$ , and klvl at the center frequency (kfco). Thus, klvl controls the amount of boost (if it is greater than 1), or cut (if it is less than 1). Setting klvl to 1 results in a flat response. Similarly to the bandpass and band-reject filters, the bandwidth is determined by  $\text{kfco} / \text{kQ}$  (which must be less than  $\text{sr} / 2$  again); however, this time it is between  $\sqrt{\text{klvl}}$  points (or, in other words, half the boost or cut in decibels). NOTE: excessively low or high values of klvl should be avoided (especially with 32-bit floats), though the opcode was tested with klvl = 0.01 and klvl = 100. klvl = 0 is always an error, unlike in the case of pareq, which does allow a zero level.



- 10: low shelf EQ, controlled by *klvl* and *kS* (*kQ* is ignored by this filter type). There is an amplitude gain of *klvl* at zero frequency, while the level of high frequencies (around  $sr / 2$ ) is not changed. At the corner frequency (*kfco*), the gain is  $\sqrt{klvl}$  (half the boost or cut in decibels). The *kS* parameter controls the steepness of the slope of the frequency response (see below).
- 12: high shelf EQ. Very similar to the low shelf EQ, but affects the high frequency range.

The default value for *imode* is zero (lowpass filter, initialization not skipped).

## Performance

*ar* -- the output signal.

*asig* -- the input signal



### NOTE

If the input contains silent sections, on Intel CPUs a significant slowdown can occur due to denormals. In such cases, it is recommended to process the input signal with "denorm" opcode before filtering it with *rbjeq* (and actually many other filters).

*kfco* -- cutoff, corner, or center frequency, depending on filter type, in Hz. It must be greater than zero, and less than  $sr / 2$  (the range of about  $sr * 0.0002$  to  $sr * 0.49$  should be safe).

*klvl* -- level (amount of boost or cut), as amplitude gain (e.g. 1: flat response, 4: 12 dB boost, 0.1: 20 dB cut); zero or negative values are not allowed. It is recognized by the peaking and shelving EQ types (8, 10, 12) only, and is ignored by other filters.

*kQ* -- resonance (also *kfco* / bandwidth in many filter types). Not used by the shelving EQs (*imode* = 10 and 12). The exact meaning of this parameter depends on the filter type (see above), but it should be always greater than zero, and usually (*kfco* / *kQ*) less than  $sr / 2$ .

*kS* -- shelf slope parameter for shelving filters. Must be greater than zero; a higher value means a steeper slope, with resonance if  $kS > 1$  (however, a too high *kS* value may make the filter unstable). If *kS* is set to exactly 1, the shelf slope is as steep as possible without a resonance. Note that the effect of *kS* - especially if it is greater than 1 - also depends on *klvl*, and it does not have any well defined unit.

## Examples

```
/* rbjeq.orc */
sr      = 44100
ksmps   = 10
nchnls  = 1

instr 1

a1      vco2      10000, 155.6           ; sawtooth wave
kfco    expon     8000, p3, 200         ; filter frequency
a1      rbjeq     a1, kfco, 1, kfco * 0.005, 1, 0 ; resonant lowpass
out a1

        endin
/* rbjeq.orc */

/* rbjeq.sco */
i 1 0 5
e
```

```
/* rbjeq.sco */
```

## Credits

Original algorithm by Robert Bristow-Johnson

Csound orchestra version by Josep M Comajuncosas, Aug 1999

Converted to C (with optimizations and bug fixes) by Istvan Varga, Dec 2002

# readclock

readclock -- Reads the value of an internal clock.

readclock

## Description

Reads the value of an internal clock.

## Syntax

`ir readclock inum`

## 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* opcode, the CPU time used is accumulated in the clock. The precision is machine dependent but is the millisecond range on UNIX and Windows systems. The *readclock* opcode reads the current value of a clock at initialization time.

## Examples

Here is an example of the readclock opcode. It uses the files *readclock.orc* [examples/readclock.orc] and *readclock.sco* [examples/readclock.sco].

### Example 321. Example of the readclock opcode.

```
/* readclock.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1.
instr 1
  ; Start clock #1.
  clockon 1
  ; Do something that keeps Csound busy.
  al oscili 10000, 440, 1
  out al
  ; Stop clock #1.
  clockoff 1
  ; Print the time accumulated in clock #1.
  il readclock 1
  print il
endin
/* readclock.orc */
```

```
/* readclock.sco */
; Initialize the function tables.
; Table 1: an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for one second starting at 0:00.
i 1 0 1
; Play Instrument #1 for one second starting at 0:01.
i 1 1 1
; Play Instrument #1 for one second starting at 0:02.
i 1 2 1
e
/* readclock.sco */
```

Its output should include lines like this:

```
instr 1:  i1 = 0.000
instr 1:  i1 = 90.000
instr 1:  i1 = 180.000
```

## See Also

*clockoff*, *clockon*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
July, 1999

Example written by Kevin Conder.

New in Csound version 3.56

# readk

readk -- Periodically reads an orchestra control-signal value from an external file.

readk

## Description

Periodically reads an orchestra control-signal value to a named external file in a specific format.

## Syntax

```
kres readk ifilename, iformat, ipol [, interp]
```

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. 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* -- if non-zero, and *iprd* implies more than one control period, interpolate the *k*- signals between the periodic reads from the external file. If the value is 0, repeat each signal between frames. Currently not supported.

## Performance

*kres* -- a control-rate signal

This opcode allows a generated control signal value to be read from a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *readk* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

```
knum      =      knum+1                                ; at each
ktemp      tempest      krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995 ; estimate
koc        specptrk      wsig, 6, .9, 0                ; and the
              dumpk3      knum, ktemp, cpsoct(koc), "what happened when", 8 0 ;& save
```

## See Also

*dumpk*, *dumpk2*, *dumpk3*, *dumpk4*, *readk2*, *readk3*, *readk4*

# readk2

readk2 -- Periodically reads two orchestra control-signal values from an external file.

readk2

## Description

Periodically reads two orchestra control-signal values from an external file.

## Syntax

```
kr1, kr2 readk2 ifilename, iformat, ipol [, interp]
```

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. 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* -- if non-zero, and *iprd* implies more than one control period, interpolate the *k*- signals between the periodic reads from the external file. If the value is 0, repeat each signal between frames. Currently not supported.

## Performance

*kr1*, *kr2* -- control-rate signals

This opcode allows two generated control signal values to be read from a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *readk2* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

```
knum      =      knum+1                                ; at each  
ktemp      tempest      krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995 ; estimate  
koc        specptrk      wsig, 6, .9, 0                ; and the  
            dumpk3      knum, ktemp, cpsoct(koc), "what happened when", 8 0 ;& save
```

## See Also

*dumpk, dumpk2, dumpk3, dumpk4, readk, readk3, readk4*



# readk3

readk3 -- Periodically reads three orchestra control-signal values from an external file.

readk3

## Description

Periodically reads three orchestra control-signal values from an external file.

## Syntax

```
kr1, kr2, kr3 readk3 ifilename, iformat, ipol [, interp]
```

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. 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* -- if non-zero, and *iprd* implies more than one control period, interpolate the *k*- signals between the periodic reads from the external file. If the value is 0, repeat each signal between frames. Currently not supported.

## Performance

*kr1*, *kr2*, *kr3* -- control-rate signals

This opcode allows three generated control signal values to be read from a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *readk3* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

```
knum      =      knum+1                                ; at each  
ktemp     tempest   krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995 ;estimated  
koc       specptrk  wsig, 6, .9, 0                        ;and the  
           dumpk3   knum, ktemp, cpsoct(koc), "what happened when", 8 0 ;& save
```

## See Also

*dumpk, dumpk2, dumpk3, dumpk4, readk, readk2, readk4*

# readk4

readk4 -- Periodically reads four orchestra control-signal values from an external file.

readk4

## Description

Periodically reads four orchestra control-signal values from an external file.

## Syntax

```
kr1, kr2, kr3, kr4 readk4 ifilename, iformat, ipol [, interp]
```

## Initialization

*ifilename* -- 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 = 32-bit long integers
- 6 = 32-bit floats
- 7 = ASCII long integers
- 8 = ASCII floats (2 decimal places)

Note that A-law and U-law output are not available, and that all formats except the last two are binary. The output file contains no header information.

*iprd* -- the period of *ksig* output in seconds, rounded to the nearest orchestra control period. A value of 0 implies one control period (the enforced minimum), which will create an output file sampled at the orchestra control rate.

*ipol* -- if non-zero, and *iprd* implies more than one control period, interpolate the k- signals between the periodic reads from the external file. If the value is 0, repeat each signal between frames. Currently not supported.

## Performance

*kr1*, *kr2*, *kr3*, *kr4* -- control-rate signals.

This opcode allows four generated control signal values to be read from a named external file. The file contains no self-defining header information. But it contains a regularly sampled time series, suitable for later input or analysis. There may be any number of *readk4* opcodes in an instrument or orchestra and they may read from the same or different files.

## Examples

```
knum      =      knum+1                                ; at each
ktemp      tempest      krms, .02, .1, 3, 2, 800, .005, 0, 60, 4, .1, .995 ; estimate
koc        specptrk      wsig, 6, .9, 0                ; and the
              dumpk3      knum, ktemp, cpsoct(koc), "what happened when", 8 0 ;& save
```

## See Also

*dumpk, dumpk2, dumpk3, dumpk4, readk, readk2, readk3*

# reinit

`reinit --` Suspends a performance while a special initialization pass is executed.

`reinit`

## Description

Suspends a performance while a special initialization pass is executed.

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 *rreturn* or *endin*, is executed. Performance will then be resumed from where it left off.

## Syntax

```
reinit label
```

## Examples

The following statements will generate an exponential control signal whose value moves from 440 to 880 exactly ten times over the duration p3. They use the files *reinit.orc* [examples/reinit.orc] and *reinit.sco* [examples/reinit.sco].

### Example 322. Example of the reinit opcode.

```
/* reinit.orc */
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1

reset:
    timeout 0, p3/10, contin
    reinit reset

contin:
    kLine expon 440, p3/10, 880
    aSig oscil 10000, kLine, 1
    out aSig
    rireturn

endin
/* reinit.orc */
```

```
/* reinit.sco */
f1 0 4096 10 1

i1 0 10
e
/* reinit.sco */
```

## See Also

*rigoto, rireturn*

# release

`release` -- Indicates whether a note is in its “release” stage.

`release`

## Description

Indicates whether a note is in its “release” stage.

## Syntax

`kflag` **release**

## Performance

*kflag* -- indicates whether the note is in its “release” stage.

*release* outputs current note state. If current note is in the “release” stage (i.e. if its duration has been extended with *xratim* opcode and if it has only just deactivated), then the *kflag* output argument is set to 1. Otherwise (in sustain stage of current note), *kflag* is set to 0.

This opcode is useful for implementing complex release-oriented envelopes.

## Examples

```
instr 1 ;allows complex ADSR envelope with MIDI events
inum notnum
icps cpsmidi
iamp ampmidi 4000
;
;----- complex envelope block -----
xratim 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
```

## See Also

*xtratim*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47



# repluck

repluck -- Physical model of the plucked string.

repluck

## Description

*repluck* is an implementation of the physical model of the plucked string. A user can control the pluck point, the pickup point, the filter, and an additional audio signal, *axcite*. *axcite* is used to excite the 'string'. Based on the Karplus-Strong algorithm.

## Syntax

ares **repluck** *iplk*, *kamp*, *icps*, *kpick*, *krefl*, *axcite*

## Initialization

*iplk* -- 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.

*icps* -- The string plays at *icps* pitch.

## Performance

*kamp* -- Amplitude of note.

*kpick* -- Proportion of the way along the string to sample the output.

*krefl* -- the coefficient of reflection, indicating the lossiness and the rate of decay. It must be strictly between 0 and 1 (it will complain about both 0 and 1).

## Performance

*axcite* -- A signal which excites the string.

## Examples

Here is an example of the repluck opcode. It uses the files *repluck.orc* [examples/repluck.orc] and *repluck.sco* [examples/repluck.sco].

### Example 323. Example of the repluck opcode.

```
/* repluck.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iplk = 0.75
  kamp = 30000
  icps = 220
```

```
kpick = 0.75
krefl = 0.5
axcite oscil 1, 1, 1

apluck repluck iplk, kamp, icps, kpick, krefl, axcite

out apluck
endin
/* repluck.orc */

/* repluck.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* repluck.sco */
```

## See Also

*wgpluck2*

## Credits

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

# reson

`reson` -- A second-order resonant filter.

`reson`

## Description

A second-order resonant filter.

## Syntax

`ares reson asig, kcf, kbw [, iscl] [, iskip]`

## Initialization

*iscl* (optional, default=0) -- 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.

*iskip* (optional, default=0) -- 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

*ares* -- the output signal at audio rate.

*asig* -- the input signal at audio rate.

*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).

*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).

## Examples

Here is an example of the `reson` opcode. It uses the files *reson.orc* [examples/reson.orc] and *reson.sco* [examples/reson.sco].

### Example 324. Example of the `reson` opcode.

```
/* reson.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
; Generate a sine waveform.
asine buzz 15000, 440, 3, 1

; Vary the cut-off frequency from 220 to 1280.
kcf line 220, p3, 1320
kbw init 20

; Run the sine through a resonant filter.
ares reson asine, kcf, kbw

; Give the filtered signal the same amplitude
; as the original signal.
al balance ares, asine
out al
endin
/* reson.orc */

/* reson.sco */
; Table #1, an ordinary sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 4 seconds.
i 1 0 4
e
/* reson.sco */
```

## See Also

*areson, aresonk, atone, atonek, port, portk, resonk, tone, tonek*

## Credits

Example written by Kevin Conder.

# resonk

resonk -- A second-order resonant filter.

resonk

## Description

A second-order resonant filter.

## Syntax

kres **resonk** ksig, kcf, kbw [, iscl] [, iskip]

## Initialization

*iscl* (optional, default=0) -- 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.

*iskip* (optional, default=0) -- 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

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*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).

*resonk* is like *reson* except its output is at control-rate rather than audio rate.

## See Also

*areson*, *aresonk*, *atone*, *atonek*, *port*, *portk*, *reson*, *tone*, *tonek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# resonr

`resonr` -- A bandpass filter with variable frequency response.

`resonr`

## Description

Implementations of a second-order, two-pole two-zero bandpass filter with variable frequency response.

## Syntax

`ares resonr asig, kcf, kbw [, iscl] [, iskip]`

## Initialization

The optional initialization variables for *resonr* are identical to the i-time variables for *reson*.

*iscl* (optional, default=0) -- 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.

*iskip* (optional, default=0) -- 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* -- 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)

*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.

## Examples

Here is an example of the *resonr* and *resonz* opcodes. It uses the files *resonr.orc*

[examples/resonr.orc] and *resonr.sco* [examples/resonr.sco].

### Example 325. Example of the resonr and resonz opcodes.

```
/* resonr.orc */
/* Written by Sean Costello */
; 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
    iamp      =      p8                ; amplitude to scale output
    ires      =      p9                ; coefficient to scale amount
    iresr     =      p10               ; coefficient to scale amount
    iresz     =      p11               ; coefficient to scale amount

; 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
/* resonr.orc */

/* resonr.sco */
/* Written by Sean Costello */
f1 0 8192 9 1 1 .25                ; cosine table for gbuzz generation

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
/* resonr.sco */
```

## Technical History

*resonr* and *resonz* were originally described in an article by Julius O. Smith and James B. Angell.<sup>1</sup> 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<sup>2</sup>, 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<sup>3</sup> features a thorough technical discussion of *reson* and *resonz*, while Dodge and Jerse's textbook<sup>4</sup> illustrates the differences in the response curves of *reson* and *resonz*.

## References

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. Steiglitz, Ken, "A Note on Constant-Gain Digital Resonators," *Computer Music Journal*, vol. 18, no. 4, pp. 8-10, Winter 1994.
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. Dodge, Charles and Jerse, Thomas A., *Computer Music: Synthesis, Composition, and Performance*. New York: Schirmer Books, 1997, 2nd edition, pp. 211-214.

## See Also

*resonz*

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55



# resonx

`resonx` -- Emulates a stack of filters using the `reson` opcode.

`resonx`

## Description

*resonx* is equivalent to a filters consisting of more layers of *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.

## Syntax

ares **resonx** asig, kcf, kbw [, inumlayer] [, iscl] [, iskip]

## Initialization

*inumlayer* (optional) -- number of elements in the filter stack. Default value is 4.

*iscl* (optional, default=0) -- 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.

*iskip* (optional, default=0) -- 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* -- input signal

*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)

## See Also

*atonex*, *tonex*

## Credits

Author: Gabriel Maldonado (adapted by John ffitich)  
Italy

New in Csound version 3.49

# resonxk

resonxk -- Control signal resonant filter stack.

resonxk

## Description

*resonxk* is equivalent to a group of resonk filters, with the same arguments, serially connected. Using a stack of a larger number of filters allows a sharper cutoff.

## Syntax

kres **resonxk** ksig, kcf, kbw[, inumlayer, iscl, istor]

## Initialization

*inumlayer* - number of elements of filter stack. Default value is 4. Maximum value is 10

*iscl* (optional, default=0) - 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.

*istor* (optional, default=0) -- 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

*kres* - output signal

*ksig* - input signal

*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)

*resonxk* is a lot faster than using individual instances in Csound orchestra of the old opcodes, because only one initialization and 'k' cycle are needed at a time, and the audio loop falls entirely inside the cache memory of processor.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# resony

resony -- A bank of second-order bandpass filters, connected in parallel.

resony

## Description

A bank of second-order bandpass filters, connected in parallel.

## Syntax

ares **resony** asig, kbf, kbw, inum, ksep [, isepmode] [, iscl] [, iskip]

## Initialization

*inum* -- number of filters

*isepmode* (optional, default=0) -- if *isepmode* = 0, the separation of center frequencies of each filter is generated logarithmically (using octave as unit of measure). If *isepmode* not equal to 0, the separation of center frequencies of each filter is generated linearly (using Hertz). Default value is 0.

*iscl* (optional, default=0) -- 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* (optional, default=0) -- 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.

## Examples

Here is an example of the *resony* opcode. It uses the files *resony.orc* [examples/resony.orc], *resony.sco* [examples/resony.sco], and *beats.wav* [examples/beats.wav].

### Example 326. Example of the *resony* opcode.

```
/* resony.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a nice sawtooth waveform.
  asig vco 32000, 220, 1

  ; Vary the base frequency from 60 to 600 Hz.
  kbf line 60, p3, 600
  kbw = 50
  inum = 2
  ksep = 1
  isepmode = 0
  iscl = 1

  al resony asig, kbf, kbw, inum, ksep, isepmode, iscl

  out al
endin
/* resony.orc */

/* resony.sco */
; Table #1, a sine wave for the vco opcode.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* resony.sco */
```

## Credits

Author: Gabriel Maldonado  
Italy  
1999

Example written by Kevin Conder.

New in Csound version 3.56

# resonz

resonz -- A bandpass filter with variable frequency response.

resonz

## Description

Implementations of a second-order, two-pole two-zero bandpass filter with variable frequency response.

## Syntax

ares **resonz** asig, kcf, kbw [, iscl] [, iskip]

## Initialization

The optional initialization variables for *resonr* and *resonz* are identical to the i-time variables for *reson*.

*iscl* -- 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)

## Technical History

*resonr* and *resonz* were originally described in an article by Julius O. Smith and James B. Angell.<sup>1</sup>

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 <sup>2</sup>, 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 <sup>3</sup> features a thorough technical discussion of *reson* and *resonz*, while Dodge and Jerse's textbook <sup>4</sup> illustrates the differences in the response curves of *reson* and *resonz*.

## References

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. Steiglitz, Ken, "A Note on Constant-Gain Digital Resonators," *Computer Music Journal*, vol. 18, no. 4, pp. 8-10, Winter 1994.
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. Dodge, Charles and Jerse, Thomas A., *Computer Music: Synthesis, Composition, and Performance*. New York: Schirmer Books, 1997, 2nd edition, pp. 211-214.

## See Also

*resonr*

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55

# resyn

resyn -- Streaming partial track additive synthesis with cubic phase interpolation with pitch control and support for timescale-modified input

resyn

## Description

The resyn opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by partials). It resynthesises the signal using linear amplitude and cubic phase interpolation to drive a bank of interpolating oscillators with amplitude and pitch scaling controls. Resyn is a modified version of sinsyn, allowing for the resynthesis of data with pitch and timescale changes.

## Syntax

asig **resyn** fin, kscal, kpitch, kmaxtracks, ifn

## Performance

*asig* -- output audio rate signal

*fin* -- input pv stream in TRACKS format

*kscal* -- amplitude scaling

*kpitch* -- pitch scaling

*kmaxtracks* -- max number of tracks in resynthesis. Limiting this will cause a non-linear filtering effect, by discarding newer and higher-frequency tracks (tracks are ordered by start time and ascending frequency, respectively)

*ifn* -- function table containing one cycle of a sinusoid (sine or cosine)

## Examples

### Example 327. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
aout resyn fst, 1, 1.5, 500, 1 ; resynthesis (up a 5th)
out aout
```

The example above shows partial tracking of an ifd-analysis signal and cubic-phase additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini;

June 2005

New plugin in version 5

November 2004.



# reverb

reverb -- Reverberates an input signal with a “natural room” frequency response.

reverb

## Description

Reverberates an input signal with a “natural room” frequency response.

## Syntax

ares **reverb** asig, krvt [, iskip]

## Initialization

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

## Performance

*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).

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.

## Examples

Here is an example of the reverb opcode. It uses the files *reverb.orc* [examples/reverb.orc] and *reverb.sco* [examples/reverb.sco].

### Example 328. Example of the reverb opcode.

```
/* reverb.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

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

```
; Instrument #1. (there may be many copies)
instr 1
; generate a source signal
a1 oscili 7000, cpspch(p4), 1
; output the direct sound
out a1
; and add to audio receiver
gal = gal + a1
endin

; (highest instr number executed last)
instr 99
; reverberate whatever is in gal
a3 reverb gal, 1.5
; and output the result
out a3
; empty the receiver for the next pass
gal = 0
endin
/* reverb.orc */

/* reverb.sco */
; Table #1, a sine wave.
f 1 0 128 10 1

; p4 = frequency (in a pitch-class)
; Play Instrument #1 for a tenth of a second, p4=6.00
i 1 0 0.1 6.00
; Play Instrument #1 for a tenth of a second, p4=6.02
i 1 1 0.1 6.02
; Play Instrument #1 for a tenth of a second, p4=6.04
i 1 2 0.1 6.04
; Play Instrument #1 for a tenth of a second, p4=6.06
i 1 3 0.1 6.06

; Make sure the reverb remains active.
i 99 0 6
e
/* reverb.sco */
```

## See Also

*alpass, comb, valpass, vcomb*

## Credits

Author: William “Pete” Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002

## reverb2

reverb2 -- Same as the nreverb opcode.

reverb2

## Description

Same as the *nreverb* opcode.

## Syntax

ares **reverb2** asig, ktime, khdif [, iskip] [,inumCombs] [, ifnCombs] [, inumAlpa

# reverbsc

reverbsc -- 8 delay line stereo FDN reverb, based on work by Sean Costello

reverbsc

## Description

8 delay line stereo FDN reverb, with feedback matrix based upon physical modeling scattering junction of 8 lossless waveguides of equal characteristic impedance. Based on Csound orchestra version by Sean Costello.

## Syntax

aoutL, aoutR **reverbsc** ainL, ainR, kfbvl, kfco[, israte[, ipitchm[, iskip]]]

## Initialization

*israte* (optional, defaults to the orchestra sample rate) -- assume a sample rate of *israte*. This is normally set to *sr*, but a different setting can be useful for special effects.

*ipitchm* (optional, defaults to 1) -- depth of random variation added to delay times, in the range 0 to 10. The default is 1, but this may be too high and may need to be reduced for held pitches such as piano tones.

*iskip* (optional, defaults to zero) -- if non-zero, initialization of the opcode is skipped, whenever possible.

## Performance

*aoutL*, *aoutR* -- output signals for left and right channel

*ainL*, *ainR* -- left and right channel input. Note that having an input signal on either the left or right channel only will still result in having reverb output on both channels, making this unit more suitable for reverberating stereo input than the *freeverb* opcode.

*kfbvl* -- feedback level, in the range 0 to 1. 0.6 gives a good small "live" room sound, 0.8 a small hall, and 0.9 a large hall. A setting of exactly 1 means infinite length, while higher values will make the opcode unstable.

*kfco* -- cutoff frequency of simple first order lowpass filters in the feedback loop of delay lines, in Hz. Should be in the range 0 to  $\text{israte}/2$  (not  $\text{sr}/2$ ). A lower value means faster decay in the high frequency range.

## Examples

```
sr      = 48000
ksmps   = 32
nchnls  = 2
0dbfs   = 1

instr 1
a1      vco2 0.85, 440, 10
kfrq    port 100, 0.004, 20000
a1      butterlp a1, kfrq
a2      linseg 0, 0.003, 1, 0.01, 0.7, 0.005, 0, 1, 0
a1      = a1 * a2
```

```
a2      = a1 * p5
a1      = a1 * p4
        denorm a1, a2
aL, aR  reverbsc a1, a2, 0.85, 12000, sr, 0.5, 1
        outs a1 + aL, a2 + aR
        endin
```

```
i 1 0 1 0.71 0.71
i 1 1 1 0 1
i 1 2 1 -0.71 0.71
i 1 3 1 1 0
i 1 4 4 0.71 0.71
e
```

## Credits

Author: Istvan Varga  
2005

## rezzy

rezzy -- A resonant low-pass filter.

rezzy

## Description

A resonant low-pass filter.

## Syntax

ares **rezzy** asig, xfco, xres [, imode, iskip]

## Initialization

*imode* (optional, default=0) -- high-pass or low-pass mode. If zero, *rezzy* is low-pass. If not zero, *rezzy* is high-pass. Default value is 0. (New in Csound version 3.50) *iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## 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. Values of 1 to 100 are typical. Resonance should be one or greater. As of version 3.50, may a-rate, i-rate, or k-rate.

*rezzy* is a resonant low-pass filter created empirically by Hans Mikelson.

## Examples

Here is an example of the rezzy opcode. It uses the files *rezzy.orc* [examples/rezzy.orc] and *rezzy.sco* [examples/rezzy.sco].

### Example 329. Example of the rezzy opcode.

```
/* rezzy.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a nice sawtooth waveform.
  asig vco 32000, 220, 1

  ; Vary the filter-cutoff frequency from .2 to 2 KHz.
  kfco line 200, p3, 2000

  ; Set the resonance amount.
  kres init 25
```

```
    al rezzy asig, kfco, kres

    out al
endin
/* rezzy.orc */

/* rezzy.sco */
; Table #1, a sine wave for the vco opcode.
f 1 0 16384 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
e
/* rezzy.sco */
```

## See Also

*biquad, moogvcf*

## Credits

Author: Hans Mikelson  
October 1998

Example written by Kevin Conder.

New in Csound version 3.49

# rigoto

rigoto -- Transfers control during a reinit pass.

rigoto

## Description

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.

## Syntax

```
rigoto label
```

## See Also

*cigoto, igoto, reinit, rireturn*



# rireturn

rireturn -- Terminates a reinit pass.

rireturn

## Description

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

## Syntax

**rireturn**

## Examples

The following statements will generate an exponential control signal whose value moves from 440 to 880 exactly ten times over the duration p3. They use the files *reinit.orc* [examples/reinit.orc] and *reinit.sco* [examples/reinit.sco].

### Example 330. Example of the reireturn opcode.

```
/* reinit.orc */
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

instr 1

reset:
    timeout 0, p3/10, contin
    reinit reset

contin:
    kLine expon 440, p3/10, 880
    aSig oscil 10000, kLine, 1
    out aSig
    reireturn

endin
/* reinit.orc */

/* reinit.sco */
f1 0 4096 10 1

i1 0 10
e
/* reinit.sco */
```

## See Also

*reinit, rigoto*

## rms

**rms** -- Determines the root-mean-square amplitude of an audio signal.

rms

## Description

Determines the root-mean-square amplitude of an audio signal.

## Syntax

```
kres rms asig [, ihp] [, iskip]
```

## Initialization

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

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

## Performance

*asig* -- input audio signal

*rms* output values *kres* will trace the root-mean-square value of the audio input *asig*. This unit is not a signal modifier, but functions rather as a signal power-gauge.

## Examples

```
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
```

## See Also

*balance*, *gain*

# **rnd**

**rnd** -- Returns a random number in a unipolar range at the rate given by the input argument.

**rnd**

## **Description**

Returns a random number in a unipolar range at the rate given by the input argument.

## **Syntax**

**rnd**(x) (init- or control-rate only)

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**

Returns a random number in the unipolar range 0 to x.

## **Examples**

Here is an example of the **rnd** opcode. It uses the files *rnd.orc* [examples/rnd.orc] and *rnd.sco* [examples/rnd.sco].

### **Example 331. Example of the rnd opcode.**

```
/* rnd.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number from 0 to 1.
  il = rnd(1)
  print il
endin
/* rnd.orc */
```

```
/* rnd.sco */
; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #1 for one second.
i 1 1 1
e
/* rnd.sco */
```

Its output should be:

```
rnd at i-rate: 0.973500    rnd at k-rate: 0.139405
rnd at i-rate: 0.973500    rnd at k-rate: 0.040065
rnd at i-rate: 0.973500    rnd at k-rate: 0.412845
rnd at i-rate: 0.973500    rnd at k-rate: 0.440650
rnd at i-rate: 0.973500    rnd at k-rate: 0.663581
rnd at i-rate: 0.973500    rnd at k-rate: 0.876723
rnd at i-rate: 0.973500    rnd at k-rate: 0.302459
rnd at i-rate: 0.973500    rnd at k-rate: 0.398580
rnd at i-rate: 0.973500    rnd at k-rate: 0.448875
rnd at i-rate: 0.973500    rnd at k-rate: 0.907728
```

## See Also

*birnd*

## Credits

Author: Barry L. Vercoe  
MIT  
Cambridge, Massachusetts  
1997

Original Example written by Kevin Conder. Modified by John Harrison.

# rnd31

rnd31 -- 31-bit bipolar random opcodes with controllable distribution.

rnd31

## Description

31-bit bipolar random opcodes with controllable distribution. These units are portable, i.e. using the same seed value will generate the same random sequence on all systems. The distribution of generated random numbers can be varied at k-rate.

## Syntax

ax **rnd31** kscl, krpow [, iseed]

ix **rnd31** iscl, irpow [, iseed]

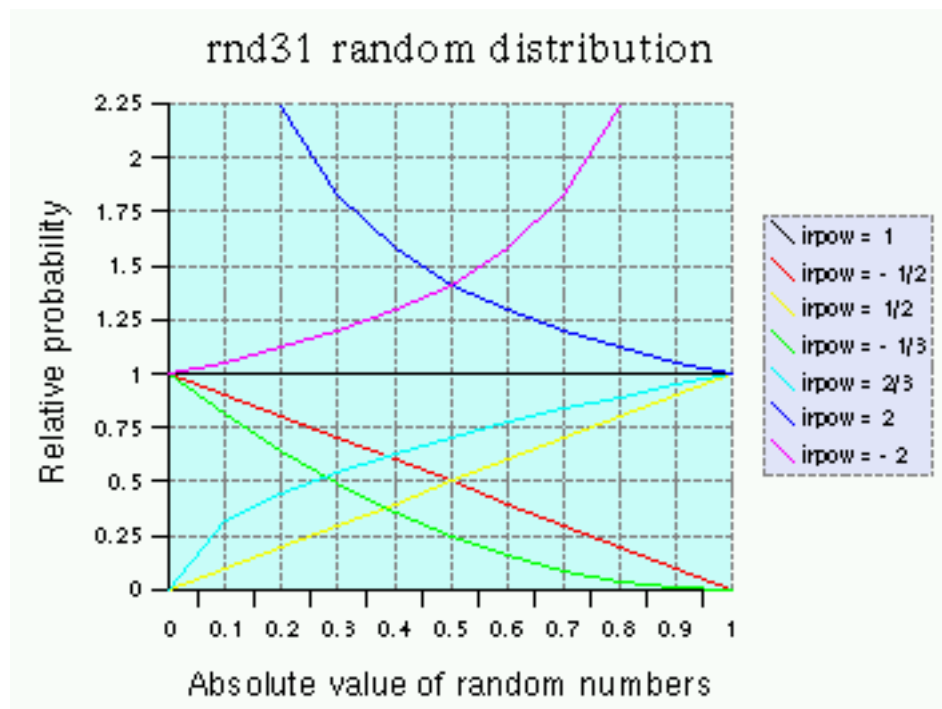
kx **rnd31** kscl, krpow [, iseed]

## Initialization

ix -- i-rate output value.

iscl -- output scale. The generated random numbers are in the range -iscl to iscl.

irpow -- controls the distribution of random numbers. If irpow is positive, the random distribution ( $x$  is in the range -1 to 1) is  $\text{abs}(x)^{(1/\text{irpow}) - 1}$ ; for negative irpow values, it is  $(1 - \text{abs}(x))^{((-1/\text{irpow}) - 1)}$ . Setting *irpow* to -1, 0, or 1 will result in uniform distribution (this is also faster to calculate).



A graph of distributions for different values of *irpow*.

*iseed* (optional, default=0) -- seed value for random number generator (positive integer in the range 1 to 2147483646 ( $2^{31} - 2$ )). Zero or negative value seeds from current time (this is also the default). Seeding from current time is guaranteed to generate different random sequences, even if multiple random opcodes are called in a very short time.

In the a- and k-rate version the seed is set at opcode initialization. With i-rate output, if *iseed* is zero or negative, it will seed from current time in the first call, and return the next value from the random sequence in successive calls; positive seed values are set at all i-rate calls. The seed is local for a- and k-rate, and global for i-rate units.



## Notes

- although seed values up to 2147483646 are allowed, it is recommended to use smaller numbers ( $< 1000000$ ) for portability, as large integers may be rounded to a different value if 32-bit floats are used.
- i-rate *rnd31* with a positive seed will always produce the same output value (this is not a bug). To get different values, set seed to 0 in successive calls, which will return the next value from the random sequence.

## Performance

*ax* -- a-rate output value.

*kx* -- k-rate output value.

*kscl* -- output scale. The generated random numbers are in the range -*kscl* to *kscl*. It is the same as *iscl*, but can be varied at k-rate.

*krpow* -- controls the distribution of random numbers. It is the same as *irpow*, but can be varied at k-rate.

## Examples

Here is an example of the *rnd31* opcode at a-rate. It uses the files *rnd31.orc* [examples/rnd31.orc] and *rnd31.sco* [examples/rnd31.sco].

### Example 332. An example of the *rnd31* opcode at a-rate.

```
/* rnd31.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Create random numbers at a-rate in the range -2 to 2 with
  ; a triangular distribution, seed from the current time.
  a31 rnd31 2, -0.5

  ; Use the random numbers to choose a frequency.
  afreq = a31 * 500 + 100
```

```
    a1 oscil 30000, afreq, 1
    out a1
endin
/* rnd31.orc */

/* rnd31.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* rnd31.sco */
```

Here is an example of the `rnd31` opcode at k-rate. It uses the files `rnd31_krate.orc` [examples/rnd31\_krate.orc] and `rnd31_krate.sco` [examples/rnd31\_krate.sco].

### **Example 333. An example of the `rnd31` opcode at k-rate.**

```
/* rnd31_krate.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
    ; Create random numbers at k-rate in the range -1 to 1
    ; with a uniform distribution, seed=10.
    k1 rnd31 1, 0, 10

    printks "k1=%f\\n", 0.1, k1
endin
/* rnd31_krate.orc */

/* rnd31_krate.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* rnd31_krate.sco */
```

Its output should include lines like this:

```
k1=0.112106
k1=-0.274665
k1=0.403933
```



Here is an example of the `rnd31` opcode that uses the number 7 as a seed value. It uses the files `rnd31_seed7.orc` [examples/rnd31\_seed7.orc] and `rnd31_seed7.sco` [examples/rnd31\_seed7.sco].

**Example 334. An example of the `rnd31` opcode that uses the number 7 as a seed value.**

```
/* rnd31_seed7.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; i-rate random numbers with linear distribution, seed=7.
; (Note that the seed was used only in the first call.)
i1 rnd31 1, 0.5, 7
i2 rnd31 1, 0.5
i3 rnd31 1, 0.5

print i1
print i2
print i3
endin
/* rnd31_seed7.orc */

/* rnd31_seed7.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* rnd31_seed7.sco */
```

Its output should include lines like this:

```
instr 1: i1 = -0.649
instr 1: i2 = -0.761
instr 1: i3 = 0.677
```

Here is an example of the `rnd31` opcode that uses the current time as a seed value. It uses the files `rnd31_time.orc` [examples/rnd31\_time.orc] and `rnd31_time.sco` [examples/rnd31\_time.sco].

**Example 335. An example of the `rnd31` opcode that uses the current time as a seed value.**

```
/* rnd31_time.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1
```

```
; Instrument #1.
instr 1
; i-rate random numbers with linear distribution,
; seeding from the current time. (Note that the seed
; was used only in the first call.)
i1 rnd31 1, 0.5, 0
i2 rnd31 1, 0.5
i3 rnd31 1, 0.5

print i1
print i2
print i3
endin
/* rnd31_time.orc */

/* rnd31_time.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* rnd31_time.sco */
```

Its output should include lines like this:

```
instr 1: i1 = -0.691
instr 1: i2 = -0.686
instr 1: i3 = -0.358
```

## Credits

Author: Istvan Varga

New in version 4.16

# rspline

rspline -- Generate random spline curves.

rspline

## Description

Generate random spline curves.

## Syntax

ares **rspline** xrangeMin, xrangeMax, kcpsMin, kcpsMax

kres **rspline** krangeMin, krangeMax, kcpsMin, kcpsMax

## Performance

*kres, ares* -- Output signal

*xrangeMin, xrangeMax* -- Range of values of random-generated points

*kcpsMin, kcpsMax* -- Range of point-generation rate. Min and max limits are expressed in cps.

*xamp* -- Amplitude factor

*rspline* (random-spline-curve generator) is similar to *jspline* but output range is defined by means of two limit values. Also in this case, real output range could be a bit greater of range values, because of interpolating curves between each pair of random-points.

At present time generated curves are quite smooth when cpsMin is not too different from cpsMax. When cpsMin-cpsMax interval is big, some little discontinuity could occur, but it should not be a problem, in most cases. Maybe the algorithm will be improved in next versions.

These opcodes are often better than *jitter* when user wants to “naturalize” or “analogize” digital sounds. They could be used also in algorithmic composition, to generate smooth random melodic lines when used together with *samphold* opcode.

Note that the result is quite different from the one obtained by filtering white noise, and they allow the user to obtain a much more precise control.

## Credits

Author: Gabriel Maldonado

New in version 4.15

# rtclock

rtclock -- Read the real time clock from the operating system.

rtclock

## Description

Read the real-time clock from the operating system.

## Syntax

ires **rtclock**

kres **rtclock**

## Performance

Read the real-time clock from operating system. Under Windows, this changes only once per second. Under GNU/Linux, it ticks every microsecond. Performance under other systems varies.

## Examples

Here is an example of the rtclock opcode. It uses the files *rtclock.orc* [examples/rtclock.orc] and *rtclock.sco* [examples/rtclock.sco].

### Example 336. Example of the rtclock opcode.

```
/* rtclock.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1
instr 1
  ; Get the system time.
  k1 rtclock
  ; Print it once per second.
  printk 1, k1
endin
/* rtclock.orc */

/* rtclock.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* rtclock.sco */
```

Its output should include lines like this:

```
i  1 time      0.00002: 1018236096.00000
i  1 time      1.00002: 1018236224.00000
```

## Credits

Author: John ffitch

Example written by Kevin Conder.

New in version 4.10

# s16b14

s16b14 -- Creates a bank of 16 different 14-bit MIDI control message numbers.

s16b14

## Description

Creates a bank of 16 different 14-bit MIDI control message numbers.

## Syntax

*i1*, ..., *i16* **s16b14** *ichan*, *ictlno\_msb1*, *ictlno\_lsb1*, *imin1*, *imax1*, *initvalue1*, *ifn1*, ..., *ifn16*

*k1*, ..., *k16* **s16b14** *ichan*, *ictlno\_msb1*, *ictlno\_lsb1*, *imin1*, *imax1*, *initvalue1*, *ifn1*, ..., *ifn16*

## Initialization

*i1* ... *i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlno\_msb1* .... *ictlno\_msb32* -- MIDI control number, most significant byte (0-127)

*ictlno\_lsb1* .... *ictlno\_lsb32* -- MIDI control number, least significant byte (0-127)

*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

*s16b14* is a bank 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.

*s16b14* allows a bank of 16 different MIDI control message numbers. It uses 14-bit values instead of MIDI's normal 7-bit values.

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 *sl6b14*, there is not an initial value input argument. The output is taken directly from the current status of internal controller array of Csound.

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# s32b14

s32b14 -- Creates a bank of 32 different 14-bit MIDI control message numbers.

s32b14

## Description

Creates a bank of 32 different 14-bit MIDI control message numbers.

## Syntax

*i1*, ..., *i32* **s32b14** *ichan*, *ictlno\_msb1*, *ictlno\_lsb1*, *imin1*, *imax1*, *initvalue1*, *ifn1*, ..., *ifn32*

*k1*, ..., *k32* **s32b14** *ichan*, *ictlno\_msb1*, *ictlno\_lsb1*, *imin1*, *imax1*, *initvalue1*, *ifn1*, ..., *ifn32*

## Initialization

*i1* ... *i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlno\_msb1* .... *ictlno\_msb32* -- MIDI control number, most significant byte (0-127)

*ictlno\_lsb1* .... *ictlno\_lsb32* -- MIDI control number, least significant byte (0-127)

*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

*s32b14* is a bank 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.

*s32b14* allows a bank of 32 different MIDI control message numbers. It uses 14-bit values instead of MIDI's normal 7-bit values.

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 *s32b14*, there is not an initial value input argument. The output is taken directly from the current status of internal controller array of Csound.

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# samphold

samphold -- Performs a sample-and-hold operation on its input.

samphold

## Description

Performs a sample-and-hold operation on its input.

## Syntax

ares **samphold** asig, agate [, ival] [, ivstor]

kres **samphold** ksig, kgate [, ival] [, ivstor]

## Initialization

*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

*kgate*, *xgate* -- controls whether to hold the signal.

*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.

## Examples

```
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
```

## See Also

*diff*, *downsamp*, *integ*, *interp*, *upsamp*

# sandpaper

sandpaper -- Semi-physical model of a sandpaper sound.

sandpaper

## Description

*sandpaper* is a semi-physical model of a sandpaper sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **sandpaper** iamp, idettack [, inum] [, idamp] [, imaxshake]

## 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

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 128.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.999 which means that the default value of *idamp* is 0.5. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the sandpaper opcode. It uses the files *sandpaper.orc* [examples/sandpaper.orc] and *sandpaper.sco* [examples/sandpaper.sco].

### Example 337. Example of the sandpaper opcode.

```
/* sandpaper.orc */
;orchestra -----

    sr =                44100
    kr =                4410
    ksmpls =            10
    nchnls =             1

instr 01                                ;an example of sandpaper blocks
  a1    line 2, p3, 2                    ;preset amplitude increase
  a2    sandpaper p4, 0.01                ;sandpaper needs a little amp help at t
  a3    product a1, a2                    ;increase amplitude
      out a3
```

```
        endin
/* sandpaper.orc */

/* sandpaper.sco */
;score -----

        i1 0 1 26000
        e
/* sandpaper.sco */
```

## See Also

*cabasa, crunch, sekere, stix*

## Credits

Author: Perry Cook, part of the PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds)

Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# scanhammer

scanhammer -- Copies from one table to another with a gain control.

scanhammer

## Description

This is a variant of *tablecopy*, copying from one table to another, starting at *ipos*, and with a gain control. The number of points copied is determined by the length of the source. Other points are not changed. This opcode can be used to “hit” a string in the scanned synthesis code.

## Syntax

**scanhammer** *isrc*, *idst*, *ipos*, *imult*

## Initialization

*isrc* -- source function table.

*idst* -- destination function table.

*ipos* -- starting position (in points).

*imult* -- gain multiplier. A value of 0 will leave values unchanged.

## See Also

*scantable*

## Credits

Author: Matt Gilliard  
April 2002

New in version 4.20

# scans

scans -- Generate audio output using scanned synthesis.

scans

## Description

Generate audio output using scanned synthesis.

## Syntax

ares **scans** kamp, kfreq, ifn, id [, iorder]

## 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, default=0) -- 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

## Examples

Here is an example of the scanned synthesis. It uses the files *scans.orc* [examples/scans.orc], *scans.sco* [examples/scans.sco], and *string-128.matrix* [examples/string-128.matrix].

### Example 338. Example of the scans opcode.

```
/* scans.orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
a0 = 0
; scanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, ksti
; scanu 1, .01, 6, 2, 3, 4, 5, 2, .1,
;ar scans kamp, kfreq, ifntraj, id
a1 scans ampdb(p4), cpspch(p5), 7, 2
out a1
endin
/* scans.orc */
```

```
/* scans.sco */
; 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.matrix"

; 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
/* scans.sco */
```

The matrix file “string-128.matrix”, as well as several other matrices, is also available in a *zipped file* [<http://www.csounds.com/scanned/zip/scanmatrices.zip>] from the *Scanned Synthesis page* [<http://www.csounds.com/scanned/>] at cSounds.com.

## Credits

Author: Paris Smaragdis  
MIT Media Lab  
Boston, Massachusetts USA

New in Csound version 4.05

# scantable

scantable -- A simpler scanned synthesis implementation.

scantable

## Description

A simpler scanned synthesis implementation. This is an implementation of a circular string scanned using external tables. This opcode will allow direct modification and reading of values with the table opcodes.

## Syntax

aout **scantable** kamp, kpch, ipos, imass, istiff, idamp, ivel

## Initialization

*ipos* -- table containing position array.

*imass* -- table containing the mass of the string.

*istiff* -- table containing the stiffness of the string.

*idamp* -- table containing the damping factors of the string.

*ivel* -- table containing the velocities.

## Performance

*kamp* -- amplitude (gain) of the string.

*kpch* -- the string's scanned frequency.

## Examples

Here is an example of the scantable opcode. It uses the files *scantable.orc* [examples/scantable.orc] and *scantable.sco* [examples/scantable.sco].

### Example 339. Example of the scantable opcode.

```
/* scantable.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Table #1 - initial position
git1 ftgen 1, 0, 128, 7, 0, 64, 1, 64, 0
; Table #2 - masses
git2 ftgen 2, 0, 128, -7, 1, 128, 1
; Table #3 - stiffness
git3 ftgen 3, 0, 128, -7, 0, 64, 100, 64, 0
; Table #4 - damping
git4 ftgen 4, 0, 128, -7, 1, 128, 1
```



```
; Table #5 - initial velocity
git5 ftgen 5, 0, 128, -7, 0, 128, 0

; Instrument #1.
instr 1
  kamp init 20000
  kpch init 220
  ipos = 1
  imass = 2
  istiff = 3
  idamp = 4
  ivel = 5

  a1 scantable kamp, kpch, ipos, imass, istiff, idamp, ivel
  a2 dcblock a1

  out a2
endin
/* scantable.orc */

/* scantable.sco */
; Play Instrument #1 for ten seconds.
i 1 0 10
e
/* scantable.sco */
```

## See Also

*scanhammer*

## Credits

Author: Matt Gilliard  
April 2002

Example written by Kevin Conder.

New in version 4.20

# scanu

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

scanu

## Description

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

## Syntax

**scanu** *init*, *irate*, *ifnvel*, *ifnmass*, *ifnstif*, *ifncentr*, *ifndamp*, *kmass*, *kstif*, *k*

## 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.

## Examples

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

## Credits

Author: Paris Smaragdis  
MIT Media Lab  
Boston, Massachusetts USA  
March 2000

New in Csound version 4.05

# schedkwhen

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

**schedkwhen**

## Description

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

## Syntax

**schedkwhen** *ktrigger*, *kmintim*, *kmaxnum*, *kinsnum*, *kwhen*, *kdur* [, *ip4*] [, *ip5*] [..

**schedkwhen** *ktrigger*, *kmintim*, *kmaxnum*, "insname", *kwhen*, *kdur* [, *ip4*] [, *ip5*] [

## Initialization

*"insname"* -- A string (in double-quotes) representing a named instrument.

*ip4*, *ip5*, ... -- Equivalent to *p4*, *p5*, etc., in a score *i statement*

## 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*.)

*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.

## Examples

Here is an example of the *schedkwhen* opcode. It uses the files *schedkwhen.orc* [examples/schedkwhen.orc] and *schedkwhen.sco* [examples/schedkwhen.sco].

### Example 340. Example of the schedkwhen opcode.

```
/* schedkwhen.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1 - oscillator with a high note.
instr 1
  ; Use the fourth p-field as the trigger.
  ktrigger = p4
  kmintim = 0
  kmaxnum = 2
  kinsnum = 2
  kwhen = 0
  kdur = 0.5

  ; Play Instrument #2 at the same time, if the trigger is set.
  schedkwhen ktrigger, kmintim, kmaxnum, kinsnum, kwhen, kdur

  ; Play a high note.
  al oscils 10000, 880, 1
  out al
endin

; Instrument #2 - oscillator with a low note.
instr 2
  ; Play a low note.
  al oscils 10000, 220, 1
  out al
endin
/* schedkwhen.orc */

/* schedkwhen.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; p4 = trigger for Instrument #2 (when p4 > 0).
; Play Instrument #1 for half a second, no trigger.
i 1 0 0.5 0
; Play Instrument #1 for half a second, trigger Instrument #2.
i 1 1 0.5 1
e
/* schedkwhen.sco */
```

## Credits

Author: Rasmus Ekman  
EMS, Stockholm, Sweden

Example written by Kevin Conder.

New in Csound version 3.59

# schedkwhennamed

`schedkwhennamed` -- Similar to `schedkwhen` but uses a named instrument at init-time.

`schedkwhennamed`

## Description

Similar to *schedkwhen* but uses a named instrument at init-time.

## Syntax

**schedkwhennamed** *ktrigger*, *kmintim*, *kmaxnum*, "*name*", *kwhen*, *kdur* [, *ip4*] [, *ip5*]

## Initialization

*ip4*, *ip5*, ... -- Equivalent to *p4*, *p5*, etc., in a score *i statement*

## Performance

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

*kmintim* -- minimum time between generated events, in seconds. If *kmintim* is less than or equal to 0, no time limit exists.

*kmaxnum* -- maximum number of simultaneous instances of named instrument allowed. If the number of extant instances of the named instrument is greater than or equal to *kmaxnum*, no new event is generated. If *kmaxnum* is less than or equal to 0, it is not used to limit event generation.

"*name*" -- the named instrument's name.

*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 greater than or equal to 0. If *kwhen* greater than 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* is 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*.)

*Note:* While waiting for events to be triggered by *schedkwhennamed*, 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.

## See Also

*schedkwhen*

## Credits

Author: Rasmus Ekman  
EMS, Stockholm, Sweden

New in Csound version 4.23

# schedule

`schedule` -- Adds a new score event.

`schedule`

## Description

Adds a new score event.

## Syntax

**schedule** *insnum*, *iwhen*, *idur* [, *ip4*] [, *ip5*] [...]

**schedule** "insname", *iwhen*, *idur* [, *ip4*] [, *ip5*] [...]

## Initialization

*insnum* -- instrument number. Equivalent to p1 in a score *i statement*. *insnum* must be a number greater than the number of the calling instrument.

"*insname*" -- A string (in double-quotes) representing a named instrument.

*iwhen* -- start time of the new event. Equivalent to p2 in a score *i statement*. *iwhen* must be nonnegative. If *iwhen* is zero, *insnum* must be greater than or equal to the p1 of the current instrument.

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

*ip4*, *ip5*, ... -- Equivalent to p4, p5, etc., 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.

## Examples

Here is an example of the `schedule` opcode. It uses the files *schedule.orc* [examples/schedule.orc] and *schedule.sco* [examples/schedule.sco].

### Example 341. Example of the schedule opcode.

```
/* schedule.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - oscillator with a high note.
```

```
instr 1
; Play Instrument #2 at the same time.
schedule 2, 0, p3

; Play a high note.
a1 oscils 10000, 880, 1
out a1
endin

; Instrument #2 - oscillator with a low note.
instr 2
; Play a low note.
a1 oscils 10000, 220, 1
out a1
endin
/* schedule.orc */

/* schedule.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for half a second.
i 1 0 0.5
; Play Instrument #1 for half a second.
i 1 1 0.5
e
/* schedule.sco */
```

## See Also

*schedwhen*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
November 1998

Example written by Kevin Conder.

New in Csound version 3.491

Based on work by Gabriel Maldonado

Thanks goes to David Gladstein, for clarifying the *iwhen* parameter.



# schedwhen

`schedwhen` -- Adds a new score event.

`schedwhen`

## Description

Adds a new score event.

## Syntax

```
schedwhen ktrigger, kinsnum, kwhen, kdur [, ip4] [, ip5] [...]
```

```
schedwhen ktrigger, "insname", kwhen, kdur [, ip4] [, ip5] [...]
```

## Initialization

*ip4, ip5, ...* -- Equivalent to *p4, p5, etc.*, in a score *i statement*.

## Performance

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

*"insname"* -- A string (in double-quotes) representing a named instrument.

*ktrigger* -- trigger value for new event

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

*kdur* -- duration of event. Equivalent to *p3* in a score *i statement*.

*schedwhen* adds a new score event. The event is only scheduled when the k-rate value *ktrigger* is first non-zero. 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.



### Warning

Support for named instruments is broken in version 4.23

## Examples

Here is an example of the `schedwhen` opcode. It uses the files `schedwhen.orc` [examples/schedwhen.orc] and `schedwhen.sco` [examples/schedwhen.sco].

### Example 342. Example of the `schedwhen` opcode.

```
/* schedwhen.orc */  
; Initialize the global variables.  
sr = 44100
```

```
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1 - oscillator with a high note.
instr 1
  ; Use the fourth p-field as the trigger.
  ktrigger = p4
  kinsnum = 2
  kwhen = 0
  kdur = p3

  ; Play Instrument #2 at the same time, if the trigger is set.
  schedwhen ktrigger, kinsnum, kwhen, kdur

  ; Play a high note.
  a1 oscils 10000, 880, 1
  out a1
endin

; Instrument #2 - oscillator with a low note.
instr 2
  ; Play a low note.
  a1 oscils 10000, 220, 1
  out a1
endin
/* schedwhen.orc */

/* schedwhen.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; p4 = trigger for Instrument #2 (when p4 > 0).
; Play Instrument #1 for half a second, trigger Instrument #2.
i 1 0 0.5 1
; Play Instrument #1 for half a second, no trigger.
i 1 1 0.5 0
e
/* schedwhen.sco */
```

## See Also

*schedule*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
November 1998

Example written by Kevin Conder.

New in Csound version 3.491

Based on work by Gabriel Maldonado

# seed

seed -- Sets the global seed value.

seed

## 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*, *randh*, *randi*, *rnd(x)* and *birnd(x)* are not affected by seed.

## Syntax

**seed** *ival*

## 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.

# sekere

sekere -- Semi-physical model of a sekere sound.

sekere

## Description

*sekere* is a semi-physical model of a sekere sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **sekere** iamp, idettack [, inum] [, idamp] [, imaxshake]

## 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

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 64.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.999 which means that the default value of *idamp* is 0.5. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the sekere opcode. It uses the files *sekere.orc* [examples/sekere.orc] and *sekere.sco* [examples/sekere.sco].

### Example 343. Example of the sekere opcode.

```
/* sekere.orc */
;orchestra -----

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

instr 01
al      sekere p4, 0.01      ;an example of a sekere
      out al
      endin
/* sekere.orc */
```

```
/* sekere.sco */
;score -----

      i1 0 1 26000
      e
/* sekere.sco */
```

## See Also

*cabasa, crunch, sandpaper, stix*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# semitone

semitone -- Calculates a factor to raise/lower a frequency by a given amount of semitones.

semitone

## Description

Calculates a factor to raise/lower a frequency by a given amount of semitones.

## Syntax

**semitone**(*x*)

This function works at a-rate, i-rate, and k-rate.

## Initialization

*x* -- a value expressed in semitones.

## Performance

The value returned by the *semitone* function is a factor. You can multiply a frequency by this factor to raise/lower it by the given amount of semitones.

## Examples

Here is an example of the semitone opcode. It uses the files *semitone.orc* [examples/semitone.orc] and *semitone.sco* [examples/semitone.sco].

### Example 344. Example of the semitone opcode.

```
/* semitone.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; The root note is A above middle-C (440 Hz)
  iroot = 440

  ; Raise the root note by three semitones to C.
  isemitone = 3

  ; Calculate the new note.
  ifactor = semitone(isemitone)
  inew = iroot * ifactor

  ; Print out all of the values.
  print iroot
  print ifactor
  print inew
endin
/* semitone.orc */
```

```
/* semitone.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* semitone.sco */
```

Its output should include lines like:

```
instr 1:  iroot = 440.000  
instr 1:  ifactor = 1.189  
instr 1:  inew = 523.229
```

## See Also

*cent, db, octave*

## Credits

Example written by Kevin Conder.

New in version 4.16

## sense

sense -- Same as the sensekey opcode.

sense

## Description

Same as the *sensekey* opcode.

## Syntax

kres **sense**



## sensekey

sensekey -- Returns the ASCII code of a key that has been pressed.

sensekey

## Description

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

## Syntax

kres **sensekey**

## Performance

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



### Note

This opcode can also be written as *sense*.

## Examples

Here is an example of the sensekey opcode. It uses the files *sensekey.orc* [examples/sensekey.orc] and *sensekey.sco* [examples/sensekey.sco].

### Example 345. Example of the sensekey opcode.

```
/* sensekey.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  k1 sensekey
  printk2 k1
endin
/* sensekey.orc */

/* sensekey.sco */
; Play Instrument #1 for thirty seconds.
i 1 0 30
e
/* sensekey.sco */
```

Here is what the output should look like when the "q" button is pressed...

```
q i1 357967744.00000
```

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
October 2000

Example written by Kevin Conder.

New in Csound version 4.09. Renamed in Csound version 4.10.

# seqtime

seqtime -- Generates a trigger signal according to the values stored in a table.

seqtime

## Description

Generates a trigger signal according to the values stored in a table.

## Syntax

```
ktrig_out seqtime ktime_unit, kstart, kloop, kinitndx, kfn_times
```

## Performance

*ktrig\_out* -- output trigger signal

*ktime\_unit* -- unit of measure of time, related to seconds.

*kstart* -- start index of looped section

*kloop* -- end index of looped section

*kinitndx* -- initial index



### Note

Although *kinitndx* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*kfn\_times* -- number of table containing a sequence of times

This opcode handles timed-sequences of groups of values stored into a table.

*seqtime* generates a trigger signal (a sequence of impulses, see also *trigger* opcode), according to the values stored in the *kfn\_times* table. This table should contain a series of delta-times (i.e. times between to adjacent events). The time units stored into table are expressed in seconds, but can be rescaled by means of *ktime\_unit* argument. The table can be filled with *GEN02* or by means of an external text-file containing numbers, with *GEN23*.

It is possible to start the sequence from a value different than the first, by assigning to *initndx* an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying *kstart* and *kloop* arguments. User must be sure that values of these arguments (as well as *initndx*) correspond to valid table numbers, otherwise Csound will crash (because no range-checking is implemented).

It is possible to disable loop (one-shot mode) by assigning the same value both to *kstart* and *kloop* arguments. In this case, the last read element will be the one corresponding to the value of such arguments. Table can be read backward by assigning a negative *kloop* value. It is possible to trigger two events almost at the same time (actually separated by a k-cycle) by giving a zero value to the corresponding delta-time. First element contained in the table should be zero, if the user intends to send a trigger impulse, it should come immediately after the orchestra instrument containing *seqtime* opcode.

## Examples

**Example 346. Example of the seqtime opcode.**

```
instr 1
icps    cpsmidi
iamp    ampmidi 5000
ktrig   seqtime 1,      1,      10,      0,      1
trigseq ktrig, 0, 10, 0, 2, kdur, kampratio, kfrequatio
        schedkwhen      ktrig, -1, -1, 2, 0, kdur, kampratio*iamp, kfrequatio*i
        endin

instr 2
**** put here your instrument code ****
out     a1
endin
```

## See Also

*GEN02, GEN23, trigseq seqtime2*

## Credits

Author: Gabriel Maldonado

November 2002. Added a note about the *kinitndx* parameter, thanks to Rasmus Ekman.

New in version 4.06

# seqtime2

seqtime2 -- Generates a trigger signal according to the values stored in a table.

seqtime2

## Description

Generates a trigger signal according to the values stored in a table.

## Syntax

ktrig\_out **seqtime2** ktrig\_in, ktime\_unit, kstart, kloop, kinitndx, kfn\_times

## Performance

*ktrig\_out* -- output trigger signal

*ktime\_unit* -- unit of measure of time, related to seconds.

*ktime\_in* -- input trigger signal.

*kstart* -- start index of looped section

*kloop* -- end index of looped section

*kinitndx* -- initial index



### Note

Although *kinitndx* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*kfn\_times* -- number of table containing a sequence of times

This opcode handles timed-sequences of groups of values stored into a table.

*seqtime2* generates a trigger signal (a sequence of impulses, see also *trigger* opcode), according to the values stored in the *kfn\_times* table. This table should contain a series of delta-times (i.e. times between to adjacent events). The time units stored into table are expressed in seconds, but can be rescaled by means of *ktime\_unit* argument. The table can be filled with *GEN02* or by means of an external text-file containing numbers, with *GEN23*.

It is possible to start the sequence from a value different than the first, by assigning to *initndx* an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying *kstart* and *kloop* arguments. User must be sure that values of these arguments (as well as *initndx*) correspond to valid table numbers, otherwise Csound will crash (because no range-checking is implemented).

It is possible to disable loop (one-shot mode) by assigning the same value both to *kstart* and *kloop* arguments. In this case, the last read element will be the one corresponding to the value of such arguments. Table can be read backward by assigning a negative *kloop* value. It is possible to trigger two events almost at the same time (actually separated by a k-cycle) by giving a zero value to the corresponding delta-time. First element contained in the table should be zero, if the user intends to send a trigger impulse, it should come immediately after the orchestra instrument containing *seqtime2* opcode.

*seqtime2* is similar to *seqtime*, the difference is that when *ktrig\_in* contains a non-zero value, current

index is reset to kinitndx value. kinitndx can be varied at performance time.

## See Also

*GEN02, GEN23, seqtime trigseq*

## Credits

Author: Gabriel Maldonado

# setctrl

setctrl -- Configurable slider controls for realtime user input.

setctrl

## 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.

## Syntax

**setctrl** inum, ival, itype

## Initialization

*inum* -- number of the slider to set

*ival* -- value to be sent to the slider

*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

Calling *setctrl* 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*.

## Examples

Here is an example of the *setctrl* opcode. It uses the files *setctrl.orc* [examples/setctrl.orc] and *setctrl.sco* [examples/setctrl.sco].

### Example 347. Example of the setctrl opcode.

```
/* setctrl.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
```

```
; Display the label "Volume" on Slider #1.
setctrl 1, "Volume", 4
; Set Slider #1's initial value to 20.
setctrl 1, 20, 1

; Capture and display the values for Slider #1.
k1 control 1
printk2 k1

; Play a simple oscillator.
; Use the values from Slider #1 for amplitude.
kamp = k1 * 128
a1 oscil kamp, 440, 1
out a1
endin
/* setctrl.orc */

/* setsctrl.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for thirty seconds.
i 1 0 30
e
/* setsctrl.sco */
```

Its output should include lines like this:

```
i1      38.00000
i1      40.00000
i1      43.00000
```

## See Also

*control*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
May 2000

Example written by Kevin Conder.

New in Csound version 4.06



# setksmps

setksmps -- Sets the local ksmps value in a user-defined opcode block.

setksmps

## Description

Sets the local ksmps value in a user-defined opcode block.

The *setksmps* statement can be used to set the local *ksmps* value of the user-defined opcode block. It has one i-time parameter specifying the new *ksmps* value (which is left unchanged if zero is used). *setksmps* should be used before any other opcodes (but allowed after *xin*), otherwise unpredictable results may occur.

## Syntax

**setksmps** iksmps

## Initialization

*iksmps* -- sets the local ksmps value.

If *iksmps* is set to zero, the *ksmps* of the caller instrument or opcode is used (this is the default behavior).



### Note

The local *ksmps* is implemented by splitting up a control period into smaller sub-periods and temporarily modifying internal Csound global variables. This also requires converting the rate of k-rate input and output arguments (input variables receive the same value in all sub-kperiods, while outputs are written only in the last one).



### Warning about local ksmps

When the local *ksmps* is not the same as the orchestra level *ksmps* value (as specified in the orchestra header). Global a-rate operations must not be used in the user-defined opcode block.

These include:

- any access to “ga” variables
- a-rate zak opcodes (*zar*, *zaw*, etc.)
- *tablera* and *tablewa* (these two opcodes may in fact work, but caution is needed)
- The *in* and *out* opcode family (these read from, and write to global a-rate buffers)

In general, the local *ksmps* should be used with care as it is an experimental feature. Though it works correctly in most cases.

The *setksmps* statement can be used to set the local *ksmps* value of the user-defined opcode block. It has one i-time parameter specifying the new *ksmps* value (which is left unchanged if zero is used). *setksmps* should be used before any other opcodes (but allowed after *xin*), otherwise unpredictable

results may occur.

## Performance

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmps]
```

## Examples

See the example for the *opcode* opcode.

## See Also

*endop*, *opcode*, *xin*, *xout*

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

## sfilist

`sfilist` -- Prints a list of all instruments of a previously loaded SoundFont2 (SF2) file.

`sfilist`

## Description

Prints a list of all instruments of a previously loaded SoundFont2 (SF2) sample file. 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 *SoundFont2 File Format Appendix*.

## Syntax

```
sfilist ifilhandle
```

## Initialization

*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.

## Performance

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

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.

## See Also

*sfinstr*, *sfinstrm*, *sfload*, *sfpassign*, *sfplay*, *sfplaym*, *sfplist*, *sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# **sfinstr**

**sfinstr** -- Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound.

**sfinstr**

## **Description**

Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound. 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 *SoundFont2 File Format Appendix*.

## **Syntax**

`ar1, ar2 sfinstr ivel, inotenum, xamp, xfreq, instrnum, ifilhandle [, iflag] [,`

## **Initialization**

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*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.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## **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.

When *iflag* = 0, *inotenum* 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.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*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.

These opcodes only support the sample structure of SF2 files. The modulator structure of the Sound-Font2 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.

## See Also

*sfilist, sfinstrm, sfload, sfpassign, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

## sfinstr3

`sfinstr3` -- Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound with cubic interpolation.

`sfinstr3`

## Description

Plays a SoundFont2 (SF2) sample instrument, generating a stereo sound with cubic interpolation. 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 *SoundFont2 File Format Appendix*.

## Syntax

```
ar1, ar2 sfinstr3 ivel, inotenum, xamp, xfreq, instrnum, ifilhandle [, iflag] [
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*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.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## 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.

When *iflag* = 0, *inotenum* 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.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfinstr3* is a cubic-interpolation version of *sfinstr*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.

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.

## See Also

*sfilist, sfinstr3m, sfinstrm, sfinstr, sfload, sfpassign, sfplay3, sfplay3m, sfplay, sfplaym, sfplist, sfpre-set*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

## **sfinstr3m**

**sfinstr3m** -- Plays a SoundFont2 (SF2) sample instrument, generating a mono sound with cubic interpolation.

**sfinstr3m**

### **Description**

Plays a SoundFont2 (SF2) sample instrument, generating a mono sound with cubic interpolation. 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 *SoundFont2 File Format Appendix*.

### **Syntax**

ares **sfinstr3m** *ivel*, *inotenum*, *xamp*, *xfreq*, *instrnum*, *ifilhandle* [, *iflag*] [, *i*

### **Initialization**

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*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.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

### **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.

When *iflag* = 0, *inotenum* 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.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfinstr3m* is a cubic-interpolation version of *sfinstrm*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.



These opcodes only support the sample structure of SF2 files. The modulator structure of the Sound-Font2 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.

## See Also

*sfilist, sfinstr3, sfinstr, sfinstrm, sfload, sfpassign, sfplay3, sfplay3m, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# **sfinstrm**

**sfinstrm** -- Plays a SoundFont2 (SF2) sample instrument, generating a mono sound.

**sfinstrm**

## **Description**

Plays a SoundFont2 (SF2) sample instrument, generating a mono sound. 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 *SoundFont2 File Format Appendix*.

## **Syntax**

ares **sfinstrm** *ivel*, *inotenum*, *xamp*, *xfreq*, *instrnum*, *ifilhandle* [, *iflag*] [, *io*]

## **Initialization**

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*instrnum* -- number of an instrument of a SF2 file.

*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.

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## **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.

When *iflag* = 0, *inotenum* 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.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*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.

## See Also

*sfilist, sfinstr, sfload, sfpassign, sfplay, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfload

`sfload --` Loads an entire SoundFont2 (SF2) sample file into memory.

`sfload`

## Description

Loads an entire SoundFont2 (SF2) sample file into memory. 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 *SoundFont2 File Format Appendix*.

*sfload* should be placed in the header section of a Csound orchestra.

## Syntax

```
ir sfload "filename"
```

## Initialization

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

*"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.

## Performance

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

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.

## See Also

*sfilist*, *sfinstr*, *sfinstrm*, *sfpassign*, *sfplay*, *sfplaym*, *sfplist*, *sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfpassign

`sfpassign` -- Assigns all presets of a SoundFont2 (SF2) sample file to a sequence of progressive index numbers.

`sfpassign`

## Description

Assigns all presets of a previously loaded SoundFont2 (SF2) sample file to a sequence of progressive index numbers. 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 *SoundFont2 File Format Appendix*.

*sfpassign* should be placed in the header section of a Csound orchestra.

## Syntax

**sfpassign** *istartindex*, *ifilhandle*

## Initialization

*istartindex* -- starting index preset by the user in bulk preset assignments.

*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.

## Performance

*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*. *istartindex* 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.

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.

## See Also

*sflist*, *sfinstr*, *sfinstrm*, *sfload*, *sfplay*, *sfplaym*, *sfplist*, *sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfplay

`sfplay` -- Plays a SoundFont2 (SF2) sample preset, generating a stereo sound.

`sfplay`

## Description

Plays a SoundFont2 (SF2) sample preset, generating a stereo sound. 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 *SoundFont2 File Format Appendix*.

## Syntax

```
ar1, ar2 sfplay ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## 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.

When *iflag* = 0, *inotenum* 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. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*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.

These opcodes only support the sample structure of SF2 files. The modulator structure of the Sound-

Font2 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.

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfpassign, sfplaym, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfplay3

`sfplay3` -- Plays a SoundFont2 (SF2) sample preset, generating a stereo sound with cubic interpolation.

`sfplay3`

## Description

Plays a SoundFont2 (SF2) sample preset, generating a stereo sound with cubic interpolation. 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 *SoundFont2 File Format Appendix*.

## Syntax

```
ar1, ar2 sfplay3 ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset]
```

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## 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.

When *iflag* = 0, *inotenum* 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 *sfplay3* will not work correctly. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfplay3* plays a preset, generating a stereo sound with cubic interpolation. *ivel* does not directly affect the amplitude of the output, but informs *sfplay3* about which sample should be chosen in multi-sample, velocity-split presets.



*sfplay3* is a cubic-interpolation version of *sfplay*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.

These opcodes only support the sample structure of SF2 files. The modulator structure of the Sound-Font2 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.

## See Also

*sfilist*, *sfinstr3*, *sfinstr3m*, *sfinstr*, *sfinstrm*, *sfload*, *sfpassign*, *sfplay3m*, *sfplaym*, *sfplay*, *sfplist*, *sfpre-set*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfplay3m

`sfplay3m` -- Plays a SoundFont2 (SF2) sample preset, generating a mono sound with cubic interpolation.

`sfplay3m`

## Description

Plays a SoundFont2 (SF2) sample preset, generating a mono sound with cubic interpolation. 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 *SoundFont2 File Format Appendix*.

## Syntax

ares **sfplay3m** *ivel*, *inotenum*, *xamp*, *xfreq*, *ipreindex* [, *iflag*] [, *ioffset*]

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## 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.

When *iflag* = 0, *inotenum* 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 *sfplay3m* will not work correctly. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably crash.

*sfplay3m* is a mono version of *sfplay3*. It should be used with mono preset, or with the stereo presets in which stereo output is not required. It is faster than *sfplay3*.

*sfplay3m* is also a cubic-interpolation version of *sfplaym*. Difference of sound-quality is noticeable specially in bass-frequency-transposed samples. In high-freq-transposed samples the difference is less noticeable, and I suggest to use linear-interpolation versions, because they are faster.

These opcodes only support the sample structure of SF2 files. The modulator structure of the Sound-Font2 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.

## See Also

*sfilist*, *sfinstr3*, *sfinstr3m*, *sfinstr*, *sfinstrm*, *sfload*, *sfpassign*, *sfplay3*, *sfplaym*, *sfplay*, *sfplist*, *sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfplaym

sfplaym -- Plays a SoundFont2 (SF2) sample preset, generating a mono sound.

sfplaym

## Description

Plays a SoundFont2 (SF2) sample preset, generating a mono sound. 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 *SoundFont2 File Format Appendix*.

## Syntax

ares **sfplaym** ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset]

## Initialization

*ivel* -- velocity value

*inotenum* -- MIDI note number value

*ipreindex* -- preset index

*iflag* (optional) -- flag regarding the behavior of *xfreq* and *inotenum*

*ioffset* (optional) -- start playing at offset, in samples.

## 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.

When *iflag* = 0, *inotenum* 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 *sfplaym* will not work correctly. *ipreindex* must contain the number of a previously assigned preset, or Csound will crash.

The *ioffset* parameter allows the sound to start from a sample different than the first one. The user should make sure that its value is within the length of the specific sound. Otherwise, Csound will probably 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*.

These opcodes only support the sample structure of SF2 files. The modulator structure of the Sound-

Font2 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.

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfpassign, sfplay, sfplist, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfplist

sfplist -- Prints a list of all presets of a SoundFont2 (SF2) sample file.

sfplist

## Description

Prints a list of all presets of a previously loaded SoundFont2 (SF2) sample file. 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 *SoundFont2 File Format Appendix*.

## Syntax

```
sfplist ifilhandle
```

## Initialization

*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.

## Performance

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

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.

## See Also

*sfilist, sfinstr, sfinstrm, sfload, sfpassign, sfplay, sfplaym, sfpreset*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# sfpreset

`sfpreset` -- Assigns an existing preset of a SoundFont2 (SF2) sample file to an index number.

`sfpreset`

## Description

Assigns an existing preset of a previously loaded SoundFont2 (SF2) sample file to an index number. 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 *SoundFont2 File Format Appendix*.

*sfpreset* should be placed in the header section of a Csound orchestra.

## Syntax

`ir` **sfpreset** `iprogram`, `ibank`, `ifilhandle`, `ipreindex`

## Initialization

*ir* -- output to be used by other SF2 opcodes. For *sfpreset*, *ir* is *ipreindex*.

*iprogram* -- program number of a bank of presets in a SF2 file

*ibank* -- number of a specific bank of a SF2 file

*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.

*ipreindex* -- preset index

## Performance

*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.

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.

## See Also

*sfilist*, *sfinstr*, *sfinstrm*, *sfload*, *sfpassign*, *sfplay*, *sfplaym*, *sfplist*

## Credits

Author: Gabriel Maldonado  
Italy  
May 2000

New in Csound Version 4.07

# shaker

shaker -- Sounds like the shaking of a maraca or similar gourd instrument.

shaker

## 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.

## Syntax

ares **shaker** kamp, kfreq, kbeans, kdamp, ktimes [, idecay]

## 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.



### Note

The argument *knum* was redundant, so it was removed in version 3.49.

## Examples

Here is an example of the shaker opcode. It uses the files *shaker.orc* [examples/shaker.orc] and *shaker.sco* [examples/shaker.sco].

### Example 348. Example of the shaker opcode.

```
/* shaker.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1
```



```
instr 1
  al shaker 10000, 440, 8, 0.999, 100, 0
  out al
endin
/* shaker.orc */
```

```
/* shaker.sco */
i 1 0 1
e
/* shaker.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

Fixed the example thanks to a message from Istvan Varga.

# sin

sin -- Performs a sine function.

sin

## Description

Returns the sine of  $x$  ( $x$  in radians).

## Syntax

**sin**( $x$ ) (no rate restriction)

## Examples

Here is an example of the sin opcode. It uses the files *sin.orc* [examples/sin.orc] and *sin.sco* [examples/sin.sco].

### Example 349. Example of the sin opcode.

```
/* sin.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 25
  il = sin(irad)

  print il
endin
/* sin.orc */

/* sin.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* sin.sco */
```

Its output should include a line like this:

```
instr 1:  il = -0.132
```

## See Also

*cos, cosh, cosinv, sinh, sininv, tan, tanh, taninv*

## Credits

Example written by Kevin Conder.

# sinh

sinh -- Performs a hyperbolic sine function.

sinh

## Description

Returns the hyperbolic sine of  $x$  ( $x$  in radians).

## Syntax

**sinh**( $x$ ) (no rate restriction)

## Examples

Here is an example of the sinh opcode. It uses the files *sinh.orc* [examples/sinh.orc] and *sinh.sco* [examples/sinh.sco].

### Example 350. Example of the sinh opcode.

```
/* sinh.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 1
  il = sinh(irad)

  print il
endin
/* sinh.orc */

/* sinh.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* sinh.sco */
```

Its output should a line like this:

```
instr 1:  il = 1.175
```

## See Also

*cos, cosh, cosinv, sin, sininv, tan, tanh, taninv*

## Credits

Example written by Kevin Conder.

# sininv

sininv -- Performs an arcsine function.

sininv

## Description

Returns the arcsine of  $x$  ( $x$  in radians).

## Syntax

**sininv**( $x$ ) (no rate restriction)

## Examples

Here is an example of the sininv opcode. It uses the files *sininv.orc* [examples/sininv.orc] and *sininv.sco* [examples/sininv.sco].

### Example 351. Example of the sininv opcode.

```
/* sininv.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 0.5
  il = sininv(irad)

  print il
endin
/* sininv.orc */

/* sininv.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* sininv.sco */
```

Its output should include a line like this:

```
instr 1:  il = 0.524
```

## See Also

*cos, cosh, cosinv, sin, sinh, tan, tanh, taninv*

## Credits

Example written by Kevin Conder.

# sinsyn

sinsyn -- Streaming partial track additive synthesis with cubic phase interpolation

sinsyn

## Description

The sinsyn opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by the partials opcode). It sinsynthesises the signal using linear amplitude and cubic phase interpolation to drive a bank of interpolating oscillators with amplitude and pitch scaling controls. Sinsyn attempts to preserve the phase of the partials in the original signal and in so doing it does not allow for pitch or timescale modifications of the signal.

## Syntax

asig **sinsyn** fin, kscal, kmaxtracks, ifn

## Performance

*asig* -- output audio rate signal

*fin* -- input pv stream in TRACKS format

*kscal* -- amplitude scaling

*kmaxtracks* -- max number of tracks in sinsynthesis. Limiting this will cause a non-linear filtering effect, by discarding newer and higher-frequency tracks (tracks are ordered by start time and ascending frequency, respectively)

*ifn* -- function table containing one cycle of a sinusoid (sine or cosine)

## Examples

### Example 352. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
    aout sinsyn fst, 1, 1.5, 500, 1 ; resynthesis (up a 5th)
out aout
```

The example above shows partial tracking of an ifd-analysis signal and cubic-phase additive resynthesis.

## Credits

Author: Victor Lazzarini;  
June 2005



New plugin in version 5

November 2004.

# sleighbells

sleighbells -- Semi-physical model of a sleighbell sound.

sleighbells

## Description

*sleighbells* is a semi-physical model of a sleighbell sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **sleighbells** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, i

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 32.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.9994 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.9994 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 0.03.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2500.

*ifreq1* (optional) -- the first resonant frequency. The default value is 5300.

*ifreq2* (optional) -- the second resonant frequency. The default value is 6500.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the sleighbells opcode. It uses the files *sleighbells.orc* [examples/sleighbells.orc] and *sleighbells.sco* [examples/sleighbells.sco].

### Example 353. Example of the sleighbells opcode.

```
/* sleighbells.orc */
sr = 22050
```

```
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1: An example of sleighbells.
instr 1
  a1 sleighbells 20000, 0.01

  out a1
endin
/* sleighbells.orc */
```

```
/* sleighbells.sco */
i 1 0.00 0.25
i 1 0.30 0.25
i 1 0.60 0.25
i 1 0.90 0.25
i 1 1.20 0.25
i 1 1.50 0.25
i 1 1.80 0.25
i 1 2.10 0.25
i 1 2.40 0.25
i 1 2.70 0.25
i 1 3.00 0.25
e
/* sleighbells.sco */
```

## See Also

*bamboo, dripwater, guiro, tambourine*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# slider16

slider16 -- Creates a bank of 16 different MIDI control message numbers.

slider16

## Description

Creates a bank of 16 different MIDI control message numbers.

## Syntax

*i1*, ..., *i16* **slider16** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum16*,

*k1*, ..., *k16* **slider16** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum16*,

## Initialization

*i1* ... *i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider16* is a bank 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.

*slider16* allows a bank of 16 different MIDI control message numbers.

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 *slider16*, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.

## See Also

*s16b14, s32b14, slider16f, slider32, slider32f, slider64, slider64f, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider16f

slider16f -- Creates a bank of 16 different MIDI control message numbers, filtered before output.

slider16f

## Description

Creates a bank of 16 different MIDI control message numbers, filtered before output.

## Syntax

*k1*, ..., *k16* **slider16f** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, *icutoff1*, ...,

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider16f* is a bank 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.

*slider16f* allows a bank of 16 different MIDI control message numbers. It filters the signal before output. This eliminates 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.



### Warning

*slider16f* does not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14, s32b14, slider16, slider32, slider32f, slider64, slider64f, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider32

slider32 -- Creates a bank of 32 different MIDI control message numbers.

slider32

## Description

Creates a bank of 32 different MIDI control message numbers.

## Syntax

*i1*, ..., *i32* **slider32** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum32*,

*k1*, ..., *k32* **slider32** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum32*,

## Initialization

*i1* ... *i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider32* is a bank 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.

*slider32* allows a bank of 32 different MIDI control message numbers.

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 *slider32*, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.



## See Also

*s16b14, s32b14, slider16, slider16f, slider32f, slider64, slider64f, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider32f

slider32f -- Creates a bank of 32 different MIDI control message numbers, filtered before output.

slider32f

## Description

Creates a bank of 32 different MIDI control message numbers, filtered before output.

## Syntax

*k1*, ..., *k32* **slider32f** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, *icutoff1*, ...,

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider32f* is a bank 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.

*slider32f* allows a bank of 32 different MIDI control message numbers. It filters the signal before output. This eliminates 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.



### Warning

*slider32f* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider64, slider64f, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

## slider64

slider64 -- Creates a bank of 64 different MIDI control message numbers.

slider64

## Description

Creates a bank of 64 different MIDI control message numbers.

## Syntax

*i1*, ..., *i64* **slider64** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum64*,

*k1*, ..., *k64* **slider64** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum64*,

## Initialization

*i1* ... *i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider64* is a bank 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.

*slider64* allows a bank of 64 different MIDI control message numbers.

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 *slider64*, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider32f, slider64f slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider64f

slider64f -- Creates a bank of 64 different MIDI control message numbers, filtered before output.

slider64f

## Description

Creates a bank of 64 different MIDI control message numbers, filtered before output.

## Syntax

*k1*, ..., *k64* **slider64f** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, *icutoff1*, ...,

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider64f* is a bank 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.

*slider64f* allows a bank of 64 different MIDI control message numbers. It filters the signal before output. This eliminates 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.



### Warning

*slider64f* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider32f, slider64, slider8, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

## slider8

slider8 -- Creates a bank of 8 different MIDI control message numbers.

slider8

## Description

Creates a bank of 8 different MIDI control message numbers.

## Syntax

*i1*, ..., *i8* **slider8** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum8*, *imi*

*k1*, ..., *k8* **slider8** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, ..., *ictlnum8*, *imi*

## Initialization

*i1* ... *i64* -- output values

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider8* is a bank 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.

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 *slider8*, there is not an initial value input argument, because the output is gotten directly from current status of internal controller array of Csound.



## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider32f, slider64, slider64f, slider8f*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# slider8f

slider8f -- Creates a bank of 8 different MIDI control message numbers, filtered before output.

slider8f

## Description

Creates a bank of 8 different MIDI control message numbers, filtered before output.

## Syntax

*k1*, ..., *k8* **slider8f** *ichan*, *ictlnum1*, *imin1*, *imax1*, *init1*, *ifn1*, *icutoff1*, ..., *ic*

## Initialization

*ichan* -- MIDI channel (1-16)

*ictlnum1* ... *ictlnum64* -- MIDI control number (0-127)

*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

*slider8f* is a bank 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.

*slider8f* allows a bank of 8 different MIDI control message numbers. It filters the signal before output. This eliminates 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.



### Warning

*slider8f* opcodes do not output the required initial value immediately, but only after some k-cycles because the filter slightly delays the output.

## See Also

*s16b14, s32b14, slider16, slider16f, slider32, slider32f, slider64, slider64f, slider8*

## Credits

Author: Gabriel Maldonado  
Italy  
December 1998

New in Csound version 3.50

Thanks goes to Rasmus Ekman for pointing out the correct MIDI channel and controller number ranges.

# sndloop

sndloop -- A sound looper with pitch control.

sndloop

## Description

This opcode records input audio and plays it back in a loop with user-defined duration and crossfade time. It also allows the pitch of the loop to be controlled, including reversed playback.

## Syntax

*asig*, *krec* **sndloop** *ain*, *kpitch*, *ktrig*, *idur*, *ifad*

## Initialisation

*idur* -- loop duration in seconds

*ifad* -- crossfade duration in seconds

## Performance

*asig* -- output sig

*krec* -- 'rec on' signal, 1 when recording, 0 otherwise

*kpitch* -- pitch control (transposition ratio); negative values play the loop back in reverse

*kon* --on signal: when 0, processing is bypassed. When switched on ( $kon \geq 1$ ), the opcode starts recording until the loop memory is full. It then plays the looped sound until it is switched off again ( $kon = 0$ ). Another recording can start again with  $kon \geq 1$ .

## Examples

### Example 354. Example

```
asig in                                ; get the signal in
ktrig line 0, 1, 1                      ; trigger signal
aout,krec sndloop asig, 1, ktrig, 4, 0.05 ; rec starts at 1 sec, for 4 secs 0
printk 1, krec                          ; prints the recording signal
out aout
```

The example above shows the basic operation of sndloop. Pitch can be controlled at the k-rate, recording is started as soon as the trigger value is  $\geq 1$ . Recording can be restarted by making the trigger 0 and then 1 again.

## Credits

Author: Victor Lazzarini;

April 2005

New plugin in version 5

April 2005.

# sndwarp

`sndwarp` -- Reads a mono sound sample from a table and applies time-stretching and/or pitch modification.

`sndwarp`

## 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 *iws*ize=*sr*/10 and *io*verlap=15. Try *ir*andw=*iws*ize\*.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.

## Syntax

ares [, ac] **sndwarp** xamp, xtimewarp, xresample, ifn1, ibeg, iwsiz, irandw, io

## 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*.

*iwsiz* -- 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 *iwsiz*.

*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

*ares* -- the single channel of output from the `sndwarp` unit generator. `sndwarp` assumes that the function table holding the sampled signal is a mono one. This simply means that `sndwarp` will index the table by single-sample frame increments. The user must be aware then that a stereo signal is used with `sndwarp`, time and pitch will be altered accordingly.

*ac* (optional) -- a 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* can greatly enhance the quality of sound.

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

*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, *xtimewarp* 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.

## Examples

Here is an example of the *sndwarp* opcode. It uses the files *sndwarp.orc* [examples/sndwarp.orc], *sndwarp.sco* [examples/sndwarp.sco], and *mary.wav* [examples/mary.wav].

### Example 355. Example of the *sndwarp* opcode.

```
/* sndwarp.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - play an audio file.
instr 1
  ; Use the audio file defined in Table #1.
  al loscil 30000, 1, 1, 1

  out al
endin

; Instrument #2 - time-stretch an audio file.
instr 2
  kamp init 6500
  ; Start at 1 second and end at 3.5 seconds.
  ktimewarp line 1, p3, 3.5
  ; Playback at the normal speed.
  kresample init 1
  ; Use the audio file defined in Table #1.
  ifn1 = 1
  ibeg = 0
  iwsiz = 4410
  irandw = 882
  ioverlap = 15
  ; Use Table #2 for the windowing function.
  ifn2 = 2
  ; Use the ktimewarp parameter as a "time" pointer.
  itimemode = 1

  al sndwarp kamp, ktimewarp, kresample, ifn1, ibeg, iwsiz, irandw, ioverlap,
  out al
endin
/* sndwarp.orc */
```

```
/* sndwarp.sco */
; Table #1: an audio file.
f 1 0 262144 1 "mary.wav" 0 0 0
; Table #2: half of a sine wave.
f 2 0 16384 9 0.5 1 0

; Play Instrument #1 for 3.5 seconds.
i 1 0 3.5
; Play Instrument #2 for 7 seconds (time-stretched).
i 2 3.5 10.5
e
/* sndwarp.sco */
```

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, 1
aresamp line 1, p3, 2
kenv line 1, p3, .1
asig sndwarp kenv, awarp, aresamp, isampfun, ibeg, iwindsize, iwindrand, ioverlap
```

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
```

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
abal balance asig, acmp

asig1,asig2,acmp1,acmp2 sndwarpst 1, atime, aresamp, sampfun, ibeg, iwindsize, ioverlap
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!

## See Also

*sndwarpst*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997

Example written by Kevin Conder.

# sndwarpst

sndwarpst -- Reads a stereo sound sample from a table and applies time-stretching and/or pitch modification.

sndwarpst

## Description

*sndwarpst* reads stereo 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 *iwsiz*=sr/10 and *ioverlap*=15. Try *irandw*=*iwsiz*\*.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.

## Syntax

*ar1*, *ar2* [, *ac1*] [, *ac2*] **sndwarpst** *xamp*, *xtimewarp*, *xresample*, *ifn1*, *ibeg*, *iwsiz*

## 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*.

*iwsiz* -- 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 *iwsiz*.

*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

*ar1*, *ar2* -- *ar1* and *ar2* are the stereo (left and right) outputs from *sndwarpst*. *sndwarpst* assumes that the function table holding the sampled signal is a stereo one. *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*, time and pitch will be altered accordingly.

*ac1*, *ac2* -- *ac1* and *ac2* 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 *sndwarpst* 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, *ac1* and *ac2* can greatly enhance the quality of sound. They are optional, but note that they must both be present in the syntax (use both or neither). An ex-

ample 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 *ac1* and *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, *xtimewarp* 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.

## Examples

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, 1
aresamp line    1, p3, 2
kenv    line    1, p3, .1
asig     sndwarp kenv, awarp, aresamp, isampfun, ibeg, iwindsize, iwindrand, ioverlap
```

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
```

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
abal          balance asig, acmp

asig1,asig2,acmp1,acmp2 sndwarpst 1, atime, aresamp, sampfun, ibeg, iwindsize, iwindrand, ioverlap
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!

## See Also

*sndwarp*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997

# soundin

soundin -- Reads audio data from an external device or stream.

soundin

## Description

Reads audio data from an external device or stream. Up to 24 channels may be read.

## Syntax

ar1[, ar2[, ar3[, ... a24]]] **soundin** ifilcod [, iskptim] [, iformat] [, iskipin

## 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, default=0) -- time in seconds of input sound to be skipped. The default value is 0. In csound 5.00 and later, this may be negative to add a delay instead of skipping time.

*iformat* (optional, default=0) -- 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
- 7 = 8-bit unsigned int (not available in Csound versions older than 5.00)
- 8 = 24-bit int (not available in Csound versions older than 5.00)
- 9 = 64-bit doubles (not available in Csound versions older than 5.00)

*iskipinit* -- switches off all initialisation if non zero (default=0). This was introduced in 4\_23f13 and csound5.

*ibufsize* -- buffer size in mono samples (not sample frames). Not available in Csound versions older than 5.00. The default buffer size is 2048.

If *iformat* = 0 it is taken from the soundfile header, and if no header from the Csound -o command-line flag. The default value is 0.

## Performance

*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* opcode 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* opcodes within a single instrument or orchestra. Two or more of them can read simultaneously from the same external file.



### Note to Windows users

Windows users typically use back-slashes, “\”, when specifying the paths of their files. As an example, a Windows user might use the path “c:\music\samples\loop001.wav”. This is problematic because back-slashes are normally used to specify special characters.

To correctly specify this path in Csound, one may alternately:

- *Use forward slashes:* c:/music/samples/loop001.wav
- *Use back-slash special characters, “\\”:* c:\\music\\samples\\loop001.wav

## Examples

Here is an example of the *soundin* opcode. It uses the files *soundin.orc* [examples/soundin.orc], *soundin.sco* [examples/soundin.sco], *beats.wav* [examples/beats.wav].

### Example 356. Example of the *soundin* opcode.

```
/* soundin.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1 - play an audio file.
instr 1
  asig soundin "beats.wav"
  out asig
endin
/* soundin.orc */

/* soundin.sco */
; Play Instrument #1, the audio file, for three seconds.
i 1 0 3
e
/* soundin.sco */
```

## See Also

*diskin*, *in*, *inh*, *ino*, *inq*, *ins*

## Credits

Authors: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997

Example written by Kevin Conder.

Warning to Windows users added by Kevin Conder, April 2002

# soundout

soundout -- Writes audio output to a disk file.

soundout

## Description

Writes audio output to a disk file.

## Syntax

```
soundout  asig1, ifilcod [, iformat]
```

## 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*.

*iformat* (optional, default=0) -- 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-line flag. The default value is 0.

## Performance

*soundout* writes audio output to a disk file.

## See Also

*out, outh, outo, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2* soundouts

## Credits

Author: Barry L. Vercoe, Matt Ingalls/Mike Berry  
MIT, Mills College  
1993-1997



# soundouts

soundouts -- Writes audio output to a disk file.

soundouts

## Description

Writes audio output to a disk file.

## Syntax

**soundouts** *asigl, asigr, ifilcod* [, *iformat*]

## Initialization

*ifilcod* -- integer or character-string denoting the destination soundfile name. An integer denotes the file soundout.ifilcod; 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 written relative to the directory given by the SFDIR environment variable if defined, or the current directory. See also *GEN01*.

*iformat* (optional, default=0) -- specifies the audio data file format:

- 1 = 8-bit signed char (high-order 8 bits of a 16-bit integer)
- 4 = 16-bit short integers
- 5 = 32-bit long integers
- 6 = 32-bit floats

If *iformat* = 0 it is taken from the Csound -o command-line flag. The default value is 0.

## Performance

*soundouts* writes stereo audio output to a disk file in raw (headerless) format without 0dBFS scaling. The expected range of the audio signals depends on the selected sample format.

## See Also

*out, outh, outh, outq, outq1, outq2, outq3, outq4, outs, outs1, outs2 soundout*

## Credits

Author: Istvan Varga

# space

*space* -- Distributes an input signal among 4 channels using cartesian coordinates.

*space*

## 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. The advantages to the former are:

1. A graphic user interface can be used to draw and edit the trajectory through the Cartesian plane
2. 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.

## Syntax

a1, a2, a3, a4   **space** asig, ifn, ktime, kverbsend, kx, ky

## Initialization

*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*.

## 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 signals from *space*, it can be used in a 2 channel orchestra. If the xy's are kept so that  $Y \geq 1$ , it should work well to do panning and fixed localization in a stereo field.

*asig* -- input audio signal.

*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.

*kreverb**send* -- 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.

## Examples

```
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 be useful for quad and stereo panning as well as fixed placement 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  
        il 0 1 -1 1  
        ;place the sound in the right speaker and far  
        il 1 1 45 45  
        ;place the sound equally between left and right and in the middle ground dist.  
        il 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!

## See Also

*spdist*, *spsend*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# spat3d

`spat3d` -- Positions the input sound in a 3D space and allows moving the sound at k-rate.

`spat3d`

## Description

This opcode positions the input sound in a 3D space, with optional simulation of room acoustics, in various output formats. *spat3d* allows moving the sound at k-rate (this movement is interpolated internally to eliminate "zipper noise" if sr not equal to kr).

## Syntax

`aW, aX, aY, aZ spat3d ain, kX, kY, kZ, idist, ift, imode, imdel, iovr [, istor]`

## Initialization

*idist* -- For modes 0 to 3, *idist* is the unit circle distance in meters. For mode 4, *idist* is the distance between microphones.

The following formulas describe amplitude and delay as a function of sound source distance from microphone(s):

$$\text{amplitude} = 1 / (0.1 + \text{distance})$$

$$\text{delay} = \text{distance} / 340 \text{ (in seconds)}$$

Distance can be calculated as:

$$\text{distance} = \sqrt{iX^2 + iY^2 + iZ^2}$$

In Mode 4, distance can be calculated as:

$$\begin{aligned} \text{distance from left mic} &= \sqrt{(iX + idist/2)^2 + iY^2 + iZ^2} \\ \text{distance from right mic} &= \sqrt{(iX - idist/2)^2 + iY^2 + iZ^2} \end{aligned}$$

With *spat3d* the distance between the sound source and any microphone should be at least  $(340 * 18) / \text{sr}$  meters. Shorter distances will work, but may produce artifacts in some cases. There is no such limitation for *spat3di* and *spat3dt*.

Sudden changes or discontinuities in sound source location can result in pops or clicks. Very fast movement may also degrade quality.

*ift* -- Function table storing room parameters (for free field spatialization, set it to zero or negative). Table size is 54. The values in the table are:

Room Parameter	Purpose
0	Early reflection recursion depth (0 is the sound source, 1 is the first reflection etc.) for spat3d and spat3di. The number of echoes for four walls (front, back, right, left) is: $N = (2 * R + 2) * R$ . If all six walls are enabled: $N = (((4 * R + 6) * R + 8) * R) / 3$
1	Late reflection recursion depth (used by spat3dt only). spat3dt skips early reflections and renders echoes up to this level. If early reflection depth is negative, spat3d and spat3di will output zero, while spat3dt will start rendering from the sound source.
2	imdel for spat3d. Overrides opcode parameter if non-negative.
3	irlen for spat3dt. Overrides opcode parameter if non-negative.
4	idist value. Overrides opcode parameter if $\geq 0$ .
5	Random seed (0 - 65535) -1 seeds from current time.
6 - 53	wall parameters (w = 6: ceil, w = 14: floor, w = 22: front, w = 30: back, w = 38: right, w = 46: left)
w + 0	Enable reflections from this wall (0: no, 1: yes)
w + 1	Wall distance from listener (in meters)
w + 2	Randomization of wall distance (0 - 1) (in units of $1 / (\text{wall distance})$ )
w + 3	Reflection level (-1 - 1)
w + 4	Parametric equalizer frequency in Hz.
w + 5	Parametric equalizer level (1.0: no filtering)
w + 6	Parametric equalizer Q (0.7071: no resonance)
w + 7	Parametric equalizer mode (0: peak EQ, 1: low shelf, 2: high shelf)

*imode* -- Output mode

- 0: B format with W output only (mono)

aout = aW

- 1: B format with W and Y output (stereo)

aleft = aW + 0.7071\*aY  
aright = aW - 0.7071\*aY

- 2: B format with W, X, and Y output (2D). This can be converted to UHJ:

aWre, aWim    hilbert aW  
aXre, aXim    hilbert aX  
aYre, aYim    hilbert aY

```
aWXr = 0.0928*aXre + 0.4699*aWre
aWXiYr = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aleft = aWXr + aWXiYr
aright = aWXr - aWXiYr
```

- 3: B format with all outputs (3D)
- 4: Simulates a pair of microphones (stereo output)

```
aW    butterlp aW, ifreq    ; recommended values for ifreq
aY    butterlp aY, ifreq    ; are around 1000 Hz
aleft = aW + aX
aright = aY + aZ
```

Mode 0 is the cheapest to calculate, while mode 4 is the most expensive.

In Mode 4, The optional lowpass filters can change the frequency response depending on direction. For example, if the sound source is located left to the listener then the high frequencies are attenuated in the right channel and slightly increased in the left. This effect can be disabled by not using filters. You can also experiment with other filters (tone etc.) for better effect.

Note that mode 4 is most useful for listening with headphones, and is also more expensive to calculate than the B-format (0 to 3) modes. The *idist* parameter in this case sets the distance between left and right microphone; for headphones, values between 0.2 - 0.25 are recommended, although higher settings up to 0.4 may be used for wide stereo effects.

More information about B format can be found here: [http://www.york.ac.uk/inst/mustech/3d\\_audio/ambis2.htm](http://www.york.ac.uk/inst/mustech/3d_audio/ambis2.htm)

*imdel* -- Maximum delay time for spat3d in seconds. This has to be longer than the delay time of the latest reflection (depends on room dimensions, sound source distance, and recursion depth; using this formula gives a safe (although somewhat overestimated) value:

$$\text{imdel} = (R + 1) * \sqrt{W*W + H*H + D*D} / 340.0$$

where R is the recursion depth, W, H, and D are the width, height, and depth of the room, respectively).

*iovr* -- Oversample ratio for spat3d (1 to 8). Setting it higher improves quality at the expense of memory and CPU usage. The recommended value is 2.

*istor* (optional, default=0) -- Skip initialization if non-zero (default: 0).

## Performance

*aW*, *aX*, *aY*, *aZ* -- Output signals

	mode 0	mode 1	mode 2	mode 3	mode 4
aW	W out	W out	W out	W out	left chn / low freq.
aX	0	0	X out	X out	left chn / high freq.
aY	0	Y out	Y out	Y out	right chn / low freq.



	mode 0	mode 1	mode 2	mode 3	mode 4
aZ	0	0	0	Z out	right chn / high fr.

*ain* -- Input signal

*kX*, *kY*, *kZ* -- Sound source coordinates (in meters)

If you encounter very slow performance (up to 100 times slower), it may be caused by denormals (this is also true of many other IIR opcodes, including *butterlp*, *pareq*, *hilbert*, and many others). Underflows can be avoided by:

- Using the *denorm* opcode on *ain* before *spat3d*.
- mixing low level DC or noise to the input signal, e.g.

```
atmp rnd31 1/1e24, 0, 0
```

```
aW, aX, aY, aZ spa3di ain + atmp, ...
```

or

```
aW, aX, aY, aZ spa3di ain + 1/1e24, ...
```

- reducing *irlen* in the case of *spat3dt* (which does not have an input signal). A value of about 0.005 is suitable for most uses, although it also depends on EQ settings. If the equalizer is not used, “*irlen*” can be set to 0.

## Examples

Here is an example of the *spat3d* opcode that outputs a stereo file. It uses the files *spat3d\_stereo.orc* [examples/spat3d\_stereo.orc] and *spat3d\_stereo.sco* [examples/spat3d\_stereo.sco].

### Example 357. Stereo example of the *spat3d* opcode.

```
/* spat3d_stereo.orc */
/* Written by Istvan Varga */
sr      = 48000
kr      = 1000
ksmps   = 48
nchnls  = 2

/* room parameters */
idep     = 3      /* early reflection depth      */

itmp     ftgen    1, 0, 64, -2, \
/* depth1, depth2, max delay, IR length, idist, seed */ \
idep, 48, -1, 0.01, 0.25, 123, \
1, 21.982, 0.05, 0.87, 4000.0, 0.6, 0.7, 2, /* ceil */ \
1, 1.753, 0.05, 0.87, 3500.0, 0.5, 0.7, 2, /* floor */ \
1, 15.220, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* front */ \
1, 9.317, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* back */ \
1, 17.545, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* right */ \
1, 12.156, 0.05, 0.87, 5000.0, 0.8, 0.7, 2 /* left */
```

```
instr 1

/* some source signal */

a1      phasor 150                ; oscillator
a1      butterbp a1, 500, 200    ; filter
a1      = taninv(a1 * 100)
a2      phasor 3                  ; envelope
a2      mirror 40*a2, -100, 5
a2      limit a2, 0, 1
a1      = a1 * a2 * 9000

kazim   line 0, 2.5, 360          ; move sound source around
kdist   line 1, 10, 4             ; distance

; convert polar coordinates
kX      = sin(kazim * 3.14159 / 180) * kdist
kY      = cos(kazim * 3.14159 / 180) * kdist
kZ      = 0

a1      = a1 + 0.000001 * 0.000001 ; avoid underflows

imode   = 1 ; change this to 3 for 8 spk in a cube,
          ; or 1 for simple stereo

aW, aX, aY, aZ spat3d a1, kX, kY, kZ, 1.0, 1, imode, 2, 2

aW      = aW * 1.4142

; stereo
;
aL      = aW + aY                /* left */
aR      = aW - aY                /* right */

; quad (square)
;
;aFL     = aW + aX + aY          /* front left */
;aFR     = aW + aX - aY          /* front right */
;aRL     = aW - aX + aY          /* rear left */
;aRR     = aW - aX - aY          /* rear right */

; eight channels (cube)
;
;aUFL    = aW + aX + aY + aZ     /* upper front left */
;aUFR    = aW + aX - aY + aZ     /* upper front right */
;aURL    = aW - aX + aY + aZ     /* upper rear left */
;aURR    = aW - aX - aY + aZ     /* upper rear right */
;aLFL    = aW + aX + aY - aZ     /* lower front left */
;aLFR    = aW + aX - aY - aZ     /* lower front right */
;aLRL    = aW - aX + aY - aZ     /* lower rear left */
;aLRR    = aW - aX - aY - aZ     /* lower rear right */

outs aL, aR

endin
/* spat3d_stereo.orc */

/* spat3d_stereo.sco */
/* Written by Istvan Varga */
i 1 0 10
e
/* spat3d_stereo.sco */
```

---

Here is an example of the `spat3d` opcode that outputs a UHJ file. It uses the files `spat3d_UHJ.orc` [examples/spat3d\_UHJ.orc] and `spat3d_UHJ.sco` [examples/spat3d\_UHJ.sco].

### Example 358. UHJ example of the `spat3d` opcode.

```
/* spat3d_UHJ.orc */
/* Written by Istvan Varga */
sr = 48000
kr = 750
ksmps = 64
nchnls = 2

itmp      ftgen      1, 0, 64, -2,
/* depth1, depth2, max delay, IR length, idist, seed */ \
3, 48, -1, 0.01, 0.25, 123, \
1, 21.982, 0.05, 0.87, 4000.0, 0.6, 0.7, 2, /* ceil */ \
1, 1.753, 0.05, 0.87, 3500.0, 0.5, 0.7, 2, /* floor */ \
1, 15.220, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* front */ \
1, 9.317, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* back */ \
1, 17.545, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* right */ \
1, 12.156, 0.05, 0.87, 5000.0, 0.8, 0.7, 2 /* left */ \

instr 1

p3 = p3 + 1.0

kazim line 0.0, 4.0, 360.0      ; azimuth
kelev line 40, p3 - 1.0, -20    ; elevation
kdist = 2.0                    ; distance
; convert coordinates
kX = kdist * cos(kelev * 0.01745329) * sin(kazim * 0.01745329)
kY = kdist * cos(kelev * 0.01745329) * cos(kazim * 0.01745329)
kZ = kdist * sin(kelev * 0.01745329)

; source signal
a1 phasor 160.0
a2 delay1 a1
a1 = a1 - a2
kffrq1 port 200.0, 0.8, 12000.0
affrq upsamp kffrq1
affrq pareq affrq, 5.0, 0.0, 1.0, 2
kffrq downsamp affrq
aenv4 phasor 3.0
aenv4 limit 2.0 - aenv4 * 8.0, 0.0, 1.0
a1 butterbp a1 * aenv4, kffrq, 160.0
aenv linseg 1.0, p3 - 1.0, 1.0, 0.04, 0.0, 1.0, 0.0
a_ = 4000000 * a1 * aenv + 0.00000001

; spatialize
a_W, a_X, a_Y, a_Z spat3d a_, kX, kY, kZ, 1.0, 1, 2, 2.0, 2

; convert to UHJ format (stereo)
aWre, aWim hilbert a_W
aXre, aXim hilbert a_X
aYre, aYim hilbert a_Y

aWXre = 0.0928*aXre + 0.4699*aWre
aWXim = 0.2550*aXim - 0.1710*aWim

aL = aWXre + aWXim + 0.3277*aYre
aR = aWXre - aWXim - 0.3277*aYre

outs aL, aR

endin
/* spat3d_UHJ.orc */
```

---

```
/* spat3d_UHJ.sco */
/* Written by Istvan Varga */
t 0 60

i 1 0.0 8.0
e
/* spat3d_UHJ.sco */
```

Here is a example of the spat3d opcode that outputs a quadrophonic file. It uses the files *spat3d\_quad.orc* [examples/spat3d\_quad.orc] and *spat3d\_quad.sco* [examples/spat3d\_quad.sco].

### Example 359. Quadrophonic example of the spat3d opcode.

```
/* spat3d_quad.orc */
/* Written by Istvan Varga */
sr      = 48000
kr      = 1000
ksmps   = 48
nchnls  = 4

/* room parameters */

idep     = 3      /* early reflection depth      */

itmp     ftgen    1, 0, 64, -2,
/* depth1, depth2, max delay, IR length, idist, seed */ \
idep, 48, -1, 0.01, 0.25, 123, \
1, 21.982, 0.05, 0.87, 4000.0, 0.6, 0.7, 2, /* ceil */ \
1, 1.753, 0.05, 0.87, 3500.0, 0.5, 0.7, 2, /* floor */ \
1, 15.220, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* front */ \
1, 9.317, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* back */ \
1, 17.545, 0.05, 0.87, 5000.0, 0.8, 0.7, 2, /* right */ \
1, 12.156, 0.05, 0.87, 5000.0, 0.8, 0.7, 2 /* left */ \

instr 1

/* some source signal */

a1      phasor 150          ; oscillator
a1      butterbp a1, 500, 200 ; filter
a1      = taninv(a1 * 100)
a2      phasor 3           ; envelope
a2      mirror 40*a2, -100, 5
a2      limit a2, 0, 1
a1      = a1 * a2 * 9000

kazim   line 0, 2.5, 360    ; move sound source around
kdist   line 1, 10, 4      ; distance

; convert polar coordinates
kX      = sin(kazim * 3.14159 / 180) * kdist
kY      = cos(kazim * 3.14159 / 180) * kdist
kZ      = 0

a1      = a1 + 0.000001 * 0.000001 ; avoid underflows
```

```
imode    = 2      ; change this to 3 for 8 spk in a cube,
                  ; or 1 for simple stereo

aW, aX, aY, aZ  spat3d a1, kX, kY, kZ, 1.0, 1, imode, 2, 2

aW        = aW * 1.4142

; stereo
;
;aL        = aW + aY          /* left          */
;aR        = aW - aY          /* right         */

; quad (square)
;
aFL        = aW + aX + aY      /* front left    */
aFR        = aW + aX - aY      /* front right   */
aRL        = aW - aX + aY      /* rear left     */
aRR        = aW - aX - aY      /* rear right    */

; eight channels (cube)
;
;aUFL      = aW + aX + aY + aZ /* upper front left */
;aUFR      = aW + aX - aY + aZ /* upper front right */
;aURL      = aW - aX + aY + aZ /* upper rear left  */
;aURR      = aW - aX - aY + aZ /* upper rear right */
;aLFL      = aW + aX + aY - aZ /* lower front left  */
;aLFR      = aW + aX - aY - aZ /* lower front right */
;aLRL      = aW - aX + aY - aZ /* lower rear left   */
;aLRR      = aW - aX - aY - aZ /* lower rear right  */

      outq aFL, aFR, aRL, aRR

      endin
/* spat3d_quad.orc */

/* spat3d_quad.sco */
/* Written by Istvan Varga */
t 0 60
i 1 0 10
e
/* spat3d_quad.sco */
```

## See Also

*spat3di*, *spat3dt*

## Credits

Author: Istvan Varga  
2001

New in version 4.12

Updated April 2002 by Istvan Varga

# spat3di

spat3di -- Positions the input sound in a 3D space with the sound source position set at i-time.

spat3di

## Description

This opcode positions the input sound in a 3D space, with optional simulation of room acoustics, in various output formats. With *spat3di*, sound source position is set at i-time.

## Syntax

aW, aX, aY, aZ **spat3di** ain, iX, iY, iZ, idist, ift, imode [, istor]

## Initialization

iX -- Sound source X coordinate in meters (positive: right, negative: left)

iY -- Sound source Y coordinate in meters (positive: front, negative: back)

iZ -- Sound source Z coordinate in meters (positive: up, negative: down)

idist -- For modes 0 to 3, *idist* is the unit circle distance in meters. For mode 4, *idist* is the distance between microphones.

The following formulas describe amplitude and delay as a function of sound source distance from microphone(s):

amplitude =  $1 / (0.1 + \text{distance})$

delay =  $\text{distance} / 340$  (in seconds)

Distance can be calculated as:

distance =  $\sqrt{iX^2 + iY^2 + iZ^2}$

In Mode 4, distance can be calculated as:

distance from left mic =  $\sqrt{(iX + idist/2)^2 + iY^2 + iZ^2}$

distance from right mic =  $\sqrt{(iX - idist/2)^2 + iY^2 + iZ^2}$

With *spat3d* the distance between the sound source and any microphone should be at least  $(340 * 18) / \text{sr}$  meters. Shorter distances will work, but may produce artifacts in some cases. There is no such limitation for *spat3di* and *spat3dt*.

Sudden changes or discontinuities in sound source location can result in pops or clicks. Very fast movement may also degrade quality.

*ift* -- Function table storing room parameters (for free field spatialization, set it to zero or negative). Table size is 54. The values in the table are:

Room Parameter	Purpose
0	Early reflection recursion depth (0 is the sound source, 1 is the first reflection etc.) for spat3d and spat3di. The number of echoes for four walls (front, back, right, left) is: $N = (2 * R + 2) * R$ . If all six walls are enabled: $N = (((4 * R + 6) * R + 8) * R) / 3$
1	Late reflection recursion depth (used by spat3dt only). spat3dt skips early reflections and renders echoes up to this level. If early reflection depth is negative, spat3d and spat3di will output zero, while spat3dt will start rendering from the sound source.
2	imdel for spat3d. Overrides opcode parameter if non-negative.
3	irlen for spat3dt. Overrides opcode parameter if non-negative.
4	idist value. Overrides opcode parameter if $\geq 0$ .
5	Random seed (0 - 65535) -1 seeds from current time.
6 - 53	wall parameters (w = 6: ceil, w = 14: floor, w = 22: front, w = 30: back, w = 38: right, w = 46: left)
w + 0	Enable reflections from this wall (0: no, 1: yes)
w + 1	Wall distance from listener (in meters)
w + 2	Randomization of wall distance (0 - 1) (in units of $1 / (\text{wall distance})$ )
w + 3	Reflection level (-1 - 1)
w + 4	Parametric equalizer frequency in Hz.
w + 5	Parametric equalizer level (1.0: no filtering)
w + 6	Parametric equalizer Q (0.7071: no resonance)
w + 7	Parametric equalizer mode (0: peak EQ, 1: low shelf, 2: high shelf)

*imode* -- Output mode

- 0: B format with W output only (mono)

aout = aW

- 1: B format with W and Y output (stereo)

aleft = aW + 0.7071\*aY  
aright = aW - 0.7071\*aY

- 2: B format with W, X, and Y output (2D). This can be converted to UHJ:

```
aWre, aWim    hilbert aW
aXre, aXim    hilbert aX
aYre, aYim    hilbert aY
aWXr  = 0.0928*aXre + 0.4699*aWre
aWXiYr = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aleft  = aWXr + aWXiYr
aright = aWXr - aWXiYr
```

- 3: B format with all outputs (3D)
- 4: Simulates a pair of microphones (stereo output)

```
aW    butterlp aW, ifreq    ; recommended values for ifreq
aY    butterlp aY, ifreq    ; are around 1000 Hz
aleft = aW + aX
aright = aY + aZ
```

Mode 0 is the cheapest to calculate, while mode 4 is the most expensive.

In Mode 4, The optional lowpass filters can change the frequency response depending on direction. For example, if the sound source is located left to the listener then the high frequencies are attenuated in the right channel and slightly increased in the left. This effect can be disabled by not using filters. You can also experiment with other filters (tone etc.) for better effect.

Note that mode 4 is most useful for listening with headphones, and is also more expensive to calculate than the B-format (0 to 3) modes. The *idist* parameter in this case sets the distance between left and right microphone; for headphones, values between 0.2 - 0.25 are recommended, although higher settings up to 0.4 may be used for wide stereo effects.

More information about B format can be found here: [http://www.york.ac.uk/inst/mustech/3d\\_audio/ambis2.htm](http://www.york.ac.uk/inst/mustech/3d_audio/ambis2.htm)

*istor* (optional, default=0) -- Skip initialization if non-zero (default: 0).

## Performance

*ain* -- Input signal

*aW*, *aX*, *aY*, *aZ* -- Output signals

	mode 0	mode 1	mode 2	mode 3	mode 4
aW	W out	W out	W out	W out	left chn / low freq.
aX	0	0	X out	X out	left chn / high freq.
aY	0	Y out	Y out	Y out	right chn / low freq.
aZ	0	0	0	Z out	right chn / high fr.

If you encounter very slow performance (up to 100 times slower), it may be caused by denormals (this is also true of many other IIR opcodes, including *butterlp*, *pareq*, *hilbert*, and many others). Underflows can be avoided by:



- Using the *denorm* opcode on *ain* before *spat3di*.
- mixing low level DC or noise to the input signal, e.g.

atmp rnd31 1/1e24, 0, 0

aW, aX, aY, aZ spat3di ain + atmp, ...

or

aW, aX, aY, aZ spa3di ain + 1/1e24, ...

- reducing *irlen* in the case of *spat3dt* (which does not have an input signal). A value of about 0.005 is suitable for most uses, although it also depends on EQ settings. If the equalizer is not used, “irlen” can be set to 0.

## Examples

See the examples for *spat3d*.

## See Also

*spat3d*, *spat3dt*

## Credits

Author: Istvan Varga  
2001

New in version 4.12

Updated April 2002 by Istvan Varga

# spat3dt

`spat3dt` -- Can be used to render an impulse response for a 3D space at i-time.

`spat3dt`

## Description

This opcode positions the input sound in a 3D space, with optional simulation of room acoustics, in various output formats. *spat3dt* can be used to render the impulse response at i-time, storing output in a function table, suitable for convolution.

## Syntax

**spat3dt** ioutft, iX, iY, iZ, idist, ift, imode, irlen [, iftnocl]

## Initialization

*ioutft* -- Output ftable number for spat3dt. W, X, Y, and Z outputs are written interleaved to this table. If the table is too short, output will be truncated.

*iX* -- Sound source X coordinate in meters (positive: right, negative: left)

*iY* -- Sound source Y coordinate in meters (positive: front, negative: back)

*iZ* -- Sound source Z coordinate in meters (positive: up, negative: down)

*idist* -- For modes 0 to 3, *idist* is the unit circle distance in meters. For mode 4, *idist* is the distance between microphones.

The following formulas describe amplitude and delay as a function of sound source distance from microphone(s):

$$\text{amplitude} = 1 / (0.1 + \text{distance})$$

$$\text{delay} = \text{distance} / 340 \text{ (in seconds)}$$

Distance can be calculated as:

$$\text{distance} = \sqrt{iX^2 + iY^2 + iZ^2}$$

In Mode 4, distance can be calculated as:

$$\begin{aligned} \text{distance from left mic} &= \sqrt{(iX + idist/2)^2 + iY^2 + iZ^2} \\ \text{distance from right mic} &= \sqrt{(iX - idist/2)^2 + iY^2 + iZ^2} \end{aligned}$$

With *spat3d* the distance between the sound source and any microphone should be at least  $(340 * 18) / \text{sr}$  meters. Shorter distances will work, but may produce artifacts in some cases. There is no

such limitation for *spat3di* and *spat3dt*.

Sudden changes or discontinuities in sound source location can result in pops or clicks. Very fast movement may also degrade quality.

*ift* -- Function table storing room parameters (for free field spatialization, set it to zero or negative). Table size is 54. The values in the table are:

Room Parameter	Purpose
0	Early reflection recursion depth (0 is the sound source, 1 is the first reflection etc.) for <i>spat3d</i> and <i>spat3di</i> . The number of echoes for four walls (front, back, right, left) is: $N = (2 * R + 2) * R$ . If all six walls are enabled: $N = (((4 * R + 6) * R + 8) * R) / 3$
1	Late reflection recursion depth (used by <i>spat3dt</i> only). <i>spat3dt</i> skips early reflections and renders echoes up to this level. If early reflection depth is negative, <i>spat3d</i> and <i>spat3di</i> will output zero, while <i>spat3dt</i> will start rendering from the sound source.
2	<i>imdel</i> for <i>spat3d</i> . Overrides opcode parameter if non-negative.
3	<i>irlen</i> for <i>spat3dt</i> . Overrides opcode parameter if non-negative.
4	<i>idist</i> value. Overrides opcode parameter if $\geq 0$ .
5	Random seed (0 - 65535) -1 seeds from current time.
6 - 53	wall parameters (w = 6: ceil, w = 14: floor, w = 22: front, w = 30: back, w = 38: right, w = 46: left)
w + 0	Enable reflections from this wall (0: no, 1: yes)
w + 1	Wall distance from listener (in meters)
w + 2	Randomization of wall distance (0 - 1) (in units of 1 / (wall distance))
w + 3	Reflection level (-1 - 1)
w + 4	Parametric equalizer frequency in Hz.
w + 5	Parametric equalizer level (1.0: no filtering)
w + 6	Parametric equalizer Q (0.7071: no resonance)
w + 7	Parametric equalizer mode (0: peak EQ, 1: low shelf, 2: high shelf)

*imode* -- Output mode

- 0: B format with W output only (mono)

$aout = aW$

- 1: B format with W and Y output (stereo)

$aleft = aW + 0.7071 * aY$   
 $aright = aW - 0.7071 * aY$

- 2: B format with W, X, and Y output (2D). This can be converted to UHJ:

```
aWre, aWim    hilbert aW
aXre, aXim    hilbert aX
aYre, aYim    hilbert aY
aWXr  = 0.0928*aXre + 0.4699*aWre
aWXiYr = 0.2550*aXim - 0.1710*aWim + 0.3277*aYre
aleft  = aWXr + aWXiYr
aright = aWXr - aWXiYr
```

- 3: B format with all outputs (3D)
- 4: Simulates a pair of microphones (stereo output)

```
aW    butterlp aW, ifreq    ; recommended values for ifreq
aY    butterlp aY, ifreq    ; are around 1000 Hz
aleft = aW + aX
aright = aY + aZ
```

Mode 0 is the cheapest to calculate, while mode 4 is the most expensive.

In Mode 4, The optional lowpass filters can change the frequency response depending on direction. For example, if the sound source is located left to the listener then the high frequencies are attenuated in the right channel and slightly increased in the left. This effect can be disabled by not using filters. You can also experiment with other filters (tone etc.) for better effect.

Note that mode 4 is most useful for listening with headphones, and is also more expensive to calculate than the B-format (0 to 3) modes. The *idist* parameter in this case sets the distance between left and right microphone; for headphones, values between 0.2 - 0.25 are recommended, although higher settings up to 0.4 may be used for wide stereo effects.

More information about B format can be found here: [http://www.york.ac.uk/inst/mustech/3d\\_audio/ambis2.htm](http://www.york.ac.uk/inst/mustech/3d_audio/ambis2.htm)

*irlen* -- Impulse response length of echoes (in seconds). Depending on filter parameters, values around 0.005-0.01 are suitable for most uses (higher values result in more accurate output, but slower rendering)

*iftnocl* (optional, default=0) -- Do not clear output ftable (mix to existing data) if set to 1, clear table before writing if set to 0 (default: 0).

## Examples

See the examples for *spat3d*.

## See Also

*spat3d*, *spat3di*

## Credits

Author: Istvan Varga  
2001

New in version 4.12

Updated April 2002 by Istvan Varga

# spdist

spdist -- Calculates distance values from xy coordinates.

spdist

## Description

*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.

## Syntax

k1 **spdist** ifn, ktime, kx, ky

## Initialization

*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*.

## 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 signals from *space*, it can be used in a 2 channel orchestra. If the xy's are kept so that  $Y \geq 1$ , it should work well to do panning and fixed localization in a stereo field.

*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.

*kx*, *ky* -- when *ifn* is 0, *space* and *spdist* will use these values as the XY coordinates to localize the signal.

## Examples

```
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 be useful for quad and stereo panning as well as fixed placement 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 dist.
```



```
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.

```
ptime          line 0, p3, 10  
kdist          spdist 1, ptime  
kfreq = (ifreq * 340) / (340 + kdist)  
asig          oscili iamp, kfreq, 1  
  
a1, a2, a3, a4  space asig, 1, ptime, .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!

## See Also

*space, spsend*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# specaddm

specaddm -- Perform a weighted add of two input spectra.

specaddm

## Description

Perform a weighted add of two input spectra.

## Syntax

```
wsig specaddm wsig1, wsig2 [, imul2]
```

## Initialization

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

## Performance

*wsig1* -- the first input spectra.

*wsig2* -- the second input spectra.

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:

$$\text{magout} = \text{mag1in} + \text{mag2in} * \text{imul2}$$

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).

## Examples

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

## See Also

*specdiff, specfilt, spechist, specscal*

# specdiff

specdiff -- Finds the positive difference values between consecutive spectral frames.

specdiff

## Description

Finds the positive difference values between consecutive spectral frames.

## Syntax

wsig **specdiff** wsignin

## Performance

*wsig* -- the output spectrum.

*wsignin* -- the input spectra.

Finds the positive difference values between consecutive spectral frames. At each new frame of *wsignin*, 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.

## Examples

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

## See Also

*specaddm*, *specfilt*, *spechist*, *specscal*

# specdisp

specdisp -- Displays the magnitude values of the spectrum.

specdisp

## Description

Displays the magnitude values of the spectrum.

## Syntax

**specdisp** *wsig*, *iprd* [, *iwtflg*]

## Initialization

*iprd* -- the period, in seconds, of each new display.

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

## Performance

*wsig* -- the input spectrum.

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

## Examples

```
      ksum      specsum  wsig,  1           ; sum the spec bins, and ksmoo
      if        ksum < 2000  kgoto  zero    ; if sufficient amplitude
      koct      specptrk wsig              ;   pitch-track the signal
      kgoto     contin
zero:
      koct      =        0                  ; else output zero
contin:
```

## See Also

*specsum*

# specfilt

specfilt -- Filters each channel of an input spectrum.

specfilt

## Description

Filters each channel of an input spectrum.

## Syntax

wsig **specfilt** wsignin, ifhtim

## Initialization

*ifhtim* -- half-time constant.

## Performance

*wsignin* -- the input spectrum.

Filters each channel of an input spectrum. At each new frame of *wsignin*, 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.

## Examples

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

## See Also

*specaddm, specdiff, spechist, specscal*

# spechist

spechist -- Accumulates the values of successive spectral frames.

spechist

## Description

Accumulates the values of successive spectral frames.

## Syntax

`wsig spechist wsign`

## Performance

*wsign* -- the input spectra.

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

## Examples

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

## See Also

*specadm, specdiff, specfilt, specscal*

# specptrk

specptrk -- Estimates the pitch of the most prominent complex tone in the spectrum.

spectrk

## Description

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

## Syntax

*koct*, *kamp* **specptrk** *wsig*, *kvar*, *ilo*, *ihi*, *istr*, *idbthresh*, *inptls*, *irolloff* [,

## 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, prorated 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).

*iwtfgr* (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*.)

## Examples

```
a1,a2    ins                                ; read a
krms      rms          a1, 20                ; find a
kvar      =            0.6 + krms/8000       ; & use t
wsig      spectrum    a1, .01, 7, 24, 15, 0, 3 ; get a 7
          specdisp    wsig, .2              ; display
koct,ka   spectrk     wsig, kvar, 7.0, 10, 9, 20, 4, .7, 1, 5, 1, .2 ; the pch
aosc      oscil       ka*ka*10, cpsoct(koct),2 ; & gener
koct      =           (koct<7.0?7.0:koct)    ; replace no
          display     koct-7.0, .25, 20      ; & displ
          display     ka, .25, 20           ; plus th
          outs        a1, aosc              ; output
```



# specscal

specscal -- Scales an input spectral datablock with spectral envelopes.

specscal

## Description

Scales an input spectral datablock with spectral envelopes.

## Syntax

wsig **specscal** wsignin, ifscale, ifthresh

## Initialization

*ifscale* -- scale function table. A function table containing values by which a value's magnitude is rescaled.

*ifthresh* -- threshold function table. If *ifthresh* is non-zero, each magnitude is reduced by its corresponding table-value (to not less than zero)

## Performance

*wsig* -- the output spectrum

*wsignin* -- the input spectra

Scales 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*.

## Examples

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

## See Also

*specaddm, specdiff, specfilt, spechist*

# specsum

specsum -- Sums the magnitudes across all channels of the spectrum.

specsum

## Description

Sums the magnitudes across all channels of the spectrum.

## Syntax

ksum **specsum** wsig [, interp]

## Initialization

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

## Performance

*ksum* -- the output signal.

*wsig* -- the input spectrum.

Sums 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.

## Examples

```
ksum      specsum  wsig, 1           ; sum the spec bins, and ksmoo
          if      ksum < 2000  kgoto  zero ; if sufficient amplitude
koct      specptrk wsig             ; pitch-track the signal
          kgoto    contin
zero:
  koct      =      0                ; else output zero
contin:
```

## See Also

*specdisp*

# spectrum

spectrum -- Generate a constant-Q, exponentially-spaced DFT.

spectrum

## Description

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

## Syntax

`wsig spectrum xsig, iprd, iocts, ifrqa [, iq] [, ihann] [, idbout] [, idsprd] [`

## 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).

*idsprd* (optional) -- if non-zero, display the composite downsampling buffer every *idsprd* 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 down-sampling, and preserves a buffer of down-sampled values in each octave (optionally displayed as a composite buffer every *idsprd* 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.

## Examples

```
asig in  
wsig spectrum asig,.02,6,12,33,0,1,1 ; get external audio  
; downsample in 6 octs & calc a 72 pt df
```

# splitrig

splitrig -- Split a trigger signal

splitrig

## Description

*splitrig* splits a trigger signal (i.e. a timed sequence of control-rate impulses) into several channels following a structure designed by the user.

## Syntax

**splitrig** ktrig, kndx, imaxtics, ifn, kout1 [,kout2,...,koutN]

## Initialization

*imaxtics* - number of tics belonging to largest pattern

*ifn* - number of table containing channel-data structuring

## Performance

*asig* - incoming (input) signal

*ktrig* - trigger signal

The *splitrig* opcode splits a trigger signal into several output channels according to one or more patterns provided by the user. Normally the regular timed trigger signal generated by metro opcode is used to be transformed into rhythmic pattern that can trig several independent melodies or percussion riffs. But you can also start from non-isocronous trigger signals. This allows to use some "interpretative" and less "mechanic" groove variations. Patterns are looped and each numtics\_of\_pattern\_N the cycle is repeated.

The scheme of patterns is defined by the user and is stored into ifn table according to the following format:

```
gil ftgen 1,0,1024, -2 \ ; table is generated with GEN02 in this case
\
numtics_of_pattern_1, \ ;pattern 1
    tic1_out1, tic1_out2, ... , tic1_outN,\
    tic2_out1, tic2_out2, ... , tic2_outN,\
    tic3_out1, tic3_out2, ... , tic3_outN,\
    .....
    ticN_out1, ticN_out2, ... , ticN_outN,\
\
numtics_of_pattern_2, \ ;pattern 2
    tic1_out1, tic1_out2, ... , tic1_outN,\
    tic2_out1, tic2_out2, ... , tic2_outN,\
    tic3_out1, tic3_out2, ... , tic3_outN,\
    .....
    ticN_out1, ticN_out2, ... , ticN_outN,\
    .....
\
numtics_of_pattern_N,\ ;pattern N
    tic1_out1, tic1_out2, ... , tic1_outN,\
    tic2_out1, tic2_out2, ... , tic2_outN,\
    tic3_out1, tic3_out2, ... , tic3_outN,\
```

```
.....  
ticN_out1, ticN_out2, ... , ticN_outN,\
```

This scheme can contain more than one pattern, each one with a different number of rows. Each pattern is preceded by a special row containing a single *numtics\_of\_pattern\_N* field; this field expresses the number of tics that makes up the corresponding pattern. Each pattern's row makes up a tic. Each pattern's column corresponds to a channel, and each field of a row is a number that makes up the value outputted by the corresponding *koutXX* channel (if number is a zero, corresponding output channel will not trigger anything in that particular arguments). Obviously, all rows must contain the same number of fields that must be equal to the number of *koutXX* channel. All patterns must contain the same number of rows, this number must be equal to the largest pattern and is defined by *imxtics* variable. Even if a pattern has less tics than the largest pattern, it must be made up of the same number of rows, in this case, some of these rows, at the end of the pattern itself, will not be used (and can be set to any value, because it doesn't matter).

The *kndx* variable chooses the number of the pattern to be played, zero indicating the first pattern. Each time the integer part of *kndx* changes, tic counter is reset to zero.

Patterns are looped and each *numtics\_of\_pattern\_N* the cycle is repeated.

examples 4 - calculate average value of asig in the time interval

This opcode can be useful in several situations, for example to implement a vu-meter

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# spsend

spsend -- Generates output signals based on a previously defined *space* opcode.

spsend

## Description

*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).

## Syntax

a1, a2, a3, a4 **spsend**

## 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 signals from *space*, it can be used in a 2 channel orchestra. If the xy's are kept so that  $Y \geq 1$ , it should work well to do panning and fixed localization in a stereo field.

## Examples

```
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 *ptime*. *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 be useful for quad and stereo panning as well as fixed placement 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
il 0 1 -1 1
;place the sound in the right speaker and far
il 1 1 45 45
;place the sound equally between left and right and in the middle ground dist.
il 2 1 0 12
e
```

The next example shows a simple intuitive use of the distance values returned by *spdist* to simulate Doppler shift.

```
ptime                line  0, p3, 10
kdist                spdist 1, ptime
kfreq = (ifreq * 340) / (340 + kdist)
asig                oscili iamp, kfreq, 1

a1, a2, a3, a4      space  asig, 1, ptime, .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!

## See Also

*space, spdist*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1998

New in Csound version 3.48

# sprintf

`sprintf` -- printf-style formatted output to a string variable

`sprintf`

## Description

`sprintf` and `sprintfk` write printf-style formatted output to a string variable, similarly to the C function `sprintf()`. `sprintf` runs at i-time only, while `sprintfk` runs both at initialization and performance time.

## Syntax

```
Sdst sprintf Sfmt, xarg1[, xarg2[, ... ]]
```

```
Sdst sprintfk Sfmt, xarg1[, xarg2[, ... ]]
```

## Initialization

*Sfmt* -- format string, has the same format as in `printf()` and other similar C functions, except length modifiers (l, ll, h, etc.) are not supported. The following conversion specifiers are allowed:

- d, i, o, u, x, X, e, E, f, F, g, G, c, s

*xarg1*, *xarg2*, ... -- input arguments (max. 30) for format, should be i-rate for all conversion specifiers except %s, which requires a string argument. `sprintfk` also allows k-rate number arguments, but these should still be valid at init time as well (unless `sprintfk` is skipped with `igoto`). Integer formats like %d round the input values to the nearest integer.

## Performance

*Sdst* -- output string variable

## Example

```
Sname    sprintf "soundin-%04d.wav", ifileno
Smsg      sprintf "The file name is: '%s'", Sname
          puts Smsg, 1
asig soundin Sname
```

## Credits

Author: Istvan Varga  
2005

# sqrt

sqrt -- Returns a square root value.

sqrt

## Description

Returns the square root of  $x$  ( $x$  non-negative).

The argument value is restricted for *log*, *log10*, and *sqrt*.

## Syntax

**sqrt**(*x*) (no rate restriction)

where the argument within the parentheses may be an expression. Value converters perform arithmetic translation from units of one kind to units of another. The result can then be a term in a further expression.

## Examples

Here is an example of the sqrt opcode. It uses the files *sqrt.orc* [examples/sqrt.orc] and *sqrt.sco* [examples/sqrt.sco].

### Example 360. Example of the sqrt opcode.

```
/* sqrt.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  i1 = sqrt(64)
  print i1
endin
/* sqrt.orc */

/* sqrt.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* sqrt.sco */
```

Its output should include lines like this:

```
instr 1:  i1 = 8.000
```

## See Also

*abs, exp, frac, int, log, log10, i*

## Credits

Example written by Kevin Conder.

## sr

sr -- Sets the audio sampling rate.

sr

## 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.

## Syntax

```
sr = iarg
```

## Initialization

sr = (optional) -- set sampling rate to *iarg* samples per second per channel. The default value is 44100.

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, *sr* may be omitted. Csound will attempt to calculate the omitted value from the specified values, but it should evaluate to an integer.

## Examples

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

## See Also

*kr, ksmps, nchnls*

# statevar

statevar -- State-variable filter.

statevar

## Description

Statevar is a new digital implementation of the analogue state-variable filter. This filter has four simultaneous outputs: high-pass, low-pass, band-pass and band-reject. This filter uses oversampling for sharper resonance (default: 3 times oversampling). It includes a resonance limiter that prevents the filter from getting unstable.

## Syntax

*ahp, alp, abp, abr* **statevar** *ain, kcf, kq [, iosamps, istor]*

## Initialization

*iosamps* -- number of times of oversampling used in the filtering process. This will determine the maximum sharpness of the filter resonance (Q). More oversampling allows higher Qs, less oversampling will limit the resonance. The default is 3 times (*iosamps*=0).

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

## Performance

*ahp* -- high-pass output signal.

*alp* -- low-pass output signal.

*abp* -- band-pass signal.

*abr* -- band-reject signal.

*asig* -- input signal.

*kcf* -- filter cutoff frequency

*kq* -- filter Q. This value is limited internally depending on the frequency and the number of times of oversampling used in the process (3-times oversampling by default).

## Examples

### Example 361. Example

```
kenv          linseg 0,0.1,1, p3-0.2,1, 0.1, 0
asig          buzz 16000*kenv, 100, 100, 1;
kf            expseg 100, p3/2, 5000, p3/2, 1000
ahp,alp,abp,abr statevar asig, kf, 200

              outs alp,ahp
```

## Credits

Author: Victor Lazzarini;  
January 2005

New plugin in version 5

January 2005.

# stix

*stix* -- Semi-physical model of a stick sound.

*stix*

## Description

*stix* is a semi-physical model of a stick sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **stix** *iamp*, *idettack* [, *inum*] [, *idamp*] [, *imaxshake*]

## 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

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 30.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.998 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.998 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 1.0.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional) -- amount of energy to add back into the system. The value should be in range 0 to 1.

## Examples

Here is an example of the *stix* opcode. It uses the files *stix.orc* [examples/stix.orc] and *stix.sco* [examples/stix.sco].

### Example 362. Example of the *stix* opcode.

```
/* stix.orc */
;orchestra -----

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

instr 01
  a1    line 20, p3, 20      ;an example of stix
  a2    stix p4, 0.01       ;stix needs a little amp help at these settings
  a3    product a1, a2      ;increase amplitude
      out a3
```



```
        endin
/* stix.orc */

/* stix.sco */
;score -----

    i1 0 1 26000
    e
/* stix.sco */
```

## See Also

*cabasa, crunch, sandpaper, sekere*

## Credits

Author: Perry Cook, part of the PhOLIES (Physically-Oriented Library of Imitated Environmental Sounds)

Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# strchar

strchar -- Return the ASCII code of a character in a string

strchar

## Description

Return the ASCII code of the character in Sstr at ipos (defaults to zero which means the first character), or zero if ipos is out of range. strchar runs at init time only.

## Syntax

```
ichr strchar Sstr[, ipos]
```

## See also

*strchark*

## Credits

Author: Istvan Varga  
2006

# strchark

strchark -- Return the ASCII code of a character in a string

strchark

## Description

Return the ASCII code of the character in Sstr at kpos (defaults to zero which means the first character), or zero if kpos is out of range. strchark runs both at init and performance time.

## Syntax

kchr **strchark** Sstr[, kpos]

## See also

*strchar*

## Credits

Author: Istvan Varga  
2006

# strcpy

strcpy -- Assign value to a string variable

strcpy

## Description

Assign to a string variable by copying the source which may be a constant or another string variable. strcpy and = copy the string at i-time only.

## Syntax

```
Sdst strcpy Ssrc
```

```
Sdst = Ssrc
```

## Example

```
Sfoo      strcpy "Hello, world !"  
          puts Sfoo, 1
```

## See also

*strcpyk*

## Credits

Author: Istvan Varga  
2005

# strcpyk

strcpyk -- Assign value to a string variable (k-rate)

strcpyk

## Description

Assign to a string variable by copying the source which may be a constant or another string variable. *strcpyk* does the assignment both at initialization and performance time.

## Syntax

Sdst **strcpyk** Ssrc

## See also

*strcpy*

## Credits

Author: Istvan Varga  
2005

# strcat

strcat -- Concatenate strings

strcat

## Description

Concatenate two strings and store the result in a variable. *strcat* runs at i-time only. It is allowed for any of the input arguments to be the same as the output variable.

## Syntax

Sdst **strcat** Ssrc1, Ssrc2

## Example

```
Sname    = "beats"  
Sname    strcat Sname, ".wav"  
asig     soundin Sname
```

## See also

*strcatk*

## Credits

Author: Istvan Varga  
2005

# strcatk

strcatk -- Concatenate strings (k-rate)

strcatk

## Description

Concatenate two strings and store the result in a variable. *strcatk* does the concatenation both at initialization and performance time. It is allowed for any of the input arguments to be the same as the output variable.

## Syntax

Sdst **strcatk** Ssrc1, Ssrc2

## See also

*strcat*

## Credits

Author: Istvan Varga  
2005

# strcmp

strcmp -- Compare strings

strcmp

## Description

Compare strings and set the result to -1, 0, or 1 if the first string is less than, equal to, or greater than the second, respectively. strcmp compares at i-time only.

## Syntax

```
ires strcmp S1, S2
```

## See also

*strcmpk*

## Credits

Author: Istvan Varga  
2005



# strcmpk

strcmpk -- Compare strings

strcmp

## Description

Compare strings and set the result to -1, 0, or 1 if the first string is less than, equal to, or greater than the second, respectively. *strcmpk* does the comparison both at initialization and performance time.

## Syntax

kres **strcmpk** S1, S2

## See also

*strcmp*

## Credits

Author: Istvan Varga  
2005

# streson

streson -- A string resonator with variable fundamental frequency.

streson

## Description

An audio signal is modified by a string resonator with variable fundamental frequency.

## Syntax

ares **streson** asig, kfr, ifdbgain

## 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

*asig* -- the input audio signal.

*kfr* -- the fundamental frequency of the string.

*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.

## Examples

Here is an example of the streson opcode. It uses the files *streson.orc* [examples/streson.orc] and *streson.sco* [examples/streson.sco].

### Example 363. Example of the streson opcode.

```
/* streson.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a normal sine wave.
  asig oscils 8000, 440, 1

  ; Vary the fundamental frequency of the string
  ; resonator linearly from 220 to 880 Hertz.
  kfr line 220, p3, 880
```

```
ifdbgain = 0.95

; Run our sine wave through the string resonator.
astres streson asig, kfr, ifdbgain

; The resonance can get quite loud.
; So we'll clip the signal at 30,000.
a1 clip astres, 1, 30000
out a1
endin
/* streson.orc */


/* streson.sco */
; Play Instrument #1 for five seconds.
i 1 0 5
e
/* streson.sco */
```

## Credits

Author: Victor Lazzarini  
Music Department  
National University of Ireland, Maynooth  
Maynooth, Co. Kildare  
1998

Example written by Kevin Conder.

New in Csound version 3.494

# strget

strget -- Set string variable to value from strset table or string p-field

strget

## Description

strget sets a string variable at initialization time to the value stored in strset table at the specified index, or a string p-field from the score. If there is no string defined for the index, the variable is set to an empty string.

## Syntax

*Sdst* **strget** *indx*

## Initialization

*indx* -- strset index, or score p-field

*Sdst* -- destination string variable

## Credits

Author: Istvan Varga  
2005

# strindex

strindex -- Return the position of the first occurrence of a string in another string

strindex

## Description

Return the position of the first occurrence of S2 in S1, or -1 if not found. If S2 is empty, 0 is returned. strindex runs at init time only.

## Syntax

ipos **strindex** S1, S2

## See also

*strindexk*

## Credits

Author: Istvan Varga  
2006

# strindexk

strindexk -- Return the position of the first occurrence of a string in another string

strindexk

## Description

Return the position of the first occurrence of S2 in S1, or -1 if not found. If S2 is empty, 0 is returned. strindexk runs both at init and performance time.

## Syntax

kpos **strindexk** S1, S2

## See also

*strindex*

## Credits

Author: Istvan Varga  
2006

# strlen

strlen -- Return the length of a string

strlen

## Description

Return the length of a string, or zero if it is empty. strlen runs at init time only.

## Syntax

```
ilen strlen Sstr
```

## See also

*strlenk*

## Credits

Author: Istvan Varga  
2006

# strlenk

strlenk -- Return the length of a string

strlenk

## Description

Return the length of a string, or zero if it is empty. strlenk runs both at init and performance time.

## Syntax

```
klen strlenk Sstr
```

## See also

*strlen*

## Credits

Author: Istvan Varga  
2006



# strlower

strlower -- Convert a string to lower case

strlower

## Description

Convert Ssrc to lower case, and write the result to Sdst. strlower runs at init time only.

## Syntax

Sdst **strlower** Ssrc

## See also

*strlowerk*

## Credits

Author: Istvan Varga  
2006

# strlowerk

strlowerk -- Convert a string to lower case

strlowerk

## Description

Convert Ssrc to lower case, and write the result to Sdst. strlowerk runs both at init and performance time.

## Syntax

Sdst **strlowerk** Ssrc

## See also

*strlower*

## Credits

Author: Istvan Varga  
2006

# strrindex

strrindex -- Return the position of the last occurrence of a string in another string

strrindex

## Description

Return the position of the last occurrence of S2 in S1, or -1 if not found. If S2 is empty, the length of S1 is returned. strrindex runs at init time only.

## Syntax

ipos **strrindex** S1, S2

## See also

*strrindexk*

## Credits

Author: Istvan Varga  
2006

# strrindexk

strrindexk -- Return the position of the last occurrence of a string in another string

strrindexk

## Description

Return the position of the last occurrence of S2 in S1, or -1 if not found. If S2 is empty, the length of S1 is returned. strrindexk runs both at init and performance time.

## Syntax

kpos **strrindexk** S1, S2

## See also

*strrindex*

## Credits

Author: Istvan Varga  
2006

# strset

**strset** -- Allows a string to be linked with a numeric value.

**strset**

## Description

Allows a string to be linked with a numeric value.

## Syntax

**strset** *iarg*, *istring*

## Initialization

*iarg* -- the numeric value.

*istring* -- the alphanumeric string (in double-quotes).

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

## 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"
```

## See Also

*pset*

# strsub

strsub -- Extract a substring

strsub

## Description

Return a substring of the source string. strsub runs at init time only.

## Syntax

Sdst **strsub** Ssrc[, istart[, iend]]

## Initialization

*istart* (optional, defaults to 0) -- start position in Ssrc, counting from 0. A negative value means the end of the string.

*iend* (optional, defaults to -1) -- end position in Ssrc, counting from 0. A negative value means the end of the string. If iend is less than istart, the output is reversed.

## See also

*strsubk*

## Credits

Author: Istvan Varga  
2006

# strsubk

strsubk -- Extract a substring

strsubk

## Description

Return a substring of the source string. strsubk runs both at init and performance time.

## Syntax

Sdst **strsubk** Ssrc, kstart, kend

## Performance

*kstart* -- start position in Ssrc, counting from 0. A negative value means the end of the string.

*kend* -- end position in Ssrc, counting from 0. A negative value means the end of the string. If *kend* is less than *kstart*, the output is reversed.

## See also

*strsub*

## Credits

Author: Istvan Varga  
2006

# strtod

strtod -- Converts a string to a float (i-rate).

strtod

## Description

Convert a string to a floating point value. It is also possible to pass an strset index or a string p-field from the score instead of a string argument. If the string cannot be parsed as a floating point or integer number, an init or perf error occurs and the instrument is deactivated.

## Syntax

```
ir strtod Sstr
```

```
ir strtod indx
```

## Initialization

*Sstr* -- String to convert.

*indx* -- index of string set by strset

## Performance

*ir* -- Value of string as float.

## Credits

Author: Istvan Varga  
2005



# strtodk

strtodk -- Converts a string to a float (k-rate).

strtodk

## Description

Convert a string to a floating point value at i- or k-rate. It is also possible to pass an strset index or a string p-field from the score instead of a string argument. If the string cannot be parsed as a floating point or integer number, an init or perf error occurs and the instrument is deactivated.



### Note

If a k-rate index variable is used, it should be valid at i-time as well.

## Syntax

kr **strtodk** Sstr

kr **strtodk** kndx

## Performance

*kr* -- Value of string as float.

*Sstr* -- String to convert.

*indx* -- index of string set by strset

## Credits

Author: Istvan Varga  
2005

# strtol

*strtol* -- Converts a string to a signed integer (i-rate).

*strtol*

## Description

Convert a string to a signed integer value. It is also possible to pass an *strset* index or a string p-field from the score instead of a string argument. If the string cannot be parsed as a floating point or integer number, an *init* or *perf* error occurs and the instrument is deactivated.

## Syntax

```
ir strtol Sstr
```

```
ir strtol indx
```

## Initialization

*Sstr* -- String to convert.

*indx* -- index of string set by *strset*

*strtol* can parse numbers in decimal, octal (prefixed by 0), and hexadecimal (with a prefix of 0x) format.

## Performance

*ir* -- Value of string as signed integer.

## Credits

Author: Istvan Varga  
2005

# strtolk

strtolk -- Converts a string to a signed integer (k-rate).

strtolk

## Description

Convert a string to a floating point value at i- or k-rate. It is also possible to pass an strset index or a string p-field from the score instead of a string argument. If the string cannot be parsed as a floating point or integer number, an init or perf error occurs and the instrument is deactivated.



### Note

If a k-rate index variable is used, it should be valid at i-time as well.

## Syntax

kr **strtolk** Sstr

kr **strtolk** kndx

strtolk can parse numbers in decimal, octal (prefixed by 0), and hexadecimal (with a prefix of 0x) format.

## Performance

*kr* -- Value of string as signed integer.

*Sstr* -- String to convert.

*indx* -- index of string set by strset

## Credits

Author: Istvan Varga  
2005

# strupper

strupper -- Convert a string to upper case

strupper

## Description

Convert Ssrc to upper case, and write the result to Sdst. strupper runs at init time only.

## Syntax

Sdst **strupper** Ssrc

## See also

*strupperk*

## Credits

Author: Istvan Varga  
2006

# strupperk

strupperk -- Convert a string to upper case

strupperk

## Description

Convert Ssrc to upper case, and write the result to Sdst. strupperk runs both at init and performance time.

## Syntax

Sdst **strupperk** Ssrc

## See also

*strupper*

## Credits

Author: Istvan Varga  
2006

# subinstr

subinstr -- Creates and runs a numbered instrument instance.

subinstr

## Description

Creates an instance of another instrument and is used as if it were an opcode.

## Syntax

```
a1, [...] [, a8] subinstr instrnum [, p4] [, p5] [...]
```

```
a1, [...] [, a8] subinstr "insname" [, p4] [, p5] [...]
```

## Initialization

*instrnum* -- Number of the instrument to be called.

*"insname"* -- A string (in double-quotes) representing a named instrument.

For more information about specifying input and output interfaces, see *Calling an Instrument within an Instrument*.

## Performance

*a1, ..., a8* -- The audio output from the called instrument. This is generated using the *signal output* opcodes.

*p4, p5, ...* -- Additional input values the are mapped to the called instrument p-fields, starting with p4.

The called instrument's p2 and p3 values will be identical to the host instrument's values. While the host instrument can *control its own duration*, any such attempts inside the called instrument will most likely have no effect.

## See Also

*Calling an Instrument within an Instrument, event, schedule, subinstrinit*

## Examples

Here is an example of the subinstr opcode. It uses the files *subinstr.orc* [examples/subinstr.orc] and *subinstr.sco* [examples/subinstr.sco].

### Example 364. Example of the subinstr opcode.

```
/* subinstr.orc */  
; Initialize the global variables.  
sr = 44100  
kr = 4410  
ksmps = 10  
nchnls = 1
```

```
; Instrument #1 - Creates a basic tone.
instr 1
  ; Print the value of p4, should be equal to
  ; Instrument #2's iamp field.
  print p4

  ; Print the value of p5, should be equal to
  ; Instrument #2's ipitch field.
  print p5

  ; Create a tone.
  asig oscils p4, p5, 0

  out asig
endin

; Instrument #2 - Demonstrates the subinstr opcode.
instr 2
  iamp = 20000
  ipitch = 440

  ; Use Instrument #1 to create a basic sine-wave tone.
  ; Its p4 parameter will be set using the iamp variable.
  ; Its p5 parameter will be set using the ipitch variable.
  abasic subinstr 1, iamp, ipitch

  ; Output the basic tone that we have created.
  out abasic
endin
/* subinstr.orc */

/* subinstr.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #2 for one second.
i 2 0 1
e
/* subinstr.sco */
```

Here is an example of the subinstr opcode using a named instrument. It uses the files *subinstr\_named.orc* [examples/subinstr\_named.orc] and *subinstr\_named.sco* [examples/subinstr\_named.sco].

### **Example 365. Example of the subinstr opcode using a named instrument.**

```
/* subinstr_named.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument "basic_tone" - Creates a basic tone.
instr basic_tone
```

```
; Print the value of p4, should be equal to
; Instrument #2's iamp field.
print p4

; Print the value of p5, should be equal to
; Instrument #2's ipitch field.
print p5

; Create a tone.
asig oscils p4, p5, 0

out asig
endin

; Instrument #1 - Demonstrates the subinstr opcode.
instr 1
  iamp = 20000
  ipitch = 440

  ; Use the "basic_tone" named instrument to create a
  ; basic sine-wave tone.
  ; Its p4 parameter will be set using the iamp variable.
  ; Its p5 parameter will be set using the ipitch variable.
  abasic subinstr "basic_tone", iamp, ipitch

  ; Output the basic tone that we have created.
  out abasic
endin
/* subinstr_named.orc */

/* subinstr_named.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* subinstr_named.sco */
```

## Credits

New in version 4.21



# subinstrinit

subinstrinit -- Creates and runs a numbered instrument instance at init-time.

subinstrinit

## Description

Same as *subinstr*, but init-time only and has no output arguments.

## Syntax

```
subinstrinit instrnum [, p4] [, p5] [...]
```

```
subinstrinit "insname" [, p4] [, p5] [...]
```

## Initialization

*instrnum* -- Number of the instrument to be called.

*"insname"* -- A string (in double-quotes) representing a named instrument.

For more information about specifying input and output interfaces, see *Calling an Instrument within an Instrument*.

## Performance

*p4*, *p5*, ... -- Additional input values the are mapped to the called instrument p-fields, starting with *p4*.

The called instrument's *p2* and *p3* values will be identical to the host instrument's values. While the host instrument can *control its own duration*, any such attempts inside the called instrument will most likely have no effect.

## See Also

*Calling an Instrument within an Instrument*, *event*, *schedule*, *subinstr*

## Credits

New in version 4.23

## sum

sum -- Sums any number of a-rate signals.

sum

## Description

Sums any number of a-rate signals.

## Syntax

```
ares sum asig1 [, asig2] [, asig3] [...]
```

## Performance

*asig1, asig2, ...* -- a-rate signals to be summed (mixed or added).

## Credits

Author: Gabriel Maldonado  
Italy  
April 1999

New in Csound version 3.54

## svfilter

**svfilter** -- A resonant second order filter, with simultaneous lowpass, highpass and bandpass outputs.

svfilter

## Description

Implementation of a resonant second order filter, with simultaneous lowpass, highpass and bandpass outputs.

## Syntax

alow, ahigh, aband **svfilter** asig, kcf, kq [, iscl]

## 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 Microprocessors* (Hayden Books, 1985).

## Examples

Here is an example of the *svfilter* opcode. It uses the files *svfilter.orc* [examples/svfilter.orc] and *svfilter.sco* [examples/svfilter.sco].

### Example 366. Example of the svfilter opcode.

```
/* svfilter.orc */
```

```
; 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 f

    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
/* svfilter.orc */

/* svfilter.sco */
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 bandpass
e
/* svfilter.sco */
```

## Credits

Author: Sean Costello  
Seattle, Washington  
1999

New in Csound version 3.55

# syncgrain

syncgrain -- Synchronous granular synthesis.

syncgrain

## Description

Syncgrain implements synchronous granular synthesis. The source sound for the grains is obtained by reading a function table containing the samples of the source waveform. For sampled-sound sources, GEN01 is used. Syncgrain will accept deferred allocation tables (with aif files).

The grain generator has full control of frequency (grains/sec), overall amplitude, grain pitch (a sampling increment) and grain size (in secs), both as fixed or time-varying (signal) parameters. An extra parameter is the grain pointer speed (or rate), which controls which position the generator will start reading samples in the table for each successive grain. It is measured in fractions of grain size, so a value of 1 (the default) will make each successive grain read from where the previous grain should finish. A value of 0.5 will make the next grain start at the midway position from the previous grain start and finish, etc.. A value of 0 will make the generator read always from a fixed position of the table (wherever the pointer was last at). A negative value will decrement pointer positions. This control gives extra flexibility for creating timescale modifications in the resynthesis.

Syncgrain will generate any number of parallel grain streams (which will depend on grain density/frequency), up to the *olaps* value (default 100). The number of streams (overlapped grains) is determined by  $\text{grainsize} * \text{grain\_freq}$ . More grain overlaps will demand more calculations and the synthesis might not run in realtime (depending on processor power).

Syncgrain can simulate FOF-like formant synthesis, provided that a suitable shape is used as grain envelope and a sinewave as the grain wave. For this use, grain sizes of around 0.04 secs can be used. The formant centre frequency is determined by the grain pitch. Since this is a sampling increment, in order to use a frequency in Hz, that value has to be scaled by  $\text{tablesize}/\text{sr}$ . Grain frequency will determine the fundamental.

Syncgrain uses floating-point indexing, so its precision is not affected by large-size tables. This opcode is based on the SndObj library SyncGrain class.

## Syntax

asig **syncgrain** kamp, kfreq, kpitch, kgrsize, kprate, ifun1, ifun2, iolaps

## Initialization

*ifun1* -- source signal function table. Deferred-allocation tables (see GEN01) are accepted, but the opcode expects a mono source.

*ifun2* -- grain envelope function table.

*iolaps* -- maximum number of overlaps,  $\max(\text{kfreq}) * \max(\text{kgrsize})$ . Estimating a large value should not affect performance, but exceeding this value will probably have disastrous consequences.

## Performance

*kamp* -- amplitude scaling

*kfreq* -- frequency of grain generation, or density, in grains/sec.

*kpitch* -- grain pitch scaling (1=normal pitch, < 1 lower, > 1 higher; negative, backwards)

*kgrsize* -- grain size in secs.

*kprate* -- readout pointer rate, in grains. The value of 1 will advance the reading pointer 1 grain ahead in the source table. Larger values will time-compress and smaller values will time-expand the source signal. Negative values will cause the pointer to run backwards and zero will freeze it.

## Examples

### Example 367. Example

```
iolaps = 2
igrsize = 0.04
ifreq = iolaps/igrsize
ips = 1/iolaps

istr = .5 /* timescale */
ipitch = 1 /* pitchscale */

a1 syncgrain 16000, ifreq, ipitch, igrsize, ips*istr, 1, 2, iolaps
    out    a1
```

## Credits

Author: Victor Lazzarini;  
January 2005

New plugin in version 5

January 2005.

# timedseq

timedseq -- Time Variant Sequencer

timedseq

## Description

An event-sequencer in which time can be controlled by a time-pointer. Sequence data are stored into a table.

## Syntax

```
ktrig timedseq ktimpnt, ifn, kp1  
      [,kp2, kp3, ...,kpN]
```

## Initialization

*ifn* -- number of table containing sequence data.

## Performance

*ktri* -- output trigger signal

*ktimpnt* -- time pointer into sequence file, in seconds.

*kp1,...,kpN* -- output p-fields of notes. kp2 meaning is relative action time and kp3 is the duration of notes in seconds.

*timedseq* is a sequencer that allows to schedule notes starting from a user sequence, and depending from an external timing given by a time-pointer value (*ktimpnt* argument). User should fill table *ifn* with a list of notes, that can be provided in an external text file by using GEN23, or by typing it directly in the orchestra (or score) file with GEN02. The format of the text file containing the sequence is made up simply by rows containing several numbers separated by space (similarly to normal Csound score). The first value of each row must be a positive or null value, except for a special case that will be explained below. This first value is normally used to define the instrument number corresponding to that particular note (like normal score). The second value of each row must contain the action time of corresponding note and the third value its duration. This is an example:

```
0 0      0.25 1  93  
0 0.25 0.25 2  63  
0 0.5   0.25 3  91  
0 0.75 0.25 4  70  
0 1     0.25 5  83  
0 1.25 0.25 6  75  
0 1.5   0.25 7  78  
0 1.75 0.25 8  78  
0 2     0.25 9  83  
0 2.25 0.25 10 70  
0 2.5   0.25 11 54  
0 2.75 0.25 12 80  
-1 3    -1   -1 -1 ;; last row of the sequence
```

In this example, the first value of each row is always zero (it is a dummy value, but this p-field can be used, for example, to express a MIDI channel or an instrument number), except the last row, that begins with -1. This value (-1) is a special value, that indicates the end of sequence. It has itself an action time, because sequences can be looped. So the previous sequence has a default duration of 3

seconds, being value 3 the last action time of the sequence.

It is important that ALL lines contains the same number of values (in the example all rows contains exactly 5 values). The number of values contained by each row, MUST be the number of kpXX output arguments (notice that, even if kp1, kp2 etc. are placed at the right of the opcode, they are output arguments, not input arguments).

ktimpnt argument provide the real temporization of the sequence. Actually the passage of time through sequence is specified by ktimpnt itself, 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 sequence file, in the same way of pvoc or lread. When ktimpnt crosses the action time of a note, a trigger signal is sent to ktrig output argument, and kp1, kp2,...kpN arguments are updated with the values of that note. This information can then be used with schedk or schedkwhen to actually activate note events. Notice that kp1,...kpn data can be further processed (for example delayed with delayk, transposed, etc.) before feeding schedk or schedkwhen.

ktimepoint can be controlled by linear signal, for example:

```
ktimpnt line      0,p3,3 ; original sequence duration was 3 secs
ktrig   timedseq ktimpnt,1,kp1,kp2,kp3,kp4,kp5
        schedk   ktrig, 105, 2, 0, kp3,kp4,kp5
```

in this case the complete sequence (with original duration of 3 seconds) will be played in p3 seconds.

You can loop a sequence by controlling it with a phasor:

```
kphs     phasor    1/3
ktimpnt =          kphs * 3
ktrig     timedseq ktimpnt,1,kp1,kp2,kp3,kp4,kp5
        schedk   ktrig, 105, 2, 0, kp3,kp4,kp5
```

Obviously you can play only a fragment of the sequence, read it backward, and non-linearly access sequence data in the same way of pvoc and lread opcodes.

With timedseq opcode you can do almost all things of a normal score, except you have the following limitations: 1. You can't have two notes exactly starting with the same action time; actually at least a k-cycle should separate timing of two notes (otherwise the schedk mechanism eats one of them). 2. all notes of the sequence must have the same number of p-fields (even if they activate different instruments). You can remedy this limitation by filling with dummy values notes that belongs to instruments with less p-fields than other ones.

## Credits

Author: Gabriel Maldonado



## tb

tb -- Table Read Access inside expressions.

tb0, tb1, tb2, tb3, tb4, tb5, tb6, tb7, tb8, tb9, tb10, tb11, tb12, tb13, tb14, tb15, tb0\_init, tb1\_init, tb2\_init, tb3\_init, tb4\_init, tb5\_init, tb6\_init, tb7\_init, tb8\_init, tb9\_init, tb10\_init, tb11\_init, tb12\_init, tb13\_init, tb14\_init, tb15\_init

## Description

Allow to read tables in function fashion, to be used inside expressions. At present time Csound only supports functions with a single input argument. However, to access table elements, user must provide two numbers, i.e. the number of table and the index of element. So, in order to allow to access a table element with a function, a previous preparation step should be done.

## Syntax

```
tb0_init ifn
```

```
tb1_init ifn
```

```
tb2_init ifn
```

```
tb3_init ifn
```

```
tb4_init ifn
```

```
tb5_init ifn
```

```
tb6_init ifn
```

```
tb7_init ifn
```

```
tb8_init ifn
```

```
tb9_init ifn
```

```
tb10_init ifn
```

```
tb11_init ifn
```

```
tb12_init ifn
```

```
tb13_init ifn
```

```
tb14_init ifn
```

```
tb15_init ifn
```

```
iout = tb0(iIndex)

kout = tb0(kIndex)

iout = tb1(iIndex)

kout = tb1(kIndex)

iout = tb2(iIndex)

kout = tb2(kIndex)

iout = tb3(iIndex)

kout = tb3(kIndex)

iout = tb4(iIndex)

kout = tb4(kIndex)

iout = tb5(iIndex)

kout = tb5(kIndex)

iout = tb6(iIndex)

kout = tb6(kIndex)

iout = tb7(iIndex)

kout = tb7(kIndex)

iout = tb8(iIndex)

kout = tb8(kIndex)

iout = tb9(iIndex)

kout = tb9(kIndex)

iout = tb10(iIndex)

kout = tb10(kIndex)

iout = tb11(iIndex)
```

```
kout = tb11(kIndex)
```

```
iout = tb12(iIndex)
```

```
kout = tb12(kIndex)
```

```
iout = tb13(iIndex)
```

```
kout = tb13(kIndex)
```

```
iout = tb14(iIndex)
```

```
kout = tb14(kIndex)
```

```
iout = tb15(iIndex)
```

```
kout = tb15(kIndex)
```

There are 16 different opcodes whose name is associated with a number from 0 to 15. User can associate a specific table with each opcode (so the maximum number of tables that can be accessed in function fashion is 16). Prior to access a table, user must associate the table with one of the 16 opcodes by means of an opcode chosen among `tb0_init...tb15_init`. For example,

```
tb0_init 1
```

associates table 1 with `tb0( )` function, so that, each element of table 1 can be accessed (in function fashion) with:

```
kvar = tb0(k_some_index_of_table1) * k_some_other_var
```

```
ivar = tb0(i_some_index_of_table1) + i_some_other_var etc...
```

By using these opcodes, user can drastically reduce the number of lines of an orchestra, improving its readability.

## Credits

Written by Gabriel Maldonado.

# tab

tab -- Fast table opcodes.

tab

## Description

Fast table opcodes. Faster than *table* and *tablew* because don't allow wrap-around and limit and don't check index validity. Have been implemented in order to provide fast access to arrays. Support non-power of two tables (can be generated by any GEN function by giving a negative length value).

## Syntax

```
ir tab_i indx, ifn[, ixmode]
```

```
kr tab kndx, ifn[, ixmode]
```

```
ar tab xndx, ifn[, ixmode]
```

```
tabw_i isig, indx, ifn [,ixmode]
```

```
tabw ksig, kndx, ifn [,ixmode]
```

```
tabw asig, andx, ifn [,ixmode]
```

## Initialization

*ifn* -- table number

*ixmode* -- defaults to zero. If zero xndx and ixoff ranges match the length of the table; if non zero xndx and ixoff have a 0 to 1 range.

*isig* -- input value to write.

*indx* -- table index

## Performance

*asig*, *ksig* -- input signal to write.

*andx*, *kndx* -- table index.

*tab* and *tabw* opcodes are similar to *table* and *tablew*, but are faster and support tables having non-power-of-two length.

Special care of index value must be taken into account. Index values out of the table allocated space will crash Csound.

## Credits

Written by Gabriel Maldonado.

# tabrec

tabrec -- Recording and playing-back control signals.

tabrec

## Description

Record and playback control-rate signals on trigger-temporization basis.

## Syntax

**tabrec**    *ktrig\_start*, *ktrig\_stop*, *knumtics*, *kfn*, *kin1* [,*kin2*,...,*kinN*]

**tabplay**   *ktrig*, *knumtics*, *kfn*, *kout1* [,*kout2*,..., *koutN*]

## Initialization

## Performance

*ktrig\_start* -- start recording when non-zero.

*ktrig\_stop* -- stop recording when *knumtics* trigger impulses are received by this input argument.

*knumtics* -- stop recording or reset playing pointer to zero when the number of tics defined by this argument is reached.

*kfn* -- table where k-rate signals are recorded.

*kin1*,...,*kinN* -- input signals to record.

*ktrig* -- starts playing when non-zero.

*kout1*,...,*koutN* -- playback output signals.

*tabrec* and *tabplay* opcodes allow to record/playback control signals on trigger-temporization basis.

*tabrec* opcode records a group of k-rate signals by storing them into *kfn* table. Each time *ktrig\_start* is triggered, *tabrec* resets the table pointer to zero and begins to record. Recording phase stops after *knumtics* trigger impluses have been received by *ktrig\_stop* argument.

*tabplay* plays back a group of k-rate signals, previously recorded by *tabrec* into a table. Each time *ktrig* argument is triggered, an internal counter is increased of one unit. After *knumtics* trigger impluses are received by *ktrig* argument, the internal counter is zeroed and playback is restarted from the beginning, in looping style.

These opcodes can be used like a sort of ``middle-term" memory that ``remembers" generated signals. Such memory can be used to supply generative music with a coherent iterative compositional structure.

## Credits

Written by Gabriel Maldonado.

# table

table -- Accesses table values by direct indexing.

table

## Description

Accesses table values by direct indexing.

## Syntax

ares **table** andx, ifn [, ixmode] [, ixoff] [, iwrap]

ires **table** indx, ifn [, ixmode] [, ixoff] [, iwrap]

kres **table** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

## Initialization

*ifn* -- function table number.

*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 `tables-ize/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 tablesizes sticks at index=size)
- 1 = wraparound.

## Performance

*table* invokes table lookup on behalf of init, control or audio indices. These indices can be raw entry numbers (0,1,2...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.

## Examples

Here is an example of the table opcode. It uses the files *table.orc* [examples/table.orc] and *table.sco* [examples/table.sco].

### Example 368. Example of the table opcode.

```
/* table.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Vary our index linearly from 0 to 1.
  kndx line 0, p3, 1

  ; Read Table #1 with our index.
  ifn = 1
  ixmode = 1
  kfreq table kndx, ifn, ixmode

  ; Generate a sine waveform, use our table values
  ; to vary its frequency.
  a1 oscil 20000, kfreq, 2
  out a1
endin
/* table.orc */

/* table.sco */
; Table #1, a line from 200 to 2,000.
f 1 0 1025 -7 200 1024 2000
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* table.sco */
```

## See Also

*tablei, table3, oscil1, oscil1i, osciln*

## Credits

Example written by Kevin Conder.

# table3

table3 -- Accesses table values by direct indexing with cubic interpolation.

table3

## Description

Accesses table values by direct indexing with cubic interpolation.

## Syntax

ares **table3** andx, ifn [, ixmode] [, ixoff] [, iwrap]

ires **table3** indx, ifn [, ixmode] [, ixoff] [, iwrap]

kres **table3** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

## Initialization

*ifn* -- function table number.

*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 `tables-ize/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 tablesizes sticks at index=size)
- 1 = wraparound.

## Performance

*table3* is identical to *tablei*, except that it uses cubic interpolation. (New in Csound version 3.50.)

## See Also

*table*, *tablei*, *oscil1*, *oscilli*, *osciln*



# tablecopy

tablecopy -- Simple, fast table copy opcode.

tablecopy

## Description

Simple, fast table copy opcode.

## Syntax

**tablecopy** *kdft*, *ksft*

## Performance

*kdft* -- Destination function table.

*ksft* -- Number of source function table.

*tablecopy* -- Simple, fast table copy opcode. 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.

## See Also

*tablegpw*, *tablemix*, *tableicopy*, *tableigpw*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tablegpw

tablegpw -- Writes a table's guard point.

tablegpw

## Description

Writes a table's guard point.

## Syntax

**tablegpw** *kfn*

## Performance

*kfn* -- Table number to be interrogated

*tablegpw* -- 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*.

## See Also

*tablecopy*, *tablemix*, *tableicopy*, *tableigpw*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tablei

tablei -- Accesses table values by direct indexing with linear interpolation.

tablei

## Description

Accesses table values by direct indexing with linear interpolation.

## Syntax

ares **tablei** andx, ifn [, ixmode] [, ixoff] [, iwrap]

ires **tablei** indx, ifn [, ixmode] [, ixoff] [, iwrap]

kres **tablei** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

## Initialization

*ifn* -- function table number. *tablei* requires 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 *tables-ize/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 *tables-ize* sticks at index=size)
- 1 = wraparound.

## Performance

*tablei* is a interpolating unit 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).

## See Also

*table*, *table3*, *oscil1*, *oscil1i*, *osciln*

# tablecopy

tablecopy -- Simple, fast table copy opcode.

tablecopy

## Description

Simple, fast table copy opcode.

## Syntax

**tablecopy** *idft*, *isft*

## Initialization

*idft* -- Destination function table.

*isft* -- Number of source function table.

## Performance

*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.

## See Also

*tablecopy*, *tablegpw*, *tablemix*, *tableigpw*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tableigpw

tableigpw -- Writes a table's guard point.

tableigpw

## Description

Writes a table's guard point.

## Syntax

```
tableigpw ifn
```

## Initialization

*ifn* -- Table number to be interrogated

## Performance

*tableigpw* -- 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*.

## See Also

*tablecopy*, *tablegpw*, *tablemix*, *tableicopy*, *tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tableikt

tableikt -- Provides k-rate control over table numbers.

tableikt

## Description

k-rate control over table numbers.

The standard Csound opcode *tablei*, when producing a k- or a-rate result, can only use an init-time variable to select the table number. *tableikt* accepts k-rate control as well as i-time. In all other respects they are similar to the original opcodes.

## Syntax

```
ares tableikt xndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

```
kres tableikt kndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*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*, ie. 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* not equal to 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* not equal to 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* not equal to 0)

*kfn* -- Table number. Must be  $\geq 1$ . 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.



### Caution with k-rate table numbers

At k-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* must be initialized at the appropriate rate using *init*. Attempting to load an i-rate value into *kfn* will result in an error.

## See Also

*tablekt*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tableimix

tableimix -- Mixes two tables.

tableimix

## Description

Mixes two tables.

## Syntax

**tableimix** *idft*, *idoff*, *ilen*, *is1ft*, *is1off*, *is1g*, *is2ft*, *is2off*, *is2g*

## Initialization

*idft* -- Destination function table.

*idoff* -- Offset to start writing from. Can be negative.

*ilen* -- Number of write operations to perform. Negative means work backwards.

*is1ft*, *is2ft* -- Source function tables. These can be the same as the destination table, if care is exercised about direction of copying data.

*is1off*, *is2off* -- Offsets to start reading from in source tables.

*is1g*, *is2g* -- Gains to apply when reading from the source tables. The results are added and the sum is written to the destination table.

## Performance

*tableimix* -- 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.



## See Also

*tablecopy*, *tablegpw*, *tablemix*, *tableicopy*, *tableigpw*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tableiw

tableiw -- Change the contents of existing function tables.

tableiw

## Description

This opcode operates 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. The valid combinations of variable types are shown by the first letter of the variable names.

## Syntax

**tableiw** isig, indx, ifn [, ixmode] [, ixoff] [, iwgmodes]

## Initialization

*isig* -- Input value to write to the table.

*indx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* not equal to 0)

*ifn* -- Table number. Must be  $\geq 1$ . 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* (optional, default=0) -- index mode.

- 0 = *indx* and *ixoff* ranges match the length of the table.
- not equal to 0 = *indx* and *ixoff* have a 0 to 1 range.

*ixoff* (optional, default=0) -- index offset.

- 0 = Total index is controlled directly by *indx*, i.e. the indexing starts from the start of the table.
- Not equal to 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* not equal to 0).

*iwgmodes* (optional, default=0) -- Wrap and guard point mode.

- 0 = Limit mode.
- 1 = Wrap mode.
- 2 = Guardpoint mode.

## Performance

### Limit mode (0)

Limit the total index ( $indx + ixoff$ ) 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.

## Wrap mode (1)

Wrap total index value into locations 0 to E, where E is either one less than 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  $igwmode = 2$ , 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.

## See Also

*tablew*, *tablewkt*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Updated August 2002, thanks go to Abram Hindle for pointing out the correct syntax.

# tablekt

tablekt -- Provides k-rate control over table numbers.

tablekt

## Description

k-rate control over table numbers.

The standard Csound opcode *table* when producing a k- or a-rate result, can only use an init-time variable to select the table number. *tablekt* accepts k-rate control as well as i-time. In all other respects they are similar to the original opcodes.

## Syntax

```
ares tablekt xndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

```
kres tablekt kndx, kfn [, ixmode] [, ixoff] [, iwrap]
```

## Initialization

*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*, ie. 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* not equal to 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* not equal to 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* not equal to 0)

*kfn* -- Table number. Must be  $\geq 1$ . 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.



### Caution with k-rate table numbers

At k-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* must be initialized at the appropriate rate using *init*. Attempting to load an i-rate value into *kfn* will result in an error.

## See Also

*tableikt*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tablemix

tablemix -- Mixes two tables.

tablemix

## Description

Mixes two tables.

## Syntax

**tablemix** *kdft*, *kdoff*, *klen*, *ks1ft*, *ks1off*, *ks1g*, *ks2ft*, *ks2off*, *ks2g*

## Performance

*kdft* -- Destination function table.

*kdoff* -- Offset to start writing from. Can be negative.

*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.

*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.

## See Also

*tablecopy, tablegpw, tableicopy, tableigpw, tableimix*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tableng

tableng -- Interrogates a function table for length.

tableng

## Description

Interrogates a function table for length.

## Syntax

```
ires tableng ifn
```

```
kres tableng kfn
```

## Initialization

*ifn* -- Table number to be interrogated

## 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 *table-copy*.

## Examples

Here is an example of the *tableng* opcode. It uses the files *tableng.orc* [examples/*tableng.orc*] and *tableng.sco* [examples/*tableng.sco*].

### Example 369. Example of the *tableng* opcode.

```
/* tableng.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Let's look at Table #1.
  ifn = 1
  ilen tableng ifn

  print ilen
endin
/* tableng.orc */
```



```
/* tablen.sco */  
; Table #1, a sine wave.  
f 1 0 16384 10 1  
  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* tablen.sco */
```

The table is 16,384 samples long. So its output should include a line like this:

```
instr 1:  ilen = 16384.000
```

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# tablera

tablera -- Reads tables in sequential locations.

tablera

## 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.

## Syntax

ares **tablera** kfn, kstart, koff

## Performance

*ares* -- 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.

*koff* -- i- or k-rate offset into table. Range unlimited - see explanation at end of this section.

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 Number	Initial kstart	Final kstart	Locations Written
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 dependent 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* not equal to 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.

# tableseg

`tableseg` -- Creates a new function table by making linear segments between values in stored function tables.

`tableseg`

## Description

*tableseg* is like *linseg* 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* (much like *lpread/lpreson* pairs work). The uses of these are described below under *vpvoc*.

Note: this opcode can also be written as *ktableseg*.

## Syntax

```
tableseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]
```

## Initialization

*ifn1*, *ifn2*, *ifn3*, etc. -- function table numbers. *ifn1*, *ifn2*, and so on, must be the same size.

*idur1*, *idur2*, etc. -- durations during which interpolation from one table to the next will take place.

## See Also

*pvbufread*, *pvcross*, *pvinterp*, *pvread*, *tablexseg*

## Credits

Author: Richard Karpen  
Seattle, Wash  
1997

# tablew

tablew -- Change the contents of existing function tables.

tablew

## Description

This opcode operates on existing function tables, changing their contents. *tablew* is for writing at k- or at a-rates, with the table number being specified at init time. The valid combinations of variable types are shown by the first letter of the variable names.

## Syntax

**tablew** asig, andx, ifn [, ixmode] [, ixoff] [, iwgmode]

**tablew** isig, indx, ifn [, ixmode] [, ixoff] [, iwgmode]

**tablew** ksig, kndx, ifn [, ixmode] [, ixoff] [, iwgmode]

## Initialization

*asig, isig, ksig* -- The value to be written into the table.

*andx, indx, kndx* -- Index into table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0)

*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 (*GENOI*) then an error will result and the instrument will be de-activated.

*ixmode* (optional, default=0) -- index mode.

- 0 = *xndx* and *ixoff* ranges match the length of the table.
- !=0 = *xndx* and *ixoff* have a 0 to 1 range.

*ixoff* (optional, default=0) -- index offset.

- 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* (optional, default=0) -- Wrap and guardpoint mode.

- 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.

### Wrap mode (1)

Wrap total index value into locations 0 to E, where E is either one less than 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  $igwmode = 2$ , 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

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.

## See Also

*tableiw*, *tablewkt*

## Credits

Author: Robin Whittle  
Australia  
May 1997





# tablewa

tablewa -- Writes tables in sequential locations.

tablewa

## 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.

## Syntax

kstart **tablewa** kfn, asig, koff

## Performance

*kstart* -- Where in table to read or write.

*kfn* -- i- or k-rate number of the 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.

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 Number	Initial kstart	Final kstart	Locations Written
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 dependent 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* not equal to 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.

## Credits

Author: Robin Whittle  
Australia

# tablewkt

tablewkt -- Change the contents of existing function tables.

tablewkt

## Description

This opcode operates on existing function tables, changing their contents. *tablewkt* 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.

## Syntax

**tablewkt** *asig*, *andx*, *kfn* [, *ixmode*] [, *ixoff*] [, *iwgmode*]

**tablewkt** *ksig*, *kndx*, *kfn* [, *ixmode*] [, *ixoff*] [, *iwgmode*]

## Initialization

*asig*, *ksig* -- The value to be written into the table.

*andx*, *kndx* -- Index into table, either a positive number range 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.

*ixmode* -- index mode. Default is zero.

- 0 = *xndx* and *ixoff* ranges match the length of the table.
- Not equal to 0 = *xndx* and *ixoff* have a 0 to 1 range.

*ixoff* -- index offset. Default is 0.

- 0 = Total index is controlled directly by *xndx*, i.e. the indexing starts from the start of the table.
- Not equal to 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* -- table writing mode. 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.

## 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.

## Caution with k-rate table numbers

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.

## See Also

*tableiw*, *tablew*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tablexkt

tablexkt -- Reads function tables with linear, cubic, or sinc interpolation.

tablexkt

## Description

Reads function tables with linear, cubic, or sinc interpolation.

## Syntax

ares **tablexkt** xndx, kfn, kwarp, iwsiz [ , ixmode] [ , ixoff] [ , iwrap]

## Initialization

*iwsiz* -- This parameter controls the type of interpolation to be used:

- 2: Use linear interpolation. This is the lowest quality, but also the fastest mode.
- 4: Cubic interpolation. Slightly better quality than *iwsiz* = 2, at the expense of being somewhat slower.
- 8 and above (up to 1024): sinc interpolation with window size set to *iwsiz* (should be an integer multiply of 4). Better quality than linear or cubic interpolation, but very slow. When transposing up, a *kwarp* value above 1 can be used for anti-aliasing (this is even slower).

*ixmode*1 (optional) -- index data mode. The default value is 0.

- 0: raw index
- any non-zero value: normalized (0 to 1)



## Notes

if *tablexkt* is used to play back samples with looping (e.g. table index is generated by lphaser), there must be at least *iwsiz* / 2 extra samples after the loop end point for interpolation, otherwise audible clicking may occur (also, at least *iwsiz* / 2 samples should be before the loop start point).

*ixoff* (optional) -- amount by which index is to be offset. For a table with origin at center, use *tablesiz* / 2 (raw) or 0.5 (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 >= *tablesiz* (or 1.0 in normalized mode) sticks at the guard point).
- any non-zero value: Index is wrapped to the allowed range (not including the guard point in this case).

**Note**

*iwrap* also applies to extra samples for interpolation.

## Performance

*ares* -- audio output

*xndx* -- table index

*kfn* -- function table number

*kwarp* -- if greater than 1, use  $\sin(x / \text{kwarp}) / x$  function for sinc interpolation, instead of the default  $\sin(x) / x$ . This is useful to avoid aliasing when transposing up (*kwarp* should be set to the transpose factor in this case, e.g. 2.0 for one octave), however it makes rendering up to twice as slow. Also, *iwsiz*e should be at least  $\text{kwarp} * 8$ . This feature is experimental, and may be optimized both in terms of speed and quality in new versions.

**Note**

*kwarp* has no effect if it is less than, or equal to 1, or linear or cubic interpolation is used.

## Credits

Author: Istvan Varga  
January 2002

New in version 4.18

# tablexseg

`tablexseg` -- Creates a new function table by making exponential segments between values in stored function tables.

`tablexseg`

## Description

*tablexseg* is like *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 *tablexseg* (much like *lpread/lpreson* pairs work). The uses of these are described below under *vpvoc*.

## Syntax

```
tablexseg ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]
```

## Initialization

*ifn1*, *ifn2*, *ifn3*, etc. -- function table numbers. *ifn1*, *ifn2*, and so on, must be the same size.

*idur1*, *idur2*, etc. -- durations during which interpolation from one table to the next will take place.

## See Also

*pvbufread*, *pvcross*, *pvinterp*, *pvread*, *tableseg*

## Credits

Author: Richard Karpen  
Seattle, WA USA  
1997



# tambourine

tambourine -- Semi-physical model of a tambourine sound.

tambourine

## Description

*tambourine* is a semi-physical model of a tambourine sound. It is one of the PhISEM percussion opcodes. PhISEM (Physically Informed Stochastic Event Modeling) is an algorithmic approach for simulating collisions of multiple independent sound producing objects.

## Syntax

ares **tambourine** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1] [, ifreq2]

## Initialization

*idettack* -- period of time over which all sound is stopped

*inum* (optional) -- The number of beads, teeth, bells, timbrels, etc. If zero, the default value is 32.

*idamp* (optional) -- the damping factor, as part of this equation:

$$\text{damping\_amount} = 0.9985 + (\text{idamp} * 0.002)$$

The default *damping\_amount* is 0.9985 which means that the default value of *idamp* is 0. The maximum *damping\_amount* is 1.0 (no damping). This means the maximum value for *idamp* is 0.75.

The recommended range for *idamp* is usually below 75% of the maximum value.

*imaxshake* (optional, default=0) -- amount of energy to add back into the system. The value should be in range 0 to 1.

*ifreq* (optional) -- the main resonant frequency. The default value is 2300.

*ifreq1* (optional) -- the first resonant frequency. The default value is 5600.

*ifreq2* (optional) -- the second resonant frequency. The default value is 8100.

## Performance

*kamp* -- Amplitude of output. Note: As these instruments are stochastic, this is only an approximation.

## Examples

Here is an example of the tambourine opcode. It uses the files *tambourine.orc* [examples/tambourine.orc] and *tambourine.sco* [examples/tambourine.sco].

### Example 370. Example of the tambourine opcode.

```
/* tambourine.orc */
sr = 22050
```

```
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1: An example of a tambourine.
instr 01
  al tambourine 15000, 0.01

  out al
endin
/* tambourine.orc */

/* tambourine.sco */
i 1 0 1
e
/* tambourine.sco */
```

## See Also

*bamboo, dripwater, guiro, sleighbells*

## Credits

Author: Perry Cook, part of the PhISEM (Physically Informed Stochastic Event Modeling)  
Adapted by John ffitch  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

Added notes by Rasmus Ekman on May 2002.

# tan

tan -- Performs a tangent function.

tan

## Description

Returns the tangent of  $x$  ( $x$  in radians).

## Syntax

**tan**( $x$ ) (no rate restriction)

## Examples

Here is an example of the tan opcode. It uses the files *tan.orc* [examples/tan.orc] and *tan.sco* [examples/tan.sco].

### Example 371. Example of the tan opcode.

```
/* tan.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 25
  il = tan(irad)

  print il
endin
/* tan.orc */

/* tan.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* tan.sco */
```

Its output should include a line like this:

```
instr 1:  il = -0.134
```

## See Also

*cos, cosh, cosinv, sin, sinh, sininv, tan, taninv*

## Credits

Example written by Kevin Conder.

# tanh

tanh -- Performs a hyperbolic tangent function.

tanh

## Description

Returns the hyperbolic tangent of  $x$  ( $x$  in radians).

## Syntax

**tanh**( $x$ ) (no rate restriction)

## Examples

Here is an example of the tanh opcode. It uses the files *tanh.orc* [examples/tanh.orc] and *tanh.sco* [examples/tanh.sco].

### Example 372. Example of the tanh opcode.

```
/* tanh.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 1
  il = tanh(irad)

  print il
endin
/* tanh.orc */

/* tanh.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* tanh.sco */
```

Its output should include a line like this:

```
instr 1:  il = 0.762
```

## See Also

*cos, cosh, cosinv, sin, sinh, sininv, tan, taninv*

## Credits

Author: John ffitch

Example written by Kevin Conder.

# taninv

taninv -- Performs an arctangent function.

taninv

## Description

Returns the arctangent of  $x$  ( $x$  in radians).

## Syntax

**taninv**( $x$ ) (no rate restriction)

## Examples

Here is an example of the taninv opcode. It uses the files *taninv.orc* [examples/taninv.orc] and *taninv.sco* [examples/taninv.sco].

### Example 373. Example of the taninv opcode.

```
/* taninv.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  irad = 0.5
  il = taninv(irad)

  print il
endin
/* taninv.orc */

/* taninv.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* taninv.sco */
```

Its output should include a line like this:

```
instr 1:  il = 0.464
```

## See Also

*cos, cosh, cosinv, sin, sinh, sininv, tan, tanh, taninv2*

## Credits

Author: John ffitch

Example written by Kevin Conder.



# taninv2

taninv2 -- Returns an arctangent.

taninv2

## Description

Returns the arctangent of  $iy/ix$ ,  $ky/kx$ , or  $ay/ax$ .

## Syntax

ares **taninv2** ay, ax

ires **taninv2** iy, ix

kres **taninv2** ky, kx

Returns the arctangent of  $iy/ix$ ,  $ky/kx$ , or  $ay/ax$ . If y is zero, *taninv2* returns zero regardless of the value of x. If x is zero, the return value is:

- $PI/2$ , if y is positive.
- $-PI/2$ , if y is negative.
- 0, if y is 0.

## Initialization

*iy*, *ix* -- values to be converted

## Performance

*ky*, *kx* -- control rate signals to be converted

*ay*, *ax* -- audio rate signals to be converted

## Examples

Here is an example of the *taninv2* opcode. It uses the files *taninv2.orc* [examples/taninv2.orc] and *taninv2.sco* [examples/taninv2.sco].

### Example 374. Example of the taninv2 opcode.

```
/* taninv2.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
```

```
instr 1
; Returns the arctangent for 1/2.
il taninv2 1, 2

print il
endin
/* taninv2.orc */

/* taninv2.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* taninv2.sco */
```

Its output should include a line like this:

```
instr 1:  il = 0.464
```

## See Also

*taninv*

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

Example written by Kevin Conder.

New in Csound version 3.48

Corrected on May 2002, thanks to Istvan Varga.

# tbvcf

tbvcf -- Models some of the filter characteristics of a Roland TB303 voltage-controlled filter.

tbvcf

## 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 untwine, 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*.

## Syntax

```
ares tbvcf asig, xfco, xres, kdist,  
      kasym [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*asig* -- input signal. Should be normalized to  $\pm 1$ .

*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.

## Examples

Here is an example of the *tbvcf* opcode. It uses the files *tbvcf.orc* [examples/tbvcf.orc] and *tbvcf.sco* [examples/tbvcf.sco].

### Example 375. Example of the *tbvcf* opcode.

```
/* tbvcf.orc */  
;-----  
; 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
/* tbvcf.orc */

/* tbvcf.sco */
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
e
/* tbvcf.sco */
```

## Credits

Author: Hans Mikelson  
December, 2000 -- January, 2001

New in Csound 4.10

# tempest

tempest -- Estimate the tempo of beat patterns in a control signal.

tempest

## Description

Estimate the tempo of beat patterns in a control signal.

## Syntax

ktemp **tempest** kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, istart

## 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.

## Examples

Here is an example of the *tempest* opcode. It uses the files *tempest.orc* [examples/tempest.orc], *tempest.sco* [examples/tempest.sco], and *beats.wav* [examples/beats.wav].

### Example 376. Example of the *tempest* opcode.

```
/* tempest.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use the "beats.wav" sound file.
  asig soundin "beats.wav"
  ; Extract the pitch and the envelope.
  kcps, krms pitchamdf asig, 150, 500, 200

  iprd = 0.01
  imindur = 0.1
  imemdur = 3
  ihp = 1
  ithresh = 30
  ihtim = 0.005
  ixfdbak = 0.05
  istartempo = 110
  ifn = 1

  ; Estimate its tempo.
  k1 tempest krms, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, istartempo
  printk2 k1

  out asig
endin
/* tempest.orc */

/* tempest.sco */
; Table #1, a declining line.
f 1 0 128 16 1 128 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* tempest.sco */
```

The tempo of the audio file “beats.wav” is 120 beats per minute. In this examples, *tempest* will print out its best guess as the audio file plays. Its output should include lines like this:

```
. i1 118.24654
. i1 121.72949
```

# tempo

tempo -- Apply tempo control to an uninterpreted score.

tempo

## Description

Apply tempo control to an uninterpreted score.

## Syntax

**tempo** ktempo, istartempo

## Initialization

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

## Performance

*ktempo* -- The tempo to which the score will be adjusted.

*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.

## Examples

Here is an example of the tempo opcode. Remember, it only works if you use the *-t* flag with Csound. The example uses the files *tempo.orc* [examples/tempo.orc] and *tempo.sco* [examples/tempo.sco].

### Example 377. Example of the tempo opcode.

```
/* tempo.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; If the fourth p-field is 1, increase the tempo.
  if (p4 == 1) kgoto speedup
  kgoto playit

speedup:
  ; Increase the tempo to 150 beats per minute.
  tempo 150, 60

playit:
  a1 oscil 10000, 440, 1
```



```
    out a1
  endin
  /* tempo.orc */

  /* tempo.sco */
  ; Table #1, a sine wave.
  f 1 0 16384 10 1

  ; p4 = plays at a faster tempo (when p4=1).
  ; Play Instrument #1 at the normal tempo, repeat 3 times.
  r3
  i 1 00.00 00.10 0
  i 1 00.25 00.10 0
  i 1 00.50 00.10 0
  i 1 00.75 00.10 0
  s

  ; Play Instrument #1 at a faster tempo, repeat 3 times.
  r3
  i 1 00.00 00.10 1
  i 1 00.25 00.10 1
  i 1 00.50 00.10 1
  i 1 00.75 00.10 1
  s

  e
  /* tempo.sco */
```

## See Also

*tempoval*

## Credits

Example written by Kevin Conder.

# tempoval

tempoval -- Reads the current value of the tempo.

tempoval

## Description

Reads the current value of the tempo.

## Syntax

```
kres tempoval
```

## Performance

*kres* -- the value of the tempo. If you use a positive value with the *-t command-line flag*, *tempoval* returns the percentage increase/decrease from the original tempo of 60 beats per minute. If you don't, its value will be 60 (for 60 beats per minute).

## Examples

Here is an example of the *tempoval* opcode. Remember, it only works if you use the *-t* flag with Csound. It uses the files *tempoval.orc* [examples/tempoval.orc] and *tempoval.sco* [examples/tempoval.sco].

### Example 378. Example of the tempoval opcode.

```
/* tempoval.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Adjust the tempo to 120 beats per minute.
  tempo 120, 60

  ; Get the tempo value.
  kval tempoval

  printks "kval = %f\\n", 0.1, kval
endin
/* tempoval.orc */
```

```
/* tempoval.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* tempoval.sco */
```

Since 120 beats per minute is a 50% increase over the original 60 beats per minute, its output should include lines like:

```
kval = 0.500000
```

## See Also

*tempo*

## Credits

Example written by Kevin Conder.

New in version 4.15

December 2002. Thanks to Drake Wilson for pointing out unclear documentation.

# tigoto

tigoto -- Transfer control at i-time when a new note is being tied onto a previously held note

tigoto

## Description

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*) It does not work 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*).

## Syntax

**tigoto** label

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

## See Also

*cigoto*, *goto*, *if*, *igoto*, *kgoto*, *timeout*

## Credits

Added a note by Jim Aikin.

# timeinstk

timeinstk -- Read absolute time in k-rate cycles.

timeinstk

## Description

Read absolute time, in k-rate cycles, since the start of an instance of an instrument. Called at both i-time as well as k-time.

## Syntax

```
kres timeinstk
```

```
kres timeinsts
```

## Performance

*timeinstk* is for time in k-rate 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.

*timeinstk* produces a k-rate variable for output. There are no input parameters.

*timeinstk* is similar to *timek* except it returns the time since the start of this instance of the instrument.

## Examples

Here is an example of the *timeinstk* opcode. It uses the files *timeinstk.orc* [examples/timeinstk.orc] and *timeinstk.sco* [examples/timeinstk.sco].

### Example 379. Example of the *timeinstk* opcode.

```
/* timeinstk.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the value from timeinstk every half-second.
  k1 timeinstk
  printks "k1 = %f samples\\n", 0.5, k1
endin
```

```
/* timeinstk.orc */

/* timeinstk.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* timeinstk.sco */
```

Its output should include lines like this:

```
k1 = 1.000000 samples
k1 = 2205.000000 samples
k1 = 4410.000000 samples
k1 = 6615.000000 samples
k1 = 8820.000000 samples
```

## See Also

*timeinsts*, *timek*, *times*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# timeinsts

`timeinsts` -- Read absolute time in seconds.

`timeinsts`

## Description

Read absolute time, in seconds, since the start of an instance of an instrument.

## Syntax

`kres timeinsts`

## Performance

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

*timeinsts* produces a k-rate variable for output. There are no input parameters.

*timeinsts* is similar to *times* except it returns the time since the start of this instance of the instrument.

## Examples

Here is an example of the `timeinsts` opcode. It uses the files *timeinsts.orc* [examples/timeinsts.orc] and *timeinsts.sco* [examples/timeinsts.sco].

### Example 380. Example of the `timeinsts` opcode.

```
/* timeinsts.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the value from timeinsts every half-second.
  k1 timeinsts
  printks "k1 = %f seconds\\n", 0.5, k1
endin
/* timeinsts.orc */
```

```
/* timeinsts.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* timeinsts.sco */
```

Its output should include lines like this:

```
k1 = 0.000227 seconds  
k1 = 0.500000 seconds  
k1 = 1.000000 seconds  
k1 = 1.500000 seconds  
k1 = 2.000000 seconds
```

## See Also

*timeinstk, timek, times*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.



# timek

timek -- Read absolute time in k-rate cycles.

timek

## Description

Read absolute time, in k-rate cycles, since the start of the performance.

## Syntax

ires **timek**

kres **timek**

## Performance

*timek* is for time in k-rate 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.

*timek* can produce a k-rate variable for output. There are no input parameters.

*timek* can also operate only at the start of the instance of the instrument. It produces an i-rate variable (starting with *i* or *gi*) as its output.

## Examples

Here is an example of the *timek* opcode. It uses the files *timek.orc* [examples/timek.orc] and *timek.sco* [examples/timek.sco].

### Example 381. Example of the *timek* opcode.

```
/* timek.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the value from timek every half-second.
  k1 timek
  printks "k1 = %f samples\\n", 0.5, k1
endin
/* timek.orc */
```

```
/* timek.sco */  
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* timek.sco */
```

Its output should include lines like this:

```
k1 = 1.000000 samples  
k1 = 2205.000000 samples  
k1 = 4410.000000 samples  
k1 = 6615.000000 samples  
k1 = 8820.000000 samples
```

## See Also

*timeinstk, timensts, times*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# times

`times` -- Read absolute time in seconds.

`times`

## Description

Read absolute time, in seconds, since the start of the performance.

## Syntax

`ires times`

`kres times`

## Performance

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

*times* can both produce a k-rate variable for output. There are no input parameters.

*times* can also operate at the start of the instance of the instrument. It produces an i-rate variable (starting with *i* or *gi*) as its output.

## Examples

Here is an example of the *times* opcode. It uses the files *times.orc* [examples/times.orc] and *times.sco* [examples/times.sco].

### Example 382. Example of the *times* opcode.

```
/* times.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print out the value from times every half-second.
  k1 times
  printks "k1 = %f seconds\\n", 0.5, k1
endin
/* times.orc */
```

```
/* times.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* times.sco */
```

Its output should include lines like this:

```
k1 = 0.000227 seconds  
k1 = 0.500000 seconds  
k1 = 1.000000 seconds  
k1 = 1.500000 seconds  
k1 = 2.000000 seconds
```

## See Also

*timeinstk, timeinsts, timek*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# timeout

timeout -- Conditional branch during p-time depending on elapsed note time.

timeout

## Description

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 *timeout* can be reinitialized for multiple activation within a single note (see example under *reinit*).

## Syntax

```
timeout istrt, idur, label
```

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

## See Also

*goto*, *if*, *igoto*, *kgoto*, *tigoto*

## Credits

Added a note by Jim Aikin.

# tival

tival -- Puts the value of the instrument's internal “tie-in” flag into the named i-rate variable.

tival

## Syntax

```
ir tival
```

## Description

Puts the value of the instrument's internal “tie-in” flag into the named i-rate variable.

## Initialization

Puts 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*.)

## See Also

=, *divz*, *init*

# tlineto

tlineto -- Generate glissandos starting from a control signal.

tlineto

## Description

Generate glissandos starting from a control signal with a trigger.

## Syntax

```
kres tlineto ksig, ktime, ktrig
```

## Performance

*kres* -- Output signal.

*ksig* -- Input signal.

*ktime* -- Time length of glissando in seconds.

*ktrig* -- Trigger signal.

*tlineto* is similar to *lineto* but can be applied to any kind of signal (not only stepped signals) without producing discontinuities. Last value of each segment is sampled and held from input signal each time *ktrig* value is set to a nonzero value. Normally *ktrig* signal consists of a sequence of zeroes (see *trigger opcode*).

The effect of glissando is quite different from *port*. Since in these cases, the lines are straight. Also the context of useage is different.

## See Also

*lineto*

## Credits

Author: Gabriel Maldonado

New in Version 4.13

# tone

tone -- A first-order recursive low-pass with variable frequency response.

tone

## Description

A first-order recursive low-pass with variable frequency response.

Tone is a 1 term IIR filter. Its formula is:

$$y_n = c1 * x_n + c2 * y_{n-1}$$

where

- $b = 2 - \cos(2 \pi \text{ hp/sr})$ ;
- $c2 = b - \sqrt{b^2 - 1.0}$
- $c1 = 1 - c2$

## Syntax

ares **tone** asig, khp [, iskip]

## Initialization

*iskip* (optional, default=0) -- 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

*ares* -- the output audio signal.

*asig* -- the input audio signal.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*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.

## See Also

*areson*, *aresonk*, *atone*, *atonek*, *port*, *portk*, *reson*, *resonk*, *tonek*



# tonek

tonek -- A first-order recursive low-pass filter with variable frequency response.

tonek

## Description

A first-order recursive low-pass filter with variable frequency response.

## Syntax

```
kres tonek ksig, khp [, iskip]
```

## Initialization

*iskip* (optional, default=0) -- 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

*kres* -- the output signal at control-rate.

*ksig* -- the input signal at control-rate.

*khp* -- the response curve's half-power point, in Hertz. Half power is defined as peak power / root 2.

*tonek* is like *tone* except its output is at control-rate rather than audio rate.

## See Also

*areson*, *aresonk*, *atone*, *atonek*, *port*, *portk*, *reson*, *resonk*, *tone*

## Credits

Author: Robin Whittle  
Australia  
May 1997

# tonex

tonex -- Emulates a stack of filters using the tone opcode.

tonex

## Description

*tonex* is equivalent to a filter consisting of more layers of *tone* 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.

## Syntax

```
ares tonex asig, khp [, inumlayer] [, iskip]
```

## Initialization

*inumlayer* (optional) -- number of elements in the filter stack. Default value is 4.

*iskip* (optional, default=0) -- 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* -- input signal

*khp* -- the response curve's half-power point. Half power is defined as peak power / root 2.

## See Also

*atonex*, *resonx*

## Credits

Author: Gabriel Maldonado (adapted by John ffitch)  
Italy

New in Csound version 3.49

# tradsyn

tradsyn -- Streaming partial track additive synthesis

tradsyn

## Description

The `tradsyn` opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by `partials`), as described in Lazzarini et al, "Time-stretching using the Instantaneous Frequency Distribution and Partial Tracking", Proc.of ICMC05, Barcelona. It resynthesises the signal using linear amplitude and frequency interpolation to drive a bank of interpolating oscillators with amplitude and pitch scaling controls.

## Syntax

asig **tradsyn** fin, kscal, kpitch, kmaxtracks, ifn

## Performance

*asig* -- output audio rate signal

*fin* -- input pv stream in TRACKS format

*kscal* -- amplitude scaling

*kpitch* -- pitch scaling

*kmaxtracks* -- max number of tracks in resynthesis. Limiting this will cause a non-linear filtering effect, by discarding newer and higher-frequency tracks (tracks are ordered by start time and ascending frequency, respectively)

*ifn* -- function table containing one cycle of a sinusoid (sine or cosine)

## Examples

### Example 383. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
aout tradsyn fst, 1, 1.5, 500, 1 ; resynthesis (up a 5th)
out aout
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini;

June 2005

New plugin in version 5

November 2004.

# transeg

transeg -- Constructs a user-definable envelope.

transeg

## Description

Constructs a user-definable envelope.

## Syntax

ares **transeg** ia, idur, itype, ib [, idur2] [, itype] [, ic] ...

kres **transeg** ia, idur, itype, ib [, idur2] [, itype] [, ic] ...

## Initialization

ia -- starting value.

ib, ic, etc. -- value after *idur* seconds.

idur, idur2, etc. -- duration in seconds of segment

itype, itype2, etc. -- 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 *GENI6*.

## See Also

*expsega*, *expsegr*, *linseg*, *linsegr*

## Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
October 2000

New in Csound version 4.09

Thanks goes to Matt Gerassimoff for pointing out the correct command syntax.

# trcross

trcross -- Streaming partial track cross-synthesis.

trcross

## Description

The trcross opcode takes two inputs containing TRACKS pv streaming signals (as generated, for instance by partials) and cross-synthesises them into a single TRACKS stream. Two different modes of operation are used: mode 0, cross-synthesis by multiplication of the amplitudes of the two inputs and mode 1, cross-synthesis by the substitution of the amplitudes of input 1 by the input 2. Frequencies and phases of input 1 are preserved in the output. The cross-synthesis is done by matching tracks between the two inputs using a 'search interval'. The matching algorithm will look for tracks in the second input that are within the search interval around each track in the first input. This interval can be changed at the control rate. Wider search intervals will find more matches.

## Syntax

```
fsig trcross fin1, fin2, ksearch,kdepth[,kmode]
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin1* -- first input pv stream in TRACKS format.

*fin2* -- second input pv stream in TRACKS format

*ksearch* -- search interval ratio, defining a 'search area' around each track of 1st input for matching purposes.

*kdepth* -- depth of effect (0-1).

*kmode* -- mode of cross-synthesis. 0, multiplication of amplitudes (filtering), 1, substitution of amplitudes of input 1 by input 2 (akin to vocoding). Defaults to 0.

## Examples

### Example 384. Example

```
ain inch 1                ; input signals
ain inch 2
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
fsl1,fsil2 pvsifd ain,2048,512,1 ; ifd analysis (second input)
fstl partials fsl1,fsil2,.003,1,3,500 ; partial tracking \ (second input
fcr trcross fst,fstl, 1.05, 1 ; cross-synthesis (mode 0)
    aout tradsyn fcr, 1, 1, 500, 1 ; resynthesis of tracks
out aout
```

The example above shows partial tracking of two ifd-analysis signals, cross-synthesis, followed by

the remix of the two parts of the spectrum and resynthesis.

## Credits

Author: Victor Lazzarini;  
February 2006

# trfilter

trfilter -- Streaming partial track filtering.

trfilter

## Description

The trfilter opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by partials) and filters it using an amplitude response curve stored in a function table. The function table can have any size (no restriction to powers-of-two). The table lookup is done by linear-interpolation. It is possible to create time-varying filter curves by updating the amplitude response table with a table-writing opcode.

## Syntax

`fsig trfilter fin, kamnt, ifn`

## Performance

*fsig* -- output pv stream in TRACKS format

*fin* -- input pv stream in TRACKS format

*kamnt* -- amount of filtering (0-1)

*ifn* -- function table number. This will contain an amplitude response curve, from 0 Hz to the Nyquist (table indexes 0 to N). Any size is allowed. Larger tables will provide a smoother amplitude response curve. Table reading uses linear interpolation.

## Examples

### Example 385. Example

```
gifn ftgen 2, 0, -22050, 5 1 1000 1 4000 0.000001 17050 0.000001 ; low-pass fil
instr 1
  ain inch 1 ; input signal
  fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
  fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
  fscl trfilter fst, 1, gifn ; filtering using function table 2
  aout tradsyn fscl, 1, 1, 500, 1 ; resynthesis
out aout
endin
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with low-pass filtering.

## Credits

Author: Victor Lazzarini;



February 2006

# trhighest

trhighest -- Extracts the highest-frequency track from a streaming track input signal.

trhighest

## Description

The trhighest opcode takes an input containing TRACKS pv streaming signals (as generated, for instance by partials) and outputs only the highest track. In addition it outputs two k-rate signals, corresponding to the frequency and amplitude of the highest track signal.

## Syntax

```
fsig, kfr, kamp trhighest finl, kscal
```

## Performance

*fsig* -- output pv stream in TRACKS format

*kfr* -- frequency (in Hz) of the highest-frequency track

*kamp* -- amplitude of the highest-frequency track

*fin* -- input pv stream in TRACKS format.

*kscal* -- amplitude scaling of output.

## Examples

### Example 386. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
fhi,kfr,kamp trhighest fst,1 ; highest freq-track
aout tradsyn fhi, 1, 1, 1, 1 ; resynthesis of highest frequency
out aout
```

The example above shows partial tracking of an ifd-analysis signal, extraction of the highest frequency and resynthesis.

## Credits

Author: Victor Lazzarini;  
February 2006

# trigger

trigger -- Informs when a krate signal crosses a threshold.

trigger

## Description

Informs when a krate signal crosses a threshold.

## Syntax

kout **trigger** ksig, kthreshold, kmode

## 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.

## Examples

Here is an example of the trigger opcode. It uses the files *trigger.orc* [examples/trigger.orc] and *trigger.sco* [examples/trigger.sco].

### Example 387. Example of the trigger opcode.

```
/* trigger.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a square-wave low frequency oscillator as the trigger.
  klf lfo 1, 10, 3
  ktr trigger klf, 1, 2

  ; When the value of the trigger isn't equal to 0, print it out.
  if (ktr == 0) kgoto contin
```

```
        ; Print the value of the trigger and the time it occurred.
        ktm times
        printks "time = %f seconds, trigger = %f\\n", 0, ktm, ktr

contin:
    ; Continue with processing.
endin
/* trigger.orc */

/* trigger.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* trigger.sco */
```

Its output should include lines like this:

```
time = 0.050340 seconds, trigger = 1.000000
time = 0.150340 seconds, trigger = 1.000000
time = 0.250340 seconds, trigger = 1.000000
time = 0.350340 seconds, trigger = 1.000000
time = 0.450340 seconds, trigger = 1.000000
time = 0.550340 seconds, trigger = 1.000000
time = 0.650340 seconds, trigger = 1.000000
time = 0.750340 seconds, trigger = 1.000000
time = 0.850340 seconds, trigger = 1.000000
time = 0.950340 seconds, trigger = 1.000000
```

## Credits

Author: Gabriel Maldonado  
Italy

Example written by Kevin Conder.

New in Csound version 3.49

# trigseq

trigseq -- Accepts a trigger signal as input and outputs a group of values.

trigseq

## Description

Accepts a trigger signal as input and outputs a group of values.

## Syntax

```
trigseq ktrig_in, kstart, kloop, kinitndx, kfn_values, kout1 [, kout2] [...]
```

## Performance

*ktrig\_in* -- input trigger signal

*kstart* -- start index of looped section

*kloop* -- end index of looped section

*kinitndx* -- initial index



### Note

Although *kinitndx* is listed as k-rate, it is in fact accessed only at init-time. So if you are using a k-rate argument, it must be assigned with *init*.

*kfn\_values* -- numer of a table containing a sequence of groups of values

*kout1* -- output values

*kout2*, ... (optional) -- more output values

This opcode handles timed-sequences of groups of values stored into a table.

*trigseq* accepts a trigger signal (*ktrig\_in*) as input and outputs group of values (contained in the *kfn\_values* table) each time *ktrig\_in* assumes a non-zero value. Each time a group of values is triggered, table pointer is advanced of a number of positions corresponding to the number of group-elements, in order to point to the next group of values. The number of elements of groups is determined by the number of *koutX* arguments.

It is possible to start the sequence from a value different than the first, by assigning to *initndx* an index different than zero (which corresponds to the first value of the table). Normally the sequence is looped, and the start and end of loop can be adjusted by modifying *kstart* and *kloop* arguments. User must be sure that values of these arguments (as well as *kinitndx*) correspond to valid table numbers, otherwise Csound will crash because no range-checking is implemented.

It is possible to disable loop (one-shot mode) by assigning the same value both to *kstart* and *kloop* arguments. In this case, the last read element will be the one corresponding to the value of such arguments. Table can be read backward by assigning a negative *kloop* value.

*trigseq* is designed to be used together with *seqtime* or *trigger* opcodes.

## See Also

*seqtime, trigger*

## Credits

Author: Gabriel Maldonado

November 2002. Added a note about the *kinitndx* parameter, thanks to Rasmus Ekman.

January 2003. Thanks to a note from Oeyvind Brandtsegg, I corrected the credits.

New in version 4.06

# trirand

trirand -- Linear distribution random number generator.

trirand

## Description

Linear distribution random number generator. This is an x-class noise generator.

## Syntax

ares **trirand** krange

ires **trirand** krange

kres **trirand** krange

## Performance

*krange* -- the range of the random numbers (*-krange* to *+krange*).

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the trirand opcode. It uses the files *trirand.orc* [examples/trirand.orc] and *trirand.sco* [examples/trirand.sco].

### Example 388. Example of the trirand opcode.

```
/* trirand.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number between -1 and 1.
  ; krange = 1

  il trirand 1

  print il
endin
/* trirand.orc */
```

```
/* trirand.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
e  
/* trirand.sco */
```

Its output should include lines like this:

```
instr 1:  i1 = 7506.261
```

## See Also

*betarand, bexprnd, cauchy, exprand, gauss, linrand, pcauchy, poisson, unirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.



# trlowest

`trlowest` -- Extracts the lowest-frequency track from a streaming track input signal.

`trlowest`

## Description

The `trlowest` opcode takes an input containing TRACKS pv streaming signals (as generated, for instance by `partials`) and outputs only the lowest track. In addition it outputs two k-rate signals, corresponding to the frequency and amplitude of the lowest track signal.

## Syntax

```
fsig, kfr, kamp trlowest finl, kscal
```

## Performance

*fsig* -- output pv stream in TRACKS format

*kfr* -- frequency (in Hz) of the lowest-frequency track

*kamp* -- amplitude of the lowest-frequency track

*fin* -- input pv stream in TRACKS format.

*kscal* -- amplitude scaling of output.

## Examples

### Example 389. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
flow,kfr,kamp trlowest fst,1 ; lowest freq-track
aout trdsyn flow, 1, 1, 1, 1 ; resynthesis of lowest frequency
out aout
```

The example above shows partial tracking of an ifd-analysis signal, extraction of the lowest frequency and resynthesis.

## Credits

Author: Victor Lazzarini;  
February 2006

# trmix

trmix -- Streaming partial track mixing.

trmix

## Description

The trmix opcode takes two inputs containing TRACKS pv streaming signals (as generated, for instance by partials) and mixes them into a single TRACKS stream. Tracks will be mixed up to the available space (defined by the original number of FFT bins in the analysed signals). If the sum of the input tracks exceeds this space, the higher-ordered tracks in the second input will be pruned.

## Syntax

```
fsig trmix fin1, fin2
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin1* -- first input pv stream in TRACKS format.

*fin2* -- second input pv stream in TRACKS format

## Examples

### Example 390. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
fslo,fshi trsplit fst, 1500 ; split partial tracks at 1500 Hz
fscl trscale fshi, 1.15 ; shift the upper tracks
fmix trmix fslo,fscl ; mix the shifted and unshifted tracks
aout tradsyn fmix, 1, 1, 500, 1 ; resynthesis of tracks
out aout
```

The example above shows partial tracking of an ifd-analysis signal, frequency splitting and pitch shifting of the upper part of the spectrum, followed by the remix of the two parts of the spectrum and resynthesis.

## Credits

Author: Victor Lazzarini;  
February 2006

# trscale

trscale -- Streaming partial track frequency scaling.

trscale

## Description

The trscale opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by partials) and scales all frequencies by a k-rate amount. It can also, optionally, scale the gain of the signal by a k-rate amount (default 1). The result is pitch shifting of the input tracks.

## Syntax

```
fsig trscale fin, kpitch[, kgain]
```

## Performance

*fsig* -- output pv stream in TRACKS format

*fin* -- input pv stream in TRACKS format

*kpitch* -- frequency scaling

*kgain* -- amplitude scaling (default 1)

## Examples

### Example 391. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
fscl trscale fst, 1.5 ; frequency scale (up a 5th)
aout tradsyn fscl, 1, 1, 500, 1 ; resynthesis
out aout
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with pitch shifting.

## Credits

Author: Victor Lazzarini;  
February 2006

# trshift

trshift -- Streaming partial track frequency scaling.

trshift

## Description

The trshift opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by partials) and shifts all frequencies by a k-rate frequency. It can also, optionally, scale the gain of the signal by a k-rate amount (default 1). The result is frequency shifting of the input tracks.

## Syntax

fsig **trshift** fin, kpsift[, kgain]

## Performance

*fsig* -- output pv stream in TRACKS format

*fin* -- input pv stream in TRACKS format

*kshift* -- frequency shift in Hz

*kgain* -- amplitude scaling (default 1)

## Examples

### Example 392. Example

```
ain inch 1 ; input signal
fs1,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fs1,fsi2,.003,1,3,500 ; partial tracking
fscl trshift fst, 150 ; frequency shift (adds 150Hz to all tracks)
aout tradsyn fscl, 1, 1, 500, 1 ; resynthesis
out aout
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis with frequency shifting.

## Credits

Author: Victor Lazzarini;  
February 2006

# trsplt

trsplt -- Streaming partial track frequency splitting.

trsplt

## Description

The trsplt opcode takes an input containing a TRACKS pv streaming signal (as generated, for instance by partials) and splits it into two signals according to a k-rate frequency 'split point'. The first output will contain all tracks up from 0Hz to the split frequency and the second will contain the tracks from the split frequency up to the Nyquist. It can also, optionally, scale the gain of the output signals by a k-rate amount (default 1). The result is two output signals containing only part of the original spectrum.

## Syntax

```
fsglow, fshi trsplt fin, ksplit[, kgainlow, kgainhigh]
```

## Performance

*fsglow* -- output pv stream in TRACKS format containing the tracks below the split point.

*fshi* -- output pv stream in TRACKS format containing the tracks above and including the split point.

*fin* -- input pv stream in TRACKS format

*ksplit* -- frequency split point in Hz

*kgainlow, kgainhigh* -- amplitude scaling of each one of the outputs (default 1).

## Examples

### Example 393. Example

```
ain inch 1 ; input signal
fsl,fsi2 pvsifd ain,2048,512,1 ; ifd analysis
fst partials fsl,fsi2,.003,1,3,500 ; partial tracking
fslo,fshi trsplt fst, 1500 ; split partial tracks at 1500 Hz
      aout trdsyn fshi, 1, 1, 500, 1 ; resynthesis of tracks above 1500Hz
out aout
```

The example above shows partial tracking of an ifd-analysis signal and linear additive resynthesis of the upper part of the spectrum (from 1500Hz).

## Credits

Author: Victor Lazzarini;  
February 2006



# turnoff

turnoff -- Enables an instrument to turn itself off.

turnoff

## Description

Enables an instrument to turn itself off.

## Syntax

**turnoff**

## Performance

*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.

## Examples

The following example uses the turnoff opcode. It will cause a note to terminate when a control signal passes a certain threshold (here the Nyquist frequency). It uses the files *turnoff.orc* [examples/turnoff.orc] and *turnoff.sco* [examples/turnoff.sco].

### Example 394. Example of the turnoff opcode.

```
/* turnoff.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  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 10000, k1, 1
  out a1
endin
/* turnoff.orc */
```

```
/* turnoff.sco */
; Table #1: an ordinary sine wave.
f 1 0 32768 10 1

; Play Instrument #1 for 4 seconds.
i 1 0 4
e
```

```
/* turnoff.sco */
```

## See Also

*ihold*



# turnoff2

turnoff2 -- Turn off instance(s) of other instruments at performance time.

turnoff2

## Description

Turn off instance(s) of other instruments at performance time.

## Syntax

**turnoff2** kinsno, kmode, krelease

## Performance

*kinsno* -- instrument to be turned off (can be fractional) if zero or negative, no instrument is turned off

*kmode* -- sum of the following values:

- 0, 1, or 2: turn off all instances (0), oldest only (1), or newest only (2)
- 4: only turn off notes with exactly matching (fractional) instrument number, rather than ignoring fractional part
- 8: only turn off notes with indefinite duration ( $p3 < 0$  or MIDI)

*krelease* -- if non-zero, the turned off instances are allowed to release, otherwise are deactivated immediately (possibly resulting in clicks)

## See Also

*turnoff*

## Credits

Author: Istvan Varga  
2005

# turnon

turnon -- Activate an instrument for an indefinite time.

turnon

## Description

Activate an instrument for an indefinite time.

## Syntax

```
turnon insnum [, itime]
```

## Initialization

*insnum* -- instrument number to be activated

*itime* (optional, default=0) -- 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*.)

# unirand

unirand -- Uniform distribution random number generator (positive values only).

unirand

## Description

Uniform distribution random number generator (positive values only). This is an x-class noise generator.

## Syntax

ares **unirand** krange

ires **unirand** krange

kres **unirand** krange

## Performance

*krange* -- the range of the random numbers (0 - *krange*).

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the unirand opcode. It uses the files *unirand.orc* [examples/unirand.orc] and *unirand.sco* [examples/unirand.sco].

### Example 395. Example of the unirand opcode.

```
/* unirand.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number between 0 and 1.
  ; krange = 1

  il unirand 1

  print il
endin
```

```
/* unirand.orc */

/* unirand.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* unirand.sco */
```

Its output should include lines like this:

```
instr 1:  i1 = 0.840
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, linrand, pcauchy, poisson, trirand, weibull*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# upsamp

upsamp -- Modify a signal by up-sampling.

upsamp

## Description

Modify a signal by up-sampling.

## Syntax

ares **upsamp** ksig

## Performance

*upsamp* converts a control signal to an audio signal. It does it by simple repetition of the kval. *upsamp* is a slightly more efficient form of the assignment, *asig = ksig*.

## Examples

```
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
```

## See Also

*diff*, *downsamp*, *integ*, *interp*, *samphold*

# urd

urd -- A discrete user-defined-distribution random generator that can be used as a function.

urd

## Description

A discrete user-defined-distribution random generator that can be used as a function.

## Syntax

```
aout = urd(ktableNum)
```

```
iout = urd(itableNum)
```

```
kout = urd(ktableNum)
```

## Initialization

*itableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

## Performance

*ktableNum* -- number of table containing the random-distribution function. Such table is generated by the user. See GEN40, GEN41, and GEN42. The table length does not need to be a power of 2

*urd* is the same opcode as *duserrnd*, but can be used in function fashion.

For a tutorial about random distribution histograms and functions see:

- D. Lorrain. "A panoply of stochastic cannons". In C. Roads, ed. 1989. Music machine. Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## See Also

*cuserrnd*, *duserrnd*

## Credits

Author: Gabriel Maldonado

New in Version 4.16

# vadd

vadd -- Performs addition between a vectorial control signal and a scalar control signal

vadd

## Description

Performs addition between a vectorial control signal and a scalar control signal

## Syntax

```
vadd ifn, kval, ielements
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ielements* - number of elements of the vector

## Performance

*kval* - scalar operand to be processed

*vadd* adds *kval* operand to each element of the vector contained in the table *ifn*.

These opcodes (*vadd*, *vmult*, *vpow*, *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* opcode to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vaddv

vaddv -- Performs addition between two vectorial control signals

vaddv

## Description

Performs addition between two vectorial control signals

## Syntax

**vaddv** ifn1, ifn2, ielements

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

## Performance

*vaddv* adds two vectorial control signals, that is, each element of the first vector is processed (only) with the corresponding element of the other vector. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The Result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy* opcode to copy it in another table.

This opcode works at k-rate.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)



# valpass

valpass -- Variably reverberates an input signal with a flat frequency response.

valpass

## Description

Variably reverberates an input signal with a flat frequency response.

## Syntax

```
ares valpass asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]
```

## Initialization

*imaxlpt* -- maximum loop time for *klpt*

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

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*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).

*xlpt* -- variable loop time in seconds, same as *ilpt* in *comb*. Loop time can be as large as *imaxlpt*.

This filter reiterates 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). Its output will begin to appear immediately.

## See Also

*alpass*, *comb*, *reverb*, *vcomb*

## Credits

Author: William “Pete” Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002

# vbap16

vbap16 -- Distributes an audio signal among 16 channels.

vbap16

## Description

Distributes an audio signal among 16 channels.

## Syntax

ar1, ..., ar16 **vbap16** asig, iazim [, ielev] [, ispread]

## Initialization

*iazim* -- azimuth angle of the virtual source

*ielev* (optional) -- elevation angle of the virtual source

*ispread* (optional) -- 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.

## Performance

*asig* -- audio signal to be panned

*vbap16* takes an input signal, *asig*, and distribute it among 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.

## Examples

### Example 396. 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.

## See Also

*vbap16move, vbap4, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: 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 4.07

# vbap16move

vbap16move -- Distribute an audio signal among 16 channels with moving virtual sources.

vbap16move

## Description

Distribute an audio signal among 16 channels with moving virtual sources.

## Syntax

```
ar1, ..., ar16 vbap16move asig, idur,  
    ispread, ifldnum, ifld1 [, ifld2] [...]
```

## Initialization

*idur* -- the duration over which the movement takes place.

*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, ...* -- azimuth angles or angular velocities, and relative durations of movement phases.

## Performance

*asig* -- audio signal to be panned

*vbap16move* allows 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.

## Examples

### Example 397. 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.

## See Also

*vbap16, vbap4, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: 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 4.07

# vbap4

vbap4 -- Distributes an audio signal among 4 channels.

vbap4

## Description

Distributes an audio signal among 4 channels.

## Syntax

ar1, ar2, ar3, ar4 **vbap4** asig, iazim [, ielev] [, ispread]

## Initialization

*iazim* -- azimuth angle of the virtual source

*ielev* (optional) -- elevation angle of the virtual source

*ispread* (optional) -- 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.

## Performance

*asig* -- audio signal to be panned

*vbap4* takes an input signal, *asig* and distributes it among 4 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.

## Examples

### Example 398. 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.

## See Also

*vbap16, vbap16move, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: 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 4.07

# vbap4move

vbap4move -- Distributes an audio signal among 4 channels with moving virtual sources.

vbap4move

## Description

Distributes an audio signal among 4 channels with moving virtual sources.

## Syntax

```
ar1, ar2, ar3, ar4 vbap4move asig,  
    idur, ispread, ifldnum, ifld1 [, ifld2] [...]
```

## Initialization

*idur* -- the duration over which the movement takes place.

*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*, ... -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*asig* -- audio signal to be panned

*vbap4move* allows 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.

## Examples

### Example 399. 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.

## See Also

*vbap16, vbap16move, vbap4, vbap8, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: 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 4.07

# vbap8

vbap8 -- Distributes an audio signal among 8 channels.

vbap8

## Description

Distributes an audio signal among 8 channels.

## Syntax

```
ar1, ..., ar8 vbap8 asig, iazim [, ielev] [, ispread]
```

## Initialization

*iazim* -- azimuth angle of the virtual source

*ielev* (optional) -- elevation angle of the virtual source

*ispread* (optional) -- 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.

## Performance

*asig* -- audio signal to be panned

*vbap8* takes an input signal, *asig*, and distributes it among 8 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.

## Examples

### Example 400. 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.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8move, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: 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 4.07

# vbap8move

vbap8move -- Distributes an audio signal among 8 channels with moving virtual sources.

vbap8move

## Description

Distributes an audio signal among 8 channels with moving virtual sources.

## Syntax

ar1, ..., ar8 **vbap8move** asig, idur, ispread, ifldnum, ifld1 [, ifld2] [...]

## Initialization

*idur* -- the duration over which the movement takes place.

*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*, ... -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*asig* -- audio signal to be panned

*vbap8move* allows 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.

## Examples

### Example 401. 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.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8, vbaplsinit, vbapz, vbapzmove*

## Credits

Author: 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 4.07

# vbaplsinit

vbaplsinit -- Configures VBAP output according to loudspeaker parameters.

vbaplsinit

## Description

Configures VBAP output according to loudspeaker parameters.

## Syntax

**vbaplsinit** *idim*, *ilsnum* [, *idir1*] [, *idir2*] [...] [, *idir32*]

## 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*, ..., *idir32* -- directions of loudspeakers. Number of directions must be less than or equal to 16. In two-dimensional loudspeaker positioning, *idirn* 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.).

## Performance

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.

## Examples

### Example 402. 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.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8, vbap8move, vbapz, vbapzmove*

## Credits

Author: 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 4.07

# vbapz

vbapz -- Writes a multi-channel audio signal to a ZAK array.

vbapz

## Description

Writes a multi-channel audio signal to a ZAK array.

## Syntax

**vbapz** inumchnls, istartndx, asig, iazim [, ielev] [, ispread]

## Initialization

*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

*iazim* -- azimuth angle of the virtual source

*ielev* (optional) -- elevation angle of the virtual source

*ispread* (optional) -- 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.

## Performance

*asig* -- audio signal to be panned

The opcode *vbapz* is the multiple channel analog of the opcodes like *vbap4*, working on *inumchnls* and using a ZAK array for output.

## Examples

### Example 403. 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.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8, vbap8move, vbaplsinit, vbapzmove*

## Credits

John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# vbapzmove

vbapzmove -- Writes a multi-channel audio signal to a ZAK array with moving virtual sources.

vbapzmove

## Description

Writes a multi-channel audio signal to a ZAK array with moving virtual sources.

## Syntax

**vbapzmove** inumchnls, istartndx, asig, idur, ispread, ifldnum, ifld1, ifld2, [...]

## Initialization

*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

*idur* -- the duration over which the movement takes place.

*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, ...* -- azimuth angles or angular velocities, and relative durations of movement phases (see below).

## Performance

*asig* -- audio signal to be panned

The opcode *vbapzmove* is the multiple channel analog of the opcodes like *vbap4move*, working on *inumchnls* and using a ZAK array for output.

## Examples

### Example 404. 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.

## See Also

*vbap16, vbap16move, vbap4, vbap4move, vbap8, vbap8move, vbaplsinit, vbapz*

## Credits

John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
May 2000

New in Csound Version 4.07

# vcella

vcella -- Cellular Automata

vcella

## Description

Unidimensional Cellular Automata applied to Csound vectors

## Syntax

**vcella** ifn1, ifn2, ielements

## Initialization

*ioutFunc* - number of the table where the state of each cell is stored

*initStateFunc* - number of a table containig the inital states of each cell

*iRuleFunc* - number of a lookup table containing the rules

*ielements* - total number of cells

*irulelen* - total number of rules

*iradius* (optional) - radius of Cellular Automata. At present time CA radius can be 1 or 2 (1 is the default)

## Performance

*ktrig* - trigger signal. Each time it is non-zero, a new generation of cells is evaluated

*kreinit* - trigger signal. Each time it is non-zero, state of all cells is forced to be that of *initStateFunc*.

*vcella* supports unidimensional cellular automata, where the state of each cell is stored in *ioutFunc*. So *ioutFunc* is a vector containing current state of each cell. This variant vector can be used together with any other vector-based opcode, such as *adsynt*, *vmap*, *vpowv* etc.

*initStateFunc* is an input vector containing the inital value of the row of cells, while *iRuleFunc* is an input vector containing the rules in the form of a lookup table. Notice that *initStateFunc* and *iRuleFunc* can be updated during the performance by means of other vector-based opcodes (for example *vcopy*) in order to force to change rules and status at performance time.

A new generation of cells is evaluated each time *ktrig* contains a non-zero value. Also the status of all cells can be forced to assume the status corresponding to the contents of *initStateFunc* each time *kreinit* contains a non-zero value.

Radius of CA algorithm can be 1 or 2 (optional *iradius* arguement).

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

## VCO

*vco* -- Implementation of a band limited, analog modeled oscillator.

*vco*

## 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.

## Syntax

```
ares vco xamp, xcps, iwave, kpw [, ifn]
      [, imaxd] [, ileak] [, inyx] [, iphs] [, iskip]
```

## Initialization

*iwave* -- determines the waveform:

- *iwave* = 1 - sawtooth
- *iwave* = 2 - Square/PWM
- *iwave* = 3 - triangle/Saw/Ramp

*ifn* (optional, default = 1) -- should be the table number of a of a stored sine wave. Must point to a valid table which contains a sine wave. Csound will report an error if this parameter is not set and table 1 doesn't exist.

*imaxd* (optional, default = 1) -- is the maximum delay time. A time of 1/1000 may be required for the pwm and triangle waveform. To bend the pitch down this value must be as large as 1/(minimum frequency).

*ileak* (optional, default = 0) -- If *ileak* is between zero and one ( $0 < \text{ileak} < 1$ ) then *ileak* is used as the leaky integrator value. Otherwise a leaky integrator value of .999 is used for the saw and square waves and .995 is used for the triangle wave. This can be used to “flatten” the square wave or “straighten” the saw wave at low frequencies by setting *ileak* to .99999 or a similar value. This should give a hollow sounding square wave.

*inyx* (optional, default = .5) -- This is used to determine the number of harmonics in the band limited pulse. All overtones up to  $\text{sr} * \text{inyx}$  will be used. The default gives  $\text{sr} * .5$  ( $\text{sr} / 2$ ). For  $\text{sr} / 4$  use *inyx* = .25. This can generate a “fatter” sound in some cases.

*iphs* (optional, default = 0) -- This is a phase value. There is an artifact (bug-like feature) in *vco* which occurs during the first half cycle of the square wave which causes the waveform to be greater in magnitude than all others. The value of *iphs* has an effect on this artifact. In particular setting *iphs* to .5 will cause the first half cycle of the square wave to resemble a small triangle wave. This may be more desirable than the large wave artifact which is the current default.

*iskip* (optional, default=0) -- if non zero skip the initialisation of the filter. (New in Csound version 4.23f13 and 5.0)

## Performance

*kpw* -- determines either the pulse width (if *iwave* is 2) or the saw/ramp character (if *iwave* is 3) The

value of *kpw* should be greater than 0 and less than 1. A value of 0.5 will generate either a square wave (if *iwave* is 2) or a triangle wave (if *iwave* is 3).

*xamp* -- determines the amplitude

*xcps* -- is the frequency of the wave in cycles per second.

## Examples

Here is an example of the *vco* opcode. It uses the files *vco.orc* [examples/vco.orc] and *vco.sco* [examples/vco.sco].

### Example 405. Example of the *vco* opcode.

```
/* vco.orc */
; Initialize the global variables.
sr = 44100
kr = 44100
ksmps = 1
nchnls = 1

; Instrument #1
instr 1
  ; Set the amplitude.
  kamp = p4

  ; Set the frequency.
  kcps = cpspch(p5)

  ; Select the wave form.
  iwave = p6

  ; Set the pulse-width/saw-ramp character.
  kpw init 0.5

  ; Use Table #1.
  ifn = 1

  ; Generate the waveform.
  asig vco kamp, kcps, iwave, kpw, ifn

  ; Output and amplification.
  out asig
endin
/* vco.orc */

/* vco.sco */
; Table #1, a sine wave.
f 1 0 65536 10 1

; Define the score.
; p4 = raw amplitude (0-32767)
; p5 = frequency, in pitch-class notation.
; p6 = the waveform (1=Saw, 2=Square/PWM, 3=Tri/Saw-Ramp-Mod)
i 1 00 02 20000 05.00 1
i 1 02 02 20000 05.00 2
i 1 04 02 20000 05.00 3

i 1 06 02 20000 07.00 1
i 1 08 02 20000 07.00 2
i 1 10 02 20000 07.00 3
```

```
i 1 12 02 20000 09.00 1
i 1 14 02 20000 09.00 2
i 1 16 02 20000 09.00 3

i 1 18 02 20000 11.00 1
i 1 20 02 20000 11.00 2
i 1 22 02 20000 11.00 3
e
/* vco.sco */
```

## See Also

*vco2*

## Credits

Author: Hans Mikelson  
December 1998

New in Csound version 3.50

November 2002. Corrected the documentation for the *kpw* parameter thanks to Luis Jure and Hans Mikelson.

## vco2

vco2 -- Implementation of a band-limited oscillator using pre-calculated tables.

vco2

### Description

*vco2* is similar to *vco*. But the implementation uses pre-calculated tables of band-limited waveforms (see also *GEN30*) rather than integrating impulses. This opcode can be faster than *vco* (especially if a low control-rate is used) and also allows better sound quality. Additionally, there are more waveforms and oscillator phase can be modulated at k-rate. The disadvantage is increased memory usage. For more details about *vco2* tables, see also *vco2init* and *vco2ft*.

### Syntax

ares **vco2** kamp, kcps [, imode] [, kpw] [, kphs] [, inyx]

### Initialization

*imode* (optional, default=0) -- a sum of values representing the waveform and its control values.

One may use any of the following values for *imode*:

- 16: enable k-rate phase control (if set, *kphs* is a required k-rate parameter that allows phase modulation)
- 1: skip initialization

One may use exactly one of these *imode* values to select the waveform to be generated:

- 14: user defined waveform -1 (requires using the *vco2init* opcode)
- 12: triangle (no ramp, faster)
- 10: square wave (no PWM, faster)
- 8:  $4 * x * (1 - x)$  (i.e. integrated sawtooth)
- 6: pulse (not normalized)
- 4: sawtooth / triangle / ramp
- 2: square / PWM
- 0: sawtooth

The default value for *imode* is zero, which means a sawtooth wave with no k-rate phase control.

*inyx* (optional, default=0.5) -- bandwidth of the generated waveform, as percentage (0 to 1) of the sample rate. The expected range is 0 to 0.5 (i.e. up to  $sr/2$ ), other values are limited to the allowed range.

Setting *inyx* to 0.25 ( $sr/4$ ), or 0.3333 ( $sr/3$ ) can produce a “fatter” sound in some cases, although it is more likely to reduce quality.



## Performance

*ares* -- the output audio signal.

*kamp* -- amplitude scale. In the case of a *imode* waveform value of 6 (a pulse waveform), the actual output level can be a lot higher than this value.

*kcps* -- frequency in Hz (should be in the range  $-sr/2$  to  $sr/2$ ).

*kpw* (optional) -- the pulse width of the square wave (*imode* waveform=2) or the ramp characteristics of the triangle wave (*imode* waveform=4). It is required only by these waveforms and ignored in all other cases. The expected range is 0 to 1, any other value is wrapped to the allowed range.



### Warning

*kpw* must not be an exact integer value (e.g. 0 or 1) if a sawtooth / triangle ramp (*imode* waveform=4) is generated. In this case, the recommended range is about 0.01 to 0.99). There is no such limitation for a square/PWM waveform.

*kphs* (optional) -- oscillator phase (depending on *imode*, this can be either an optional i-rate parameter that defaults to zero or required k-rate). Similarly to *kpw*, the expected range is 0 to 1.



### Note

When a low control-rate is used, pulse width (*kpw*) and phase (*kphs*) modulation is internally converted to frequency modulation. This allows for faster processing and reduced artifacts. But in the case of very long notes and continuous fast changes in *kpw* or *kphs*, the phase may drift away from the requested value. In most cases, the phase error is at most 0.037 per hour (assuming a sample rate of 44100 Hz).

This is a problem mainly in the case of pulse width (*kpw*), where it may result in various artifacts. While future releases of *vco2* may fix such errors, the following workarounds may also be of some help:

- Use *kpw* values only in the range 0.05 to 0.95. (There are more artifacts around integer values)
- Try to avoid modulating *kpw* by asymmetrical waveforms like a sawtooth wave. Relatively slow ( $\leq 20$  Hz) symmetrical modulation (e.g. sine or triangle), random splines (also slow), or a fixed pulse width is a lot less likely to cause synchronization problems.
- In some cases, adding random jitter (for example: random splines with an amplitude of about 0.01) to *kpw* may also fix the problem.

## Examples

Here is an example of the *vco2* opcode. It uses the files *vco2.orc* [examples/*vco2.orc*] and *vco2.sco* [examples/*vco2.sco*].

### Example 406. Example of the *vco2* opcode.

```
/* vco2.orc */
sr      = 44100
ksmps  = 10
nchnls = 1
```

```
; user defined waveform -1: trapezoid wave with default parameters (can be
; accessed at ftables starting from 10000)
itmp      ftgen 1, 0, 16384, 7, 0, 2048, 1, 4096, 1, 4096, -1, 4096, -1, 2048, 0
ift       vco2init -1, 10000, 0, 0, 0, 1
; user defined waveform -2: fixed table size (4096), number of partials
; multiplier is 1.02 (~238 tables)
itmp      ftgen 2, 0, 16384, 7, 1, 4095, 1, 1, -1, 4095, -1, 1, 0, 8192, 0
ift       vco2init -2, ift, 1.02, 4096, 4096, 2

instr 1
kcps      expon p4, p3, p5                ; instr 1: basic vco2 example
a1        vco2 12000, kcps                ; (sawtooth wave with default
out a1                                         ; parameters)
endin

instr 2
kcps      expon p4, p3, p5                ; instr 2:
kpw       linseg 0.1, p3/2, 0.9, p3/2, 0.1 ; PWM example
a1        vco2 10000, kcps, 2, kpw
out a1
endin

instr 3
kcps      expon p4, p3, p5                ; instr 3: vco2 with user
a1        vco2 14000, kcps, 14            ; defined waveform (-1)
aenv      linseg 1, p3 - 0.1, 1, 0.1, 0   ; de-click envelope
out a1 * aenv
endin

instr 4
kcps      expon p4, p3, p5                ; instr 4: vco2ft example,
kfn       vco2ft kcps, -2, 0.25           ; with user defined waveform
a1        oscilikt 12000, kcps, kfn       ; (-2), and sr/4 bandwidth
out a1
endin
/* vco2.orc */

/* vco2.sco */
i 1 0 3 20 2000
i 2 4 2 200 400
i 3 7 3 400 20
i 4 11 2 100 200

f 0 14

e
/* vco2.sco */
```

## See Also

*vco*, *vco2ft*, *vco2ift*, and *vco2init*.

## Credits

Author: Istvan Varga

New in version 4.22

## vco2ft

*vco2ft* -- Returns a table number at k-time for a given oscillator frequency and waveform.

*vco2ft*

## Description

*vco2ft* returns the function table number to be used for generating the specified waveform at a given frequency. This function table number can be used by any Csound opcode that generates a signal by reading function tables (like *oscilikt*). The tables must be calculated by *vco2init* before *vco2ft* is called and shared as Csound ftables (*ibasfn*).

## Syntax

kfn **vco2ft** kcps, iwave [, inyx]

## Initialization

*iwave* -- the waveform for which table number is to be selected. Allowed values are:

- 0: sawtooth
- 1:  $4 * x * (1 - x)$  (integrated sawtooth)
- 2: pulse (not normalized)
- 3: square wave
- 4: triangle

Additionally, negative *iwave* values select user defined waveforms (see also *vco2init*).

*inyx* (optional, default=0.5) -- bandwidth of the generated waveform, as percentage (0 to 1) of the sample rate. The expected range is 0 to 0.5 (i.e. up to  $sr/2$ ), other values are limited to the allowed range.

Setting *inyx* to 0.25 ( $sr/4$ ), or 0.3333 ( $sr/3$ ) can produce a “fatter” sound in some cases, although it is more likely to reduce quality.

## Performance

*kfn* -- the ftable number, returned at k-rate.

*kcps* -- frequency in Hz, returned at k-rate. Zero and negative values are allowed. However, if the absolute value exceeds  $sr/2$  (or  $sr*inyx$ ), the selected table will contain silence.

## Examples

See the example for the *vco2* opcode.

## See Also

*vco2ift*, *vco2init*, and *vco2*.

## Credits

Author: Istvan Varga

New in version 4.22

## vco2ift

*vco2ift* -- Returns a table number at i-time for a given oscillator frequency and waveform.

*vco2ift*

## Description

*vco2ift* is the same as *vco2ft*, but works at i-time. It is suitable for use with opcodes that expect an i-rate table number (for example, *oscili*).

## Syntax

```
ifn vco2ift icps, iwave [, inyx]
```

## Initialization

*ifn* -- the ftable number.

*icps* -- frequency in Hz. Zero and negative values are allowed. However, if the absolute value exceeds *sr/2* (or *sr\*inyx*), the selected table will contain silence.

*iwave* -- the waveform for which table number is to be selected. Allowed values are:

- 0: sawtooth
- 1:  $4 * x * (1 - x)$  (integrated sawtooth)
- 2: pulse (not normalized)
- 3: square wave
- 4: triangle

Additionally, negative *iwave* values select user defined waveforms (see also *vco2init*).

*inyx* (optional, default=0.5) -- bandwidth of the generated waveform, as percentage (0 to 1) of the sample rate. The expected range is 0 to 0.5 (i.e. up to *sr/2*), other values are limited to the allowed range.

Setting *inyx* to 0.25 (*sr/4*), or 0.3333 (*sr/3*) can produce a “fatter” sound in some cases, although it is more likely to reduce quality.

## See Also

*vco2ft*, *vco2init*, and *vco2*.

## Credits

Author: Istvan Varga

New in version 4.22

# vco2init

`vco2init` -- Calculates tables for use by `vco2` opcode.

`vco2init`

## Description

`vco2init` calculates tables for use by `vco2` opcode. Optionally, it is also possible to access these tables as standard Csound function tables. In this case, `vco2ft` can be used to find the correct table number for a given oscillator frequency.

In most cases, this opcode is called from the orchestra header. Using `vco2init` in instruments is possible but not recommended. This is because replacing tables during performance can result in a Csound crash if other opcodes are accessing the tables at the same time.

Note that `vco2init` is not required for `vco2` to work (tables are automatically allocated by the first `vco2` call, if not done yet), however it can be useful in some cases:

- Pre-calculate tables at orchestra load time. This is useful to avoid generating the tables during performance, which could interrupt real-time processing.
- Share the tables as Csound ftables. By default, the tables can be accessed only by `vco2`.
- Change the default parameters of tables (e.g. size) or use an user-defined waveform specified in a function table.

## Syntax

```
ifn vco2init iwave [, ibasfn] [, ipmul] [, iminsiz] [, imaxsiz] [, isrcft]
```

## Initialization

*ifn* -- the first free ftable number after the allocated tables. If *ibasfn* was not specified, -1 is returned.

*iwave* -- sum of the following values selecting which waveforms are to be calculated:

- 16: triangle
- 8: square wave
- 4: pulse (not normalized)
- 2:  $4 * x * (1 - x)$  (integrated sawtooth)
- 1: sawtooth

Alternatively, *iwave* can be set to a negative integer that selects an user-defined waveform. This also requires the *isrcft* parameter to be specified. `vco2` can access waveform number -1. However, other user-defined waveforms are usable only with `vco2ft` or `vco2ift`.

*ibasfn* (optional, default=-1) -- ftable number from which the table set(s) can be accessed by opcodes other than `vco2`. This is required by user defined waveforms, with the exception of -1. If this value is less than 1, it is not possible to access the tables calculated by `vco2init` as Csound function tables.

*ipmul* (optional, default=1.05) -- multiplier value for number of harmonic partials. If one table has *n* partials, the next one will have  $n * ipmul$  (at least  $n + 1$ ). The allowed range for *ipmul* is 1.01 to 2. Zero or negative values select the default (1.05).

*iminsiz* (optional, default=-1) -- minimum table size.

*imaxsiz* (optional, default=-1) -- maximum table size.

The actual table size is calculated by multiplying the square root of the number of harmonic partials by *iminsiz*, rounding up the result to the next power of two, and limiting this not to be greater than *imaxsiz*.

Both parameters, *iminsiz* and *imaxsiz*, must be power of two, and in the allowed range. The allowed range is 16 to 262144 for *iminsiz* to up to 16777216 for *imaxsiz*. Zero or negative values select the default settings:

- The minimum size is 128 for all waveforms except pulse (*iwave*=4). Its minimum size is 256.
- The default maximum size is usually the minimum size multiplied by 64, but not more than 16384 if possible. It is always at least the minimum size.

*isrcft* (optional, default=-1) -- source ftable number for user-defined waveforms (if *iwave* < 0). *isrcft* should point to a function table containing the waveform to be used for generating the table array. The table size is recommended to be at least *imaxsiz* points. If *iwave* is not negative (built-in waveforms are used), *isrcft* is ignored.



## Warning

The number and size of tables is not fixed. Orchestras should not depend on these parameters, as they are subject to changes between releases.

If the selected table set already exists, it is replaced. If any opcode is accessing the tables at the same time, it is very likely that a crash will occur. This is why it is recommended to use *vco2init* only in the orchestra header.

These tables should not be replaced/overwritten by GEN routines or the *ftgen* opcode. Otherwise, unpredictable behavior or a Csound crash may occur if *vco2* is used. The first free ftable after the table array(s) is returned in *ifn*.

## Examples

See the example for the *vco2* opcode.

## See Also

*vco2ft*, *vco2ift*, and *vco2*.

## Credits

Author: Istvan Varga

New in version 4.22

## vcomb

vcomb -- Variably reverberates an input signal with a “colored” frequency response.

vcomb

## Description

Variably reverberates an input signal with a “colored” frequency response.

## Syntax

```
ares vcomb asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]
```

## Initialization

*imaxlpt* -- maximum loop time for *klpt*

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

*insmps* (optional, default=0) -- delay amount, as a number of samples.

## Performance

*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).

*xlpt* -- variable loop time in seconds, same as *ilpt* in *comb*. Loop time can be as large as *imaxlpt*.

This filter reiterates 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 will appear only after *ilpt* seconds.

## See Also

*alpass*, *comb*, *reverb*, *valpass*

## Credits

Author: William “Pete” Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002



# vcopy

vcopy -- Copies between two vectorial control signals

vcopy

## Description

Copies between two vectorial control signals

## Syntax

**vcopy** ifn, ifn2, ielements

## Initialization

*ifn* - number of the table where the vectorial signal will be copied

*ifn* - number of the table hosting the vectorial signal to be copied

*ielements* - number of elements of the vector

## Performance

*vcopy* copies *ifn2* to *ifn1*. Useful to keep old vector values, by storing them in another table.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexp*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

*Note:* *bmscan* not yet available on Canonical Csound

## Examples

Here is an example of the *vcopy* opcode. It uses the files *vcopy.csd* [examples/vcopy.csd].

### Example 407. Example of the vcopy opcode.

```
<CsoundSynthesizer>
<CsOptions>
;use appropriate realtime flags
;-+rtaudio=jack -odac:alsa_pcm:playback_ -B256 -b256
</CsOptions>
<CsInstruments>

sr=44100
kr=4410
ksmps=10
nchnls=2

      instr 1 ;table playback
ar lposcil 1, 1, 0, 262144, 1
outs ar,ar
      endin
```

```
instr 2
vcopy 2, 1, 20000 ;copy vector from sample to empty table
vmult 5, 20000, 262144 ;scale noise to make it audible
vcopy 1, 5, 20000 ;put noise into sample
turnoff
endin

instr 3
vcopy 1, 2, 20000 ;put original information back in
turnoff
endin

</CsInstruments>
<CsScore>
f1 0 262144 -1 "beats.aiff" 0 4 0
f2 0 262144 2 0

f5 0 262144 21 3 30000

i1 0 4
i2 3 1

s
i1 0 4
i3 3 1
s

i1 0 4

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

## vcopy\_i

vcopy\_i -- Copies between two vectorial control signals.

vcopy\_i

## Description

Copies between two vectorial control signals

## Syntax

**vcopy\_i** ifn, ifn2, ielements

## Initialization

*ifn* - number of the table where the vectorial signal will be copied

*ifn* - number of the table hosting the vectorial signal to be copied

*ielements* - number of elements of the vector

## Performance

*vcopy* copies *ifn2* to *ifn1*. Useful to keep old vector values, by storing them in another table. This opcode is exactly the same as *vcopy* but performs all the copying on the initialization pass only.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexp*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

*Note:* *bmscan* not yet available on Canonical Csound

## Examples

Here is an example of the *vcopy* opcode. It uses the files *vcopy.csd* [examples/vcopy.csd].

### Example 408. Example of the vcopy opcode.

```
<CsoundSynthesizer>
<CsOptions>
;use appropriate realtime flags
;--rtaudio=jack -odac:alsa_pcm:playback_ -B256 -b256
</CsOptions>
<CsInstruments>

sr=44100
kr=4410
ksmps=10
nchnls=2

instr 1 ;table playback
ar lposcil 1, 1, 0, 262144, 1
outs ar,ar
endin
```

```
instr 2
vcopy 2, 1, 20000 ;copy vector from sample to empty table
vmult 5, 20000, 262144 ;scale noise to make it audible
vcopy 1, 5, 20000 ;put noise into sample
turnoff
endin

instr 3
vcopy 1, 2, 20000 ;put original information back in
turnoff
endin

</CsInstruments>
<CsScore>
f1 0 262144 -1 "beats.aiff" 0 4 0
f2 0 262144 2 0

f5 0 262144 21 3 30000

i1 0 4
i2 3 1

s
i1 0 4
i3 3 1
s

i1 0 4

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vdelay

`vdelay` -- An interpolating variable time delay.

`vdelay`

## Description

This is an interpolating variable time delay, it is not very different from the existing implementation (*deltapi*), it is only easier to use.

## Syntax

ares **vdelay** asig, adel, imaxdel [, iskip]

## 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.

## Examples

```
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.

## See Also

*vdelay3*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

# vdelay3

*vdelay3* -- An variable time delay with cubic interpolation.

*vdelay3*

## Description

*vdelay3* is experimental. It is the same as *vdelay* except that it uses cubic interpolation. (New in Version 3.50.)

## Syntax

ares **vdelay3** asig, adel, imaxdel [, iskip]

## 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* (optional) -- 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.

## Examples

```
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.

## See Also

*vdelay*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995



# vdelayx

vdelayx -- A variable delay opcode with high quality interpolation.

vdelayx

## Description

A variable delay opcode with high quality interpolation.

## Syntax

*aout* **vdelayx** *ain*, *adl*, *imd*, *iws* [, *ist*]

## Initialization

*aout* -- output audio signal

*ain* -- input audio signal

*adl* -- delay time in seconds

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

This opcode uses high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.



### Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
where *a* is the output gain, and *dt* is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## See Also

*vdelayxq, vdelayxs, vdelayxw, vdelayxwq, vdelayxws*

# vdelayxq

vdelayxq -- A 4-channel variable delay opcode with high quality interpolation.

vdelayxq

## Description

A 4-channel variable delay opcode with high quality interpolation.

## Syntax

*aout1*, *aout2*, *aout3*, *aout4* **vdelayxq** *ain1*, *ain2*, *ain3*, *ain4*, *adl*, *imd*, *iws* [, *ist*]

## Initialization

*aout1*, *aout2*, *aout3*, *aout4* -- output audio signals.

*ain1*, *ain2*, *ain3*, *ain4* -- input audio signals.

*adl* -- delay time in seconds

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* (optional) -- skip initialization if not zero

## Performance

This opcode uses high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The multichannel opcodes (eg. *vdelayxq*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



## Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
  
where *a* is the output gain, and *dt* is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## See Also

*vdelayx, vdelayxs, vdelayxw, vdelayxwq, vdelayxws*

# vdelayxs

vdelayxs -- A stereo variable delay opcode with high quality interpolation.

vdelayxs

## Description

A stereo variable delay opcode with high quality interpolation.

## Syntax

```
aout1, aout2 vdelayxs ain1, ain2, adl, imd, iws [, ist]
```

## Initialization

*aout1, aout2* -- output audio signals

*ain1, ain2* -- input audio signals

*adl* -- delay time in seconds

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* -- skip initialization if not zero

## Performance

This opcode uses high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The multichannel opcodes (eg. *vdelayxq*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



## Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
where  $a$  is the output gain, and  $dt$  is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## See Also

*vdelayx, vdelayxq, vdelayxw, vdelayxwq, vdelayxws*

# vdelayxw

vdelayxw -- Variable delay opcodes with high quality interpolation.

vdelayxw

## Description

Variable delay opcodes with high quality interpolation.

## Syntax

*aout* **vdelayxw** *ain*, *adl*, *imd*, *iws* [, *ist*]

## Initialization

*aout* -- output audio signal

*ain* -- input audio signal

*adl* -- delay time in seconds

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* -- skip initialization if not zero

## Performance

These opcodes use high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The vdelayxw opcodes change the position of the write tap in the delay line (unlike all other delay ugens that move the read tap), and are most useful for implementing Doppler effects where the position of the listener is fixed, and the sound source is moving.



### Notes

- Delay time is measured in seconds (unlike in vdelay and vdelay3), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In vdelayxw\*, changing the delay time has some effects on output volume:  
$$a = 1 / (1 + dt)$$
where  $a$  is the output gain, and  $dt$  is the change of delay time per seconds.
- These opcodes are best used in the double-precision version of Csound.

## See Also

*vdelayx*, *vdelayxq*, *vdelayxs*, *vdelayxwq*, *vdelayxws*



# vdelayxwq

vdelayxwq -- Variable delay opcodes with high quality interpolation.

vdelayxwq

## Description

Variable delay opcodes with high quality interpolation.

## Syntax

aout1, aout2, aout3, aout4 **vdelayxwq** ain1, ain2, ain3, ain4, adl, imd, iws [, i

## Initialization

*ain1, ain2, ain3, ain4* -- input audio signals

*aout1, aout2, aout3, aout4* -- output audio signals

*adl* -- delay time in seconds

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* -- skip initialization if not zero

## Performance

These opcodes use high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The *vdelayxw* opcodes change the position of the write tap in the delay line (unlike all other delay ugens that move the read tap), and are most useful for implementing Doppler effects where the position of the listener is fixed, and the sound source is moving.

The multichannel opcodes (eg. *vdelayxq*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



## Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:

$$a = 1 / (1 + dt)$$

where *a* is the output gain, and *dt* is the change of delay time per seconds.

- These opcodes are best used in the double-precision version of Csound.

## See Also

*vdelayx, vdelayxq, vdelayxs, vdelayxw, vdelayxws*

# vdelayxws

vdelayxws -- Variable delay opcodes with high quality interpolation.

vdelayxws

## Description

Variable delay opcodes with high quality interpolation.

## Syntax

```
aout1, aout2 vdelayxws ain1, ain2, adl, imd, iws [, ist]
```

## Initialization

*ain1, ain2* -- input audio signals

*aout1, aout2* -- output audio signals

*adl* -- delay time in seconds

*imd* -- max. delay time (seconds)

*iws* -- interpolation window size (see below)

*ist* -- skip initialization if not zero

## Performance

These opcodes use high quality (and slow) interpolation, that is much more accurate than the currently available linear and cubic interpolation. The *iws* parameter sets the number of input samples used for calculating one output sample (allowed values are any integer multiply of 4 in the range 4 - 1024); higher values mean better quality and slower speed.

The *vdelayxw* opcodes change the position of the write tap in the delay line (unlike all other delay ugens that move the read tap), and are most useful for implementing Doppler effects where the position of the listener is fixed, and the sound source is moving.

The multichannel opcodes (eg. *vdelayx*) allow delaying 2 or 4 variables at once (stereo or quad signals); this is much more efficient than using separate opcodes for each channel.



## Notes

- Delay time is measured in seconds (unlike in *vdelay* and *vdelay3*), and must be a-rate.
- The minimum allowed delay is  $iws/2$  samples.
- Using the same variables as input and output is allowed in these opcodes.
- In *vdelayxw\**, changing the delay time has some effects on output volume:

$$a = 1 / (1 + dt)$$

where *a* is the output gain, and *dt* is the change of delay time per seconds.

- These opcodes are best used in the double-precision version of Csound.

## See Also

*vdelayx, vdelayxq, vdelayxs, vdelayxw, vdelayxwq*

## vdivv

vdivv -- Performs division between two vectorial control signals

vdivv

## Description

Performs division between two vectorial control signals

## Syntax

**vdivv** ifn1, ifn2, ielements

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

## Performance

*vdivv* divides two vectorial control signals, that is, each element of *ifn1* is divided by the corresponding element of *ifn2*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy* opcode to copy it in another table.

This opcode works at k-rate.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# **vdelayk**

vdelayk -- k-rate variable time delay.

vdelayk

## **Description**

Variable delay applied to a k-rate signal

## **Syntax**

```
kout vdelayk  iksig, kdel, imaxdel [, iskip, imode]
```

## **Initialization**

*imaxdel* - maximum value of delay in seconds.

*iskip* (optional) - Skip initialization if present and non zero.

*imode* (optional) - if non-zero it suppresses linear interpolation. While, normally, interpolation increases the quality of a signal, it should be suppressed if using vdelay with discrete control signals, such as, for example, trigger signals.

## **Performance**

*kout* - delayed output signal

*ksig* - input signal

*kdel* - delay time in seconds can be varied at k-rate

*vdelayk* is similar to *vdelay*, but works at k-rate. It is designed to delay control signals, to be used, for example, in algorithmic composition.

## **Credits**

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vecdelay

vecdelay -- Vectorial Control-rate Delay Paths

vecdelay

## Description

Generate a sort of 'vectorial' delay

## Syntax

```
vecdelay ifn, ifnIn, ifnDel, ielements, imaxdel [, iskip]
```

## Initialization

*ifn* - number of the table containing the output vector

*ifnIn* - number of the table containing the input vector

*ifnDel* - number of the table containing a vector whose elements contain delay values in seconds

*ielements* - number of elements of the two vectors

*imaxdel* - Maximum value of delay in seconds.

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

## Performance

*vecdelay* is similar to *vdelay*, but it works at k-rate and, instead of delaying a single signal, it delays a vector. *ifnIn* is the input vector of signals, *ifn* is the output vector of signals, and *ifnDel* is a vector containing delay times for each element, expressed in seconds. Elements of *ifnDel* can be updated at k-rate. Each single delay can be different from that of the other elements, and can vary at k-rate. *imaxdel* sets the maximum delay allowed for all elements of *ifnDel*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# veloc

veloc -- Get the velocity from a MIDI event.

veloc

## Description

Get the velocity from a MIDI event.

## Syntax

ival **veloc** [ilow] [, ihigh]

## Initialization

*ilow*, *ihigh* -- low and hi ranges for mapping

## Performance

Get the MIDI byte value (0 - 127) denoting the velocity of the current event.

## Examples

Here is an example of the veloc opcode. It uses the files *veloc.orc* [examples/veloc.orc] and *veloc.sco* [examples/veloc.sco].

### Example 409. Example of the veloc opcode.

```
/* veloc.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  il veloc

  print il
endin
/* veloc.orc */

/* veloc.sco */
; Play Instrument #1 for 12 seconds.
i 1 0 12
e
/* veloc.sco */
```



## See Also

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

## Credits

Author: Barry L. Vercoe - Mike Berry  
MIT - Mills  
May 1997

Example written by Kevin Conder.

## vexp

vexp -- Performs power-of operations between a vectorial control signal and a scalar control signal

vexp

## Description

Performs power-of operations between a vectorial control signal and a scalar control signal

## Syntax

**vexp** ifn, kval, ielements

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ielements* - number of elements of the vector

## Performance

*kval* - scalar operand to be processed

*vexp* rises *kval* to each element contained in the table *ifn*.

These opcodes (*vadd*, *vmult*, *vpow*, *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* opcode to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vexpseg

vexpseg -- Vectorial envelope generator

vexpseg

## Description

Generate exponential vectorial segments

## Syntax

```
vexpseg    ifnout, ielements, ifn1, idur1, ifn2 [, idur2, ifn3 [...]]
```

## Initialization

*ifnout* - number of table hosting output vectorial signal

*ifn1* - starting vector

*ifn2, ifn3, etc.* - vector after idurx seconds

*idur1* - duration in seconds of first segment.

*dur2, idur3, etc.* - duration in seconds of subsequent segments.

*ielements* - number of elements of vectors.

## Performance

These opcodes are similar to linseg and expseg, but operate with vectorial signals instead of with scalar signals.

Output is a vectorial control signal hosted by ifnout (that must be previously allocated), while each break-point of the envelope is actually a vector of values. All break-points must contain the same number of elements (ielements).

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as bmscan, vcella, adsynt, adsynt2 etc.

## Example

Here is an example of the vexpseg opcode. It uses the files *vexpseg.csd* [examples/vexpseg.csd].

### Example 410. Example of the vexpseg opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
-odac -B441 -b441  
</CsOptions>  
<CsInstruments>
```

```
sr=44100  
ksmps=10  
nchnls=2
```

```
gilen init 32

gitable1 ftgen 0, 0, gilen, 10, 1
gitable2 ftgen 0, 0, gilen, 10, 1

gitable3 ftgen 0, 0, gilen, -7, 30, gilen, 35
gitable4 ftgen 0, 0, gilen, -7, 400, gilen, 450
gitable5 ftgen 0, 0, gilen, -7, 5000, gilen, 5500

instr 1
vcopy gitable2, gitable1, gilen
turnoff
endin

instr 2
vexpseg gitable2, 16, gitable3, 2, gitable4, 2, gitable5
endin

instr 3
kcount init 0
if kcount < 16 then
    kval table kcount, gitable2
    printk 0,kval
    kcount = kcount +1
else
    turnoff
endif

endin

</CsInstruments>
<CsScore>
i1 0 1
s
i2 0 10
i3 0 1
i3 1 1
i3 1.5 1
i3 2 1
i3 2.5 1
i3 3 1
i3 3.5 1
i3 4 1
i3 4.5 1

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

## vexpv

vexpv -- Performs exponential operations between two vectorial control signals

vexpv

## Description

Performs exponential operations between two vectorial control signals

## Syntax

**vexpv** ifn1, ifn2, ielements

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

## Performance

*vexpv* divides two vectorial control signals, that is, each element of *ifn2* is risen to the corresponding element of *ifn1*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy* opcode to copy it in another table.

This opcode works at k-rate.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vibes

vibes -- Physical model related to the striking of a metal block.

vibes

## Description

Audio output is a tone related to the striking of a metal block as found in a vibraphone. The method is a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

ares **vibes** kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec

## 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* [examples/marmstk1.wav] is a suitable function from measurements and can be loaded with a *GENO1* table. It is also available at <ftp://ftp.cs.bath.ac.uk/pub/dream/documentation/sounds/modelling/>.

*ivfn* -- shape of vibrato, usually a sine table, created by a function

*idec* -- time before end of note when damping is introduced

*idoubles* (optional) -- percentage of double strikes. Default is 40%.

*itriples* (optional) -- 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

## Examples

Here is an example of the vibes opcode. It uses the files *vibes.orc* [examples/vibes.orc], *vibes.sco* [examples/vibes.sco], and *marmstk1.wav* [examples/marmstk1.wav].

### Example 411. Example of the vibes opcode.

```
/* vibes.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
```

```
nchnls = 1

; Instrument #1.
instr 1
  ; kamp = 20000
  ; kfreq = 440
  ; ihrd = 0.5
  ; ipos = 0.561
  ; imp = 1
  ; kvibf = 6.0
  ; kvamp = 0.05
  ; ivibfn = 2
  ; idec = 0.1

  al vibes 20000, 440, 0.5, 0.561, 1, 6.0, 0.05, 2, 0.1

  out a1
endin
/* vibes.orc */

/* vibes.sco */
; Table #1, the "marmstk1.wav" audio file.
f 1 0 256 1 "marmstk1.wav" 0 0 0
; Table #2, a sine wave for the vibrato.
f 2 0 128 10 1

; Play Instrument #1 for four seconds.
i 1 0 4
e
/* vibes.sco */
```

## See Also

*marimba*

## Credits

Author: John fitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# vibr

vibr -- Easier-to-use user-controllable vibrato.

vibr

## Description

Easier-to-use user-controllable vibrato.

## Syntax

kout **vibr** kAverageAmp, kAverageFreq, ifn

## Initialization

*ifn* -- Number of vibrato table. It normally contains a sine or a triangle wave.

## Performance

*kAverageAmp* -- Average amplitude value of vibrato

*kAverageFreq* -- Average frequency value of vibrato (in cps)

*vibr* is an easier-to-use version of *vibrato*. It has the same generation-engine of *vibrato*, but the parameters corresponding to missing input arguments are hard-coded to default values.

## Examples

Here is an example of the vibr opcode. It uses the files *vibr.orc* [examples/vibr.orc] and *vibr.sco* [examples/vibr.sco].

### Example 412. Example of the vibr opcode.

```
/* vibr.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Create a vibrato waveform.
  kaverageamp init 7500
  kaveragefreq init 5
  ifn = 1
  kvamp vibr kaverageamp, kaveragefreq, ifn

  ; Generate a tone including the vibrato.
  a1 oscili 10000+kvamp, 440, 2

  out a1
endin
/* vibr.orc */
```



```
/* vibr.sco */
; Table #1, a sine wave for the vibrato.
f 1 0 256 10 1
; Table #1, a sine wave for the oscillator.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* vibr.sco */
```

## See Also

*jitter, jitter2, vibrato*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Version 4.15

# vibrato

vibrato -- Generates a natural-sounding user-controllable vibrato.

vibrato

## Description

Generates a natural-sounding user-controllable vibrato.

## Syntax

kout **vibrato** kAverageAmp, kAverageFreq, kRandAmountAmp, kRandAmountFreq, kAmpMi.

## Initialization

*ifn* -- Number of vibrato table. It normally contains a sine or a triangle wave.

*iphs* -- (optional) Initial phase of table, 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

*kAverageAmp* -- Average amplitude value of vibrato

*kAverageFreq* -- Average frequency value of vibrato (in cps)

*kRandAmountAmp* -- Amount of random amplitude deviation

*kRandAmountFreq* -- Amount of random frequency deviation

*kAmpMinRate* -- Minimum frequency of random amplitude deviation segments (in cps)

*kAmpMaxRate* -- Maximum frequency of random amplitude deviation segments (in cps)

*kcpsMinRate* -- Minimum frequency of random frequency deviation segments (in cps)

*kcpsMaxRate* -- Maximum frequency of random frequency deviation segments (in cps)

*vibrato* outputs a natural-sounding user-controllable vibrato. The concept is to randomly vary both frequency and amplitude of the oscillator generating the vibrato, in order to simulate the irregularities of a real vibrato.

In order to have a total control of these random variations, several input arguments are present. Random variations are obtained by two separated segmented lines, the first controlling amplitude deviations, the second the frequency deviations. Average duration of each segment of each line can be shortened or enlarged by the arguments *kAmpMinRate*, *kAmpMaxRate*, *kcpsMinRate*, *kcpsMaxRate*, and the deviation from the average amplitude and frequency values can be independently adjusted by means of *kRandAmountAmp* and *kRandAmountFreq*.

## Examples

Here is an example of the vibrato opcode. It uses the files *vibrato.orc* [examples/vibrato.orc] and *vibrato.sco* [examples/vibrato.sco].

### Example 413. Example of the vibrato opcode.

```
/* vibrato.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Create a vibrato waveform.
  kaverageamp init 2500
  kaveragefreq init 6
  krandamountamp init 0.3
  krandamountfreq init 0.5
  kampminrate init 3
  kampmaxrate init 5
  kcpsminrate init 3
  kcpsmaxrate init 5
  ifn = 1
  kvamp vibrato kaverageamp, kaveragefreq, krandamountamp, \
               krandamountfreq, kampminrate, kampmaxrate, \
               kcpsminrate, kcpsmaxrate, ifn

  ; Generate a tone including the vibrato.
  al oscili 10000+kvamp, 440, 2

  out a1
endin
/* vibrato.orc */

/* vibrato.sco */
; Table #1, a sine wave for the vibrato.
f 1 0 256 10 1
; Table #1, a sine wave for the oscillator.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* vibrato.sco */
```

## See Also

*jitter, jitter2, vibr*

## Credits

Author: Gabriel Maldonado

Example written by Kevin Conder.

New in Version 4.15

# vincr

vincr -- Accumulates audio signals.

vincr

## Description

*vincr* increments an audio variable of another signal, i.e. accumulates output.

## Syntax

**vincr** *asig*, *aincr*

## Performance

*asig* -- audio variable to be incremented

*aincr* -- incrementing signal

*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).

## Examples

See the *fout* opcode for an example.

## See Also

*clear*

## Credits

Author: Gabriel Maldonado  
Italy  
1999

New in Csound version 3.56

# vlimit

vlimit -- Limiting and Wrapping Vectorial Signals

vlimit

## Description

Limits elements of vectorial control signals.

## Syntax

```
vlimit ifn, kmin, kmax, ielements
```

## Initialization

*ifn* - number of the table hosting the vector to be processed

*ielements* - number of elements of the vector

## Performance

*kmin* - minimum threshold value

*kmax* - maximum threshold value

*vlimit* set lower and upper limits on each element of the vector they process.

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate with a vectorial signal instead of with a scalar signal.

Result overrides old values of *ifn1*, if these are out of min/max interval. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

All these opcodes are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

*Note:* *bmscan* not yet available on Canonical Csound

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vlinseg

vlinseg -- Vectorial envelope generator

vlinseg

## Description

Generate linear vectorial segments

## Syntax

```
vlinseg    ifnout, ielements, ifn1, idur1, ifn2 [, idur2, ifn3 [...]]
```

## Initialization

*ifnout* - number of table hosting output vectorial signal

*ifn1* - starting vector

*ifn2, ifn3, etc.* - vector after idurx seconds

*idur1* - duration in seconds of first segment.

*dur2, idur3, etc.* - duration in seconds of subsequent segments.

*ielements* - number of elements of vectors.

## Performance

These opcodes are similar to linseg and expseg, but operate with vectorial signals instead of with scalar signals.

Output is a vectorial control signal hosted by ifnout (that must be previously allocated), while each break-point of the envelope is actually a vector of values. All break-points must contain the same number of elements (ielements).

All these operators are designed to be used together with other opcodes that operate with vectorial signals such as bmscan, vcella, adsynt, adsynt2 etc.

## Example

Here is an example of the vlinseg opcode. It uses the files *vlinseg.csd* [examples/vlinseg.csd].

### Example 414. Example of the vlinseg opcode.

```
<CsoundSynthesizer>  
<CsOptions>  
-odac -B441 -b441  
</CsOptions>  
<CsInstruments>
```

```
sr=44100  
ksmps=10  
nchnls=2
```

```
gilen init 32

gitable1 ftgen 0, 0, gilen, 10, 1
gitable2 ftgen 0, 0, gilen, 10, 1

gitable3 ftgen 0, 0, gilen, -7, 30, gilen, 35
gitable4 ftgen 0, 0, gilen, -7, 400, gilen, 450
gitable5 ftgen 0, 0, gilen, -7, 5000, gilen, 5500

instr 1
vcopy gitable2, gitable1, gilen
turnoff
endin

instr 2
vlinseg gitable2, 16, gitable3, 2, gitable4, 2, gitable5
endin

instr 3
kcount init 0
if kcount < 16 then
    kval table kcount, gitable2
    printk 0,kval
    kcount = kcount +1
else
    turnoff
endif

endin

</CsInstruments>
<CsScore>
i1 0 1
s
i2 0 10
i3 0 1
i3 1 1
i3 1.5 1
i3 2 1
i3 2.5 1
i3 3 1
i3 3.5 1
i3 4 1
i3 4.5 1

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vlowres

vlowres -- A bank of filters in which the cutoff frequency can be separated under user control.

vlowres

## Description

A bank of filters in which the cutoff frequency can be separated under user control

## Syntax

ares **vlowres** asig, kfco, kres, iord, ksep

## 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 *kfco* and *ksep* parameters.

## Examples

Here is an example of the vlowres opcode. It uses the files *vlowres.orc* [examples/vlowres.orc], *vlowres.sco* [examples/vlowres.sco], and *beats.wav* [examples/beats.wav].

### Example 415. Example of the vlowres opcode.

```
/* vlowres.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Use a nice sawtooth waveform.
  asig vco 32000, 220, 1

  ; Vary the cutoff frequency from 30 to 300 Hz.
  kfco line 30, p3, 300
  kres = 25
  iord = 2
  ksep = 20

  ; Apply the filters.
```



```
avlr vlowres asig, kfco, kres, iord, ksep  
  
; It gets loud, so clip the output amplitude to 30,000.  
al clip avlr, 1, 30000  
out al  
endin  
/* vlowres.orc */
```

```
/* vlowres.sco */  
; Table #1, a sine wave.  
f 1 0 16384 10 1  
  
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* vlowres.sco */
```

## Credits

Author: Gabriel Maldonado  
Italy

Example written by Kevin Conder.

New in Csound version 3.49

# vmap

vmap -- Maps elements from a vectorial control signal

vmap

## Description

Maps elements from a vectorial control signal onto itself according to the indices of a second vectorial control signal

## Syntax

```
vmap ifn, ifn2, ielements
```

## Initialization

*ifn* - number of the table where the vectorial signal will be copied

*ifn* - number of the table hosting the vectorial signal to be copied

*ielements* - number of elements of the vector

## Performance

*vmap* maps elements of *ifn1* according to the values of table *ifn2*. Elements of *ifn1* are treated as indexes of table *ifn2*, so element values of *ifn1* must not exceed the length of *ifn2* table otherwise a Csound crash due to an illegal memory access error will occur. Elements of *ifn1* are treated as integers, so any fractional part will be truncated.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

*Note:* *bmscan* not yet available on Canonical Csound

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vmirror

vmirror -- Limiting and Wrapping Vectorial Signals

vmirror

## Description

'Reflects' elements of vectorial control signals on thresholds.

## Syntax

```
vmirror ifn, kmin, kmax, ielements
```

## Initialization

*ifn* - number of the table hosting the vector to be processed

*ielements* - number of elements of the vector

## Performance

*kmin* - minimum threshold value

*kmax* - maximum threshold value

*vmirror* 'reflects' each element of corresponding vector if it exceeds low or high thresholds.

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate with a vectorial signal instead of with a scalar signal.

Result overrides old values of *ifn1*, if these are out of min/max interval. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

All these opcodes are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

*Note:* *bmscan* not yet available on Canonical Csound

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vmult

vmult -- Performs multiplication between a vectorial control signal and a scalar control signal

vmult

## Description

Performs multiplication between a vectorial control signal and a scalar control signal

## Syntax

```
vmult ifn, kval, ielements
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ielements* - number of elements of the vector

## Performance

*kval* - scalar operand to be processed

*vmult* multiplies each elements of the vector contained in the table *ifn* by *kval* operand.

These opcodes (*vadd*, *vmult*, *vpow*, *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* opcode to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Example

Here is an example of the *vmult* opcode. It uses the files *vmult.csd* [examples/vmult.csd].

### Example 416. Example of the vmult opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr=44100
kr=4410
ksmps=10
nchnls=2

      instr 1 ;table playback
ar lposcil 1, 1, 0, 262144, 1
out ar,ar
```

```
        endin

        instr 2
vcopy 2, 1, 40000 ;copy vector from sample to empty table
vmult 5, 10000, 262144 ;scale noise to make it audible
vcopy 1, 5, 40000 ;put noise into sample
turnoff
        endin

        instr 3
vcopy 1, 2, 40000 ;put original information back in
turnoff
        endin

</CsInstruments>
<CsScore>
f1  0 262144   -1 "beats.aiff" 0 4 0
f2  0 262144    2  0

f5  0 262144   21  3 30000

i1 0 4
i2 3 1

s
i1 0 4
i3 3 1
s

i1 0 4

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

## vmultv

vmultv -- Performs multiplication between two vectorial control signals

vmultv

## Description

Performs multiplication between two vectorial control signals

## Syntax

```
vmultv ifn1, ifn2, ielements
```

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

## Performance

*vmultv* multiplies two vectorial control signals, that is, each element of the first vector is processed (only) with the corresponding element of the other vector. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The Result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy* opcode to copy it in another table.

This opcode works at k-rate.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# voice

voice -- An emulation of a human voice.

voice

## Description

An emulation of a human voice.

## Syntax

ares **voice** kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn

## Initialization

*ifn*, *ivfn* -- two table numbers containing the carrier waveform and the vibrato waveform. The files *impuls20.aiff* [examples/impuls20.aiff], *ahh.aiff* [examples/ahh.aiff], *eee.aiff* [examples/eee.aiff], or *ooo.aiff* [examples/ooo.aiff] are suitable for the first of these, and a sine wave for the second. These files are available from <ftp://ftp.cs.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

## Examples

Here is an example of the voice opcode. It uses the files *voice.orc* [examples/voice.orc], *voice.sco* [examples/voice.sco], and *impuls20.aiff* [examples/impuls20.aiff].

### Example 417. Example of the voice opcode.

```
/* voice.orc */
; Initialize the global variables.
sr = 22050
kr = 2205
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 3
  kfreq = 0.8
  kphoneme = 6
  kform = 0.488
  kvibf = 0.04
  kvamp = 1
  ifn = 1
  ivfn = 2

  av voice kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn

  ; It tends to get loud, so clip voice's amplitude at 30,000.
  al clip av, 2, 30000
  out al
endin
/* voice.orc */

/* voice.sco */
; Table #1, an audio file for the carrier waveform.
f 1 0 256 1 "impuls20.aiff" 0 0 0
; Table #2, a sine wave for the vibrato waveform.
f 2 0 256 10 1

; Play Instrument #1 for a half-second.
i 1 0 0.5
e
/* voice.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

Example written by Kevin Conder.

New in Csound version 3.47



# vport

vport -- Vectorial Control-rate Delay Paths

vport

## Description

Generate a sort of 'vectorial' portamento

## Syntax

```
vport ifn, khtime, ielements [, ifnInit]
```

## Initialization

*ifn* - number of the table containing the output vector

*ielements* - number of elements of the two vectors

*ifnInit* (optional) - number of the table containing a vector whose elements contain initial portamento values.

## Performance

*vport* is similar to *port*, but operates with vectorial signals, instead of with scalar signals. Each vector element is treated as an independent control signal. Input vector input and output vectors are placed in the same table and output vector overrides input vector. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

## vpow

vpow -- Performs power-of operations between a vectorial control signal and a scalar control signal

vpow

## Description

Performs power-of operations between a vectorial control signal and a scalar control signal

## Syntax

```
vpow ifn, kval, ielements
```

## Initialization

*ifn* - number of the table hosting the vectorial signal to be processed

*ielements* - number of elements of the vector

## Performance

*kval* - scalar operand to be processed

*vpow* elevates each element of the vector contained in the table *ifn* to the power of *kval*.

These opcodes (*vadd*, *vmult*, *vpow*, *vexp*) perform numeric operations between a vectorial control signal (hosted by the table *ifn*), and a scalar signal (*kval*). Result is a new vector that overrides old values of *ifn*. All these opcodes work at k-rate.

In all these opcodes, the resulting vectors are stored in *ifn*, overriding the initial vectors. If you want to keep initial vector, use *vcopy* opcode to copy it in another table. All these operators are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

## vpowv

vpowv -- Performs power-of operations between two vectorial control signals

vpowv

## Description

Performs power-of operations between two vectorial control signals

## Syntax

**vpowv** ifn1, ifn2, ielements

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

## Performance

*vpowv* divides two vectorial control signals, that is, each element of *ifn1* is risen to the corresponding element of *ifn2*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy* opcode to copy it in another table.

This opcode works at k-rate.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

## vpvoc

vpvoc -- Implements signal reconstruction using an fft-based phase vocoder and an extra envelope.

vpvoc

## Description

Implements signal reconstruction using an fft-based phase vocoder and an extra envelope.

## Syntax

ares **vpvoc** ktmpnt, kfmod, ifile [, ispecwp] [, ifn]

## Initialization

*ifile* -- the pvoc number (n in pvoc.n) or the name in quotes of the analysis file made using pvanal. (See *pvoc*.)

*ispecwp* (optional, default=0) -- if non-zero, attempts to preserve the spectral envelope while its frequency content is varied by *kfmod*. The default value is zero.

*ifn* (optional, default=0) -- 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

*ktmpnt* -- The passage of time, in seconds, through the analysis file. *ktmpnt* must always be positive, but can move forwards or backwards in time, be stationary or discontinuous, as a pointer into the analysis file.

*kfmod* -- 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 originally 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* or *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 *tableseg* 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*.

## Examples

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.

```
ptime    line          0, p3,3 ; time pointer, in seconds, into file
         tablexseg      1, p3*.5, 2, p3*.5, 3
apv       vpvoc         ptime,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.

## See Also

*pvoc*

## Credits

Authors: Dan Ellis and Richard Karpen  
Seattle, WA USA  
1997

# vrandh

vrandh -- Generate a sort of 'vectorial band-limited noise'

vrandh

## Description

Copies between two vectorial control signals

## Syntax

**vrandh** ifn, krange, kcps, ielements

## Initialization

*ifn* - number of the table where the vectorial signal will be generated

*ielements* - number of elements of the vector

## Performance

*krange* - range of random elements (from -krange to krange)

*kcps* - rate of generated elements in cycles per seconds

This opcode is similar to *randh*, but operates with vectors instead of with scalar values..

The output is a vector contained in ifn (that must be previously allocated).

All these operators are designed to be used together with other opocdes that operate with vector such as *bmscan*, *adsynt* etc.

*Note:* *bmscan* not yet available on Canonical Csound

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vrandi

vrandi -- Generate a sort of 'vectorial band-limited noise'

vrandi

## Description

Copies between two vectorial control signals

## Syntax

**vrandi** ifn, krange, kcps, ielements

## Initialization

*ifn* - number of the table where the vectorial signal will be generated

*ielements* - number of elements of the vector

## Performance

*krange* - range of random elements (from -krange to krange)

*kcps* - rate of generated elements in cycles per seconds

This opcode is similar to *randi*, but operates with vectors instead of with scalar values..

The output is a vector contained in ifn (that must be previously allocated).

All these operators are designed to be used together with other opocdes that operate with vector such as *bmscan*, *adsynt* etc.

*Note:* bmscan not yet available on Canonical Csound

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vstaudio, vstaudiog

vstaudio, vstaudiog -- VST audio output.

vstaudio

## Syntax

```
aout1,aout2 vstaudio instance, [ain1, ain2]
```

```
aout1,aout2 vstaudiog instance, [ain1, ain2]
```

## Description

*vstaudio* and *vstaudiog* are used for sending and receiving audio from a VST plugin.

*vstaudio* is used within an instrument definition that contains a *vstmidiout* or *vstnote* opcode. It outputs audio for only that one instrument. Any audio remaining in the plugin after the end of the note, for example a reverb tail, will be cut off and should be dealt with using a damping envelope.

*vstaudiog* (*vstaudio* global) is used in a separate instrument to process audio from any number of VST notes or MIDI events that share the same VST plugin instance (*instance*). The *vstaudiog* instrument must be numbered higher than all the instruments receiving notes or MIDI data, and the note controlling the *vstplug* instrument must have an indefinite duration, or at least a duration as long as the VST plugin is active.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other *vst4cs* opcodes.

## Performance

*aout1*, *aout2* - the audio output received from the plugin.

*ain1*, *ain2* - the audio input sent to the plugin.

## Examples

See *vstmidiout* and *vstparamset* for examples.

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's *vst~* object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.



# vstbankload

vstbankload -- Loads parameter banks to a VST plugin.

vstbankload

## Syntax

**vstbankload** instance, ipath

## Description

*vstbankload* is used for loading parameter banks to a VST plugin.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

*ipath* - the full pathname of the parameter bank (. fxb file).

## Examples

### Example 418.

```
/* orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll",1
instr 4
vstbankload gihandle1,"c:/vstplugins/cheeze/chengo'scheese.fxb"
vstinfo gihandle1
endin
```

```
/* sco */
i 3 0 21
i4 1 1 57 32
```

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstedit

vstedit -- Opens the GUI editor widow for a VST plugin.

vstedit

## Syntax

**vstedit** instance

## Description

*vstedit* opens the custom GUI editor widow for a VST plugin. Note that not all VST plugins have custom GUI editors.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstinit

vstinit -- Load a VST plugin into memory for use with the other vst4cs opcodes.

vstinit

## Syntax

```
instance vstinit ilibrarypath [,iverbose]
```

## Description

*vstinit* is used to load a VST plugin into memory for use with the other vst4cs opcodes. Both VST effects and instruments (synthesizers) can be used.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

*ilibrarypath* - the full path to the vst plugin shared library (dll, on Windows). Remember to use '/' instead of '\' as separator.

*iverbose* - show plugin information and parameters when loading.

## Examples

### Example 419. Loading a VST Plugin

```
/* orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll",1
instr 1
    giHandle2 vstinit "c:/vstplugins/crazy diamonds.dll",1
endin
```

```
/* sco */
i 1 0 1
e
```

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstinfo

vstinfo -- Displays the parameters and the programs of a VST plugin.

vstinfo

## Syntax

**vstinfo** instance

## Description

*vstinfo* displays the parameters and the programs of a VST plugin.

Note: The *verbose* flag in *vstinit* gives the same information as *vstinfo*. *vstinfo* is useful after loading parameter banks, or when the plugin changes parameters dynamically.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Examples

### Example 420.

```
/* orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll",1
instr 4
vstbankload gihandle1,"c:/vstplugins/cheeze/chengo'scheese.fxb"
vstinfo gihandle1
endin
```

```
/* sco */
i 3 0 21
i4 1 1 57 32
e
```

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstmidiout

vstmidiout -- Sends MIDI information to a VST plugin.

vstmidiout

## Syntax

**vstmidiout** instance, kstatus, kchan, kdata1, kdata2

## Description

*vstmidiout* is used for sending MIDI information to a VST plugin.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Performance

*kstatus* - the type of midi message to be sent. Currently noteon (144), note off (128), Control Change (176), Program change (192), Aftertouch (208) and Pitch Bend (224) are supported.

*kchan* - the MIDI channel transmitted on.

*kdata1*, *kdata2* - the MIDI data pair, which varies depending on kstatus. e.g. note/velocity for note on and note off, Controller number/value for control change.

## Examples

### Example 421.

```
/* orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll",1
instr 3
ain1 = 0
ab1, ab2 vstaudio gihandle1, ain1, ain1
outs ab1, ab2
endin
instr 4
vstmidiout gihandle1,144,1,p4,p5
endin
```

```
/* sco */
i 3 0 21
i4 1 1 57 32
i4 3 1 60 100
i4 5 1 62 100
i4 7 1 64 100
i4 9 1 65 100
i4 11 1 67 100
```

```
i4 13 1 69 100  
i4 15 3 71 100  
i4 18 3 72 100  
e
```

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstnote

`vstnote` -- Sends a MIDI note with definite duration to a VST plugin.

`vstnote`

## Syntax

**vstnote** *instance*, *kchan*, *knote*, *kveloc*, *kdur*

## Description

*vstnote* sends a MIDI note with definite duration to a VST plugin.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Performance

*kchan* - The midi channel to trasnmit the note on.

*knote* - The midi note number to send.

*kveloc* - The midi note's velocity.

*kdur* - The midi note's duration in seconds.

Note: Be sure the instrument containing `vstnote` is not finished before the duration of the note, otherwise you'll have a 'hung' note.

## Examples

### Example 422.

```
/* orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
gihandle5 vstinit "c:/vstplugins/cheeze/cheeze machine.dll",1
instr 3
ain1 = 0
ga1, ga2 vstplugg gihandle5, ain1, ain1
endin
instr 4
vstnote giHandle5, 1, p4, p5, p3
endin
instr 10
outs ga1, ga2
endin

/* sco */
i 3 0 21
```

```
i 10 0 21
i4 1 3 57 55
i4 3 3 60 100
i4 5 3 62 100
i4 7 3 64 100
i4 9 2 65 100
i4 11 1 67 100
i4 13 1 69 100
i4 15 3 71 100
i4 18 3 72 100
```

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.



# vstparamset,vstparamget

vstparamset,vstparamget -- Used for parameter communication to and from a VST plugin.

vstparamset

## Syntax

**vstparamset** instance, kparam, kvalue

kvalue **vstparamget** instance, kparam

## Description

*vstparamset* and *vstparamget* are used for parameter communication to and from a VST plugin.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

## Performance

*kparam* - The number of the parameter to set or get.

*kvalue* - the value to set, or the the value returned by the plugin.

Parameters vary according to the plugin. To find out what parameters are available, use the verbose option when loading the plugin with vstinit.

## Examples

### Example 423.

```
/* orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
gihandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll",1
instr 3
ain1 = 0
ab1, ab2 vstaudio gihandle1, ain1, ain1
outs ab1, ab2
endin
instr 4
vstmidiout gihandle1,144,1,p4,p5
kline line 0,p3,1
vstparamset gihandle1, 3, kline
endin

/* sco */
i 3 0 21
i4 1 1 57 32
i4 3 1 60 100
```

```
i4 5 1 62 100  
i4 7 1 64 100  
i4 9 1 65 100  
i4 11 1 67 100  
i4 13 1 69 100  
i4 15 3 71 100  
i4 18 3 72 100  
e
```

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vstprogset

vstprogset -- Loads parameter banks to a VST plugin.

vstprogset

## Syntax

**vstprogset** instance, kprogram

## Description

*vstprogset* sets one of the programs in an . fxb bank.

## Initialization

*instance* - the number which identifies the plugin, to be passed to other vst4cs opcodes.

*kprogram* - the number of the program to set.

## Examples

### Example 424.

```
/* orc */
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2
giHandle1 vstinit "c:/vstplugins/cheeze/cheeze machine.dll",1
instr 4
vstbankload gihandle1,"c:/vstplugins/cheeze/chengo'scheese.fxb"
vstprogset gihandle1, 4
vstinfo gihandle1
endin
```

```
/* sco */
i 3 0 21
i4 1 1 57 32
e
```

## Credits

By: Andres Cabrera and Michael Gogins

Uses code from Hermann Seib's VSTHost and Thomas Grill's vst~ object.

VST is a trademark of Steinberg Media Technologies GmbH. VST Plug-In Technology by Steinberg.

# vsubv

vsubv -- Performs subtraction between two vectorial control signals

vsubv

## Description

Performs subtraction between two vectorial control signals

## Syntax

**vsubv** ifn1, ifn2, ielements

## Initialization

*ifn1* - number of the table hosting the first vector to be processed

*ifn2* - number of the table hosting the second vector to be processed

*ielements* - number of elements of the two vectors

## Performance

*vsubv* subtracts two vectorial control signals, that is, each element of *ifn2* is subtracted from the corresponding element of *ifn1*. Each vectorial signal is hosted by a table (*ifn1* and *ifn2*). The number of elements contained in both vectors must be the same.

The result is a new vectorial control signal that overrides old values of *ifn1*. If you want to keep the old *ifn1* vector, use *vcopy* opcode to copy it in another table.

This opcode works at k-rate.

All these operators (*vaddv*, *vsubv*, *vmultv*, *vdivv*, *vpowv*, *vexpv*, *vcopy* and *vmap*) are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtablei

vtablei -- Read vectors (from tables -or arrays of vectors).

vtablei

## Description

This opcode reads vectors from tables.

## Syntax

**vtablei** *indx*, *ifn*, *interp*, *ixmode*, *iout1* [, *iout2*, *iout3*, .... , *ioutN* ]

## Initialization

*indx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*ifn* - table number

*iout1...ioutN* - output vector elements

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

*interp* - vtable (vector table) family of opcodes allows the user to switch between interpolated or non-interpolated output by means of the *interp* argument.

## Performance

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of *outN* or *inargN* arguments, and must remain fixed for all indexes of each table.

vtable (vector table) family of opcodes allows the user to switch between interpolated or non-interpolated output by means of the *interp* argument.

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtable, in order to correct eventual out-of-range values.

Notice that vtablei output arguments are placed at the left of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## Examples

Here is an example of the vtablei opcode. It uses the files *vtablei.csd* [examples/vtablei.csd]

### Example 425. Example of the vtablei opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr      =      44100
kr      =      100
ksmps   =      441
nchnls  =      2

gindx init 0

      instr      1
kindex init 0
ktrig metro 0.5
if ktrig = 0 goto noevent
event "i", 2, 0, 0.5, kindex
kindex = kindex + 1
noevent:

      endin

      instr 2
iout1 init 0
iout2 init 0
iout3 init 0
iout4 init 0
indx = p4
vtablei indx, 1, 1, 0, iout1,iout2, iout3, iout4
print iout1, iout2, iout3, iout4
turnoff
      endin

</CsInstruments>
<CsScore>
f 1 0 32 10 1
i 1 0 20

</CsScore>
</CsoundSynthesizer>
```

## Credits

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vtablek

vtablek -- Read vectors (from tables -or arrays of vectors).

vtablek

## Description

This opcode reads vectors from tables at k-rate.

## Syntax

**vtablek** kndx, kfn, kinterp, ixmode, kout1 [, kout2, kout3, .... , koutN ]

## Initialization

*ixmode* - index data mode. The default value is 0.

== 0 index is treated as a raw table location,

== 1 index is normalized (0 to 1).

*kinterp* - switch between interpolated or non-interpolated output. 0 -> non-interpolation , non-zero -> interpolation activated

## Performance

*kndx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*kfn* - table number

*kout1...koutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

vtablek allows the user to switch between interpolated or non-interpolated output at k-rate by means of kinterp argument.

vtablek allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also ixmode argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtable, in order to correct eventual out-of-range values.

Notice that vtablek output arguments are placed at the left of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as fin or trigseq).

## Examples

Here is an example of the vtablek opcode. It uses the files *vtablek.csd* [examples/vtablek.csd].

**Example 426. Example of the vtablek opcode.**

```
<CsoundSynthesizer>
<CsOptions>
-odac -B441 -b441
</CsOptions>
<CsInstruments>

sr      =      44100
kr      =      100
ksmps   =      441
nchnls  =      2

gkindx init -1

      instr      1
kindex init 0
ktrig metro 0.5
if ktrig = 0 goto noevent
gkindx = gkindx + 1
noevent:

      endin

      instr 2
kout1 init 0
kout2 init 0
kout3 init 0
kout4 init 0
vtablek gkindx, 1, 1, 0, kout1,kout2, kout3, kout4
printk2 kout1
printk2 kout2
printk2 kout3
printk2 kout4
      endin

</CsInstruments>
<CsScore>
f 1 0 32 10 1
i 1 0 20
i 2 0 20
</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)



# vtablea

vtablea -- Read vectors (from tables -or arrays of vectors).

vtablea

## Description

This opcode reads vectors from tables at a-rate.

## Syntax

**vtablea** *andx*, *kfn*, *kinterp*, *ixmode*, *aout1* [, *aout2*, *aout3*, .... , *aoutN* ]

## Initialization

*ixmode* - index data mode. The default value is 0. == 0 index is treated as a raw table location, == 1 index is normalized (0 to 1).

## Performance

*andx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0). *kfn* - table number *kinterp* - switch between interpolated or non-interpolated output. 0 -> non-interpolation , non-zero -> interpolation activated *aout1...aoutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

**vtablea** allows the user to switch between interpolated or non-interpolated output at k-rate by means of *kinterp* argument.

**vtablea** allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also *ixmode* argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtablea, in order to correct eventual out-of-range values.

Notice that vtablea output arguments are placed at the left of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as *fin* or *trigseq*).

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtablewi

vtablewi -- Write vectors (to tables -or arrays of vectors).

vtablewi

## Description

This opcode writes vectors to tables at init time.

## Syntax

**vtablewi** *indx*, *ifn*, *ixmode*, *inarg1* [, *inarg2*, *inarg3* , .... , *inargN* ]

## Initialization

*indx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*ifn* - table number

*ixmode* - index data mode. The default value is 0. == 0 index is treated as a raw table location, == 1 index is normalized (0 to 1).

*inarg1...inargN* - output vector elements

## Performance

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtable, in order to correct eventual out-of-range values.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtablewk

vtablewk -- Write vectors (to tables -or arrays of vectors).

vtablewk

## Description

This opcode writes vectors to tables at k-rate.

## Syntax

**vtablewk** kndx, kfn, ixmode, kinarg1 [, kinarg2, kinarg3 , .... , kinargN ]

## Initialization

*ixmode* - index data mode. The default value is 0. == 0 index is treated as a raw table location, == 1 index is normalized (0 to 1).

## Performance

*kndx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*kfn* - table number

*kinarg1...kinargN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

**vtablewk** allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also *ixmode* argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtable, in order to correct eventual out-of-range values.

## Examples

Here is an example of the vtablewk opcode. It uses the files *vtablewk.csd* [examples/vtablewk.csd].

### Example 427. Example of the vtablewk opcode.

```
<CsoundSynthesizer>
<CsOptions>
-odac -b441 -B441
</CsOptions>
<CsInstruments>
```

```
sr=44100
kr=4410
ksmps=10
nchnls=2

        instr 1
vcopy
ar random 0, 1
vtablewa ar
out ar,ar
        endin

</CsInstruments>
<CsScore>
f1  0 262144    -1 "beats.aiff" 0 4 0
f2  0 262144     2  0

i1 0 4
i2 3 1

s
i1 0 4
i3 3 1
s

i1 0 4

</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vtablewa

vtablewa -- Write vectors (to tables -or arrays of vectors).

vtablewa

## Description

This opcode writes vectors to tables at a-rate.

## Syntax

**vtablewa** andx, kfn, ixmode, ainarg1 [, ainarg2, ainarg3 , .... , ainargN ]

## Initialization

*ixmode* - index data mode. The default value is 0. == 0 index is treated as a raw table location, == 1 index is normalized (0 to 1).

## Performance

*andx* - Index into f-table, either a positive number range matching the table length (*ixmode* = 0) or a 0 to 1 range (*ixmode* != 0).

*kfn* - table number

*ainarg1...ainargN* - input vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

vtablewa allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also *ixmode* argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtablewa, in order to correct eventual out-of-range values.

## Examples

Here is an example of the vtablewa opcode. It uses the files *vtablewa.csd* [examples/vtablewa.csd].

### Example 428. Example of the vtablek opcode.

```
<CsoundSynthesizer>
<CsOptions>
;-ovtablewa.wav -W -b441 -B441
-odac -b441 -B441
</CsOptions>
<CsInstruments>
```

```
sr=44100
kr=441
ksmps=100
nchnls=2

      instr 1
ilen = ftilen (1)

knew1 oscil 10000, 440, 3
knew2 oscil 15000, 440, 3, 0.5
kindex phasor 0.3
asig oscil 1, sr/ilen , 1
vtablewk kindex*ilen, 1, 0, knew1, knew2
out asig,asig
      endin

</CsInstruments>
<CsScore>
f1  0 262144  -1 "beats.aiff" 0 4 0
f2  0 262144   2  0
f3  0 1024  10 1

i1 0 10
</CsScore>
</CsoundSynthesizer>
```

## Credits

Written by Gabriel Maldonado.

Example written by Andres Cabrera.

New in Csound 5 (Previously available only on CsoundAV)

# vtabi

vtabi -- Read vectors (from tables -or arrays of vectors).

vtabi

## Description

This opcode reads vectors from tables.

## Syntax

```
vtabi   indx, ifn, iout1 [, iout2, iout3, .... , ioutN ]
```

## Initialization

*indx* - Index into f-table, either a positive number range matching the table length

*ifn* - table number

*iout1...ioutN* - output vector elements

## Performance

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtable, in order to correct eventual out-of-range values.

Notice that vtabi output arguments are placed at the left of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as fin or trigseq).

The **vtab** family is similar to **vtable**, but is much faster because interpolation is not available, table number cannot be changed after initialization, and only raw indexing is supported.

## Examples

For an example of the vtabi opcode usage, see *vtablei*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtabk

vtabk -- Read vectors (from tables -or arrays of vectors).

vtabk

## Description

This opcode reads vectors from tables at k-rate.

## Syntax

**vtabk** kndx, ifn, kout1 [, kout2, kout3, .... , koutN ]

## Initialization

*ifn* - table number

## Performance

*kndx* - Index into f-table, either a positive number range matching the table length

*kout1...koutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtable, in order to correct eventual out-of-range values.

Notice that *vtabk* output arguments are placed at the left of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as fin or trigseq).

The **vtab** family is similar to **vtable**, but is much faster because interpolation is not available, table number cannot be changed after initialization, and only raw indexing is supported.

## Examples

For an example of the vtabk opcode usage, see *vtablek*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)



## vtaba

vtaba -- Read vectors (from tables -or arrays of vectors).

vtaba

## Description

This opcode reads vectors from tables at a-rate.

## Syntax

```
vtaba  andx, ifn, aout1 [, aout2, aout3, .... , aoutN ]
```

## Initialization

*ifn* - table number

## Performance

*andx* - Index into f-table, either a positive number range matching the table length

*aout1...aoutN* - output vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtaba, in order to correct eventual out-of-range values.

Notice that **vtaba** output arguments are placed at the left of the opcode name, differently from usual (this style is already used in other opcodes using undefined lists of output arguments such as fin or trigseq).

The **vtab** family is similar to the **vtable** family, but is much faster because interpolation is not available, table number cannot be changed after initialization, and only raw indexing is supported.

## Examples

The usage of *vtaba* is similar to *vtablek*.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# **vtabwi**

vtabwi -- Write vectors (to tables -or arrays of vectors).

vtabwi

## **Description**

This opcode writes vectors to tables at init time.

## **Syntax**

```
vtabwi  indx, ifn, inarg1 [, inarg2, inarg3 , .... , inargN ]
```

## **Initialization**

*indx* - Index into f-table, either a positive number range matching the table length (ixmode = 0) or a 0 to 1 range (ixmode != 0).

*ifn* - table number

*inarg1...inargN* - output vector elements

## **Performance**

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtable, in order to correct eventual out-of-range values.

## **Credits**

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtabwk

vtabwk -- Write vectors (to tables -or arrays of vectors).

vtabwk

## Description

This opcode writes vectors to tables at a-rate.

## Syntax

**vtabwk** kndx, ifn, kinarg1 [, kinarg2, kinarg3 , .... , kinargN ]

## Initialization

*ifn* - table number

## Performance

*kndx* - Index into f-table, either a positive number range matching the table length (ixmode = 0) or a 0 to 1 range (ixmode != 0). *kinarg1...kinargN* - input vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

vtabwk allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also ixmode argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtabwk, in order to correct eventual out-of-range values.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vtabwa

vtabwa -- Write vectors (to tables -or arrays of vectors).

vtabwa

## Description

This opcode writes vectors to tables at a-rate.

## Syntax

**vtabwa** andx, ifn, ainarg1 [, ainarg2, ainarg3 , .... , ainargN ]

## Initialization

*ifn* - table number

## Performance

*andx* - Index into f-table, either a positive number range matching the table length (ixmode = 0) or a 0 to 1 range (ixmode != 0). *ainarg1...ainargN* - input vector elements

This opcode is useful in all cases in which one needs to access sets of values associated to unique indexes (for example, multi-channel samples, STFT bin frames, spectral formants, p-field based scores etc.) . The number of elements of each vector frame is automatically determined by the number of outN or inargN arguments, and must remain fixed for all indexes of each table.

vtabwa allows also to switch the table number at k-rate (but this is possible only when vector frames of each used table have the same number of elements, otherwise unpredictable results could occur), as well as to choose indexing style (raw or normalized, see also ixmode argument of table opcode ).

Notice that no wrap nor limit mode is implemented. So, if an index attempt to access to a zone not allocated by the table, Csound will probably crash. However this drawback can be easily avoided by using wrap or limit opcodes applied to indexes before using vtabwa, in order to correct eventual out-of-range values.

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# vwrap

vwrap -- Limiting and Wrapping Vectorial Signals

vwrap

## Description

Wraps elements of vectorial control signals.

## Syntax

**vwrap** ifn, kmin, kmax, ielements

## Initialization

*ifn* - number of the table hosting the vector to be processed

*ielements* - number of elements of the vector

## Performance

*kmin* - minimum threshold value

*kmax* - maximum threshold value

*vwrap* wraps around each element of corresponding vector if it exceeds low or high thresholds.

These opcodes are similar to *limit*, *wrap* and *mirror*, but operate with a vectorial signal instead of with a scalar signal.

Result overrides old values of *ifn1*, if these are out of min/max interval. If you want to keep input vector, use *vcopy* opcode to copy it in another table.

All these opcodes are designed to be used together with other opcodes that operate with vectorial signals such as *bmscan*, *vcella*, *adsynt*, *adsynt2* etc.

*Note:* *bmscan* not yet available on Canonical Csound

## Credits

Written by Gabriel Maldonado.

New in Csound 5 (Previously available only on CsoundAV)

# waveset

waveset -- A simple time stretch by repeating cycles.

waveset

## Description

A simple time stretch by repeating cycles.

## Syntax

ares **waveset** ain, krep [, ilen]

## Initialization

*ilen* (optional, default=0) -- the length (in samples) of the audio signal. If *ilen* is set to 0, it defaults to half the given note length (p3).

## Performance

*ain* -- the input audio signal.

*krep* -- the number of times the cycle is repeated.

The input is read and each complete cycle (two zero-crossings) is repeated *krep* times.

There is an internal buffer as the output is clearly slower than the input. Some care is taken if the buffer is too short, but there may be strange effects.

## Examples

Here is an example of the waveset opcode. It uses the files *waveset.orc* [examples/waveset.orc], *waveset.sco* [examples/waveset.sco], and *beats.wav* [examples/beats.wav].

### Example 429. Example of the waveset opcode.

```
/* waveset.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - play an audio file.
instr 1
  asig soundin "beats.wav"
  out asig
endin

; Instrument #2 - stretch the audio file with waveset.
instr 2
  asig soundin "beats.wav"
  a1 waveset asig, 2
```

```
    out a1
endin
/* waveset.orc */
```

```
/* waveset.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for four seconds.
i 2 3 4
e
/* waveset.sco */
```

## Credits

Author: John ffitch  
February 2001

Example written by Kevin Conder.

New in version 4.11

# weibull

weibull -- Weibull distribution random number generator (positive values only).

weibull

## Description

Weibull distribution random number generator (positive values only). This is an x-class noise generator

## Syntax

ares **weibull** ksigma, ktau

ires **weibull** ksigma, ktau

kres **weibull** ksigma, ktau

## Performance

*ksigma* -- scales the spread of the distribution.

*ktau* -- if greater than one, numbers near *ksigma* are favored. If smaller than one, small values are favored. If t equals 1, the distribution is exponential. Outputs only positive numbers.

For more detailed explanation of these distributions, see:

1. C. Dodge - T.A. Jerse 1985. Computer music. Schirmer books. pp.265 - 286
2. D. Lorrain. A panoply of stochastic cannons. In C. Roads, ed. 1989. Music machine . Cambridge, Massachusetts: MIT press, pp. 351 - 379.

## Examples

Here is an example of the weibull opcode. It uses the files *weibull.orc* [examples/weibull.orc] and *weibull.sco* [examples/weibull.sco].

### Example 430. Example of the weibull opcode.

```
/* weibull.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Generate a random number in a Weibull distribution.
  ; ksigma = 1
  ; ktau = 1
```



```
    i1 weibull 1, 1

    print i1
endin
/* weibull.orc */

/* weibull.sco */
; Play Instrument #1 for one second.
i 1 0 1
e
/* weibull.sco */
```

Its output should include lines like this:

```
instr 1:  i1 = 1.834
```

## See Also

*seed, betarand, bexprnd, cauchy, exprand, gauss, linrand, pcauchy, poisson, trirand, unirand*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Example written by Kevin Conder.

# wgbow

wgbow -- Creates a tone similar to a bowed string.

wgbow

## Description

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

## Syntax

ares **wgbow** kamp, kfreq, kpres, krat, kvibf, kvamp, ifn [, iminfreq]

## Initialization

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

*iminfreq* (optional) -- 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

## Examples

Here is an example of the wgbow opcode. It uses the files *wgbow.orc* [examples/wgbow.orc] and *wgbow.sco* [examples/wgbow.sco].

### Example 431. Example of the wgbow opcode.

```
/* wgbow.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
```

```
instr 1
  kamp = 31129.60
  kfreq = 440
  kpres = 3.0
  krat = 0.127236
  kvibf = 6.12723
  ifn = 1

  ; Create an amplitude envelope for the vibrato.
  kv linseg 0, 0.5, 0, 1, 1, p3-0.5, 1
  kvamp = kv * 0.01

  a1 wgbow kamp, kfreq, kpres, krat, kvibf, kvamp, ifn
  out a1
endin
/* wgbow.orc */

/* wgbow.sco */
; Table #1, a sine wave.
f 1 0 128 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* wgbow.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# wgbowedbar

wgbowedbar -- A physical model of a bowed bar.

wgbowedbar

## Description

A physical model of a bowed bar, belonging to the Perry Cook family of waveguide instruments.

## Syntax

ares **wgbowedbar** kamp, kfreq, kpos, kbowpres, kgain [, iconst] [, itvel] [, ibowpos]

## Initialization

*iconst* (optional, default=0) -- an integration constant. Default is zero.

*itvel* (optional, default=0) -- either 0 or 1. When *itvel* = 0, the bow velocity follows an ADSR style trajectory. When *itvel* = 1, the value of the bow velocity decays in an exponentially.

*ibowpos* (optional, default=0) -- the position on the bow, which affects the bow velocity trajectory.

*ilow* (optional, default=0) -- 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.

## Examples

Here is an example of the wgbowedbar opcode. It uses the files *wgbowedbar.orc* [examples/wgbowedbar.orc] and *wgbowedbar.sco* [examples/wgbowedbar.sco].

### Example 432. Example of the wgbowedbar opcode.

```
/* wgbowedbar.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

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
           out a1
           endin
/* wgbowedbar.orc */

/* wgbowedbar.sco */
i1          0 3 32000 7.00 0
e
/* wgbowedbar.sco */
```

## Credits

Author: John fitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 4.07

# wgbrass

wgbrass -- Creates a tone related to a brass instrument.

wgbrass

## Description

Audio output is a tone related to a brass instrument, using a physical model developed from Perry Cook, but re-coded for Csound.

## Syntax

ares **wgbrass** kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn [, iminfreq]

## 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



### NOTE

This is rather poor, and at present uncontrolled. Needs revision, and possibly more parameters.

## Examples

Here is an example of the wgbrass opcode. It uses the files *wgbrass.orc* [examples/wgbrass.orc] and *wgbrass.sco* [examples/wgbrass.sco].

### Example 433. Example of the wgbrass opcode.

```
/* wgbrass.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
```

```
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 31129.60
  kfreq = 440
  ktens = 0.4
  iatt = 0.1
  kvibf = 6.137
  ifn = 1

  ; Create an amplitude envelope for the vibrato.
  kvamp line 0, p3, 0.5

  al wgbrass kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn
  out al
endin
/* wgbrass.orc */

/* wgbrass.sco */
; Table #1, a sine wave.
f 1 0 128 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* wgbrass.sco */
```

## Credits

Author: John fitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# wgclar

wgclar -- Creates a tone similar to a clarinet.

wgclar

## Description

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

## Syntax

ares **wgclar** kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, ifn [, imin

## 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* (optional) -- 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

## Examples

Here is an example of the wgclar opcode. It uses the files *wgclar.orc* [examples/wgclar.orc] and *wgclar.sco* [examples/wgclar.sco].

### Example 434. Example of the wgclar opcode.

```
/* wgclar.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
```



```
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp init 31129.60
  kfreq = 440
  kstiff = -0.3
  iatt = 0.1
  idetk = 0.1
  kngain = 0.2
  kvibf = 5.735
  kvamp = 0.1
  ifn = 1

  a1 wgclar kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, ifn

  out a1
endin
/* wgclar.orc */

/* wgclar.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
e
/* wgclar.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# wgflute

wgflute -- Creates a tone similar to a flute.

wgflute

## Description

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

## Syntax

ares **wgflute** kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn [, iminfreq]

## 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* (optional) -- 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.

*ijetrf* (optional, default=0.5) -- amount of reflection in the breath jet that powers the flute. Default value is 0.5.

*iendrf* (optional, default=0.5) -- reflection coefficient of the breath jet. Default value is 0.5. Both *ijetrf* and *iendrf* are used in the calculation of the pressure differential.

## 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

## Examples

Here is an example of the wgflute opcode. It uses the files *wgflute.orc* [examples/wgflute.orc] and *wgflute.sco* [examples/wgflute.sco].

### Example 435. Example of the wgflute opcode.

```
/* wgflute.orc */
```

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 31129.60
  kfreq = 440
  kjet = 0.32
  iatt = 0.1
  idetk = 0.1
  kngain = 0.15
  kvibf = 5.925
  kvamp = 0.05
  ifn = 1

  al wgflute kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn
  out al
endin
/* wgflute.orc */

/* wgflute.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* wgflute.sco */
```

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47

# wgpluck

wgpluck -- A high fidelity simulation of a plucked string.

wgpluck

## Description

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

## Syntax

ares **wgpluck** *icps*, *iamp*, *kpick*, *iplk*, *idamp*, *ifilt*, *axcite*

## 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 *idamp*, 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* -- a 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. It uses the files *wgpluck.orc* [examples/wgpluck.orc] and *wgpluck.sco* [examples/wgpluck.sco].

### Example 436. An example of the wgpluck opcode.

```
/* wgpluck.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  icps = 220
  iamp = 20000
```

```
kpick = 0.5
iplk = 0
idamp = 10
ifilt = 1000

axcite oscil 1, 1, 1
apluck wgpluck icps, iamp, kpick, iplk, idamp, ifilt, axcite

out apluck
endin
/* wgpluck.orc */

/* wgpluck.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* wgpluck.sco */
```

The following example produces a shorter, brighter note. It uses the files *wgpluck\_brighter.orc* [examples/wgpluck\_brighter.orc] and *wgpluck\_brighter.sco* [examples/wgpluck\_brighter.sco].

**Example 437. An example of the wgpluck opcode with a shorter, brighter note.**

```
/* wgpluck_brighter.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
icps = 220
iamp = 20000
kpick = 0.5
iplk = 0
idamp = 30
ifilt = 10

axcite oscil 1, 1, 1
apluck wgpluck icps, iamp, kpick, iplk, idamp, ifilt, axcite

out apluck
endin
/* wgpluck_brighter.orc */

/* wgpluck_brighter.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1
```

```
; Play Instrument #1 for two seconds.  
i 1 0 2  
e  
/* wgpluck_brighter.sco */
```

# wgpluck2

wgpluck2 -- Physical model of the plucked string.

wgpluck2

## 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. Based on the Karplus-Strong algorithm.

## Syntax

ares **wgpluck2** *iplk*, *kamp*, *icps*, *kpick*, *krefl*

## Initialization

*iplk* -- 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.

*icps* -- The string plays at *icps* pitch.

## Performance

*kamp* -- Amplitude of note.

*kpick* -- Proportion of the way along the string to sample the output.

*krefl* -- the coefficient of reflection, indicating the lossiness and the rate of decay. It must be strictly between 0 and 1 (it will complain about both 0 and 1).

## Examples

Here is an example of the *wgpluck2* opcode. It uses the files *wgpluck2.orc* [examples/wgpluck2.orc] and *wgpluck2.sco* [examples/wgpluck2.sco].

### Example 438. Example of the *wgpluck2* opcode.

```
/* wgpluck2.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  iplk = 0.75
  kamp = 30000
  icps = 220
  kpick = 0.75
  krefl = 0.5

  apluck wgpluck2 iplk, kamp, icps, kpick, krefl

  out apluck
```

```
endin
/* wgpluck2.orc */

/* wgpluck2.sco */
; Play Instrument #1 for two seconds.
i 1 0 2
e
/* wgpluck2.sco */
```

## See Also

*repluck*

## Credits

Author: John ffitch (after Perry Cook)  
University of Bath, Codemist Ltd.  
Bath, UK

New in Csound version 3.47



# wguide1

`wguide1` -- A simple waveguide model consisting of one delay-line and one first-order lowpass filter.

`wguide1`

## Description

A simple waveguide model consisting of one delay-line and one first-order lowpass filter.

## Syntax

ares **wguide1** asig, xfreq, kcutoff, kfeedback

## Performance

*asig* -- the input of excitation noise.

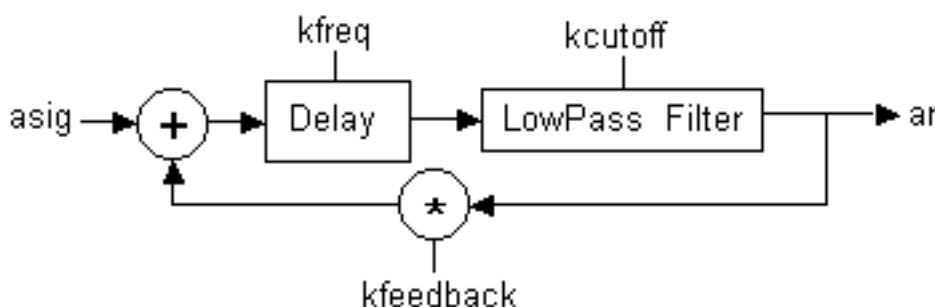
*xfreq* -- the frequency (i.e. the inverse of delay time) Changed to x-rate in Csound version 3.59.

*kcutoff* -- the filter cutoff frequency in Hz.

*kfeedback* -- the feedback factor.

*wguide1* is the most elemental waveguide model, consisting of one delay-line and one first-order lowpass filter.

Implementing waveguide algorithms as opcodes, instead of orc instruments, allows the user to set *kr* different than *sr*, allowing better performance particularly when using real-time.



`wguide1`.

## Examples

Here is an example of the `wguide1` opcode. It uses the files `wguide1.orc` [examples/wguide1.orc] and `wguide1.sco` [examples/wguide1.sco].

### Example 439. Example of the `wguide1` opcode.

```
/* wguide1.orc */
```

```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1 - a simple noise waveform.
instr 1
  ; Generate some noise.
  asig noise 20000, 0.5

  out asig
endin

; Instrument #2 - a waveguide example.
instr 2
  ; Generate some noise.
  asig noise 20000, 0.5

  ; Run it through a wave-guide model.
  kfreq init 200
  kcutoff init 3000
  kfeedback init 0.8
  awg1 wguidel asig, kfreq, kcutoff, kfeedback

  out awg1
endin
/* wguidel.orc */

/* wguidel.sco */
; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 2 2
e
/* wguidel.sco */
```

## See Also

*wguide2*

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

Example written by Kevin Conder.

New in Csound version 3.49

## wguide2

wguide2 -- A model of beaten plate consisting of two parallel delay-lines and two first-order low-pass filters.

wguide2

### Description

A model of beaten plate consisting of two parallel delay-lines and two first-order lowpass filters.

### Syntax

ares **wguide2** asig, xfreq1, xfreq2, kcutoff1, kcutoff2, kfeedback1, kfeedback2

### Performance

*asig* -- the input of excitation noise

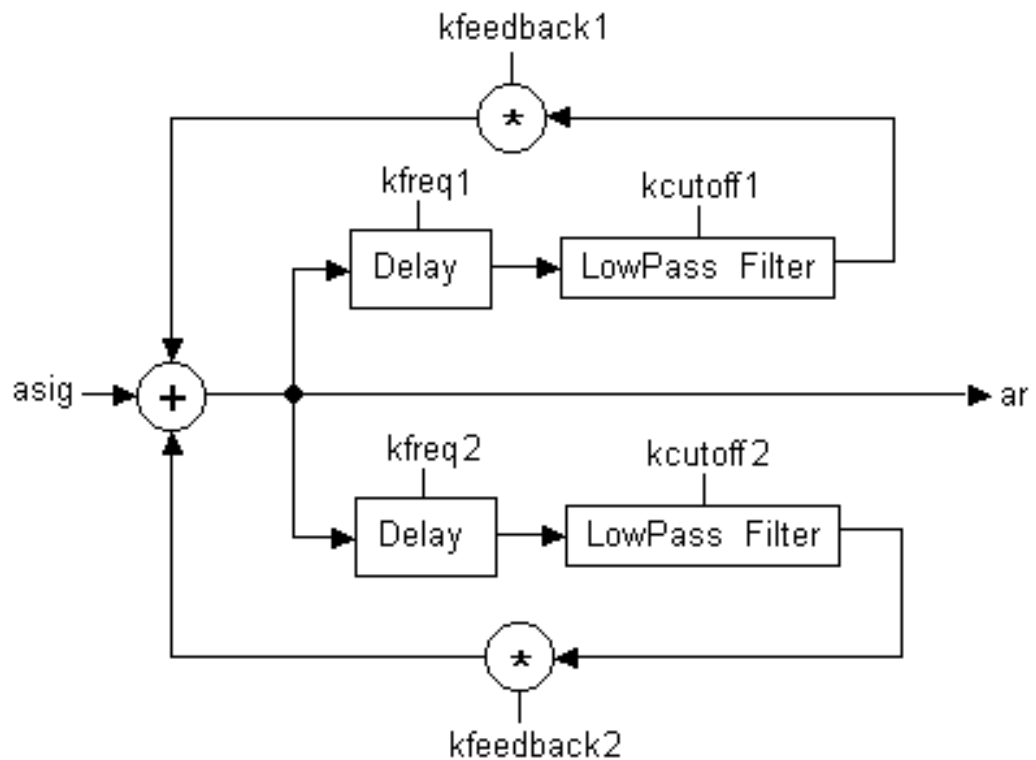
*xfreq1*, *xfreq2* -- the frequency (i.e. the inverse of delay time) Changed to x-rate in Csound version 3.59.

*kcutoff1*, *kcutoff2* -- the filter cutoff frequency in Hz.

*kfeedback1*, *kfeedback2* -- the feedback factor

*wguide2* is a model of beaten plate consisting of two parallel delay-lines 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 orc instruments, allows the user to set *kr* different than *sr*, allowing better performance particularly when using real-time.



`wguide2.`

## See Also

`wguide1`

## Credits

Author: Gabriel Maldonado  
Italy  
October 1998

New in Csound version 3.49

# wrap

`wrap` -- Wraps-around the signal that exceeds the low and high thresholds.

`wrap`

## Description

Wraps-around the signal that exceeds the low and high thresholds.

## Syntax

`ares wrap asig, klow, khigh`

`ires wrap isig, ilow, ihigh`

`kres wrap ksig, klow, khigh`

## Initialization

*isig* -- input signal

*ilow* -- low threshold

*ihigh* -- high threshold

## Performance

*xsig* -- input signal

*klow* -- low threshold

*khigh* -- high threshold

`wrap` wraps-around the signal that exceeds the low and high thresholds.

This opcode is useful in several situations, such as table indexing or for clipping and modeling a-rate, i-rate or k-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.

## See Also

*limit*, *mirror*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.49

# wterrain

wterrain -- A simple wave-terrain synthesis opcode.

wterrain

## Description

A simple wave-terrain synthesis opcode.

## Syntax

aout **wterrain** kamp, kpch, k\_xcenter, k\_ycenter, k\_xradius, k\_yradius, itabx, it

## Initialization

*itabx, itaby* -- The two tables that define the terrain.

## Performance

The output is the result of drawing an ellipse with axes *k\_xradius* and *k\_yradius* centered at (*k\_xcenter*, *k\_ycenter*), and traversing it at frequency *kpch*.

## Examples

Here is an example of the wterrain opcode. It uses the files *wterrain.orc* [examples/wterrain.orc] and *wterrain.sco* [examples/wterrain.sco].

### Example 440. Example of the wterrain opcode.

```
/* wterrain.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

instr 1
kdclk   linseg  0, 0.01, 1, p3-0.02, 1, 0.01, 0
kcx     line    0.1, p3, 1.9
krx     linseg  0.1, p3/2, 0.5, p3/2, 0.1
kpch    line    cpspch(p4), p3, p5 * cpspch(p4)
a1      wterrain 10000, kpch, kcx, kcx, -krx, krx, p6, p7
a1      dcblock a1
        out      a1*kdclk
endin
/* wterrain.orc */

/* wterrain.sco */
f1      0      8192      10      1 0 0.33 0 0.2 0 0.14 0 0.11
f2      0      4096      10      1
i1      0      4      7.00 1 1 1
```

```
i1      4      4      6.07 1 1 2
i1      8      8      6.00 1 2 2
e
/* wterrain.sco */
```

## Credits

Author: Matthew Gillard  
New in version 4.19

## xadsr

`xadsr` -- Calculates the classical ADSR envelope.

`xadsr`

## Description

Calculates the classical ADSR envelope

## Syntax

```
ares xadsr iatt, idec, islev, irel [, idel]
```

```
kres xadsr iatt, idec, islev, irel [, idel]
```

## Initialization

*iatt* -- duration of attack phase

*idec* -- duration of decay

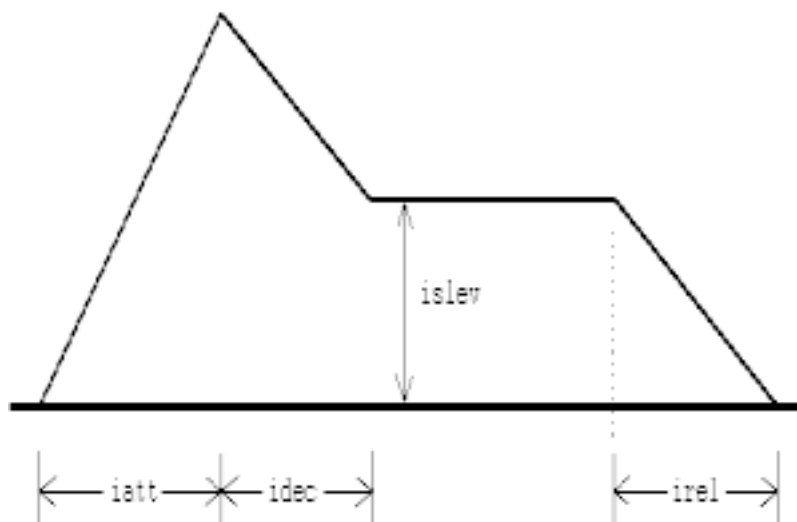
*islev* -- level for sustain phase

*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:



Picture of an ADSR envelope.

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 *xadsr* is identical to *adsr* except it uses exponential, rather than linear, line segments.



*xadsr* is new in Csound version 3.51.

## See Also

*adsr*, *madsr*, *mxadsr*

## Credits

Author: John ffitch

# xin

xin -- Passes variables from a user-defined opcode block,

xin

## Description

The *xin* and *xout* opcodes copy variables to and from the opcode definition, allowing communication with the calling instrument.

The types of input and output variables are defined by the parameters *intypes* and *outtypes*.



### Notes

- *xin* and *xout* should be called only once, and *xin* should precede *xout*, otherwise an init error and deactivation of the current instrument may occur.
- These opcodes actually run only at i-time. Performance time copying is done by the user opcode call. This means that skipping *xin* or *xout* with *kgoto* has no effect, while skipping with *igoto* affects both init and performance time operation.

## Syntax

```
xinarg1 [, xinarg2] ... [xinargN] xin
```

## Performance

*xinarg1*, *xinarg2*, ... - input arguments. The number and type of variables must agree with the user-defined opcode's *intypes* declaration. However, *xin* does not check for incorrect use of init-time and control-rate variables.

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmps]
```

## Examples

See the example for the *opcode* opcode.

## See Also

*endop, opcode, setksmps, xout*

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

# xout

`xout --` Retrieves variables from a user-defined opcode block,

`xout`

## Description

The *xin* and *xout* opcodes copy variables to and from the opcode definition, allowing communication with the calling instrument.

The types of input and output variables are defined by the parameters *intypes* and *outtypes*.



### Notes

- *xin* and *xout* should be called only once, and *xin* should precede *xout*, otherwise an init error and deactivation of the current instrument may occur.
- These opcodes actually run only at i-time. Performance time copying is done by the user opcode call. This means that skipping *xin* or *xout* with *kgoto* has no effect, while skipping with *igoto* affects both init and performance time operation.

## Syntax

```
xout xoutarg1 [, xoutarg2] ... [, xoutargN]
```

## Performance

*xoutarg1*, *xoutarg2*, ... - output arguments. The number and type of variables must agree with the user-defined opcode's *outtypes* declaration. However, *xout* does not check for incorrect use of init-time and control-rate variables.

The syntax of a user-defined opcode block is as follows:

```
opcode name, outtypes, intypes
xinarg1 [, xinarg2] [, xinarg3] ... [xinargN] xin
[setksmps iksmps]
... the rest of the instrument's code.
xout xoutarg1 [, xoutarg2] [, xoutarg3] ... [xoutargN]
endop
```

The new opcode can then be used with the usual syntax:

```
[xinarg1] [, xinarg2] ... [xinargN] name [xoutarg1] [, xoutarg2] ... [xoutargN] [, iksmps]
```

## Examples

See the example for the *opcode* opcode.

## See Also

*endop, opcode, setksmps, xin*

## Credits

Author: Istvan Varga, 2002; based on code by Matt J. Ingalls

New in version 4.22

# xscanmap

xscanmap -- Allows the position and velocity of a node in a scanned process to be read.

xscanmap

## Description

Allows the position and velocity of a node in a scanned process to be read.

## Syntax

`kpos, kvel xscanmap iscan, kamp, kvamp [, iwhich]`

## Initialization

*iscan* -- which scan process to read

*iwhich* (optional) -- which node to sense. The default is 0.

## Performance

*kamp* -- amount to amplify the *kpos* value.

*kvamp* -- amount to amplify the *kvel* value.

The internal state of a node is read. This includes its position and velocity. They are amplified by the *kamp* and *kvamp* values.

## Credits

Author: John ffitich

New in version 4.20

## xscansmap

xscansmap -- Allows the position and velocity of a node in a scanned process to be read.

xscansmap

### Description

Allows the position and velocity of a node in a scanned process to be read.

### Syntax

**xscansmap** *kpos*, *kvel*, *iscan*, *kamp*, *kvamp* [, *iwhich*]

### Initialization

*iscan* -- which scan process to read

*iwhich* (optional) -- which node to sense. The default is 0.

### Performance

*kpos* -- the node's position.

*kvel* -- the node's velocity.

*kamp* -- amount to amplify the *kpos* value.

*kvamp* -- amount to amplify the *kvel* value.

The internal state of a node is read. This includes its position and velocity. They are amplified by the *kamp* and *kvamp* values.

### Credits

New in version 4.21

November 2002. Thanks to Rasmus Ekman for pointing this opcode out.

## xscans

xscans -- Fast scanned synthesis waveform and the wavetable generator.

xscans

## Description

Experimental version of *scans*. Allows much larger matrices and is faster and smaller but removes some (unused?) flexibility. If liked, it will replace the older opcode as it is syntax compatible but extended.

## Syntax

ares **xscans** kamp, kfreq, ifntraj, id [, iorder]

## Initialization

*ifntraj* -- table 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* -- 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.

*iorder* (optional, default=0) -- 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

## Matrix Format

The new matrix format is a list of connections, one per line linking point x to point y. There is no weight given to the link; it is assumed to be unity. The list is proceeded by the line <MATRIX> and ends with a </MATRIX> line

For example, a circular string of 8 would be coded as

```
<MATRIX>
0 1
1 0
1 2
2 1
2 3
3 2
3 4
4 3
</MATRIX>
```



```
4 5
5 4
5 6
6 5
6 7
7 6
0 7
</MATRIX>
```

## Examples

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

## See Also

*scans*, *xscanu*

# xscanu

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

xscanu

## Description

Experimental version of *scanu*. Allows much larger matrices and is faster and smaller but removes some (unused?) flexibility. If liked, it will replace the older opcode as it is syntax compatible but extended.

## Syntax

**xscanu** init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, kstif, l

## 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  $\text{init} > 0$ , the length of it should be the same as the intended mass number, otherwise it can be anything.

*irate* -- update rate.

*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* --

- *either* an 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.
- *or* a string giving the name of a file in the MATRIX format

*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  $\text{init} < 0$ , the position of the left hammer ( $\text{ileft} = 0$  is hit at leftmost,  $\text{ileft} = 1$  is hit at rightmost).

*iright* -- If  $\text{init} < 0$ , the position of the right hammer ( $\text{iright} = 0$  is hit at leftmost,  $\text{iright} = 1$  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.

## Matrix Format

The new matrix format is a list of connections, one per line linking point *x* to point *y*. There is no weight given to the link; it is assumed to be unity. The list is preceded by the line `<MATRIX>` and ends with a `</MATRIX>` line

For example, a circular string of 8 would be coded as

```
<MATRIX>
0 1
1 0
1 2
2 1
2 3
3 2
3 4
4 3
4 5
5 4
5 6
6 5
6 7
7 6
0 7
</MATRIX>
```

## Examples

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

## See Also

*scanu*, *xscans*

# xtratim

xtratim -- Extend the duration of real-time generated events.

xtratim

## Description

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

## Syntax

**xtratim** iextradur

## Initialization

*iextradur* -- additional duration of current instrument instance

## Performance

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

This opcode is useful for implementing complex release-oriented envelopes.

## Examples

```
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
```

## See Also

*linenr, release*

## Credits

Author: Gabriel Maldonado  
Italy

New in Csound version 3.47

# xyin

xyin -- Sense the cursor position in an output window

xyin

## Description

Sense the cursor position in an output window. 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.

## Syntax

kx, ky **xyin** iprd, ixmin, ixmax, iymax [, ixinit] [, iyinit]

## 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.

## 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.

## Examples

Here is an example of the *xyin* opcode. It uses the files *xyin.orc* [examples/*xyin.orc*] and *xyin.sco* [examples/*xyin.sco*].

### Example 441. Example of the *xyin* opcode.

```
/* xyin.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Print and capture values every 0.1 seconds.
  iprd = 0.1
  ; The x values are from 1 to 30.
  ixmin = 1
  ixmax = 30
  ; The y values are from 1 to 30.
  iymin = 1
  iymax = 30
```

```
; The initial values for X and Y are both 15.
ixinit = 15
iyinit = 15

; Get the values kx and ky using the xyin opcode.
kx, ky xyin iprd, ixmin, ixmax, iymmin, iymax, ixinit, iyinit

; Print out the values of kx and ky.
printks "kx=%f, ky=%f\\n", iprd, kx, ky

; Play an oscillator, use the x values for amplitude and
; the y values for frequency.
kamp = kx * 1000
kcps = ky * 220
a1 oscil kamp, kcps, 1

out a1
endin
/* xyin.orc */

/* xyin.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 30 seconds.
i 1 0 30
e
/* xyin.sco */
```

As the values of kx and ky change, they will be printed out like this:

```
kx=8.612036, ky=22.677933
kx=10.765685, ky=15.644135
```

## Credits

Example written by Kevin Conder.

# zaci

*zaci* -- Clears one or more variables in the za space.

*zaci*

## Description

Clears one or more variables in the za space.

## Syntax

**zaci** *kfirst*, *klast*

## Performance

*kfirst* -- first zk or za location in the range to clear.

*klast* -- last zk or za location in the range to clear.

*zaci* 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

Here is an example of the *zaci* opcode. It uses the files *zaci.orc* [examples/*zaci.orc*] and *zaci.sco* [examples/*zaci.sco*].

### Example 442. Example of the *zaci* opcode.

```
/* zaci.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
  ; Generate a simple sine waveform.
  asin oscil 20000, 440, 1

  ; Send the sine waveform to za variable #1.
  zaw asin, 1
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read za variable #1.
  a1 zar 1

  ; Generate the audio output.
```



```
out a1

; Clear the za variables, get them ready for
; another pass.
zacl 0, 1
endin
/* zacl.orc */


/* zacl.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e
/* zacl.sco */
```

## See Also

*zamod, zar, zaw, zawm, ziw, ziwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# zakinit

zakinit -- Establishes zak space.

zakinit

## Description

Establishes zak space. Must be called only once.

## Syntax

**zakinit** *isizea*, *isizek*

## 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*.

## Examples

Here is an example of the *zakinit* opcode. It uses the files *zakinit.orc* [examples/zakinit.orc] and *zakinit.sco* [examples/zakinit.sco].

### Example 443. Example of the *zakinit* opcode.

```
/* zakinit.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 3 a-rate variables and 5 k-rate variables.
zakinit 3, 5

; Instrument #1 -- a simple waveform.
instr 1
; Generate a simple sine waveform.
asin oscil 20000, 440, 1

; Send the sine waveform to za variable #1.
zaw asin, 1
```

```
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read za variable #1.
  al zar 1

  ; Generate audio output.
  out al

  ; Clear the za variables, get them ready for
  ; another pass.
  zacl 0, 3
endin
/* zakinit.orc */
```

```
/* zakinit.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e
/* zakinit.sco */
```

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# zamod

zamod -- Modulates one a-rate signal by a second one.

zamod

## Description

Modulates one a-rate signal by a second one.

## Syntax

ares **zamod** asig, kzamod

## Performance

*asig* -- the input signal

*kzamod* -- controls which za 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 *asig*.

*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 *zkmmod*.

## Examples

Here is an example of the zamod opcode. It uses the files *zamod.orc* [examples/zamod.orc] and *zamod.sco* [examples/zamod.sco].

### Example 444. Example of the zamod opcode.

```
/* zamod.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 2 a-rate variables and 2 k-rate variables.
zakinit 2, 2

; Instrument #1 -- a simple waveform.
instr 1
  ; Vary an a-rate signal linearly from 20,000 to 0.
  asig line 20000, p3, 0

  ; Send the signal to za variable #1.
  zaw asig, 1
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Generate a simple sine wave.
  asin oscil 1, 440, 1
```

```
; Modify the sine wave, multiply its amplitude by
; za variable #1.
al zamod asin, -1

; Generate the audio output.
out al

; Clear the za variables, prepare them for
; another pass.
zacl 0, 2
endin
/* zamod.orc */

/* zamod.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 0 2
e
/* zamod.sco */
```

## See Also

*zacl, ziw, ziwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# zar

zar -- Reads from a location in za space at a-rate.

zir

## Description

Reads from a location in za space at a-rate.

## Syntax

ares **zar** kndx

## Performance

*kndx* -- points to the za location to be read.

*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.

## Examples

Here is an example of the zar opcode. It uses the files *zar.orc* [examples/zar.orc] and *zar.sco* [examples/zar.sco].

### Example 445. Example of the zar opcode.

```
/* zar.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
  ; Generate a simple sine waveform.
  asin oscil 20000, 440, 1

  ; Send the sine waveform to za variable #1.
  zaw asin, 1
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read za variable #1.
  a1 zar 1

  ; Generate audio output.
  out a1

  ; Clear the za variables, get them ready for
  ; another pass.
```

```
    zacl 0, 1
endin
/* zar.orc */

/* zar.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e
/* zar.sco */
```

## See Also

*zarg, zir, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# zarg

*zarg* -- Reads from a location in za space at a-rate, adds some gain.

*zarg*

## Description

Reads from a location in za space at a-rate, adds some gain.

## Syntax

ares **zarg** *kndx*, *kgain*

## Initialization

*kndx* -- points to the za location to be read.

*kgain* -- multiplier for the a-rate signal.

## Performance

*zarg* 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* also multiplies the a-rate signal by a k-rate value *kgain*.

## Examples

Here is an example of the *zarg* opcode. It uses the files *zarg.orc* [examples/zarg.orc] and *zarg.sco* [examples/zarg.sco].

### Example 446. Example of the *zarg* opcode.

```
/* zarg.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
  ; Generate a simple sine waveform, with an amplitude
  ; between 0 and 1.
  asin oscil 1, 440, 1

  ; Send the sine waveform to za variable #1.
  zaw asin, 1
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read za variable #1, multiply its amplitude by 20,000.
  a1 zarg 1, 20000
```



```
; Generate audio output.
out a1

; Clear the za variables, get them ready for
; another pass.
zacl 0, 1
endin
/* zarg.orc */

/* zarg.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e
/* zarg.sco */
```

## See Also

*zar, zir, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

## zaw

*zaw* -- Writes to a *za* variable at a-rate without mixing.

*zaw*

## Description

Writes to a *za* variable at a-rate without mixing.

## Syntax

**zaw** *asig*, *kndx*

## Performance

*asig* -- value to be written to the *za* location.

*kndx* -- points to the *zk* or *za* location to which to write.

*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.

## Examples

Here is an example of the *zaw* opcode. It uses the files *zaw.orc* [examples/zaw.orc] and *zaw.sco* [examples/zaw.sco].

### Example 447. Example of the *zaw* opcode.

```
/* zaw.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
; Generate a simple sine waveform.
asin oscil 20000, 440, 1

; Send the sine waveform to za variable #1.
zaw asin, 1
endin

; Instrument #2 -- generates audio output.
instr 2
; Read za variable #1.
a1 zar 1
```

```
; Generate the audio output.
out a1

; Clear the za variables, get them ready for
; another pass.
zacl 0, 1
endin
/* zaw.orc */

/* zaw.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e
/* zaw.sco */
```

## See Also

*zawm, ziw, ziwm, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

## zawm

`zawm` -- Writes to a `za` variable at a-rate with mixing.

`zawm`

## Description

Writes to a `za` variable at a-rate with mixing.

## Syntax

**zawm** *asig*, *kndx* [, *imix*]

## Initialization

*imix* (optional, default=1) -- indicates if mixing should occur.

## Performance

*asig* -- value to be written to the `za` location.

*kndx* -- points to the `zk` or `za` location to which to write.

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.

`zawm` is a mixing opcode, it adds the signal to the current value of the variable. If no *imix* is specified, mixing always occurs. *imix* = 0 will cause overwriting like `ziw`, `zkw`, and `zaw`. 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

Here is an example of the `zawm` opcode. It uses the files `zawm.orc` [examples/zawm.orc] and `zawm.sco` [examples/zawm.sco].

### Example 448. Example of the `zawm` opcode.

```
/* zawm.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1
```

```
; Instrument #1 -- a basic instrument.
instr 1
; Generate a simple sine waveform.
asin oscil 15000, 440, 1

; Mix the sine waveform with za variable #1.
zawm asin, 1
endin

; Instrument #2 -- another basic instrument.
instr 2
; Generate another waveform with a different frequency.
asin oscil 15000, 880, 1

; Mix this sine waveform with za variable #1.
zawm asin, 1
endin

; Instrument #3 -- generates audio output.
instr 3
; Read za variable #1, containing both waveforms.
al zar 1

; Generate the audio output.
out al

; Clear the za variables, get them ready for
; another pass.
zacl 0, 1
endin
/* zawm.orc */

/* zawm.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
; Play Instrument #3 for one second.
i 3 0 1
e
/* zawm.sco */
```

## See Also

*zaw, ziw, ziwm, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

## zfilter2

`zfilter2` -- Performs filtering using a transposed form-II digital filter lattice with radial pole-shearing and angular pole-warping.

`zfilter2`

## Description

General purpose custom filter with time-varying pole control. The filter coefficients implement the following difference equation:

$$(1)*y(n) = b0*x[n] + b1*x[n-1] + \dots + bM*x[n-M] - a1*y[n-1] - \dots - aN*y[n-N]$$

the system function for which is represented by:

$$H(Z) = \frac{B(Z)}{A(Z)} = \frac{b0 + b1*Z^{-1} + \dots + bM*Z^{-M}}{1 + a1*Z^{-1} + \dots + aN*Z^{-N}}$$

## Syntax

`ares zfilter2 asig, kdamp, kfreq, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN`

## 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.

## Examples

A controllable second-order IIR filter operating on an a-rate signal:

```
a1 zfilter2 asig, kdamp, kfreq, 1, 2, 1, ia1, ia2 ; controllable a-rate ; IIR
```

## See Also

*filter2*

## Credits

Author: Michael A. Casey  
M.I.T.  
Cambridge, Mass.  
1997

# zir

zir -- Reads from a location in zk space at i-rate.

zir

## Description

Reads from a location in zk space at i-rate.

## Syntax

ir **zir** indx

## Initialization

*indx* -- points to the zk location to be read.

## Performance

*zir* reads the signal at *indx* location in zk space.

## Examples

Here is an example of the zir opcode. It uses the files *zir.orc* [examples/zir.orc] and *zir.sco* [examples/zir.sco].

### Example 449. Example of the zir opcode.

```
/* zir.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple instrument.
instr 1
; Set the zk variable #1 to 32.594.
ziw 32.594, 1
endin

; Instrument #2 -- prints out zk variable #1.
instr 2
; Read the zk variable #1 at i-rate.
il zir 1

; Print out the value of zk variable #1.
print il
endin
/* zir.orc */
```



```
/* zir.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
; Play Instrument #2 for one second.  
i 2 0 1  
e  
/* zir.sco */
```

## See Also

*zar, zarg, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

## ziw

ziw -- Writes to a zk variable at i-rate without mixing.

ziw

## Description

Writes to a zk variable at i-rate without mixing.

## Syntax

**ziw** *isig*, *indx*

## Initialization

*isig* -- initializes the value of the zk location.

*indx* -- points to the zk or za location to which to write.

## Performance

*ziw* writes *isig* into the zk variable specified by *indx*.

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.

## Examples

Here is an example of the *ziw* opcode. It uses the files *ziw.orc* [examples/ziw.orc] and *ziw.sco* [examples/ziw.sco].

### Example 450. Example of the *ziw* opcode.

```
/* ziw.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple instrument.
instr 1
  ; Set zk variable #1 to 64.182.
  ziw 64.182, 1
endin

; Instrument #2 -- prints out zk variable #1.
instr 2
  ; Read zk variable #1 at i-rate.
  il zir 1
```

```
    ; Print out the value of zk variable #1.  
    print i1  
endin  
/* ziw.orc */
```

```
/* ziw.sco */  
; Play Instrument #1 for one second.  
i 1 0 1  
; Play Instrument #2 for one second.  
i 2 0 1  
e  
/* ziw.sco */
```

## See Also

*zaw, zawm, ziwm, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# ziwm

ziwm -- Writes to a zk variable to an i-rate variable with mixing.

ziwm

## Description

Writes to a zk variable to an i-rate variable with mixing.

## Syntax

```
ziwm isig, indx [, imix]
```

## Initialization

*isig* -- initializes the value of the zk location.

*indx* -- points to the zk location location to which to write.

*imix* (optional, default=1) -- indicates if mixing should occur.

## Performance

*ziwm* is a mixing opcode, it adds the signal to the current value of the variable. If no *imix* is specified, mixing always occurs. *imix* = 0 will cause overwriting like *ziw*, *zkw*, and *zaw*. 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

Here is an example of the *ziwm* opcode. It uses the files *ziwm.orc* [examples/ziwm.orc] and *ziwm.sco* [examples/ziwm.sco].

### Example 451. Example of the *ziwm* opcode.

```
/* ziwm.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple instrument.
instr 1
    ; Add 20.5 to zk variable #1.
```

```
    ziwm 20.5, 1
  endin

; Instrument #2 -- another simple instrument.
instr 2
  ; Add 15.25 to zk variable #1.
  ziwm 15.25, 1
  endin

; Instrument #3 -- prints out zk variable #1.
instr 3
  ; Read zk variable #1 at i-rate.
  il zir 1

  ; Print out the value of zk variable #1.
  ; It should be 35.75 (20.5 + 15.25)
  print il
  endin
/* ziwm.orc */

/* ziwm.sco */
; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
; Play Instrument #3 for one second.
i 3 0 1
e
/* ziwm.sco */
```

## See Also

*zaw, zawm, ziw, zkw, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# zkcl

zkcl -- Clears one or more variables in the zk space.

zkcl

## Description

Clears one or more variables in the zk space.

## Syntax

**zkcl** kfirst, klast

## Performance

*ksig* -- the input signal

*kfirst* -- first zk or za location in the range to clear.

*klast* -- last zk or za location in the range to clear.

*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.

## Examples

Here is an example of the zkcl opcode. It uses the files *zkcl.orc* [examples/zkcl.orc] and *zkcl.sco* [examples/zkcl.sco].

### Example 452. Example of the zkcl opcode.

```
/* zkcl.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
  ; Linearly vary a k-rate signal from 220 to 1760.
  kline line 220, p3, 1760

  ; Add the linear signal to zk variable #1.
  zkw kline, 1
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read zk variable #1.
  kfreq zkr 1
```

```
; Use the value of zk variable #1 to vary
; the frequency of a sine waveform.
a1 oscil 20000, kfreq, 1

; Generate the audio output.
out a1

; Clear the zk variables, get them ready for
; another pass.
zkcl 0, 1
endin
/* zkcl.orc */

/* zkcl.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for three seconds.
i 1 0 3
; Play Instrument #2 for three seconds.
i 2 0 3
e
/* zkcl.sco */
```

## See Also

*zacr, zkwm, zkw, zkmod, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# zkmod

zkmod -- Facilitates the modulation of one signal by another.

zkmod

## Description

Facilitates the modulation of one signal by another.

## Syntax

kres **zkmod** ksig, kzkmod

## 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

*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.

## Examples

Here is an example of the zkmod opcode. It uses the files *zkmod.orc* [examples/zkmod.orc] and *zkmod.sco* [examples/zkmod.sco].

### Example 453. Example of the zkmod opcode.

```
/* zkmod.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 2

; Initialize the ZAK space.
; Create 2 a-rate variables and 2 k-rate variables.
zakinit 2, 2

; Instrument #1 -- a signal with jitter.
instr 1
; Generate a k-rate signal goes from 30 to 2,000.
kline line 30, p3, 2000

; Add the signal into zk variable #1.
zkw kline, 1
endin

; Instrument #2 -- generates audio output.
instr 2
; Create a k-rate signal modulated the jitter opcode.
kamp init 20
kcpsmin init 40
```



```
kcpsmax init 60
kjtr jitter kamp, kcpsmin, kcpsmax

; Get the frequency values from zk variable #1.
kfreq zkr 1
; Add the the frequency values in zk variable #1 to
; the jitter signal.
kjfreq zkmod kjtr, 1

; Use a simple sine waveform for the left speaker.
aleft oscil 20000, kfreq, 1
; Use a sine waveform with jitter for the right speaker.
aright oscil 20000, kjfreq, 1

; Generate the audio output.
outs aleft, aright

; Clear the zk variables, prepare them for
; another pass.
zkcl 0, 2
endin
/* zkmod.orc */

/* zkmod.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
; Play Instrument #2 for 2 seconds.
i 2 0 2
e
/* zkmod.sco */
```

## See Also

*zamod, zkcl, zkr, zkwm, zkw*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

## zkr

*zkr* -- Reads from a location in zk space at k-rate.

*zkr*

## Description

Reads from a location in zk space at k-rate.

## Syntax

kres **zkr** kndx

## Initialization

*kndx* -- points to the zk location to be read.

## Performance

*zkr* reads the array of floats at *kndx* in zk space.

## Examples

Here is an example of the *zkr* opcode. It uses the files *zkr.orc* [examples/zkr.orc] and *zkr.sco* [examples/zkr.sco].

### Example 454. Example of the *zkr* opcode.

```
/* zkr.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
  ; Linearly vary a k-rate signal from 440 to 880.
  kline line 440, p3, 880

  ; Add the linear signal to zk variable #1.
  zkw kline, 1
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read zk variable #1.
  kfreq zkr 1

  ; Use the value of zk variable #1 to vary
  ; the frequency of a sine waveform.
  a1 oscil 20000, kfreq, 1
```

```
; Generate the audio output.
out al

; Clear the zk variables, get them ready for
; another pass.
zkcl 0, 1
endin
/* zkr.orc */

/* zkr.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for one second.
i 1 0 1
; Play Instrument #2 for one second.
i 2 0 1
e
/* zkr.sco */
```

## See Also

*zar, zarg, zir, zkcl, zkmod, zkwm, zkw*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

## zkw

**zkw** -- Writes to a zk variable at k-rate without mixing.

**zkw**

## Description

Writes to a zk variable at k-rate without mixing.

## Syntax

**zkw** *ksig*, *kndx*

## Performance

*ksig* -- value to be written to the zk location.

*kndx* -- points to the zk or za location to which to write.

**zkw** writes *ksig* into the zk variable specified by *kndx*.

## Examples

Here is an example of the **zkw** opcode. It uses the files *zkw.orc* [examples/zkw.orc] and *zkw.sco* [examples/zkw.sco].

### Example 455. Example of the **zkw** opcode.

```
/* zkw.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a simple waveform.
instr 1
  ; Linearly vary a k-rate signal from 100 to 1,000.
  kline line 100, p3, 1000

  ; Add the linear signal to zk variable #1.
  zkw kline, 1
endin

; Instrument #2 -- generates audio output.
instr 2
  ; Read zk variable #1.
  kfreq zkr 1

  ; Use the value of zk variable #1 to vary
  ; the frequency of a sine waveform.
  a1 oscil 20000, kfreq, 1
```

```
; Generate the audio output.
out a1

; Clear the zk variables, get them ready for
; another pass.
zkcl 0, 1
endin
/* zkw.orc */

/* zkw.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
; Play Instrument #2 for two seconds.
i 2 0 2
e
/* zkw.sco */
```

## See Also

*zaw, zawm, ziw, ziwm, zkr, zkwm*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.

# zkwm

zkwm -- Writes to a zk variable at k-rate with mixing.

zkwm

## Description

Writes to a zk variable at k-rate with mixing.

## Syntax

**zkwm** ksig, kndx [, imix]

## Initialization

*imix* (optional) -- points to the zk location location to which to write.

## Performance

*ksig* -- value to be written to the zk location.

*kndx* -- points to the zk or za location to which to write.

*zkwm* is a mixing opcode, it adds the signal to the current value of the variable. If no *imix* is specified, mixing always occurs. *imix* = 0 will cause overwriting like *ziw*, *zkw*, and *zaw*. 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

Here is an example of the *zkwm* opcode. It uses the files *zkwm.orc* [examples/zkwm.orc] and *zkwm.sco* [examples/zkwm.sco].

### Example 456. Example of the zkwm opcode.

```
/* zkwm.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Initialize the ZAK space.
; Create 1 a-rate variable and 1 k-rate variable.
zakinit 1, 1

; Instrument #1 -- a basic instrument.
instr 1
    ; Generate a k-rate signal.
```

```
; The signal goes from 30 to 20,000 then back to 30.
kramp linseg 30, p3/2, 20000, p3/2, 30

; Mix the signal into the zk variable #1.
zkwm kramp, 1
endin

; Instrument #2 -- another basic instrument.
instr 2
; Generate another k-rate signal.
; This is a low frequency oscillator.
klfo lfo 3500, 2

; Mix this signal into the zk variable #1.
zkwm klfo, 1
endin

; Instrument #3 -- generates audio output.
instr 3
; Read zk variable #1, containing a mix of both signals.
kamp zkr 1

; Create a sine waveform. Its amplitude will vary
; according to the values in zk variable #1.
a1 oscil kamp, 880, 1

; Generate the audio output.
out a1

; Clear the zk variable, get it ready for
; another pass.
zkcl 0, 1
endin
/* zkwm.orc */

/* zkwm.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; Play Instrument #1 for 5 seconds.
i 1 0 5
; Play Instrument #2 for 5 seconds.
i 2 0 5
; Play Instrument #3 for 5 seconds.
i 3 0 5
e
/* zkwm.sco */
```

## See Also

*zaw, zawm, ziw, ziwm, zkcl, zkw, zkr*

## Credits

Author: Robin Whittle  
Australia  
May 1997

Example written by Kevin Conder.



---

# **Score Statements and GEN Routines**

## **Score Statements**

## a Statement (or Advance Statement)

a Statement (or Advance Statement) -- Advance score time by a specified amount.

a

### Description

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

### Syntax

a p1 p2 p3

### Performance

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.

## b Statement

**b** Statement -- This statement resets the clock.

**b** Statement

### Description

This statement resets the clock.

### Syntax

**b** p1

### Performance

*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*.

### Examples

```
i1      0      2
i1      10     888

b 5
i2      1      1      440      ; set the clock "forward"
i2      2      1      480      ; start time = 6
                                   ; start time = 7

b -1
i3      3      2      3.1415    ; set the clock back
i3      5.5    1      1.1111    ; start time = 2
                                   ; start time = 4.5

b 0
i4      10     200    7          ; reset clock to normal
                                   ; start time = 10
```

### Credits

Explanation suggested and example provided by Paul Winkler. (Csound Version 4.07)

## e Statement

*e* Statement -- This statement may be used to mark the end of the last section of the score.

*e* statement

### Description

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

### Syntax

*e* anything

### Performance

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.

## f Statement (or Function Table Statement)

f Statement (or Function Table Statement) -- Causes a GEN subroutine to place values in a stored function table.

f Statement (or Function Table Statement)

### Description

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

### Syntax

**f** p1 p2 p3 p4 ...

### Performance

*p1* -- Table number 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*, *oscill*, *oscilli*, etc.

Table space is allocated in primary memory, along with instrument data space. The maximum table number used to be 200. This has been changed to be limited by memory only. (Currently there is an internal soft limit of 300, this is automatically extended as 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*).

### See also

*GEN ROUTINES*

## Credits

Updated August 2002 thanks to a note from Rasmus Ekman. There is no longer a hard limit of 200 function tables.

## i Statement (Instrument or Note Statement)

i Statement (Instrument or Note Statement) -- Makes an instrument active at a specific time and for a certain duration.

i

### 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.

### Syntax

i p1 p2 p3 p4 ...

### Initialization

*p1* -- Instrument number, usually a non-negative integer. An optional fractional part can provide an additional tag for specifying ties between particular notes of consecutive clusters. A negative *p1* (including tag) can be used to turn off a particular “held” note.

*p2* -- Starting time in arbitrary units called beats.

*p3* -- Duration time in beats (usually positive). A negative value will initiate a held note (see also *ihold*). A zero value will invoke an initialization pass without performance (see also *instr*).

*p4* ... -- Parameters whose significance is determined by the instrument.

### Performance

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 *linsegs*, 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* opcode 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.

## Special Considerations

The maximum instrument number used to be 200. This has been changed to be limited by memory only (currently there is an internal soft limit of 200; this is automatically extended as required).

## Examples

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 e
    iportime  init      abs(p3)/7          ; Portamento time dep on note
    iamp0     init      p5                  ; Set default amps
    iamp1     init      p5
    iamp2     init      p5
```



```

    itie      tival
    if itie == 1      igoto nofadein
    iamp0      init      0
                                ; Check if this note is tied,
                                ; if not fade in

nofadein:
    if p3 < 0      igoto nofadeout
    iamp2      init      0
                                ; Check if this note is held,

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
    kcps      port      icps, iportime, icps
                                ; Init pitch for untied note
                                ; Drift towards target pitch

    kpw      oscil      .4, rnd(1), 1, rnd(.7)
    ar      vco      kamp, kcps, 3, kpw+.5, 1, 1/icps
                                ; A simple triangle-saw oscil

    ; (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 t

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

```

## Credits

Additional text (Csound Version 4.07) explaining tied notes, edited by Rasmus Ekman from a note by David Kirsh, posted to the Csound mailing list. Example instrument by Rasmus Ekman.

Updated August 2002 thanks to a note from Rasmus Ekman. There is no longer a hard limit of 200 instruments.

## m Statement (Mark Statement)

m Statement (Mark Statement) -- Sets a named mark in the score.

m

### Description

Sets a named mark in the score, which can be used by an *n statement*.

### Syntax

**m** p1

### Initialization

*p1* -- Name of mark.

### Performance

This can be helpful in setting a up verse and chorus structure in the score. Names may contain letters and numerals.

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April, 1998

New in Csound version 3.48

## **n Statement**

`n Statement` -- Repeats a section.

`n`

### **Description**

Repeats a section from the referenced *m statement*.

### **Syntax**

`n p1`

### **Initialization**

*p1* -- Name of mark to repeat.

### **Performance**

This can be helpful in setting a up verse and chorus structure in the score. Names may contain letters and numerals.

### **Credits**

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April 1998

New in Csound version 3.48

## q Statement

q Statement -- This statement may be used to quiet an instrument.

q statement

### Description

This statement may be used to quiet an instrument.

### Syntax

q *p1* *p2* *p3*

### Performance

*p1* -- Instrument number to mute/unmute.

*p2* -- Action time of function generation (or destruction) in beats.

*p3* -- determines whether the instrument is muted/unmuted. The value of 0 means the instrument is muted, other values mean it is unmuted.

Note that this does not affect instruments that are already running at time *p2*. It blocks any attempt to start one afterwards.

## r Statement (Repeat Statement)

r Statement (Repeat Statement) -- Starts a repeated section.

r

### Description

Starts a repeated section, which lasts until the next *s*, *r* or *e statement*.

### Syntax

**r** p1 p2

### Initialization

*p1* -- Number of times to repeat the section.

*p2* -- Macro(name) to advance with each repetition (optional).

### Performance

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.

### Examples

Here is an example of the *r* statement. It uses the files *r.orc* [examples/r.orc] and *r.sco* [examples/r.sco].

#### Example 1. Example of the *r* statement.

```
/* r.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; The score's p4 parameter has the number of repeats.
kreps = p4
; The score's p5 parameter has our note's frequency.
kcps = p5

; Print the number of repeats.
printks "Repeated %i time(s).\n", 1, kreps

; Generate a nice beep.
a1 oscil 20000, kcps, 1
```

```
    out a1
  endin
/* r.orc */

/* r.sco */
; Table #1, a sine wave.
f 1 0 16384 10 1

; We'll repeat this section 6 times. Each time it
; is repeated, its macro REPS_MACRO is incremented.
r6 REPS_MACRO

; Play Instrument #1.
; p4 = the r statement's macro, REPS_MACRO.
; p5 = the frequency in cycles per second.
i 1 00.10 00.10 $REPS_MACRO 1760
i 1 00.30 00.10 $REPS_MACRO 880
i 1 00.50 00.10 $REPS_MACRO 440
i 1 00.70 00.10 $REPS_MACRO 220

; Marks the end of the section.
s

e
/* r.sco */
```

## Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK  
April, 1998

New in Csound version 3.48

Example written by Kevin Conder

## s Statement

s Statement -- Marks the end of a section.

s

### Description

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

### Syntax

**s** anything

### Initialization

All p-fields are ignored.

### Performance

Sorting of the *i statement*, *f statement* and *a statement* 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.

## t Statement (Tempo Statement)

t Statement (Tempo Statement) -- Sets the tempo.

t

### Description

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

### Syntax

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

### Initialization

*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.

### Performance

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.



## v Statement

*v* Statement -- Provides for locally variable time warping of score events.

*v*

### Description

The *v statement* provides for locally variable time warping of score events.

### Syntax

**v** p1

### Initialization

**p1** -- Time warp factor (must be positive).

### Performance

The *v statement* takes effect with the following *i statement*, and remains in effect until the next *v statement*, *s statement*, 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 are unaffected by the *v statement* (see *Carry*).

```
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.

## x Statement

x Statement -- Skip the rest of the current section.

x

## Description

This statement may be used to skip the rest of the current section.

## Syntax

**x** anything

## Initialization

All pfields are ignored.

# GEN Routines

## Sine/Cosine Generators:

- *GEN09* - Composite waveforms made up of weighted sums of simple sinusoids.
- *GEN10* - Composite waveforms made up of weighted sums of simple sinusoids.
- *GEN11* - Additive set of cosine partials.
- *GEN19* - Composite waveforms made up of weighted sums of simple sinusoids.
- *GEN30* - Generates harmonic partials by analyzing an existing table.
- *GEN33* - Generate composite waveforms by mixing simple sinusoids.
- *GEN34* - Generate composite waveforms by mixing simple sinusoids.

## Line/Exponential Segment Generators:

- *GEN05* - Constructs functions from segments of exponential curves.
- *GEN06* - Generates a function comprised of segments of cubic polynomials.
- *GEN07* - Constructs functions from segments of straight lines.
- *GEN08* - Generate a piecewise cubic spline curve.
- *GEN16* - Creates a table from a starting value to an ending value.
- *GEN25* - Construct functions from segments of exponential curves in breakpoint fashion.
- *GEN27* - Construct functions from segments of straight lines in breakpoint fashion.

## File Access GEN Routines:

- *GEN01* - Transfers data from a soundfile into a function table.
- *GEN23* - Reads numeric values from a text file.
- *GEN28* - Reads a text file which contains a time-tagged trajectory.

## Numeric Value Access GEN Routines

- *GEN02* - Transfers data from immediate pfields into a function table.
- *GEN17* - Creates a step function from given x-y pairs.

## Window Function GEN Routines

- *GEN20* - Generates functions of different windows.

## Random Function GEN Routines

- *GEN21* - Generates tables of different random distributions.
- *GEN40* - Generates a random distribution using a distribution histogram.
- *GEN41* - Generates a random list of numerical pairs.
- *GEN42* - Generates a random distribution of discrete ranges of values.
- *GEN43* - Loads a PVOCEX file containing a PV analysis.

## Waveshaping GEN Routines

- *GEN03* - Generates a stored function table by evaluating a polynomial.
- *GEN13* - Stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind.
- *GEN14* - Stores a polynomial whose coefficients derive from Chebyshevs of the second kind.
- *GEN15* - Creates two tables of stored polynomial functions.

## Amplitude Scaling GEN Routines

- *GEN04* - Generates a normalizing function.
- *GEN12* - Generates the log of a modified Bessel function of the second kind.
- *GEN24* - Reads numeric values from another allocated function-table and rescales them.

## Mixing GEN Routines

- *GEN18* - Writes composite waveforms made up of pre-existing waveforms.
- *GEN31* - Mixes any waveform specified in an existing table.
- *GEN32* - Mixes any waveform, resampled with either FFT or linear interpolation.

## GEN01

GEN01 -- Transfers data from a soundfile into a function table.

GEN01

### Description

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

### Syntax

```
f# time size 1 filcod skiptime format channel
```

### Performance

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

*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.

*format* -- specifies the audio data-file format:

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

If *format* = 0 the sample format is taken from the soundfile header, or by default from the CSound -o command-line flag.



### Note

- Reading stops at end-of-file or when the table is full. Table locations not filled will contain zeros.
- If p4 is positive, the table will be post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.

### Examples

Here is a simple example of the GEN01 routine. It uses the files *gen01.orc* [examples/gen01.orc], *gen01.sco* [examples/gen01.sco], and *beats.wav* [examples/beats.wav]. It uses the audio file “beats.wav”, here is its diagram:



Diagram of the waveform generated by GEN01.

### Example 2. A simple example of the GEN01 routine.

```
/* gen01.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
    kamp = 30000
    kcps = 1
    ifn = 1
    ibas = 1

    ; Play the audio sample stored in Table #1.
    a1 loscil kamp, kcps, ifn, ibas
    out a1
endin
/* gen01.orc */
```

```
/* gen01.sco */
; Table #1: read an audio file (using GEN01).
f 1 0 131072 1 "beats.wav" 0 4 0

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen01.sco */
```

Here is another example of the GEN01 routine. Csound will automatically compute the tablesizes because we have set it to 0. This example uses the files *gen01computed.orc* [examples/gen01computed.orc], *gen01computed.sco* [examples/gen01computed.sco], and *beats.wav* [examples/beats.wav].

### Example 3. An example of the GEN01 routine with a computed tablesizes.

```
/* gen01computed.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
    kamp = 30000
```

```
kcps = 1
ifn = 1
ibas = 1

; Play the audio sample stored in Table #1.
a1 loscil kamp, kcps, ifn, ibas
out a1
endin
/* gen01computed.orc */

/* gen01computed.sco */
; Table #1: an audio file (using GEN01).
; Since our table size is 0, Csound will compute it.
f 1 0 0 1 "beats.wav" 0 0 0

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen01computed.sco */
```

## Credits

Examples written by Kevin Conder

December 2002. Thanks goes to Kanata Motohashi for fixing mistakes in the examples.

September 2003. Thanks goes to Dr. Richard Boulanger for pointing out the references to the AIFF file format. GEN01 also works with WAV files.



## GEN02

GEN02 -- Transfers data from immediate pfields into a function table.

GEN02

### Description

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

### Syntax

```
f # time size 2 v1 v2 v3 ...
```

### 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*, etc. -- 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 1000). 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.

### Examples

Here is a simple example of the GEN02 routine. It uses the files *gen02.orc* [examples/gen02.orc] and *gen02.sco* [examples/gen02.sco]. It places 12 values plus an explicit wrap-around guard value into a table of size next-highest power of 2. Rescaling is inhibited. Here is its diagram:



Diagram of the waveform generated by GEN02.

#### Example 4. A simple example of the GEN02 routine.

```
/* gen02.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
```

```
kamp tablei kndx, ifn, ixmode

; Create a sine wave, use the Table #1 values to control
; the amplitude. This creates a sound with a long attack.
a1 oscil kamp*30000, 440, 2
out a1
endin
/* gen02.orc */


/* gen02.sco */
; Table #1: an envelope with a long attack (using GEN02).
f 1 0 16 2 0 1 2 3 4 5 6 7 8 9 10 11 0
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen02.sco */
```

## See Also

*GEN17*

## Credits

December 2002. Thanks to Rasmus Ekman, corrected the limit of the *PMAX* variable.

## GEN03

GEN03 -- Generates a stored function table by evaluating a polynomial.

GEN03

### Description

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

### Syntax

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

### 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 1)th stored value respectively in the generated function table.

*c0*, *c1*, *c2*, ... *cn* -- coefficients of the  $n$ th-order polynomial

$c0 + c1x + c2x^2 + \dots + cnx^n$

Coefficients may be positive or negative real numbers; a zero denotes a missing term in the polynomial. The coefficient list begins in *p7*, providing a current upper limit of 144 terms.



### Note

- The defined segment  $[fn(xval1), fn(xval2)]$  is evenly distributed. Thus a 512-point table over the interval  $[-1,1]$  will have its origin at location 257 (at the start of the 2nd half). Provided the extended guard point is requested, both  $fn(-1)$  and  $fn(1)$  will exist in the table.
- *GEN03* is useful in conjunction with *table* or *tablei* for audio waveshaping (sound modification by non-linear distortion). Coefficients to produce a particular formant from a sinusoidal lookup index of known amplitude can be determined at preprocessing time using algorithms such as Chebyshev formulae. See also *GEN13*.

### Examples

Here is a simple example of the GEN03 routine. It uses the files *gen03.orc* [examples/gen03.orc] and *gen03.sco* [examples/gen03.sco]. It fills 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. Here is its diagram:



Diagram of the waveform generated by GEN03.

**Example 5. A simple example of the GEN03 routine.**

```
/* gen03.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kamp table kndx, ifn, ixmode

; Create a sine wave, use the Table #1 values to control
; the amplitude.
a1 oscil kamp*30000, 440, 2
out a1
endin
/* gen03.orc */

/* gen03.sco */
; Table #1: a polynomial function (using GEN03).
f 1 0 1025 3 -1 1 5 4 3 2 2 1
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen03.sco */
```

**See Also**

*GEN13*, *GEN14*, and *GEN15*.

## GEN04

GEN04 -- Generates a normalizing function.

GEN04

### Description

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

### Syntax

```
f # time size 4 source# sourcemode
```

### 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  $1/(\text{absolute maximum so far scanned})$ . Stored values will thus begin with  $1/(\text{first value scanned})$ , then get progressively smaller as new maxima are encountered. For a source table which is normalized (values  $\leq 1$ ), the derived values will range from  $1/(\text{first value scanned})$  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.

### Examples

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

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

## GEN05

GEN05 -- Constructs functions from segments of exponential curves.

GEN05

### Description

Constructs functions from segments of exponential curves.

### Syntax

```
f # time size 5 a n1 b n2 c ...
```

### 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, . . . These must be nonzero and must be alike in sign.

*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  $n1 + n2 + \dots$  will normally equal *size* for fully specified functions. If the sum is smaller, the function locations not included will be set to zero; if the sum is greater, only the first *size* locations will be stored.



### Note

- If p4 is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.
- Discrete-point linear interpolation implies an increase or decrease along a segment by equal differences between adjacent locations; exponential interpolation implies that the progression is by equal ratio. In both forms the interpolation from *a* to *b* is such as to assume that the value *b* will be attained in the  $n + 1$ th location. For discontinuous functions, and for the segment encompassing the end location, this value will not actually be reached, although it may eventually appear as a result of final scaling.

### Examples

Here is a simple example of the GEN05 routine. It uses the files *gen05.orc* [examples/gen05.orc] and *gen05.sco* [examples/gen05.sco]. It will create a nice percussive amplitude envelope. Here is its diagram:



Diagram of the waveform generated by GEN05.

#### Example 6. A simple example of the GEN05 routine.

```
/* gen05.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  ; Create an index over the length of our entire note.
  kcps init 1/p3
  kndx phasor kcps

  ; Read Table #1 with our index.
  ifn = 1
  ixmode = 1
  kamp table kndx, ifn, ixmode

  ; Create a sine wave, use the Table #1 values to control
  ; the amplitude. This creates a nice percussive sound.
  a1 oscil kamp*30000, 440, 2
  out a1
endin
/* gen05.orc */

/* gen05.sco */
; Table #1: a percussive envelope (using GEN05).
f 1 0 64 5 1 2 120 60 1 1 0.001 1
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen05.sco */
```

## See Also

*GEN06*, *GEN07*, and *GEN08*

## Credits

Example written by Kevin Conder

## GEN06

GEN06 -- Generates a function comprised of segments of cubic polynomials.

GEN06

### Description

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

### Syntax

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

### Initialization

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

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

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

*n1, n2, n3 ...* -- number of stored values between specified points. Cannot be negative, but a zero is meaningful for specifying discontinuities. The sum  $n1 + n2 + \dots$  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.

### Examples

Here is a simple example of the GEN06 routine. It uses the files *gen06.orc* [examples/gen06.orc] and *gen06.sco* [examples/gen06.sco]. It 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. Here is its diagram:



Diagram of the waveform generated by GEN06.

#### Example 7. A simple example of the GEN06 routine.



```
/* gen06.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kval table kndx, ifn, ixmode

; Generate a sine waveform, use our Table #1 value to
; vary its frequency by 100 Hz from its base frequency.
ibasefreq = 440
kfreq = kval * 100
a1 oscil 20000, ibasefreq + kfreq, 2
out a1
endin
/* gen06.orc */

/* gen06.sco */
; Table #1: a curve (using GEN06).
f 1 0 65 6 0 16 0.5 16 1 16 0 16 -1
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen06.sco */
```

## See Also

*GEN05*, *GEN07*, and *GEN08*

## GEN07

GEN07 -- Constructs functions from segments of straight lines.

GEN07

### Description

Constructs functions from segments of straight lines.

### Syntax

```
f #      time      size  7  a  n1  b  n2  c  ...
```

### 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, . . .

*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  $n1 + n2 + \dots$  will normally equal *size* for fully specified functions. If the sum is smaller, the function locations not included will be set to zero; if the sum is greater, only the first *size* locations will be stored.



### Note

- If p4 is positive, functions are post-normalized (rescaled to a maximum absolute value of 1 after generation). A negative p4 will cause rescaling to be skipped.
- Discrete-point linear interpolation implies an increase or decrease along a segment by equal differences between adjacent locations; exponential interpolation implies that the progression is by equal ratio. In both forms the interpolation from *a* to *b* is such as to assume that the value *b* will be attained in the  $n + 1$ th location. For discontinuous functions, and for the segment encompassing the end location, this value will not actually be reached, although it may eventually appear as a result of final scaling.

### Examples

Here is a simple example of the GEN07 routine. It uses the files *gen07.orc* [examples/gen07.orc] and *gen07.sco* [examples/gen07.sco]. It will create a single-cycle sawtooth whose discontinuity is mid-way in the stored function. Here is its diagram:



Diagram of the waveform generated by GEN07.

#### Example 8. A simple example of the GEN07 routine.

```
/* gen07.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  kcps = 440
  ifn = 1

  ; Play the sine wave stored in Table #1.
  al oscil kamp, kcps, ifn
  out al
endin
/* gen07.orc */

/* gen07.sco */
; Table #1: a sawtooth wave (using GEN07).
f 1 0 256 7 0 128 1 0 -1 128 0

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen07.sco */
```

## See Also

*GEN05*, *GEN06*, and *GEN08*

## GEN08

GEN08 -- Generate a piecewise cubic spline curve.

GEN08

### Description

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

### Syntax

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

### 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 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  $n1 + n2 + \dots$  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

Here is a simple example of the GEN08 routine. It uses the files *gen08.orc* [examples/gen08.orc] and *gen08.sco* [examples/gen08.sco]. It will create a curve with a smooth hump in the middle, going briefly negative outside the hump then flat at its ends. Here is its diagram:



Diagram of the waveform generated by GEN08.

#### Example 9. A simple example of the GEN08 routine.

```
/* gen08.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kval table kndx, ifn, ixmode

; Generate a sine waveform, use our Table #1 value to
; vary its frequency by 100 Hz from its base frequency.
ibasefreq = 440
kfreq = kval * 100
a1 oscil 20000, ibasefreq + kfreq, 2
out a1
endin
/* gen08.orc */

/* gen08.sco */
; Table #1: a curve with a smooth hump (using GEN08).
f 1 0 65 8 0 16 0 16 1 16 0 16 0
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for two seconds.
i 1 0 2
e
/* gen08.sco */
```

## See Also

*GEN05*, *GEN06*, and *GEN07*

## GEN09

GEN09 -- Generate composite waveforms made up of weighted sums of simple sinusoids.

GEN09

### Description

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 3 p-fields using *GEN09*.

### Syntax

```
f # time size 9 pna stra phsa pnb strb phsb ...
```

### 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. Partial 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 (0-360).



### 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

Here is a simple example of the GEN09 routine. It uses the files *gen09.orc* [examples/gen09.orc] and *gen09.sco* [examples/gen09.sco]. It will generate a cosine wave, a sine wave with an initial phase of 90 degrees. Here is its diagram:



Diagram of the waveform generated by GEN09.

#### Example 10. A simple example of the GEN09 routine.

```
/* gen09.orc */  
; Initialize the global variables.  
sr = 44100
```

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

; Instrument #1.
instr 1
  kamp = 30000
  kcps = 440
  ifn = 1

  ; Play the waveform stored in Table #1.
  al oscil kamp, kcps, ifn
  out al
endin
/* gen09.orc */

/* gen09.sco */
; Table #1: a cosine wave (using GEN09).
; This is a sine wave with an initial phase of 90 degrees.
f 1 0 16384 9 1 1 90

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen09.sco */
```

Here is another example of the GEN09 routine. It uses the files *gen09square.orc* [examples/gen09square.orc] and *gen09square.sco* [examples/gen09square.sco]. It combines partials 1, 3 and 9 in the relative strengths in which they are found in a square wave, except that partial 9 is upside down. It will be rescaled, here is its diagram:



Diagram of the waveform generated by GEN09.

### Example 11. A square wave generated by the GEN09 routine.

```
/* gen09square.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  kcps = 440
  ifn = 1

  ; Play the waveform stored in Table #1.
  al oscil kamp, kcps, ifn
  out al
endin
/* gen09square.orc */
```

```
/* gen09square.sco */  
; Table #1: an approximation of a square wave (using GEN09).  
f 1 0 16384 9 1 3 0 3 1 0 9 0.3333 180  
  
; Play Instrument #1 for 2 seconds.  
i 1 0 2  
e  
/* gen09square.sco */
```

## See Also

*GEN10, GEN19*

## Credits

The simple example was written by Kevin Conder.



## GEN10

GEN10 -- Generate composite waveforms made up of weighted sums of simple sinusoids.

GEN10

### Description

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 1 pfield using *GEN10*.

### Syntax

**f** # time size 10 str1 str2 str3 str4 ...

### Initialization

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

*str1*, *str2*, *str3*, etc. -- relative strengths of the fixed harmonic partial numbers 1,2,3, etc., beginning in p5. Partial 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

Here is a simple example of the GEN10 routine. It uses the files *gen10.orc* [examples/gen10.orc] and *gen10.sco* [examples/gen10.sco]. It will generate a simple sine wave. Here is its diagram:



Diagram of the waveform generated by GEN10.

### Example 12. A simple example of the GEN10 routine.

```
/* gen10.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
```

```
kcps = 440
ifn = 1

; Play the sine wave stored in Table #1.
a1 oscil kamp, kcps, ifn
out a1
endin
/* gen10.orc */

/* gen10.sco */
; Table #1: a simple sine wave (using GEN10).
f 1 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen10.sco */
```

## See Also

*GEN09*, *GEN11*, and *GEN19*.

## Credits

Example written by Kevin Conder

## GEN11

GEN11 -- Generates an additive set of cosine partials.

GEN11

### Description

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

### Syntax

```
f # time size ll nh [lh] [r]
```

### 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 the dynamic buzz units. However, the spectral content is invariant and care is necessary, lest the higher partials exceed the Nyquist during sampling to produce fold-over.

### Examples

Here is a simple example of the GEN11 routine. It uses the files *gen11.orc* [examples/gen11.orc] and *gen11.sco* [examples/gen11.sco]. It will generate a simple cosine wave. Here is its diagram:



Diagram of the waveform generated by GEN11.

**Example 13. A simple example of the GEN11 routine.**

```
/* gen11.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
  kamp = 30000
  kcps = 440
  ifn = 1

  ; Play the cosine wave stored in Table #1.
  al oscil kamp, kcps, ifn
  out al
endin
/* gen11.orc */

/* gen11.sco */
; Table #1: a simple cosine wave (using GEN11).
f 1 0 16384 11 1 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen11.sco */
```

**See Also**

*GEN10*

**Credits**

Example written by Kevin Conder

## GEN12

GEN12 -- Generates the log of a modified Bessel function of the second kind.

GEN12

### Description

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

### Syntax

**f** # time size 12 xint

### 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 +*xint*] over which the function is defined.



### Note

- This subroutine draws the natural log of a modified Bessel function of the second kind, order 0 (commonly written as  $I$  subscript 0), over the x-interval requested. The call should have rescaling inhibited.
- The function is useful as an amplitude scaling factor in cycle-synchronous amplitude-modulated FM. (See Palamin & Palamin, *J. Audio Eng. Soc.*, 36/9, Sept. 1988, pp.671-684.) The algorithm is interesting because it permits the normally symmetric FM spectrum to be made asymmetric around a frequency other than the carrier, and is thereby useful for formant positioning. By using a table lookup index of  $I(r - 1/r)$ , where  $I$  is the FM modulation index and  $r$  is an exponential parameter affecting partial strengths, the Palamin algorithm becomes relatively efficient, requiring only oscil's, table lookups, and a single *exp* call.

### Examples

Here is a simple example of the GEN12 routine. It uses the files *gen12.orc* [examples/gen12.orc] and *gen12.sco* [examples/gen12.sco]. It generates the function  $\ln(I_0(x))$  from 0 to 20. Here is its diagram:



Diagram of the waveform generated by GEN12.

#### Example 14. A simple example of the GEN12 routine.

```
/* gen12.orc */
; Initialize the global variables.
sr = 44100
```

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

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kamp tablei kndx, ifn, ixmode

; Create a sine wave, use the Table #1 values to control
; the amplitude. This creates a sound with a long attack.
a1 oscil kamp*30000, 440, 2
out a1
endin
/* gen12.orc */

/* gen12.sco */
; Table #1: a modified Bessel function (using GEN12).
f 1 0 2049 12 20
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen12.sco */
```

## Credits

Example written by Kevin Conder

## GEN13

GEN13 -- Stores a polynomial whose coefficients derive from the Chebyshev polynomials of the first kind.

GEN13

### Description

Uses 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.

### Syntax

```
f # time size 13 xint xamp h0 h1 h2 ...
```

### 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*, etc. -- relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude

$xamp * \text{int}(\text{size}/2)/xint$

is waveshaped using this function table. These values thus describe a frequency spectrum associated with a particular factor *xamp* of the input signal.

*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.

### Examples

Here is a simple example of the GEN13 routine. It uses the files *gen13.orc* [examples/gen13.orc] and *gen13.sco* [examples/gen13.sco]. It creates a function which, under waveshaping, will split a sinusoid into 3 odd-harmonic partials of relative strength 5:3:1. Here is its diagram:



Diagram of the waveform generated by GEN13.

### Example 15. A simple example of the GEN13 routine.

```
/* gen13.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kval table kndx, ifn, ixmode

; Generate a sine waveform, use our Table #1 value to
; vary its frequency by 100 Hz from its base frequency.
ibasefreq = 440
kfreq = kval * 100
a1 oscil 20000, ibasefreq + kfreq, 2
out a1
endin
/* gen13.orc */

/* gen13.sco */
; Table #1: a polynomial function (using GEN13).
f 1 0 1025 13 1 1 0 5 0 3 0 1
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen13.sco */
```

### See Also

*GEN03*, *GEN14*, and *GEN15*.



## GEN14

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

GEN14

### Description

Uses 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.

### Syntax

**f** # time size 14 xint xamp h0 h1 h2 ...

### 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 therefore 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*, etc. -- relative strength of partials 0 (DC), 1 (fundamental), 2 ... that will result when a sinusoid of amplitude

$xamp * \text{int}(\text{size}/2)/xint$

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  $+,+,-,-,+,+,\dots$  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.

## Examples

Here is a simple example of the GEN14 routine. It uses the files *gen14.orc* [examples/gen14.orc] and *gen14.sco* [examples/gen14.sco]. It creates a function which, under waveshaping, will split a sinusoid into 3 odd-harmonic partials of relative strength 5:3:1. Here is its diagram:



Diagram of the waveform generated by GEN14.

### Example 16. A simple example of the GEN14 routine.

```
/* gen14.orc */
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kval table kndx, ifn, ixmode

; Generate a sine waveform, use our Table #1 value to
; vary its frequency by 100 Hz from its base frequency.
ibasefreq = 440
kfreq = kval * 100
a1 oscil 20000, ibasefreq + kfreq, 2
out a1
endin
/* gen14.orc */

/* gen14.sco */
; Table #1: a polynomial function (using GEN14).
f 1 0 1025 14 1 1 0 5 0 3 0 1
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 2 seconds.
i 1 0 2
e
/* gen14.sco */
```

## See Also

*GEN03*, *GEN13*, and *GEN15*.

## Credits

Example written by Kevin Conder

## GEN15

GEN15 -- Creates two tables of stored polynomial functions.

GEN15

### Description

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

### Syntax

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

### 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

$xamp * \text{int}(\text{size}/2)/xint$

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.

### See Also

*GEN03*, *GEN13*, and *GEN14*.

## GEN16

GEN16 -- Creates a table from a starting value to an ending value.

GEN16

### Description

Creates a table from *beg* value to *end* value of *dur* steps.

### Syntax

```
f # time size 16 beg dur type end
```

### 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:

$$\text{beg} + (\text{end} - \text{beg}) * (1 - \exp(i * \text{type} / (\text{dur} - 1))) / (1 - \exp(\text{type}))$$

*end* -- value after *dur* segments



### Note

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*.

### Credits

Author: John ffitch  
University of Bath, Codemist. Ltd.  
Bath, UK  
October, 2000

New in Csound version 4.09

## GEN17

GEN17 -- Creates a step function from given x-y pairs.

GEN17

### Description

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

### Syntax

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

### 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*).

### Examples

```
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.

### See Also

GEN02

## GEN18

GEN18 -- Writes composite waveforms made up of pre-existing waveforms.

GEN18

### Description

Writes composite waveforms made up of pre-existing waveforms. Each contributing waveform requires 4 pfields and can overlap with other waveforms.

### Syntax

```
f # time size 18 fna ampa starta finisha fna ampa starta finisha ...
```

### Initialization

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

*fna, fnb, etc.* -- pre-existing table number to be written into the table.

*ampa, ampb, etc.* -- strength of waveforms. These are relative strengths, since the composite waveform may be rescaled later. Negative values are permitted and imply a 180 degree phase shift.

*starta, startb, etc.* -- where to start writing the fn into the table.

*finisha, finishb, etc.* -- where to stop writing the fn into the table.

### Examples

```
f 1 0 4096 10 1
f 2 0 1025 18 1 1 0 512 1 1 513 1025
```

f2 consists of two copies of f1 written in to locations 0-512 and 513-1025.

### Deprecated Names

*GEN18* was called *GEN22* in version 4.18. The name was changed due to a conflict with DirectC-sound.

### Credits

Author: William "Pete" Moss  
University of Texas at Austin  
Austin, Texas USA  
January 2002

New in version 4.18, changed in version 4.19

## GEN19

GEN19 -- Generate composite waveforms made up of weighted sums of simple sinusoids.

GEN19

### Description

These subroutines generate composite waveforms made up of weighted sums of simple sinusoids. The specification of each contributing partial requires 4 p-fields using *GEN19*.

### Syntax

```
f # time size 19 pna stra phsa dcoa pnb strb phsb dcob ...
```

### 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. Partial 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).



### 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

Here is a simple example of the GEN19 routine. It uses the files *gen19.orc* [examples/gen19.orc] and *gen19.sco* [examples/gen19.sco]. It will generate a nice bell curve, here is its diagram:



Diagram of the waveform generated by GEN19.

#### Example 17. A simple example of the GEN19 routine.

```
/* gen19.orc */
```



```
; Initialize the global variables.
sr = 44100
kr = 4410
ksmps = 10
nchnls = 1

; Instrument #1.
instr 1
; Create an index over the length of our entire note.
kcps init 1/p3
kndx phasor kcps

; Read Table #1 with our index.
ifn = 1
ixmode = 1
kval table kndx, ifn, ixmode

; Generate a sine waveform, use our Table #1 value to
; vary its frequency by 100 Hz from its base frequency.
ibasefreq = 440
kfreq = kval * 100
a1 oscil 20000, ibasefreq + kfreq, 2
out a1
endin
/* gen19.orc */

/* gen19.sco */
; Table #1: a bell curve (using GEN19).
f 1 0 16384 -19 1 1 260 1
; Table #2, a sine wave.
f 2 0 16384 10 1

; Play Instrument #1 for 3 seconds.
i 1 0 3
e
/* gen19.sco */
```

## See Also

*GEN09* and *GEN10*

## Credits

Example written by Kevin Conder

## GEN20

GEN20 -- Generates functions of different windows.

GEN20

### Description

This subroutine generates functions of different windows. These windows are usually used for spectrum analysis or for grain envelopes.

### Syntax

```
f # time size 20 window max [opt]
```

### 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 *Window Functions*

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK

New in Csound version 3.2

## GEN21

GEN21 -- Generates tables of different random distributions.

GEN21

### Description

This generates tables of different random distributions. (See also *betarand*, *bexprnd*, *cauchy*, *exprand*, *gauss*, *linrand*, *pcauchy*, *poisson*, *trirand*, *unirand*, and *weibull*)

### Syntax

```
f # time size 21 type level [arg1 [arg2]]
```

### Initialization

*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 follow:

- 1 = Uniform (positive numbers only)
- 2 = Linear (positive numbers only)
- 3 = Triangular (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.

## Credits

Author: Paris Smaragdis  
MIT, Cambridge  
1995

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK

New in Csound version 3.2

## GEN22

GEN22 -- Deprecated.

GEN22

### Description

Deprecated as of version 4.19. Use the *GEN18* routine instead.

## GEN23

GEN23 -- Reads numeric values from a text file.

GEN23

### Description

This subroutine reads numeric values from an external ASCII file.

### Syntax

```
f # time size -23 "filename.txt"
```

### 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).

### Credits

Author: Gabriel Maldonado  
Italy  
February, 1998

New in Csound version 3.47

## GEN24

GEN24 -- Reads numeric values from another allocated function-table and rescales them.

GEN24

### Description

This subroutine reads numeric values from another allocated function-table and rescales them according to the max and min values given by the user.

### Syntax

```
f # time size -24 ftable min max
```

### Initialization

*#, time, size* -- the usual GEN parameters. See *f* statement.

*ftable* -- ftable must be an already allocated table with the same size as this function.

*min, max* -- the rescaling range.



### Note

This GEN is useful, for example, to eliminate the starting offset in exponential segments allowing a real starting from zero.

### Credits

Author: Gabriel Maldonado

New in Csound version 4.16



## GEN25

GEN25 -- Construct functions from segments of exponential curves in breakpoint fashion.

GEN25

### Description

These subroutines are used to construct functions from segments of exponential curves in breakpoint fashion.

### Syntax

```
f # time size 25 x1 y1 x2 y2 x3 ...
```

### 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. These must be non-zero and must be alike in sign.



### 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.

### See Also

*f statement*, *GEN27*

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK

New in Csound version 3.49

## GEN27

GEN27 -- Construct functions from segments of straight lines in breakpoint fashion.

GEN27

### Description

Construct functions from segments of straight lines in breakpoint fashion.

### Syntax

```
f # time size 27 x1 y1 x2 y2 x3 ...
```

### 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.



### 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.

### Examples

```
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.

### See Also

*f statement*, *GEN25*

### Credits

Author: John ffitch  
University of Bath/Codemist Ltd.  
Bath, UK

New in Csound version 3.49

## GEN28

GEN28 -- Reads a text file which contains a time-tagged trajectory.

GEN28

### 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.

### Syntax

```
f # time size 28 ifilcod
```

### 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.

### Examples

```
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.

## Credits

Author: Richard Karpen  
Seattle, Wash  
1998

New in Csound version 3.48

## GEN30

GEN30 -- Generates harmonic partials by analyzing an existing table.

GEN30

### Description

Extracts a range of harmonic partials from an existing waveform.

### Syntax

```
f # time size 30 src minh maxh [ref_sr] [interp]
```

### Performance

*src* -- source ftable

*minh* -- lowest harmonic number

*maxh* -- highest harmonic number

*ref\_sr* (optional) -- maxh is scaled by (sr / ref\_sr). The default value of ref\_sr is sr. If *ref\_sr* is zero or negative, it is now ignored.

*interp* (optional) -- if non-zero, allows changing the amplitude of the lowest and highest harmonic partial depending on the fractional part of *minh* and *maxh*. For example, if *maxh* is 11.3 then the 12th harmonic partial is added with 0.3 amplitude. This parameter is zero by default.

GEN30 does not support tables with an extended guard point (ie. table size = power of two + 1). Although such tables will work both for input and output, when reading source table(s), the guard point is ignored, and when writing the output table, guard point is simply copied from the first sample (table index = 0).

The reason of this limitation is that GEN30 uses FFT, which requires power of two table size. GEN32 allows using linear interpolation for resampling and phase shifting, which makes it possible to use any table size (however, for partials calculated with FFT, the power of two limitation still exists).

### Credits

Author: Istvan Varga

New in version 4.16

## GEN31

GEN31 -- Mixes any waveform specified in an existing table.

GEN31

### Description

This routine is similar to GEN09, but allows mixing any waveform specified in an existing table.

### Syntax

```
f # time size 31 src pna stra phsa pnb strb phsb ...
```

### Performance

*src* -- source table number

*pna*, *pnb*, ... -- partial number, must be a positive integer

*stra*, *strb*, ... -- amplitude scale

*phsa*, *phsb*, ... -- start phase (0 to 1)

*GEN31* does not support tables with an extended guard point (ie. table size = power of two + 1). Although such tables will work both for input and output, when reading source table(s), the guard point is ignored, and when writing the output table, guard point is simply copied from the first sample (table index = 0).

The reason of this limitation is that *GEN31* uses FFT, which requires power of two table size. *GEN32* allows using linear interpolation for resampling and phase shifting, which makes it possible to use any table size (however, for partials calculated with FFT, the power of two limitation still exists).

### Credits

Author: Istvan Varga

New in version 4.15

## GEN32

GEN32 -- Mixes any waveform, resampled with either FFT or linear interpolation.

GEN32

### Description

This routine is similar to *GEN31*, but allows specifying source ftable for each partial. Tables can be resampled either with FFT, or linear interpolation.

### Syntax

```
f # time size 32 srca pna stra phsa srcb pnb strb phsb ...
```

### Performance

*srca*, *srcb* -- source table number. A negative value can be used to read the table with linear interpolation (by default, the source waveform is transposed and phase shifted using FFT); this is less accurate, but faster, and allows non-integer and negative partial numbers.

*pna*, *pnb*, ... -- partial number, must be a positive integer if source table number is positive (i.e. resample with FFT).

*stra*, *strb*, ... -- amplitude scale

*phsa*, *phsb*, ... -- start phase (0 to 1)

### Examples

```
itmp    ftgen 1, 0, 16384, 7, 1, 16384, -1      ; sawtooth
itmp    ftgen 2, 0, 8192, 10, 1                ; sine
; mix tables
itmp    ftgen 5, 0, 4096, -32, -2, 1.5, 1.0, 0.25, 1, 2, 0.5, 0, \
                                     1, 3, -0.25, 0.5
; window
itmp    ftgen 6, 0, 16384, 20, 3, 1
; generate band-limited waveforms
inote   = 0
loop0:
icps    = 440 * exp(log(2) * (inote - 69) / 12)      ; one table for
inumh   = sr / (2 * icps)                            ; each MIDI note number
ift     = int(inote + 256.5)
itmp    ftgen ift, 0, 4096, -30, 5, 1, inumh
inote   = inote + 1
if (inote < 127.5) igoto loop0

instr 1

kcps    expon 20, p3, 16000
kft     = int(256.5 + 69 + 12 * log(kcps / 440) / log(2))
kft     = (kft > 383 ? 383 : kft)

a1      phasor kcps
a1      tableikt a1, kft, 1, 0, 1

out a1 * 10000

endin
```

```
instr 2

kcps    expon 20, p3, 16000
kft     = int(256.5 + 69 + 12 * log(kcps / 440) / log(2))
kft     = (kft > 383 ? 383 : kft)

kgdur   limit 10 / kcps, 0.1, 1
a1      grain2 kcps, 0.02, kgdur, 30, kft, 6, -0.5

out a1 * 2000

endin

-----
score:
-----

t 0 60
i 1 0 10
i 2 12 10
e
```

## Credits

Author: Rasmus Ekman

Programmer: Istvan Varga

New in version 4.17



## GEN33

GEN33 -- Generate composite waveforms by mixing simple sinusoids.

GEN33

### Description

These routines generate composite waveforms by mixing simple sinusoids, similarly to *GEN09*, but the parameters of the partials are specified in an already existing table, which makes it possible to calculate any number of partials in the orchestra.

The difference between *GEN33* and *GEN34* is that *GEN33* uses inverse FFT to generate output, while *GEN34* is based on the algorithm used in *oscils* opcode. *GEN33* allows integer partials only, and does not support power of two plus 1 table size, but may be significantly faster with a large number of partials. On the other hand, with *GEN34*, it is possible to use non-integer partial numbers and extended guard point, and this routine may be faster if there is only a small number of partials (note that *GEN34* is also several times faster than *GEN09*, although the latter may be more accurate).

### Syntax

```
f # time size 33 src nh scl [fmode]
```

### Initialization

*size* -- number of points in the table. Must be power of two and at least 4.

*src* -- source table number. This table contains the parameters of each partial in the following format:

stra, pna, phsa, strb, pnb, phsb, ...

the parameters are:

- *stra*, *strb*, etc.: relative strength of partials. The actual amplitude depends on the value of *scl*, or normalization (if enabled).
- *pn*, *pnb*, etc.: partial number, or frequency, depending on *fmode* (see below); zero and negative values are allowed, however, if the absolute value of the partial number exceeds ( $\text{size} / 2$ ), the partial will not be rendered. With *GEN33*, partial number is rounded to the nearest integer.
- *phsa*, *phsb*, etc.: initial phase, in the range 0 to 1.

Table length (not including the guard point) should be at least  $3 * nh$ . If the table is too short, the number of partials (*nh*) is reduced to  $(\text{table length}) / 3$ , rounded towards zero.

*nh* -- number of partials. Zero or negative values are allowed, and result in an empty table (silence). The actual number may be reduced if the source table (*src*) is too short, or some partials have too high frequency.

*scl* -- amplitude scale.

*fmode* (optional, default = 0) -- a non-zero value can be used to set frequency in Hz instead of partial numbers in the source table. The sample rate is assumed to be *fmode* if it is positive, or  $-(sr * fmode)$  if any negative value is specified.

## Examples

```
; partials 1, 4, 7, 10, 13, 16, etc. with base frequency of 400 Hz

ibsfrq = 400
; estimate number of partials
inumh = int(1.5 + sr * 0.5 / (3 * ibsfrq))
; source table length
isrcln = int(0.5 + exp(log(2) * int(1.01 + log(inumh * 3) / log(2))))
; create empty source table
itmp    ftgen 1, 0, isrcln, -2, 0
ifpos   = 0
ifrq    = ibsfrq
inumh   = 0
l1:
    tableiw ibsfrq / ifrq, ifpos, 1          ; amplitude
    tableiw ifrq, ifpos + 1, 1              ; frequency
    tableiw 0, ifpos + 2, 1                 ; phase
ifpos   = ifpos + 3
ifrq    = ifrq + ibsfrq * 3
inumh   = inumh + 1
        if (ifrq < (sr * 0.5)) igoto l1

; store output in ftable 2 (size = 262144)

itmp    ftgen 2, 0, 262144, -34, 1, inumh, 1, -1
```

## See Also

*GEN09, GEN34*

## Credits

Programmer: Istvan Varga  
March 2002

New in version 4.19

## GEN34

GEN34 -- Generate composite waveforms by mixing simple sinusoids.

GEN34

### Description

These routines generate composite waveforms by mixing simple sinusoids, similarly to *GEN09*, but the parameters of the partials are specified in an already existing table, which makes it possible to calculate any number of partials in the orchestra.

The difference between *GEN33* and *GEN34* is that *GEN33* uses inverse FFT to generate output, while *GEN34* is based on the algorithm used in *oscils* opcode. *GEN33* allows integer partials only, and does not support power of two plus 1 table size, but may be significantly faster with a large number of partials. On the other hand, with *GEN34*, it is possible to use non-integer partial numbers and extended guard point, and this routine may be faster if there is only a small number of partials (note that *GEN34* is also several times faster than *GEN09*, although the latter may be more accurate).

### Syntax

```
f # time size 34 src nh scl [fmode]
```

### Initialization

*size* -- number of points in the table. Must be power of two or a power of two plus 1.

*src* -- source table number. This table contains the parameters of each partial in the following format:

stra, pna, phsa, strb, pnb, phsb, ...

the parameters are:

- *stra*, *strb*, etc.: relative strength of partials. The actual amplitude depends on the value of *scl*, or normalization (if enabled).
- *pn*a, *pn*b, etc.: partial number, or frequency, depending on *fmode* (see below); zero and negative values are allowed, however, if the absolute value of the partial number exceeds ( $\text{size} / 2$ ), the partial will not be rendered.
- *phsa*, *phsb*, etc.: initial phase, in the range 0 to 1.

Table length (not including the guard point) should be at least  $3 * nh$ . If the table is too short, the number of partials (*nh*) is reduced to  $(\text{table length}) / 3$ , rounded towards zero.

*nh* -- number of partials. Zero or negative values are allowed, and result in an empty table (silence). The actual number may be reduced if the source table (*src*) is too short, or some partials have too high frequency.

*scl* -- amplitude scale.

*fmode* (optional, default = 0) -- a non-zero value can be used to set frequency in Hz instead of partial numbers in the source table. The sample rate is assumed to be *fmode* if it is positive, or  $-(sr * fmode)$  if any negative value is specified.

## Examples

```
; partials 1, 4, 7, 10, 13, 16, etc. with base frequency of 400 Hz

ibsfrq = 400
; estimate number of partials
inumh = int(1.5 + sr * 0.5 / (3 * ibsfrq))
; source table length
isrcln = int(0.5 + exp(log(2) * int(1.01 + log(inumh * 3) / log(2))))
; create empty source table
itmp    ftgen 1, 0, isrcln, -2, 0
ifpos   = 0
ifrq    = ibsfrq
inumh   = 0
l1:
    tableiw ibsfrq / ifrq, ifpos, 1          ; amplitude
    tableiw ifrq, ifpos + 1, 1              ; frequency
    tableiw 0, ifpos + 2, 1                  ; phase
ifpos   = ifpos + 3
ifrq    = ifrq + ibsfrq * 3
inumh   = inumh + 1
        if (ifrq < (sr * 0.5)) igoto l1

; store output in ftable 2 (size = 262144)

itmp    ftgen 2, 0, 262144, -34, 1, inumh, 1, -1
```

## See Also

*GEN09, GEN33*

## Credits

Programmer: Istvan Varga  
March 2002

New in version 4.19

## GEN40

GEN40 -- Generates a random distribution using a distribution histogram.

GEN40

### Description

Generates a continuous random distribution function starting from the shape of a user-defined distribution histogram.

### Syntax

```
f # time size -40 shapetab
```

### Performance

The shape of histogram must be stored in a previously defined table, in fact shapetab argument must be filled with the number of such table.

Histogram shape can be generated with any other GEN routines. Since no interpolation is used when GEN40 processes the translation, it is suggested that the size of the table containing the histogram shape to be reasonably big, in order to obtain more precision (however after the processing the shapetab can be destroyed in order to re-gain memory).

This subroutine is designed to be used together with cusernd opcode (see cusernd for more information).

### Credits

Author: Gabriel Maldonado

## GEN41

GEN41 -- Generates a random list of numerical pairs.

GEN41

### Description

Generates a discrete random distribution function by giving a list of numerical pairs.

### Syntax

```
f # time size -41 value1 prob1 value2 prob2 value3 prob3 ... valueN probN
```

### Performance

The first number of each pair is a value, and the second is the probability of that value to be chosen by a random algorithm. Even if any number can be assigned to the probability element of each pair, it is suggested to give it a percent value, in order to make it clearer for the user.

This subroutine is designed to be used together with `dusernd` and `urd` opcodes (see `dusernd` for more information).

### Credits

Author: Gabriel Maldonado

## GEN42

GEN42 -- Generates a random distribution of discrete ranges of values.

GEN42

### Description

Generates a random distribution function of discrete ranges of values by giving a list of groups of three numbers.

### Syntax

```
f # time size -42 min1 max1 prob1 min2 max2 prob2 min3 max3 prob3 ... minN ma
```

### Performance

The first number of each group is a the minimum value of the first range, the second is the maximum value and the third is the probability of that an element belonging to that range of values can be chosen by a random algorithm. Even if any number can be assigned to the probability element of each group, it is suggested to give it a percent value, in order to make it clearer to the user.

This subroutine is designed to be used together with `dusernd` and `urd` opcodes (see `dusernd` for more information). Since both `dusernd` and `urd` do not use any interpolation, it is suggested to give a size reasonably big.

### Credits

Author: Gabriel Maldonado

## GEN43

GEN43 -- Loads a PVOCEX file containing a PV analysis.

GEN43

### Description

This subroutine loads a PVOCEX file containing the PV analysis (amp-freq) of a soundfile and calculates the average magnitudes of all analysis frames of one or all audio channels. It then creates a table with these magnitudes for each PV bin.

### Syntax

```
f # time size 43 filecod channel
```

### Initialisation

*size* -- number of points in the table, power-of-two or power-of-two plus 1. GEN 43 does not make any distinction between these two sizes, but it requires the table to be at least the  $\text{fftsize}/2$ . PV bins cover the positive spectrum from 0Hz (table index 0) to the Nyquist (table index  $\text{fftsize}/2+1$ ) in equal-size frequency increments (of size  $\text{sr}/\text{fftsize}$ ).

*filecod* -- a pvocex file (which can be generated by pvanal).

*channel* -- audio channel number from which the magnitudes will be extracted; a 0 will average the magnitudes from all channels.

Reading stops at the end of the file.



### Note

if p4 is positive, the table will be post-normalised. A negative p4 will cause post-normalisation to be skipped.

### Examples

```
f1 0 512 43 "viola.pvx" 1
f1 0 -1024 -43 "noiseprint.pvx" 0
```

This table can be used as a masking table for pvstencil and pvsmaska. The first example uses a 1024-point FFT phase vocoder analysis file from which the first channel is used. The second uses all channels of a 2048-point file, without post-normalisation. For noise reduction applications with pvstencil, it is easiest to skip table normalisation (negative GEN code).

### Credits

Author: Victor Lazzarini



## GEN51

GEN51 -- This subroutine fills a table with a fully customized micro-tuning scale, in the manner of Csound opcodes cpstun, cpstuni and cpstmid.

GEN51

### Description

This subroutine fills a table with a fully customized micro-tuning scale, in the manner of Csound opcodes cpstun, cpstuni and cpstmid.

### Syntax

```
f # time size -51 numgrades interval basefreq basekey tuningRatio1 tuningRatio2
```

### Performance

The first four parameters (i.e. p5, p6, p7 and p8) define the following generation directives:

*p5 (numgrades)* -- the number of grades of the micro-tuning scale

*p6 (interval)* -- the frequency range covered before repeating the grade ratios, for example 2 for one octave, 1.5 for a fifth etcetera

*p7 (basefreq)* -- the base frequency of the scale in cps

*p8 (basekey)* -- the integer index of the scale to which to assign basefreq unmodified

The other parameters define the ratios of the scale:

*p9...pN (tuningRatio1...etc.)* -- the tuning ratios of the scale

For example, for a standard 12-grade scale with the base-frequency of 261 cps assigned to the key-number 60, the corresponding f-statement in the score to generate the table should be:

```

;               numgrades      basefreq      tuning-ratios  (eq.temp
;               interval      basekey
f1 0 128 -51    12             2       261      60          1   1.059463 1.12246 1
```

After the gen has been processed, the table f1 is filled with 64 different frequency values. The 60th element is filled with the frequency value of 261, and all other elements (preceding and subsequent) of the table are filled according to the tuning ratios

Another example with a 24-grade 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      basekey
f1 0 128 -2     24             1.5     440      48          1   1.01  1.02  1.
..etc...
```

>

### Credits

Author: Gabriel Maldonado

## GEN52

GEN52 -- Creates an interleaved multichannel table from the specified source tables, in the format expected by the "ftconv" opcode.

GEN52

### Description

GEN52 creates an interleaved multichannel table from the specified source tables, in the format expected by the "ftconv" opcode. It can also be used to extract a channel from a multichannel table and store it in a normal mono table, copy tables with skipping some samples, adding delay, or store in reverse order, etc.

### Syntax

```
f # time size 52 nchannels channel_params_1 channel_params_2 ...
```

### Example

```
                ; source tables
f 1 0 16384 10 1
f 2 0 16384 10 0 1
; create 2 channel interleaved table
f 3 0 32768 -52 2 1 0 1 2 0 1
; extract first channel
f 4 0 16384 -52 1 3 0 2
; extract second channel
f 5 0 16384 -52 1 3 1 2
```

### Credits

Author: Istvan Varga

---

# The Utility Programs

Dan Ellis, MIT Media Lab

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 the *-R flag* (or its default). If both the *SR header* and the command-line flag are present, the flag value will override the header.

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.

**Credits.** Dan Ellis MIT Media Lab Cambridge, Massachusetts

## Analysis File Generation (HETRO,LPANAL, PVANAL, CVANAL, ATSA)

The following utilities exist for Soundfile analysis:

- *HETRO*: Heterodyne analysis for the Csound *adsyn* generator.
- *LPANAL*: Linear predicitive coding analysis for the Csound *Linear Predictive Coding (LPC) Resynthesis* opcodes.
- *PVANAL*: Phase vocoder analysis for the Csound *pvoc* generator.
- *CVANAL*: Impulse Response Fourier Analysis for *convolve* operator.
- *ATSA*: ATS analysis for use with the Csound *ATS Resynthesis* opcodes.

## hetro

hetro -- Decomposes an input soundfile into component sinusoids.

hetro

## Description

Hetrodyne filter analysis for the Csound *adsyn* generator.

## Syntax

```
csound -U hetro [flags] infilename outfilename
```

```
hetro [flags] infilename outfilename
```

## Initialization

*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).

## Performance

As of Csound 4.08, *hetro* can write SDIF output files if the output file name ends with ".sdif". See the *sdif2ad* utility for more information about the Csound's SDIF support.

## Examples

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

This will analyze 2.5 seconds of channel 1 of a file "audiofile.test", recorded at 44.1 kHz, beginning .5 seconds from the start, and place the result in a file "adsynfile7". We request just the first 16 harmonics of the sound, with 256 initial breakpoint values per amplitude or frequency track, and a peak summation amplitude of 24000. The fundamental is estimated to begin at 100 Hz. Amplitude thresholding is at 54 db down.

The Butterworth LPF is not enabled.

## File Format

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

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

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

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

## Credits

October 2002. Thanks to Rasmus Ekman, added a note about the SDIF format.

## lpanal

lpanal -- Performs both linear predictive analysis on a soundfile.

lpanal

## Description

Linear predictive analysis for the Csound *Linear Predictive Coding (LPC) Resynthesis* opcodes.

## Syntax

```
csound -U lpanal [flags] infilename outfilename
```

```
lpanal [flags] infilename outfilename
```

## Initialization

*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.



## Examples

```
lpanal -a -p26 -d2.5 -P100 -Q400 audiofile.test lpfil22
```

will analyze the first 2.5 seconds of file "audiofile.test", producing  $\text{srate}/200$  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.

## pvanal

pvanal -- Converts a soundfile into a series of short-time Fourier transform frames.

pvanal

## Description

Fourier analysis for the Csound *pvoc* generator

## Syntax

```
csound -U pvanal [flags] infilename outfilename
```

```
pvanal [flags] infilename outfilename
```

## Pvanal extension to create a PVOC-EX file.

The standard Csound utility program pvanal has been extended to enable a PVOC-EX format file to be created, using the existing interface. To create a PVOC-EX file, the file name must be given the required extension, “.pvx”, e.g “test.pvx”. The requirement for the FFT size to be a power of two is here relaxed, and any positive value is accepted; odd numbers are rounded up internally. However, power-of-two sizes are still to be preferred for all normal applications.

The channel select flags are ignored, and all source channels will be analysed and written to the output file, up to a compiler-set limit of eight channels. The analysis window size (iwinsize) is set internally to double the FFT size.

## Initialization

*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*.

*-H* -- Use a Hamming window instead of the default von Hann window.

*-K* -- Use a Kaiser window instead of the default von Hann window. The Kaiser parameter default is 6.8, but can be set with the *-B* option.

*-B beta* -- Set the beta parameter for any Kaiser window used to the floating point value beta.

## Examples

```
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.

## Credits

Author: Dan Ellis

MIT Media Lab

Cambridge, Massachusetts

1990

## cvanal

`cvanal` -- Converts a soundfile into a single Fourier transform frame.

`cvanal`

### Description

Impulse Response Fourier Analysis for *convolve* operator

### Syntax

```
CSound -U cvanal [flags] infilename outfilename
```

### Initialization

*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.

### Examples

```
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.

### Credits

Author: Greg Sullivan

Based on algorithm given in *Elements Of Computer Music*, by F. Richard Moore.

## atsa

atsa -- Performs ATS analysis on a soundfile.

atsa

## Description

ATS analysis for use with the Csound *ATS Resynthesis* opcodes.

## Syntax

```
csound -U atsa [flags] infilename outfilename
```

## Initialization

The following flags can be set for atsa (The default values are stated in parenthesis):

- b start (0.000000 seconds)
- e duration (0.000000 seconds or end)
- l lowest frequency (20.000000 Hertz)
- H highest frequency (20000.000000 Hertz)
- d frequency deviation (0.100000 of partial freq.)
- c window cycles (4 cycles)
- w window type (type: 1) (Options: 0=BLACKMAN, 1=BLACKMAN\_H, 2=HAMMING, 3=VONHANN)
- h hop size (0.250000 of window size)
- m lowest magnitude (-60.000000)
- t track length (3 frames)
- s min. segment length (3 frames)
- g min. gap length (3 frames)
- T SMR threshold (30.000000 dB SPL)
- S min. segment SMR (60.000000 dB SPL)
- P last peak contribution (0.000000 of last peak's parameters)
- M SMR contribution (0.500000)
- F File Type (type: 4) (Options: 1=amp.and freq. only, 2=amp.,freq. and phase, 3=amp.,freq. and residual, 4=amp.,freq.,phase, and residual)

ATS analysis was devised by Juan Pampin. For complete information on ATS visit: <http://www-ccrma.stanford.edu/~juan/ATS.html>.

Analysis parameters must be carefully tuned for the Analysis Algorithm (ATSA) to properly capture the nature of the signal to be analyzed. As there are a significant number of them, ATSH offers the possibility of Saving/Loading them in a Binary File carrying the extension "\*.apf". The extension is not mandatory, but recommended. A brief explanation of each Analysis Parameters follows:

1. Start (secs.): the starting time of the analysis in seconds.
2. Duration (secs.): the duration time of the analysis in seconds. A zero means the whole duration of the input sound file.
3. Lowest Frequency (Hz.): this parameter will partially determine the size of the Analysis Window to be used. To compute the size of the Analysis Window, the period of the Lowest Frequency in samples (SR / LF) is multiplied by the number of cycles of it the user wants to fit in the Analysis Window (see parameter 6). This value is rounded to the next power of two to determine the size of the FFT for the analysis. The remaining samples are zero-padded. If the signal is a single, harmonic sound, then the value of the Lowest Frequency should be its fundamental frequency or a sub-harmonic of it. If it is not harmonic, then its lowest significant frequency component may be a good starting value.

4. Highest Frequency (Hz.): highest frequency to be taken into account for Peak Detection. Once it is determined that no relevant information is found beyond a certain frequency, the analysis may be faster and more accurate setting the Highest Frequency parameter to that value.
5. Frequency Deviation (Ratio): frequency deviation allowed for each peak in the Peak Continuation Algorithm, as a ratio of the frequency involved. For instance, considering a peak at 440 Hz and a Deviation of .1 will produce that the Peak Continuation Algorithm will only try to find candidates for its continuation between 396 and 484 Hz (10% above and below the frequency of the peak). A small value is likely to produce more trajectories whilst a large value will reduce them, but at the cost of rendering information difficult to be further processed.
6. Number of Cycles of Lowest Frequency to fit in Analysis Window: this will also partially determine the size of the Fourier Analysis Window to be used. See Parameter 3. For single harmonic signals, it is supposed to be more than one (typically 4).
7. Hop Size (Ratio): size of the gap between one Analysis Window and the next expressed as a ratio of the Window Size. For instance, a Hop Size value of .25 will "jump" by 512 samples (Windows will overlap for a 75% of their size). This parameter will also determine the size of the analysis frames obtained. Signals that change their spectra very fast (such as Speech sounds) may need a high frame rate in order to properly track their changes.
8. Amplitude Threshold (dB): the highest amplitude value to be taken into account for Peak Detection.
9. Window Type: the shape of the smoothing function to be used for the Fourier Analysis. There are four choices available at present: Blackman, Blackman-Harris, Von Hann, and Hanning. Precise specifications about them are easily found on D.S.P. bibliography.
10. Track Length (Frames): The Peak Continuation Algorithm will "look-back" by Length frames in order to do its job better, preventing frequency trajectories from curving too much and losing stability. However, a large value for this parameter will slow down the analysis significantly.
11. Minimal Segment Length (Frames): once the analysis is done, the spectral data can be further "cleaned" up during post-processing. Trajectories shorter than this value are suppressed if their average SMR is below Minimal Segment SMR (see parameters 16 and 14). This might help to avoid non-relevant sudden changes while keeping a high frame rate, reducing also the number of intermittent sinusoids during synthesis.
12. Minimal Gap Length (Frames): as parameter 11, this one is also used to clean up the data during post-processing. In this case, gaps (zero amplitude values, i.e. theoretical "silence") longer than Length frames are filled up with amplitude/frequency values obtained by linear interpolation of the adjacent active frames. This parameter prevents sudden interruptions of stable trajectories while keeping a high frame rate.
13. SMR Threshold (dB SPL): also a post-processing parameter, the SMR Threshold is used to eliminate partials with low averages.
14. -Minimal Segment SMR (dB SPL): this parameter is used in combination with parameter 11. Short segments with SMR average below this value will be removed during post-processing.
15. Last Peak Contribution (0 to 1): as explained in Parameter 10, the Peak Continuation Algorithm "looks-back" several number of frames to do its job better. This parameter will help to weight the contribution of the first precedent peak over the others. A zero value means that all precedent peaks (to the size of Parameter 10) are equally taken in account.
16. SMR Contribution (0 to 1): In addition to the proximity in frequency of the peaks, the ATS Peak Continuation Algorithm may use psycho-acoustic information (the Signal-to-Mask-Ratio, or SMR) to improve the perceptual results. This parameter indicates how much the SMR information is used during tracking. For instance, a value of .5 makes the Peak Continuation Algorithm to use a 50% of SMR information and a 50% of Frequency Proximity information to decide which is the best candidate to continue a sinusoidal track.

## Examples

The following command:

```
atsa -b0.1 -e1 -l100 -H10000 -w2 audiofile.wav audiofile.ats
```

Generates the ATS analysis file 'audiofile.ats' from the original 'audiofile.wav' file. It begins analysis from second 0.1 of the file and the analysis is performed for 1 second thereafter. The lowest frequency stored is 100 Hz and the highest is 10kHz. A Hamming window is used for each analysis frame.

## File Queries (SNDINFO)

The following utilities exist for Soundfile query:

- *SNDINFO*: Displays information about a soundfile.



## sndinfo

sndinfo -- Displays information about a soundfile.

sndinfo

### Description

Get basic information about one or more soundfiles.

### Syntax

```
csound -U sndinfo [options] soundfilenames ...
```

```
sndinfo [options] soundfilenames ...
```

### Initialization

*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.

There are two option types:

1. *-i* or *-il* will print instrument information, which includes looping. The option continues until a *-i0* option.
2. The other option is *-b* which prints the broadcast information for WAV files. It can similarly be negated with *-b0*.

### Examples

```
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, ba
      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

## **File Conversion (DNOISE, PVLOOK, SDIF2AD, SRCONV)**

The following utilities exist for file conversion:

- *DNOISE*: Reduces noise in a file.
- *PVLOOK*: View formatted text output of STFT analysis files.
- *SDIF2AD*: Converts SDIF files to files usable by adsynt.
- *SRCONV*: Converts the sample rate of an audio file.

## dnoise

dnoise -- Reduces noise in a file.

dnoise

### Description

This is a noise reduction scheme using frequency-domain noise-gating.

### Syntax

```
dnoise [flags] -i noise_ref_file -o output_soundfile input_soundfile
```

### Initialization

Dnoise specific flags:

- *(no flag)* input soundfile to be denoised
- *-i fname* input reference noise soundfile
- *-o fname* output soundfile
- *-N fnum* # of bandpass filters (default: 1024)
- *-w fovlp* filter overlap factor: {0,1,(2),3} DON'T USE *-w* AND *-M*
- *-M awlen* analysis window length (default: N-1 unless *-w* is specified)
- *-L swlen* synthesis window length (default: M)
- *-D dfac* decimation factor (default: M/8)
- *-b btim* begin time in noise reference soundfile (default: 0)
- *-B smpst* starting sample in noise reference soundfile (default: 0)
- *-e etim* end time in noise reference soundfile (default: end of file)
- *-E smpend* final sample in noise reference soundfile (default: end of file)
- *-t thr* threshold above noise reference in dB (default: 30)
- *-S gfact* sharpness of noise-gate turnoff, range: 1 to 5 (default: 1)
- *-n numfrm* number of FFT frames to average over (default: 5)
- *-m mingain* minimum gain of noise-gate when off in dB (default: -40)

Soundfile format options:

- *-A* AIFF format output
- *-W* WAV format output
- *-J* IRCAM format output

- *-h* skip soundfile header (not valid for AIFF/WAV output)
- *-8* 8-bit unsigned char sound samples
- *-c* 8-bit signed\_char sound samples
- *-a* alaw sound samples
- *-u* ulaw sound samples
- *-s* short\_int sound samples
- *-l* long\_int sound samples
- *-f* float sound samples. Floats also supported for WAV files. (New in Csound 3.47.)

Additional options:

- *-R* verbose - print status info
- *-H [N]* print a heartbeat character at each soundfile write.
- *--fname* output to log file fname
- *-V* verbose - print status info



## Note

DNOISE also looks at the environment variable SFOUTYP to determine soundfile output format.

The *-i* flag is used for a reference noise file (normally created from a short section of the denoised file, where only noise is audible). The input soundfile to be denoised can be given anywhere on the command line, without a flag.

## Performance

This is a noise reduction scheme using frequency-domain noise-gating. This should work best in the case of high signal-to-noise with hiss-type noise.

The algorithm is that suggested by Moorer & Berger in “Linear-Phase Bandsplitting: Theory and Applications” presented at the 76th Convention 1984 October 8-11 New York of the Audio Engineering Society (preprint #2132) except that it uses the Weighted Overlap-Add formulation for short-time Fourier analysis-synthesis in place of the recursive formulation suggested by Moorer & Berger. The gain in each frequency bin is computed independently according to

$$\text{gain} = g0 + (1-g0) * [\text{avg} / (\text{avg} + th*th*nref)] ^ sh$$

where *avg* and *nref* are the mean squared signal and noise respectively for the bin in question. (This is slightly different than in Moorer & Berger.)

The critical parameters *th* and *g0* are specified in dB and internally converted to decimal values. The *nref* values are computed at the start of the program on the basis of a noise\_soundfile (specified in the command line) which contains noise without signal.

The *avg* values are computed over a rectangular window of *m* FFT frames looking both ahead and behind the current time. This corresponds to a temporal extent of *m*\*D/R (which is typically

$(m \cdot N/8)/R$ ). The default settings of N, M, and D should be appropriate for most uses. A higher sample rate than 16 KHz might indicate a higher N.

## Credits

Author: Mark Dolson

August 26, 1989

Author: John ffitch

December 30, 2000

Updated by Rasmus Ekman on March 11, 2002.

## pvlook

pvlook -- View formatted text output of STFT analysis files.

pvlook

### Description

View formatted text output of STFT analysis files created with *pvanal*.

### Syntax

```
csound -U pvlook [flags] infilename
```

```
pvlook [flags] infilename
```

### Initialization

*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.

### Examples

```
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
```







[illegible]

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...*

## Credits

Author: Richard Karpen

Seattle, Wash

1993 (New in Csound version 3.57)

## sdif2ad

sdif2ad -- Converts SDIF files to files usable by adsynt.

sdif2ad

### Description

Convert files Sound Description Interchange Format (SDIF) to the format usable by Csound's *adsyn* opcode. As of Csound version 4.10, *sdif2ad* was available only as a standalone program for Windows console and DOS.

### Syntax

```
Csound -U sdif2ad [flags] infilename  
outfilename
```

### Initialization

Flags:

- *-sN* -- apply amplitude scale factor N
- *-pN* -- 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 is there to facilitate 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 *sdif2ad* can be used to convert any SDIF file containing a stream of 1TRC data to the Csound *adsyn* format. *sdif2ad* 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 *sdif2ad*, require amplitudes to be within a particular range. *sdif2ad* 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 *sdif2ad*. 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>

### Credits

Author: Richard Dobson

Somerset, England

August, 2000

New in Csound version 4.08

## srconv

srconv -- Converts the sample rate of an audio file.

srconv

## Description

Converts the sample rate of an audio file at sample rate  $R_{in}$  to a sample rate of  $R_{out}$ . Optionally the ratio ( $R_{in} / R_{out}$ ) may be linearly time-varying according to a set of (time, ratio) pairs in an auxiliary file.

## Syntax

**srconv** [*flags*] *infile*

## Initialization

Flags:

- *-P num* = pitch transposition ratio (*srate* / *r*) [don't specify both *P* and *r*]
- *-P num* = pitch transposition ratio (*srate* / *r*) [don't specify both *P* and *r*]
- *-Q num* = quality factor (1, 2, 3, or 4: default = 2)
- *-i filnam* = break file
- *-r num* = output sample rate (must be specified)
- *-o fnam* = sound output filename
- *-A* = create an AIFF format output soundfile
- *-J* = create an IRCAM format output soundfile
- *-W* = create a WAV 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* = Do not generate PEAK chunks
- *-R* = continually rewrite header while writing soundfile (WAV/AIFF)

- *-H#* = print a heartbeat style 1, 2 or 3 at each soundfile write
- *-N* = notify (ring the bell) when score or miditrack is done
- *--fnam* = log output to file

This program performs arbitrary sample-rate conversion with high fidelity. The method is to step through the input at the desired sampling increment, and to compute the output points as appropriately weighted averages of the surrounding input points. There are two cases to consider:

1. sample rates are in a small-integer ratio - weights are obtained from table.
2. sample rates are in a large-integer ratio - weights are linearly interpolated from table.

Calculate increment: if decimating, then window is impulse response of low-pass filter with cutoff frequency at half of output sample rate; if interpolating, then window is impulse response of low-pass filter with cutoff frequency at half of input sample rate.

## Credits

Author: Mark Dolson

August 26, 1989

Author: John ffitch

December 30, 2000

## Other Csound Utilities (MAKECSD, CS)

The following utilities exist for Soundfile analysis:

- *MAKECSD*: Creates a CSD file from the specified input files.
- *CS*: Starts Csound with a set of options that can be controlled by environment variables, and input and output files determined by the specified filename stem.
- *SCALE*: Scale the amplitude of a sound file.
- *MIXER*: Mixes together a number of soundfiles.
- *EXTRACTOR*: Extract a section of audio from an audio file.

## makecsd

`makecsd --` Creates a CSD file from the specified input files.

`makecsd`

### Description

Creates a CSD file from the specified input files. The first input file that has a .orc extension (case is not significant) is put to the <CsInstruments> section, and the first input file that has a .sco extension becomes <CsScore>. Any remaining files are Base64 encoded and added as <CsFileB> tags. An empty <CsOptions> section is always added.

Some text filtering is performed on the orchestra and score file:

- newlines are converted to the native format of the system on which `makecsd` is being run.
- blank lines are removed from the beginning and end of files.
- any trailing whitespace is removed from the end of lines.
- optionally, tabs can be expanded to spaces with an user specified tabstop size.

### Syntax

```
makecsd [OPTIONS ... ] infile1 [ infile2 [ ... ] ]
```

### Initialization

Flags:

- `-t n` = expand tabs to spaces using tabstop size `n` (default: disabled). This applies only to the orchestra and score file.
- `-w n` = set Base64 line width to `n` (default: 72). Note: the orchestra and score are not wrapped.
- `-o fname` = output file name (default: stdout)

### Examples

```
makecsd -t 6 -w 78 -o file.csd file.mid file.orc file.sco sample.aif
```

This creates a CSD from `file.orc` and `file.sco` (tabs are expanded to spaces assuming a tabstop size of 6 characters), and `file.mid` and `sample.aif` are added as <CsFileB> tags containing Base64 encoded data with a line width of 78 characters. The output file is `file.csd`.

### Credits

Author: Istvan Varga

Jan 2003

## CS

**cs** -- Starts Csound with a set of options that can be controlled by environment variables, and input and output files determined by the specified filename stem.

cs

## Description

Starts Csound with a set of options that can be controlled by environment variables, and input and output files determined by the specified filename stem.

## Syntax

**cs** [-OPTIONS] <name> [CSOUND OPTIONS ... ]

## Initialization

Flags:

- - *OPTIONS* = OPTIONS is a sequence of alphabetic characters that can be used for selecting the Csound executable to be run, as well as the command line flags (see below). There is a default for the option 'r' (selects real-time output), but it can be overridden.
- <name> = this is the filename stem for selecting input files; it may contain a path. Files that have .csd, .orc, or .sco extension are searched, and either a CSD or an orc/sco pair that matches <name> the best are selected. MIDI files with a .mid extension are also searched, and if one that matches <name> at least as close as the CSD or orc/sco pair, it is used with the -F flag.



### NOTE

The MIDI file is not used if any -M or -F flag is specified by the user - new in version 4.24.0) Unless there is any option (-n or -o) related to audio output, an output file name with the appropriate extension is automatically generated (based on the name of selected input files and format options). The output file is always written to the current directory.



### NOTE

file name extensions are not case sensitive.

- [*CSOUND OPTIONS ...*] = any number of additional options for Csound that are simply copied to the final command line to be executed.

The command line that is executed is generated from four parts:

1. Csound executable (possibly with options). This is exactly one of the following (the last one has the highest precedence):
  - a built-in default
  - the value of the CSOUND environment variable
  - environment variables with a name in the format of CSOUND\_x where x is an uppercase letter selected by characters of the -OPTIONS string. Thus, if the -dcba option is used, and



the environment variables CSOUND\_B and CSOUND\_C are defined, the value of CSOUND\_B will take effect.

2. Any number of option lists, added in the following order:
  - either some built-in defaults, or the value of the CSFLAGS environment variable if it is defined.
  - environment variables with a name in the format of CSFLAGS\_x where x is an uppercase letter selected by characters of the -OPTIONS string. Thus, if the -dcb option is used, and the environment variables CSFLAGS\_A and CSFLAGS\_C are defined as '-M 1 -o dac' and '-m231 -H0', respectively, the string '-m231 -H0 -M 1 -o dac' will be added.
3. The explicit options of [CSOUND OPTIONS ... ].
4. Any options and file names generated from <name>.



## NOTE

Quoted options that contain spaces are allowed.

## Examples

Assuming the following environment variables:

```
CSOUND      = csoundfltk.exe -W
CSOUND_D    = csound64.exe -J
CSOUND_R    = csoundfltk.exe -h

CSFLAGS     = -d -m135 -H1 -s
CSFLAGS_D   = -f
CSFLAGS_R   = -m0 -H0 -o dac1 -M "MIDI Yoke NT: 1" -b 200 -B 6000
```

And a directory that contains:

```
foo.orc      piano.csd
foo.sco      piano.mid
im.csd       piano2.mid
ImproSculpt2_share.csd  foobar.csd
```

The following commands will execute as shown:

```
cs foo          => csoundfltk.exe -W -d -m135 -H1 -s -o foo.wav \
                  foo.orc foo.sco

cs foob         => csoundfltk.exe -W -d -m135 -H1 -s          \
                  -o foobar.wav foobar.csd

cs -r imp -i adc => csoundfltk.exe -h -d -m135 -H1 -s -m0 -H0 \
                  -o dac1 -M "MIDI Yoke NT: 1" \
                  -b 200 -B 6000 -i adc \
                  ImproSculpt2_share.csd

cs -d im        => csound64.exe -J -d -m135 -H1 -s -f -o im.sf \
                  im.csd

cs piano        => csoundfltk.exe -W -d -m135 -H1 -s          \
                  -F piano.mid -o piano.wav \
```

```
                                piano.csd
cs piano2                      => csoundfltk.exe -W -d -m135 -H1 -s      \
                                -F piano2.mid -o piano2.wav           \
                                piano.csd
```

## Credits

Author: Istvan Varga

Jan 2003

## scale

scale -- Scale the amplitude of a sound file.

scale

## Description

Takes a sound file and scales it by applying a gain, either constant or variable. The scale can be specified as a multiplier, a maximum or a percentage of 0db.

## Syntax

**scale** [OPTIONS ... ] infile

## Initialization

Flags:

- *-A* = Generate an AIFF outout file.
- *-W* = Generate an WAV outout file.
- *-h* = Generate an outout file with no header.
- *-c* = Generate 8-bit signed\_char sound samples.
- *-a* = Generate alaw sound samples.
- *-u* = Generate ulaw sound samples.
- *-s* = Generate short integer sound samples.
- *-l* = Generate long (32 bit) integer sound samples.
- *-f* = Generate floating point samples.
- *-F arg* = Specifies the gain to be applied. If arg is a floating point number that gain is applied uniformly to the input. Alternatively it could be a file name which specifies a breakpoint file for varying the gain for different periods.
- *-M fnum* = Scales the input so the maximum absolute displacement is the value given.
- *-P fnum* = Scales the input so the maximum absolute displacement is the percentage given of 0db.
- *-R* = Continually rewrite the header while writing soundfile (WAV/AIFF).
- *-H integer* = Show a "heart-beat" to indicate progress, in style 1, 2 or 3.
- *-N* = Alert call (usually ringing the bell) when finished.
- *-o fname* = output file name (default: test.wav)

## Examples

```
scale -s -W -F 0.96 -o out.wav sound.wav
```

This creates a new sound file with a constant gain of 0.96. It is particularly useful if the input file is in floating point format.

## Credits

Author: John ffitch

1994

## mixer

mixer -- Mixes together a number of soundfiles.

mixer

## Description

Mixes together a number of soundfiles, starting at different times and with individual channel selection from the input files.

## Syntax

```
mixer [OPTIONS ... ] infile [[OPTIONS  
... ] infile] ...
```

## Initialization

Flags:

- *-A* = Generate an AIFF outout file.
- *-W* = Generate an WAV outout file.
- *-h* = Generate an outout file with no header.
- *-c* = Generate 8-bit signed\_char sound samples.
- *-a* = Generate alaw sound samples.
- *-u* = Generate ulaw sound samples.
- *-s* = Generate short integer sound samples.
- *-l* = Generate long (32 bit) integer sound samples.
- *-f* = Generate floating point samples.
- *-F arg* = Specifies the gain to be applied to the following input file. If arg is a floating point number that gain is applied uniformly to the input. Alternatively it could be a file name which specifies a breakpoint file for varying the gain for different periods.
- *-S integer* = Indicate at which sample to start to mix in the next input file.
- *-T fpnum* = Indicate at which time (in seconds) to start to mix in the next input file.
- *-1* = Mix in channel 1 from next sound file.
- *-2* = Mix in channel 2 from next sound file.
- *-3* = Mix in channel 3 from next sound file.
- *-4* = Mix in channel 4 from next sound file.
- *-^ intx inty* = Mix in channel x from next sound file as channel y in the output.
- *-v* = Verbose mode.
- *-R* = Continually rewrite the header while writing soundfile (WAV/AIFF).

- *-H integer* = Show a "heart-beat" to indicate progress, in style 1, 2 or 3.
- *-N* = Alert call (usually ringing the bell) when finished.
- *-o fname* = output file name (default: test.wav)

## Examples

The default values are

```
mixer -s -otest -F 1.0 -S 0
```

For example

```
mixer -F 0.96 in1.wav -S 300 -2 in2.aiff -S 300  
-^4 1 in3.wav -o out.wav
```

This creates a new sound file with a constant gain of 0.96 from in1.wav with the second channel of in2.aiff mixed in after 300 samples and channel 4 of in3.wav output as channel 1 after 300 samples.

## Credits

Author: John ffitch

1994

## extractor

extractor -- Extract a section of audio from an audio file.

extractor

## Description

Extract a section of audio, by time or sample, from an existing sound file.

## Syntax

**extractor** [OPTIONS ... ] infile

## Initialization

Flags:

- *-S integer* = Start the extract at the given sample number.
- *-Z integer* = End the extract at the given sample number.
- *-Q integer* = Extract given number of samples.
- *-T fnum* = Start the extract at the given time in seconds.
- *-E fnum* = End the extract at the given time in seconds.
- *-D fnum* = Extract given time in seconds.
- *-v* = Verbose mode.
- *-R* = Continually rewrite the header while writing soundfile (WAV/AIFF).
- *-H integer* = Show a "heart-beat" to indicate progress, in style 1, 2 or 3.
- *-N* = Alert call (usually ringing the bell) when finished.
- *-v* = Verbose mode.
- *-o fname* = output file name (default: test.wav)

## Examples

The default values are

```
extractor -S 0 -Z end-of-file -otest
```

For example

```
extractor -S 10234 -D 2.13 in.aiff -o out.wav
```

This creates a new sound file taken from sample 10234 and lasting 2.13 seconds.

## Credits

Author: John ffitich

1994



---

# Cscore

*Cscore* is a program for generating and manipulating numeric score files. It comprises a number of function subprograms, called into operation by a user-written control program, and can be invoked either as a standalone score preprocessor, or as part of the Csound run-time system:

```
Cscore [scorefilein] [scorefileout]
```

or

```
Csound [-C] [otherflags] [orchname] [scorename]
```

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

The user-written control program is also in C, and is compiled and linked to the function programs (or the entire Csound) by the user. It is not essential to know the C language well to write this program, since the function calls have a simple syntax, and are powerful enough to do most of the complicated work. Additional power can come from C later as the need arises.

## Events, Lists, and Operations

An event in *Cscore* is equivalent to one statement of a *standard numeric score* or time-warped score (see any score.srt), stored internally in time-warped format. It is either created in-line, or read in from an existing score file (either format). Its main components are an opcode and an array of pfield values. It is stored somewhere in memory, organized by a structure that starts as follows:

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

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

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

Any function that creates or modifies a list will return a pointer to the new list. The list pointer can be used to access any of its component event pointers, in the form of *a-e[n]*. Event pointers and list pointers are thus primary tools for manipulating the data of a score file. Pointers and lists of pointers can be copied and reordered without modifying the data values they refer to. This means that notes and phrases can be copied and manipulated from a high level of control. Alternatively, the data within an event or group of events can be modified without changing the event or list pointers. The *Cscore* function subprograms enable scores to be created and manipulated in this way.

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

the symbols e, f are pointers to events (notes);	
the symbols a, b are pointers to lists (arrays) of such events;	
the letters ev at the end of a function name signify operation on an event;	
the letter l at the start of a function name signifies operation on a list.	
the symbol fp is a score input stream file pointer (FILE *);	
calling syntax	description
e = createv(n);	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

## 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 *fable* 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 */
    lrele(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);
    lrele(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-->0) {
            (*f)-p[6] -= dim;
            (*f)-p[2] += cue[cuecount]; /* add values of cue */
            f++;
        }
        printf("; diagnostic: cue = %d\n", cue[cuecount]);
        cuecount++;
        dim += 2000;
        lput(a);
    }
    putstr("e");
}
```

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

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

```
f 1 0 257 10 1
f 2 0 257 7 0 300 1 212 .8
t 0 120
; diagnostic: cue = 0
i 1 1 3 0 440 10000
i 1 4 3 0 256 10000
i 1 7 3 0 880 10000
; diagnostic: cue = 10
i 1 11 3 0 440 8000
i 1 14 3 0 256 8000
i 1 17 3 0 880 8000
; diagnostic: cue = 17
i 1 28 3 0 440 4000
i 1 31 3 0 256 4000
i 1 34 3 0 880 4000
e;
```

## More Advanced Examples

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

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

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

```
./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 pulsel6[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 = pulsel6;
    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 = pulsel6, n1 = 48; n1--; p += 4)  /* show vals & diffs */
/*   printf("%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] = pulsel6[(int)(4. * e-p2orig)];
    e-p[3] = pulsel6[(int)(4. * (e-p2orig + e-p3orig))] - e-p[2];
  }
}

lplay(a); /* now play modified list */
}
```

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

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

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

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

## Compiling a Cscore Program

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

```
cscore -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 -lcsound -lX11 -lm (X11 if your installation included it)
```

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

---

# Extending Csound

## Adding Unit Generators

If the existing Csound unit generators do not suit your needs, it is relatively easy to extend Csound by writing new unit generators in C or C++. The translator, loader, and run-time monitor will treat your module just like any other provided you follow some conventions.

Historically, this has been done with builtin unit generators, that is, with code that is statically linked with the rest of the Csound executable.

Today, the preferred method is to create plugin unit generators. These are dynamic link libraries (DLLs) on Windows, and loadable modules (shared libraries that are `dlopened`) on Linux. Csound searches for and loads these plugins at run time. The advantage of this method, of course, is that plugins created by any developer at any time can be used with already existing versions of Csound.

## Creating a Builtin Unit Generator

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`. The examples given are for Csound 5. For earlier versions, all opcode functions omit the first parameter (`CSOUND *csound`).

```
/* newgen.h - define a structure */

/* Declares Csound structures and functions. */
#include "csoundCore.h"

typedef struct
{
    OPDS h;
    MYFLT *result, *istrt, *incr, *itime, *icontin; /* required header */
    MYFLT curval, vincr; /* addr outarg, inargs */
    long countdown; /* private dataspace */
} RMP; /* ditto */

/* newgen.c - init and perf code */
/* Declares Csound structures and functions. */
#include "csoundCore.h"
/* Declares RMP structure. */
#include "newgen.h"

int rampset (CSOUND *csound, RMP * p) /* at note initialization: */
{
    if (*p->icontin == FL(0.0))
        p->curval = *p->istrt; /* optionally get new start value */
    p->vincr = *p->incr / csound->esr; /* set s-rate increment per sec. */
    p->countdown = *p->itime * csound->esr; /* counter for itime seconds */
    return OK;
}

int ramp (CSOUND *csound, RMP * p) /* during note performance: */
{
    MYFLT *rsltp = p->result; /* init an output array pointer */
    int nn = csound->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);
    return OK;
}

```

Now we add this module to the translator table in `entry1.c`, under the opcode name `rampt`:

```

#include "newgen.h"

int rampset(CSOUND *, RMP *), ramp(CSOUND *, RMP *);

/*  opname      dsblksiz  thread      outtypes  intypes  iopadr      kopadr      aopa
{ "ramp",      S(RMP),   5,          "a",       "iiio",      (SUBR) rampset, (SUBR) NUL

```

Finally you must relink Csound with the new module. Add the name of the C file to the `libCsoundSources` list in the `SConstruct` file:

```

libCsoundSources = Split(''
Engine/auxfd.c
...
OOps/newgen.c
...
Top/utility.c
'' )

```

Run `scons` just as you would for any other Csound build, and the new module will be built into your Csound.

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 values of type `MYFLT` (which is double if the macro `USE_DOUBLE` is defined, and float otherwise). There are no restrictions on names, but the input-output argument types are further defined by character strings in `entry1.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 `entry1.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 Csound instance and 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 `entry1.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 (but should only read) public members of the CSOUND structure defined in `csoundCore.h`, the most useful of which are:

OPARMS	*oparms	
MYFLT	esr	user-defined sampling rate
MYFLT	ekr	user-defined control rate
int	ksmps	user-defined ksmps
int	nchnls	user-defined nchnls
int	oparms->odebug	command-line -v flag
int	oparms->msglevel	command-line -m level
MYFLT	tpidsr	2 * PI / esr

## 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 = csound->FTFind(csound, p->ifuncno);
```

where MYFLT \*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 `csoundCore.h`, the `csoundFTFind()` code in `fgens.c`, and the code for `oscset()` and `koscil()` in `OOps/ugens2.c`.

## Additional Space

Sometimes the space requirement of a module is too large to be part of a structure (upper limit 65279 bytes, due to the unsigned short `dsblksiz` parameter and reserved codes `>= 0xFF00`), 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:

```
csound->AuxAlloc(csound, npoints * sizeof(MYFLT), &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

```
void *auxp = p->auxch.auxp;
```

will find the allocated space for init-time and perf-time use. See the LINSEG structure in ugens1.h and the code for lsgset() and klnseg() in Oops/ugens1.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:

```
p->mfp = csound->ldmemfile(csound, filename);
```

where char \*filename 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 ugens3.h and the code for adset() and adsyn() in Oops/ugens3.c.

## String arguments

To permit a string input argument (MYFLT \*ifilnam, say) in our defined structure (\*p), assign it the argtype *S* in entry1.c, and include the following code in the init module:

```
strcpy(filename, (char*) p->ifilnam);
```

See the code for `adset()` in `OOps/ugens3.c`, `lprdset()` in `OOps/ugens5.c`, and `pvset()` in `OOps/ugens8.c`.

## Adding a Plugin Unit Generator

The procedure for creating a plugin unit generator is very similar to the procedure for creating a builtin. The actual unit generator code would normally be identical. The differences are as follows.

Again supposing that your unit generator is named `newgen`, perform the following steps:

1. Write your `newgen.c` and `newgen.h` file as you would for a builtin unit generator. Put these files in the `csound5/Opcodes` directory.
2. `#include "csdl.h"` in your unit generator sources, instead of `csoundCore.h`.
3. Add your `OENTRY` records and unit generator registration functions at the bottom of your C file. Example (but you can have as many unit generators in one plugin as you like):

```
#define S sizeof

static OENTRY localops[] = {
{
    { "rampt", S(RMP), 5, "a", "iiio", (SUBR) rampset, (SUBR) NULL, (
};

/*
 * The following macro from csdl.h defines
 * the "csound_opcode_init()" opcode registration
 * function for the localops table.
 */
LINKAGE
```

4. Add your plugin as a new target in the plugin opcodes section of the `SConstruct` build file:

```
pluginEnvironment.SharedLibrary('newgen',
    Split(''Opcodes/newgen.c
    Opcodes/another_file_used_by_newgen.c
    Opcodes/yet_another_file_used_by_newgen.c''))
```

5. Run the Csound 5 build in the regular way.

## OENTRY Reference

The `OENTRY` structure (see `H/csoundCore.h`, `Engine/entry1.c`, and `Engine/rdorch.c`) contains the following public fields:

`opname`, `dsblksiz`, `thread`, `outtypes`, `intypes`, `iopadr`, `kopadr`, `aopadr`

**dsblksiz** There are two types of opcodes, polymorphic and non-polymorphic. For non-polymorphic opcodes, the `dsblksiz` flag specifies the size of the opcode structure in bytes, and arguments are always passed to the opcode at the same rate. Polymorphic opcodes can accept arguments at different rates, and those arguments are actually dispatched to other opcodes as determined by the `dsblksiz` flag and the following naming convention (note: the following list is not complete, see `Engine/entry1.c` for all possible special `dsblksiz` codes):

0xffff The type of the first output argument determines which unit generator function is actually called: XXX -> XXX.a, XXX.i, or XXX.k.

0xfffe The types of the first two input arguments determine which unit generator function is actually called: XXX -> XXX.aa, XXX.ak, XXX.ka, or XXX.kk, as in the `oscil` unit generator.

0xfffd Refers to one input argument of type a or k, as in the `peak` unit generator.

thread Specifies the rate(s) at which the unit generator's functions are called, as follows:

**Table 1.**

0	i-rate <i>or</i> k-rate (B out only)
1	i-rate
2	k-rate
3	i-rate <i>and</i> k-rate
4	a-rate
5	i-rate <i>and</i> a-rate
7	i-rate <i>and</i> (k-rate <i>or</i> a-rate)

outypes Lists the return values of the unit generator functions, if any. The types allowed are (note: the following list is not complete, see `Engine/entry1.c` for all possible output types):

**Table 2.**

i	i-rate scalar
k	k-rate scalar
a	a-rate vector
x	k-rate vector or a-rate vector
f	f-rate streaming pvoc fsig type
m	multiple a-rate output arguments

intypes Lists the arguments the unit generator functions take, if any. The types allowed are (note: the following list is not complete, see `Engine/entry1.c` for all possible input types):

**Table 3.**

i	i-rate scalar
k	k-rate scalar
a	a-rate vector
x	k-rate vector or a-rate vector
f	f-rate streaming pvoc fsig type
S	String
B	
l	
m	Begins an indefinite list of i-rate



	arguments (any count)
M	Begins an indefinite list of arguments (any rate, any count)
n	Begins an indefinite list of i-rate arguments (any odd count)
o	Optional i-rate, defaulting to 0
p	Optional i-rate, defaulting to 1
q	Optional i-rate, defaulting to 10
v	Optional i-rate, defaulting to 0.5
j	Optional i-rate, defaulting to -1
h	Optional i-rate, defaulting to 127
y	Begins an indefinite list of a-rate arguments (any count)
z	Begins an indefinite list of k-rate arguments (any count)
Z	Begins an indefinite list of alternating k-rate and a-rate arguments (kaka...) (any count)

iopadr    The address of the unit generator function (of type `int (*SUBR)(CSOUND *, void *)`) that is called at i-time, or NULL for no function.

kopadr    The address of the unit generator function (of type `int (*SUBR)(CSOUND *, void *)`) that is called at k-rate, or NULL for no function.

aopadr    The address of the unit generator function (of type `int (*SUBR)(CSOUND *, void *)`) that is called at a-rate, or NULL for no function.

---

# Appendix A. Pitch Conversion

**Table A.1. Pitch Conversion**

Note	Hz	cpspch	MIDI
C-1	8.176	3.00	0
C#-1	8.662	3.01	1
D-1	9.177	3.02	2
D#-1	9.723	3.03	3
E-1	10.301	3.04	4
F-1	10.913	3.05	5
F#-1	11.562	3.06	6
G-1	12.250	3.07	7
G#-1	12.978	3.08	8
A-1	13.750	3.09	9
A#-1	14.568	3.10	10
B-1	15.434	3.11	11
C0	16.352	4.00	12
C#0	17.324	4.01	13
D0	18.354	4.02	14
D#0	19.445	4.03	15
E0	20.602	4.04	16
F0	21.827	4.05	17
F#0	23.125	4.06	18
G0	24.500	4.07	19
G#0	25.957	4.08	20
A0	27.500	4.09	21
A#0	29.135	4.10	22
B0	30.868	4.11	23
C1	32.703	5.00	24
C#1	34.648	5.01	25
D1	36.708	5.02	26
D#1	38.891	5.03	27
E1	41.203	5.04	28
F1	43.654	5.05	29
F#1	46.249	5.06	30
G1	48.999	5.07	31
G#1	51.913	5.08	32
A1	55.000	5.09	33
A#1	58.270	5.10	34
B1	61.735	5.11	35
C2	65.406	6.00	36
C#2	69.296	6.01	37
D2	73.416	6.02	38

Pitch Conversion

Note	Hz	cpspch	MIDI
D#2	77.782	6.03	39
E2	82.407	6.04	40
F2	87.307	6.05	41
F#2	92.499	6.06	42
G2	97.999	6.07	43
G#2	103.826	6.08	44
A2	110.000	6.09	45
A#2	116.541	6.10	46
B2	123.471	6.11	47
C3	130.813	7.00	48
C#3	138.591	7.01	49
D3	146.832	7.02	50
D#3	155.563	7.03	51
E3	164.814	7.04	52
F3	174.614	7.05	53
F#3	184.997	7.06	54
G3	195.998	7.07	55
G#3	207.652	7.08	56
A3	220.000	7.09	57
A#3	233.082	7.10	58
B3	246.942	7.11	59
C4	261.626	8.00	60
C#4	277.183	8.01	61
D4	293.665	8.02	62
D#4	311.127	8.03	63
E4	329.628	8.04	64
F4	349.228	8.05	65
F#4	369.994	8.06	66
G4	391.995	8.07	67
G#4	415.305	8.08	68
A4	440.000	8.09	69
A#4	466.164	8.10	70
B4	493.883	8.11	71
C5	523.251	9.00	72
C#5	554.365	9.01	73
D5	587.330	9.02	74
D#5	622.254	9.03	75
E5	659.255	9.04	76
F5	698.456	9.05	77
F#5	739.989	9.06	78
G5	783.991	9.07	79
G#5	830.609	9.08	80
A5	880.000	9.09	81
A#5	932.328	9.10	82

Pitch Conversion

Note	Hz	cpSPch	MIDI
B5	987.767	9.11	83
C6	1046.502	10.00	84
C#6	1108.731	10.01	85
D6	1174.659	10.02	86
D#6	1244.508	10.03	87
E6	1318.510	10.04	88
F6	1396.913	10.05	89
F#6	1479.978	10.06	90
G6	1567.982	10.07	91
G#6	1661.219	10.08	92
A6	1760.000	10.09	93
A#6	1864.655	10.10	94
B6	1975.533	10.11	95
C7	2093.005	11.00	96
C#7	2217.461	11.01	97
D7	2349.318	11.02	98
D#7	2489.016	11.03	99
E7	2637.020	11.04	100
F7	2793.826	11.05	101
F#7	2959.955	11.06	102
G7	3135.963	11.07	103
G#7	3322.438	11.08	104
A7	3520.000	11.09	105
A#7	3729.310	11.10	106
B7	3951.066	11.11	107
C8	4186.009	12.00	108
C#8	4434.922	12.01	109
D8	4698.636	12.02	110
D#8	4978.032	12.03	111
E8	5274.041	12.04	112
F8	5587.652	12.05	113
F#8	5919.911	12.06	114
G8	6271.927	12.07	115
G#8	6644.875	12.08	116
A8	7040.000	12.09	117
A#8	7458.620	12.10	118
B8	7902.133	12.11	119
C9	8372.018	13.00	120
C#9	8869.844	13.01	121
D9	9397.273	13.02	122
D#9	9956.063	13.03	123
E9	10548.08	13.04	124
F9	11175.30	13.05	125
F#9	11839.82	13.06	126

Pitch Conversion

---

Note	Hz	cpspch	MIDI
G9	12543.85	13.07	127

---

# Appendix B. Sound Intensity Values

**Table B.1. Sound Intensity Values (for a 1000 Hz tone)**

Dynamics	Intensity (W/m <sup>2</sup> )	Level (dB)
pain	1	120
fff	10 <sup>-2</sup>	100
f	10 <sup>-4</sup>	80
p	10 <sup>-6</sup>	60
ppp	10 <sup>-8</sup>	40
threshold	10 <sup>-12</sup>	0

---

# Appendix C. Formant Values

**Table C.1. alto “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	800	1150	2800	3500	4950
amp (dB)	0	-4	-20	-36	-60
bw (Hz)	80	90	120	130	140

**Table C.2. alto “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	1600	2700	3300	4950
amp (dB)	0	-24	-30	-35	-60
bw (Hz)	60	80	120	150	200

**Table C.3. alto “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	1700	2700	3700	4950
amp (dB)	0	-20	-30	-36	-60
bw (Hz)	50	100	120	150	200

**Table C.4. alto “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	450	800	2830	3500	4950
amp (dB)	0	-9	-16	-28	-55
bw (Hz)	70	80	100	130	135

**Table C.5. alto “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	325	700	2530	3500	4950
amp (dB)	0	-12	-30	-40	-64
bw (Hz)	50	60	170	180	200

**Table C.6. bass “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	600	1040	2250	2450	2750
amp (dB)	0	-7	-9	-9	-20

Values	f1	f2	f3	f4	f5
bw (Hz)	60	70	110	120	130

**Table C.7. bass “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	1620	2400	2800	3100
amp (dB)	0	-12	-9	-12	-18
bw (Hz)	40	80	100	120	120

**Table C.8. bass “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	250	1750	2600	3050	3340
amp (dB)	0	-30	-16	-22	-28
bw (Hz)	60	90	100	120	120

**Table C.9. bass “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	750	2400	2600	2900
amp (dB)	0	-11	-21	-20	-40
bw (Hz)	40	80	100	120	120

**Table C.10. bass “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	600	2400	2675	2950
amp (dB)	0	-20	-32	-28	-36
bw (Hz)	40	80	100	120	120

**Table C.11. countertenor “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	660	1120	2750	3000	3350
amp (dB)	0	-6	-23	-24	-38
bw (Hz)	80	90	120	130	140

**Table C.12. countertenor “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	440	1800	2700	3000	3300
amp (dB)	0	-14	-18	-20	-20



Values	f1	f2	f3	f4	f5
bw (Hz)	70	80	100	120	120

**Table C.13. countertenor “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	270	1850	2900	3350	3590
amp (dB)	0	-24	-24	-36	-36
bw (Hz)	40	90	100	120	120

**Table C.14. countertenor “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	430	820	2700	3000	3300
amp (dB)	0	-10	-26	-22	-34
bw (Hz)	40	80	100	120	120

**Table C.15. countertenor “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	370	630	2750	3000	3400
amp (dB)	0	-20	-23	-30	-34
bw (Hz)	40	60	100	120	120

**Table C.16. soprano “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	800	1150	2900	3900	4950
amp (dB)	0	-6	-32	-20	-50
bw (Hz)	80	90	120	130	140

**Table C.17. soprano “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	2000	2800	3600	4950
amp (dB)	0	-20	-15	-40	-56
bw (Hz)	60	100	120	150	200

**Table C.18. soprano “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	270	2140	2950	3900	4950
amp (dB)	0	-12	-26	-26	-44

Values	f1	f2	f3	f4	f5
bw (Hz)	60	90	100	120	120

**Table C.19. soprano “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	450	800	2830	3800	4950
amp (dB)	0	-11	-22	-22	-50
bw (Hz)	40	80	100	120	120

**Table C.20. soprano “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	325	700	2700	3800	4950
amp (dB)	0	-16	-35	-40	-60
bw (Hz)	50	60	170	180	200

**Table C.21. tenor “a”**

Values	f1	f2	f3	f4	f5
freq (Hz)	650	1080	2650	2900	3250
amp (dB)	0	-6	-7	-8	-22
bw (Hz)	80	90	120	130	140

**Table C.22. tenor “e”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	1700	2600	3200	3580
amp (dB)	0	-14	-12	-14	-20
bw (Hz)	70	80	100	120	120

**Table C.23. tenor “i”**

Values	f1	f2	f3	f4	f5
freq (Hz)	290	1870	2800	3250	3540
amp (dB)	0	-15	-18	-20	-30
bw (Hz)	40	90	100	120	120

**Table C.24. tenor “o”**

Values	f1	f2	f3	f4	f5
freq (Hz)	400	800	2600	2800	3000
amp (dB)	0	-10	-12	-12	-26

---

Values	f1	f2	f3	f4	f5
bw (Hz)	70	80	100	130	135

**Table C.25. tenor “u”**

Values	f1	f2	f3	f4	f5
freq (Hz)	350	600	2700	2900	3300
amp (dB)	0	-20	-17	-14	-26
bw (Hz)	40	60	100	120	120

---

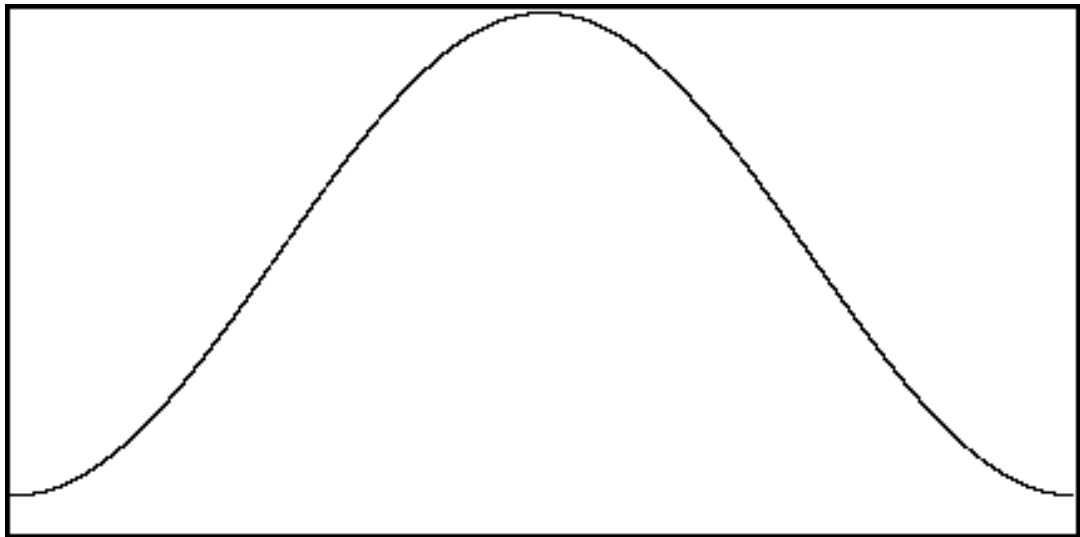
# Appendix D. Window Functions

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.**

## Example D.1. Hamming window function statement

```
f81 0 8192 20 1 1
```

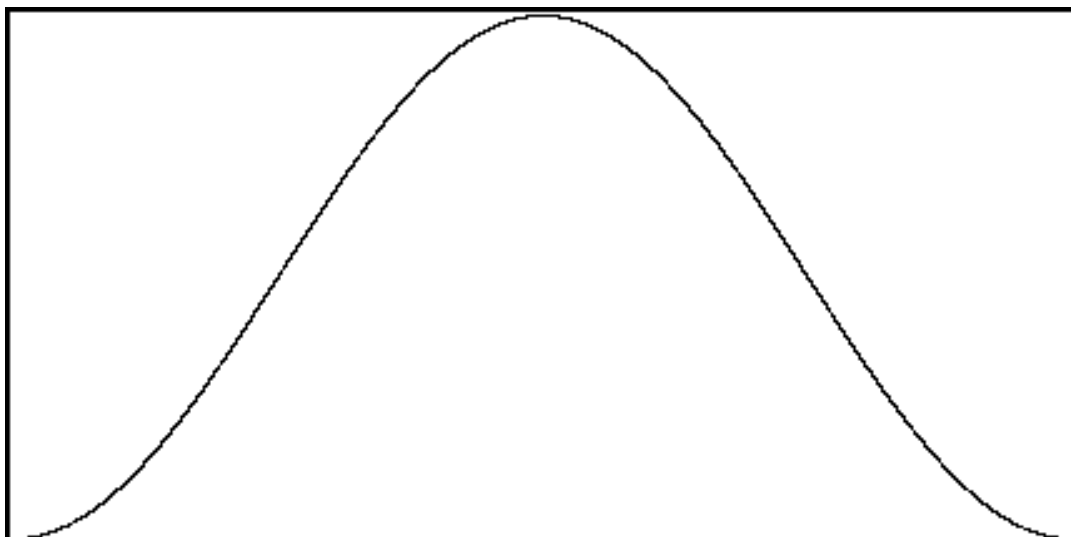


Hamming Window Function.

**Hanning.**

## Example D.2. Hanning window function statement

```
f82 0 8192 20 2 1
```

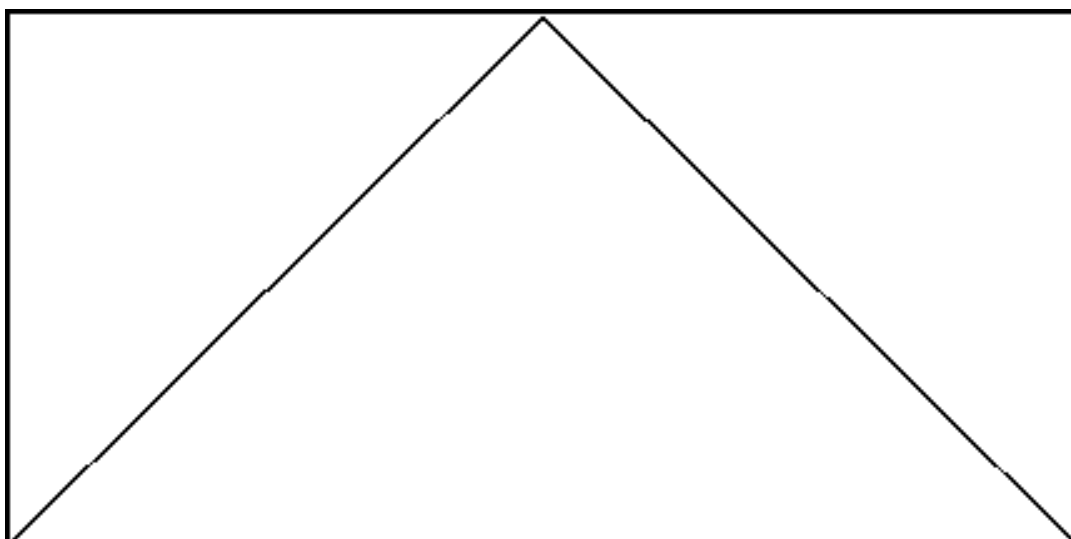


Hanning Window Function

**Bartlett.**

**Example D.3. Bartlett window function statement**

```
f83    0    8192    20    3    1
```

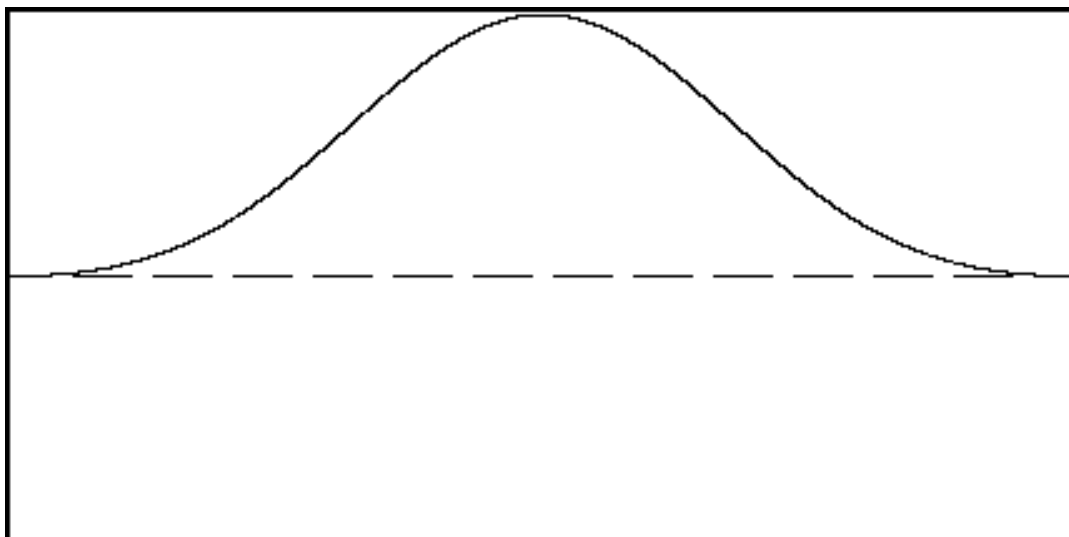


Bartlett Window Function

**Blackman.**

**Example D.4. Blackman window function statement**

```
f84    0    8192    20    4    1
```

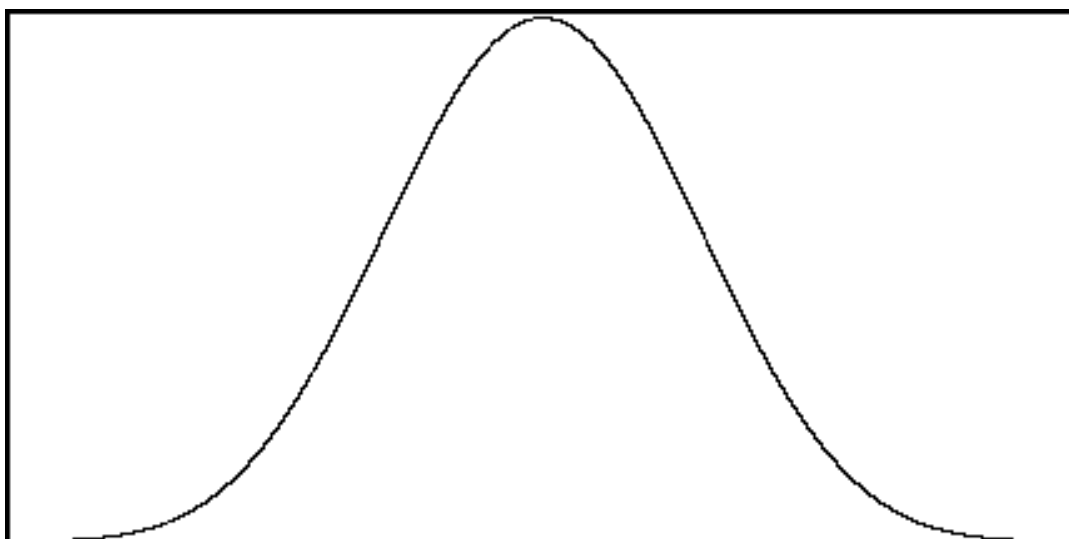


Blackman Window Function

**Blackman-Harris.**

**Example D.5. Blackman-Harris window function statement**

```
f85    0    8192    20    5    1
```

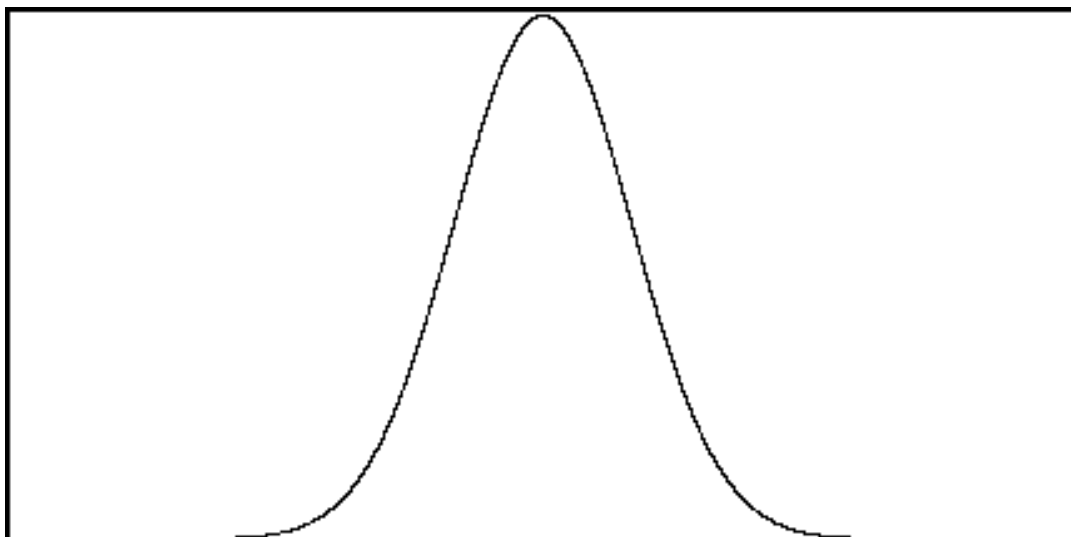


Blackman-Harris Window Function

**Gaussian.**

**Example D.6. Gaussian window function statement**

```
f86    0    8192    20    6    1
```



Gaussian Window Function

**Rectangle.**

**Example D.7. Rectangle window function statement**

```
f88  0  8192  -20  8  .1
```

*Note:* Vertical scale is exaggerated in this diagram.

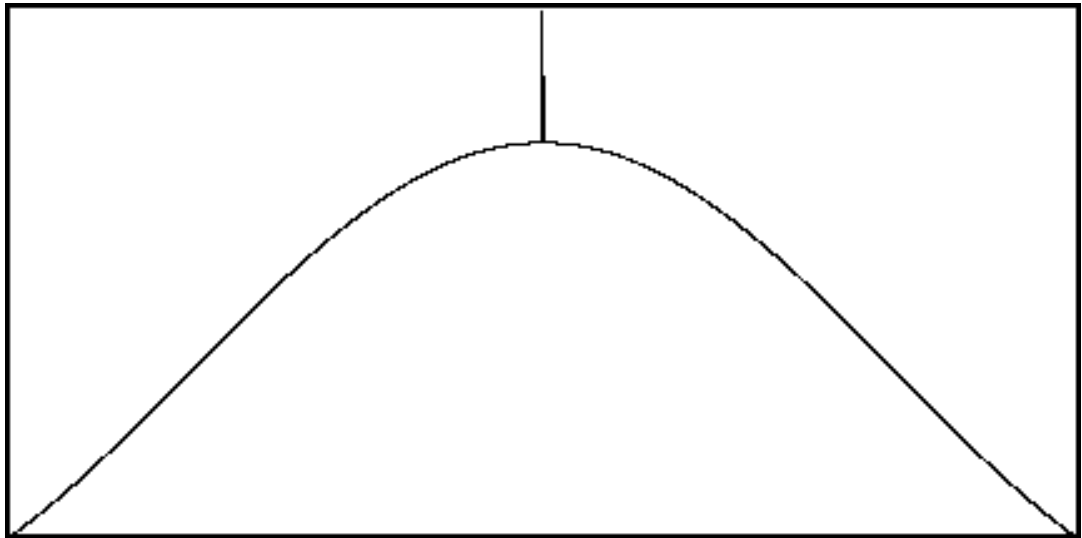


Rectangle Window Function

**Sync.**

**Example D.8. Sync window function statement**

```
f89  0  4096  -20  9  .75
```



Sync Window Function



---

# Appendix E. SoundFont2 File Format

Beginning with Csound Version 4.07, *Csound supports the 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.

---

# Appendix F. Csound64

Csound64 is a version of Csound that uses 64-bit DOUBLE's internally to do processing versus regular Csound's 32-bit FLOAT's. This larger resolution for processing internally yields a much "cleaner" sound but at the expense of extended processing time. Because it does require much longer to process, Csound64 is typically used after a work is finished for a final production run.

## Notes On Using Csound64.

1. hetro files generated for Csound will work with Csound64.
2. PVOC-EX analysis and pvanal files generated for Csound will not work with Csound64. For Csound64, use of pvanal and pvoc opcodes are not currently supported. If your work file uses pvoc, use Csound. (This is a temporary issue relating to older file formats and is currently being addressed and worked on.)
3. lpanal files generated for Csound will not work with Csound64. For Csound64, use of lpanal and lpc opcodes are not currently supported. If your work file uses lpc, use Csound. (This is a temporary issue relating to older file formats and is currently being addressed and worked on.)
4. cvanal files generated for Csound will not work with Csound64. To generated cv files usable by Csound64, use the following command line:

```
csound64 -U cvanal
```

instead of either of the following:

```
csound -U cvanal  
cvanal
```

This will generate a 64-bit cv file. If you were working with 32-bit Csound and using a 32-bit cv file, the cv file will not work with Csound64. When you switch to Csound64, you will need to use a 64-bit generated cv file.

---

# Glossary

## G

### Guard Point

A guard point is the last position on a function table. If the length is, say 1024, the table will have 1024+1 (1025) points: the extra point is the guard point.

In any case, for a 1024-point table, the first point is index 0 and the last 1023; index 1024 is not really used)

The reason for a guard-point is that some opcodes interpolate to obtain a table value, in which case, when the table index is say, 1023.5, we need the value of the 1024 pos in order to interpolate.

There are two ways of filling this point (writing the value that goes in it):

1. Default way: by copying the value of the 1st point in the table
2. Extended Guard-Point: extending the contour of the table (continuing to calculate the table for one extra point)

In general the first mode is used for wrap-around applications, such as an oscillator (which loops continuously reading the table). The second use is for one-shot readouts, such as envelopes, where the last point needs to be interpolated correctly following the table contour (we are not looping back to the beginning of the table)

---

# Appendix G. Quick Reference

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

`#define NAME # replacement text #`

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

`#include "filename"`

`#undef NAME`

`$NAME`

`a % b (no rate restriction)`

`a && b (logical AND; not audio-rate)`

`(a > b ? v1 : v2)`

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

`(a < b ? v1 : v2)`

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

`a * b (no rate restriction)`

`+ a (no rate restriction)`

`# a (no rate restriction)`

`a / b (no rate restriction)`

`ar = xarg`

`ir = iarg`

`kr = karg`

`(a == b ? v1 : v2)`

`a ^ b (b not audio-rate)`

`a || b (logical OR; not audio-rate)`

**0dbfs** = iarg

**a(x)** (control-rate args only)

**abs(x)** (no rate restriction)

ir **active** insnum

kr **active** kinsnum

ar **adsr** iatt, idec, islev, irel [, idel]

kr **adsr** iatt, idec, islev, irel [, idel]

ar **adsyn** kamod, kfmod, ksmod, ifilcod

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

kaft **aftouch** [imin] [, imax]

ar **alpass** asig, krvt, ilpt [, iskip] [, insmps]

**ampdbfs(x)** (no rate restriction)

**ampdb(x)** (no rate restriction)

iamp **ampmidi** iscal [, ifn]

kr **aresonk** ksig, kcf, kbw [, iscl] [, iskip]

ar **areson** asig, kcf, kbw [, iscl] [, iskip]

kr **atonek** ksig, khp [, iskip]

ar **atone** asig, khp [, iskip]

ar **atonex** asig, khp [, inumlayer] [, iskip]

a1, a2 **babo** asig, ksrcx, ksrcy, ksrcz, irx, iry, irz [, idiff] [, ifno]

ar **balance** asig, acomp [, ihp] [, iskip]

ar **bamboo** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1]

a1 **bbcutm** asource, ibps, isubdiv, ibarlength, iphrasebars, inumrepeats [, istut]

a1,a2 **bbcuts** asource1, asource2, ibps, isubdiv, ibarlength, iphrasebars, inumre

ar **betarand** krange, kalpha, kbeta

ir **betarand** krange, kalpha, kbeta

kr **betarand** krange, kalpha, kbeta

ar **bexprnd** krange

ir **bexprnd** krange

kr **bexprnd** krange

ar **biquada** asig, ab0, ab1, ab2, aa0, aa1, aa2 [, iskip]

ar **biquad** asig, kb0, kb1, kb2, ka0, ka1, ka2 [, iskip]

**birnd**(x) (init- or control-rate only)

ar **bqrez** asig, xfco, xres [, imode]

ar **butbp** asig, kfreq, kband [, iskip]

ar **butbr** asig, kfreq, kband [, iskip]

ar **buthp** asig, kfreq [, iskip]

ar **butlp** asig, kfreq [, iskip]

ar **butterbp** asig, kfreq, kband [, iskip]

ar **butterbr** asig, kfreq, kband [, iskip]

ar **butterhp** asig, kfreq [, iskip]

ar **butterlp** asig, kfreq [, iskip]

kr **button** knum

ar **buzz** xamp, xcps, knh, ifn [, iphs]

ar **cabasa** iamp, idettack [, inum] [, idamp] [, imaxshake]

ar **cauchy** kalpha

ir **cauchy** kalpha

kr **cauchy** kalpha

**cent**(x)

**cggoto** condition, label

ival **chanctrl** ichnl, ictrlno [, ilow] [, ihigh]

kval **chanctrl** ichnl, ictrlno [, ilow] [, ihigh]

kr **checkbox** knum

**cigoto** condition, label

**ckgoto** condition, label

**clear** avar1 [, avar2] [, avar3] [...]

ar **clfilt** asig, kfreq, itype, inpol [, ikind] [, ipbr] [, isba] [, iskip]

ar **clip** asig, imeth, ilimit [, iarg]

**clockoff** inum

**clockon** inum

**cngoto** condition, label

ar **comb** asig, krvt, ilpt [, iskip] [, insmps]

kr **control** knum

ar1 [, ar2] [, ar3] [, ar4] **convle** ain, ifilcod [, ichannel]

ar1 [, ar2] [, ar3] [, ar4] **convolve** ain, ifilcod [, ichannel]

**cosh**(x) (no rate restriction)

**cosinv**(x) (no rate restriction)

**cos**(x) (no rate restriction)

icps **cps2pch** ipch, iequal

icps **cpsmidib** [irange]

kcps **cpsmidib** [irange]

icps **cpsmidi**

**cpsoct** (oct) (no rate restriction)

**cpspch** (pch) (init- or control-rate args only)

icps **cpstmid** ifn

icps **cpstuni** index, ifn

kcps **cpstun** ktrig, kindex, kfn

icps **cpsxpch** ipch, iequal, irepeat, ibase

**cpuprc** insnum, ipercent

ar **cross2** ain1, ain2, isize, ioverlap, iwin, kbias

ar **crunch** iamp, idettack [, inum] [, idamp] [, imaxshake]

idest **ctrl14** ichan, ictrlno1, ictrlno2, imin, imax [, ifn]

kdest **ctrl14** ichan, ictrlno1, ictrlno2, kmin, kmax [, ifn]

idest **ctrl21** ichan, ictrlno1, ictrlno2, ictrlno3, imin, imax [, ifn]

kdest **ctrl21** ichan, ictrlno1, ictrlno2, ictrlno3, kmin, kmax [, ifn]

idest **ctrl7** ichan, ictrlno, imin, imax [, ifn]

kdest **ctrl7** ichan, ictrlno, kmin, kmax [, ifn]

**ctrlinit** ichn1, ictrlno1, ival1 [, ictrlno2] [, ival2] [, ictrlno3] [, ival3] [...]

aout **cuserrnd** kmin, kmax, ktableNum

iout **cuserrnd** imin, imax, itableNum

kout **cuserrnd** kmin, kmax, ktableNum

ar **dam** asig, kthreshold, icomp1, icomp2, irtime, iftime



**dbamp**(x) (init-rate or control-rate args only)

**dbfsamp**(x) (init-rate or control-rate args only)

**db**(x)

ar **dcblock** ain [, igain]

ar **dconv** asig, isize, ifn

ar **delay1** asig [, iskip]

ar **delayr** idlt [, iskip]

ar **delay** asig, idlt [, iskip]

**delayw** asig

ar **deltap3** xdlt

ar **deltapi** xdlt

ar **deltapn** xnumsamps

ar **deltap** kdlt

aout **deltapx** adel, iwsiz

**deltapxw** ain, adel, iwsiz

ar **diff** asig [, iskip]

kr **diff** ksig [, iskip]

ar1 [,ar2] [, ar3] [, ar4] **diskin** ifilcod, kpitch [, iskiptim] [, iwraparound]

**dispfft** xsig, iprd, iwsiz [, iwtyp] [, idbout] [, iwtflg]

**display** xsig, iprd [, inprds] [, iwtflg]

ar **distort1** asig, kpregain, kpostgain, kshapel, kshape2

ar **divz** xa, xb, ksubst

ir **divz** ia, ib, isubst

kr **divz** ka, kb, ksubst

kr **downsamp** asig [, iwlen]

ar **dripwater** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq

**dumpk2** ksig1, ksig2, ifilename, iformat, iprd

**dumpk3** ksig1, ksig2, ksig3, ifilename, iformat, iprd

**dumpk4** ksig1, ksig2, ksig3, ksig4, ifilename, iformat, iprd

**dumpk** ksig, ifilename, iformat, iprd

aout **duserrrnd** ktableNum

iout **duserrrnd** itableNum

kout **duserrrnd** ktableNum

**elseif** xa R xb **then**

**else**

**endif**

**endin**

**endop**

ar **envlpxr** xamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod] [,irind]

kr **envlpxr** kamp, irise, idur, idec, ifn, iatss, iatdec [, ixmod] [,irind]

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

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

**event** "scorechar", kinsnum, kdelay, kdur, [, kp4] [, kp5] [, ...]

**event** "scorechar", "insname", kdelay, kdur, [, kp4] [, kp5] [, ...]

ar **expon** ia, idur1, ib

kr **expon** ia, idur1, ib

ar **exprand** krange

ir **exprand** krange

kr **exprand** krange

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

ar **expsegr** ia, idur1, ib [, idur2] [, ic] [...], irel, iz

kr **expsegr** ia, idur1, ib [, idur2] [, ic] [...], irel, iz

ar **expseg** ia, idur1, ib [, idur2] [, ic] [...]

kr **expseg** ia, idur1, ib [, idur2] [, ic] [...]

**exp**(x) (no rate restriction)

ir **filelen** ifilcod

ir **filenchnls** ifilcod

ir **filepeak** ifilcod [, ichnl]

ir **filesr** ifilcod

ar **filter2** asig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN

kr **filter2** ksig, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN

**fini** ifilename, iskipframes, iformat, in1 [, in2] [, in3] [, ...]

**fink** ifilename, iskipframes, iformat, kin1 [, kin2] [, kin3] [, ...]

**fin** ifilename, iskipframes, iformat, ain1 [, ain2] [, ain3] [, ...]

ihandle **fiopen** ifilename, imode

ar **flanger** asig, adel, kfeedback [, imaxd]

**flashtxt** iwhich, String

ihandle **FLbox** "label", itype, ifont, isize, iwidth, iheight, ix, iy [, image]

kout, ihandle **FLbutBank** itype, inumx, inumy, iwidth, iheight, ix, iy, iopcode [

kout, ihandle **FLbutton** "label", ion, ioff, itype, iwidth, iheight, ix, iy, iopco

**FLcolor2** ired, igreen, iblue

**FLcolor** ired, igreen, iblue

kout, ihandle **FLcount** "label", imin, imax, istep1, istep2, itype, iwidth, iheight,

inumsnap **FLgetsnap** index

**FLgroupEnd**

**FLgroup** "label", iwidth, iheight, ix, iy [, iborder] [, image]

**FLhide** ihandle

koutx, kouty, ihandlex, ihandley **FLjoy** "label", iminx, imaxx, iminy, imaxy, iexp,

kout **FLkeyb** kparam1 [, kparam2] ... [, kparamN]

kout, ihandle **FLknob** "label", imin, imax, iexp, itype, idisp, iwidth, ix, iy [,

**FLlabel** isize, ifont, ialign, ired, igreen, iblue

**FLloadsnap** "filename"

**FLpackEnd**

**FLpack** iwidth, iheight, ix, iy, itype, ispace, iborder

**FLpanelEnd**

**FLpanel** "label", iwidth, iheight [, ix] [, iy] [, iborder]

**FLprintk2** kval, idisp

**FLprintk** itime, kval, idisp

kout, ihandle **FLroller** "label", imin, imax, istep, iexp, itype, idisp, iwidth, iheight,

**FLrun**

**FLsavesnap** "filename"

**FLscrollEnd**

**FLscroll** iwidth, iheight [, ix] [, iy]

**FLsetAlign** ialign, ihandle

**FLsetBox** itype, ihandle

**FLsetColor2** ired, igreen, iblue, ihandle

**FLsetColor** ired, igreen, iblue, ihandle

**FLsetFont** ifont, ihandle

**FLsetPosition** ix, iy, ihandle

**FLsetSize** iwidth, iheight, ihandle

inumsnap, inumval **FLsetsnap** index [, ifn]

**FLsetTextColor** isize, ihandle

**FLsetText** "itext", ihandle

**FLsetTextSize** isize, ihandle

**FLsetTextType** itype, ihandle

**FLsetVal\_i** kvalue, ihandle

**FLsetVal** ktrig, kvalue, ihandle

**FLshow** ihandle

**FLslidBnk** "names", inumsliders [, ioutable] [, iwidth] [, iheight] [, ix] [, iy

kout, ihandle **FLslider** "label", imin, imax, iexp, itype, idisp, iwidth, iheight

**FLtabsEnd**

**FLtabs** iwidth, iheight, ix, iy

kout, ihandle **FLtext** "label", imin, imax, istep, itype, iwidth, iheight, ix, iy

**FLupdate**

ihandle **FLvalue** "label", iwidth, iheight, ix, iy

ar **fmb3** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn

ar **fmbell** 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 **fmpercfl** 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 **fmvoice** kamp, kfreq, kvowel, ktilt, kvibamt, kvibrate, ifn1, ifn2, ifn3, ifn4, ivfn

ar **fmwurlie** kamp, kfreq, kc1, kc2, kvdepth, kvrate, ifn1, ifn2, ifn3, ifn4, ivfn

ar **fof2** xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, ifnc

ar **fof** xamp, xfund, xform, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, ifnc

ar **fog** xamp, xdens, xtrans, aspd, koct, kband, kris, kdur, kdec, iolaps, ifna, ifnb, ifnc

ar **fold** asig, kincr

ar **follow2** asig, katt, krel

ar **follow** asig, idt

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

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

**foutir** ihandle, iformat, iflag, iout1 [, iout2, iout3,...,ioutN]

**fouti** ihandle, iformat, iflag, iout1 [, iout2, iout3,...,ioutN]

**foutk** ifilename, iformat, kout1 [, kout2, kout3,...,koutN]

**fout** ifilename, iformat, aout1 [, aout2, aout3,...,aoutN]

**fprintks** "filename", "string", [, kval1] [, kval2] [...]

**fprints** "filename", "string" [, kval1] [, kval2] [...]

**frac**(x) (init-rate or control-rate args only)

**ftchnls**(x) (init-rate args only)

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

**ftlen**(x) (init-rate args only)

**ftloadk** "filename", ktrig, iflag, ifn1 [, ifn2] [...]

**ftload** "filename", iflag, ifn1 [, ifn2] [...]

**ftlptim**(x) (init-rate args only)

**ftmorf** kftndx, iftn, iresfn

**ftsavk** "filename", ktrig, iflag, ifn1 [, ifn2] [...]

**ftsav** "filename", iflag, ifn1 [, ifn2] [...]

**ftsr**(x) (init-rate args only)

ar **gain** asig, krms [, ihp] [, iskip]

ar **gauss** krange

ir **gauss** krange

kr **gauss** krange

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

ar **gogobel** kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivfn

**goto** label

ar **grain2** kcps, kfmd, kgdur, iovrlp, kfn, iwfn [, irpow] [, iseed] [, imode]

ar **grain3** kcps, kphs, kfmd, kpmd, kgdur, kdens, imaxovr, kfn, iwfn, kfrpow, kpr

ar **grain** xamp, xpitch, xdens, kampoff, kpitchoff, kgdur, igfn, iwfn, imgdur [,

ar **granule** xamp, ivoice, iratio, imode, ithd, ifn, ipshift, igskip, igskip\_os,

ar **guiro** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifreq1]

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

ar1, ar2 **hilbert** asig

aleft, aright **hrtfer** asig, kaz, kelev, "HRTFcompact"

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

**i**(x) (control-rate args only)

**if** ia R ib **igoto** label

**if** ka R kb **kgoto** label

**if** ia R ib **goto** label

**if** xa R xb **then**

**igoto** label

**ihold**

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, ar13, ar14, ar15

ar1 **inch** ksig1

ar1, ar2, ar3, ar4, ar5, ar6 **inh**

**initc14** ichan, ictrlno1, ictrlno2, ivalue

**initc21** ichan, ictrlno1, ictrlno2, ictrlno3, ivalue

**initc7** ichan, ictrlno, ivalue

ar **init** iarg

ir **init** iarg

kr **init** iarg

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8 **ino**

ar1, ar2, ar3, a4 **inq**

ar1 **in**

ar1, ar2 **ins**

**instr** i, j, ...



ar **integ** asig [, iskip]

kr **integ** ksig [, iskip]

ar **interp** ksig [, iskip] [, imodel]

**int**(x) (init-rate or control-rate args only)

kvalue **invalue** "channel name"

ar1, ar2, ar3, ar4, ar5, ar6, ar7, ar8, ar9, ar10, ar11, ar12, ar13, ar14, ar15

**inz** ksig1

kout **jitter2** ktotamp, kamp1, kcps1, kamp2, kcps2, kamp3, kcps3

kout **jitter** kamp, kcpsMin, kcpsMax

ar **jspline** xamp, kcpsMin, kcpsMax

kr **jspline** kamp, kcpsMin, kcpsMax

**kgoto** label

**kr** = iarg

**ksmps** = iarg

**ktableseg** ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]

kr **lfo** kamp, kcps [, itype]

ar **lfo** kamp, kcps [, itype]

ar **limit** asig, klow, khigh

ir **limit** isig, ilow, ihigh

kr **limit** ksig, klow, khigh

ar **linenr** xamp, irise, idec, iatdec

kr **linenr** kamp, irise, idec, iatdec

ar **linen** xamp, irise, idur, idec

kr **linen** kamp, irise, idur, idec

ar **line** ia, idur1, ib

kr **line** ia, idur1, ib

kr **lineto** ksig, ktime

ar **linrand** krange

ir **linrand** krange

kr **linrand** krange

ar **linsegr** ia, idur1, ib [, idur2] [, ic] [...], irel, iz

kr **linsegr** ia, idur1, ib [, idur2] [, ic] [...], irel, iz

ar **linseg** ia, idur1, ib [, idur2] [, ic] [...]

kr **linseg** ia, idur1, ib [, idur2] [, ic] [...]

a1, a2 **locsend**

a1, a2, a3, a4 **locsend**

a1, a2 **locsig** asig, kdegree, kdistance, kverbsend

a1, a2, a3, a4 **locsig** asig, kdegree, kdistance, kverbsend

**log10**(x) (no rate restriction)

**logbtwo**(x) (init-rate or control-rate args only)

**log**(x) (no rate restriction)

ksg **loopseg** kfreq, ktrig, ktime0, kvalue0 [, ktime1] [, kvalue1] [, ktime2] [,

ax, ay, az **lorenz** ksv, krv, kbv, kh, ix, iy, iz, iskip

ar [,ar2] **loscil3** xamp, kcps, ifn [, ibas] [, imod1] [, ibeg1] [, iend1] [, imod

ar [,ar2] **loscil** xamp, kcps, ifn [, ibas] [, imod1] [, ibeg1] [, iend1] [, imod

ar **lowpass2** asig, kcf, kq [, iskip]

ar **lowres** asig, kcutoff, kresonance [, iskip]

ar **lowresx** asig, kcutoff, kresonance [, inumlayer] [, iskip]

ar **lpf18** asig, kfco, kres, kdist

ar **lpfreson** asig, kfrqratio

ar **lphasor** xtrns [, ilps] [, ilpel] [, imodel] [, istrtr] [, istor]

**lpinterp** islot1, islot2, kmix

ar **lposcil3** kamp, kfregratio, kloop, kend, ifn [, iphs]

ar **lposcil** kamp, kfregratio, kloop, kend, ifn [, iphs]

krmsr, krms0, kerr, kcps **lpread** ktimepnt, ifilcod [, inpoles] [, ifrmrate]

ar **lpreson** asig

ksig **lpshold** kfreq, ktrig, ktime0, kvalue0 [, ktime1] [, kvalue1] [, ktime2] [

**lpslot** islot

ar **maca** asig1 [, asig2] [, asig3] [, asig4] [, asig5] [...]

ar **mac** asig1, ksig1 [, asig2] [, ksig2] [, asig3] [, ksig3] [...]

ar **madsr** iatt, idec, islev, irel [, idel] [, ireltim]

kr **madsr** iatt, idec, islev, irel [, idel] [, ireltim]

ar **mandol** kamp, kfreq, kpluck, kdetune, kgain, ksize, ifn [, iminfreq]

ar **marimba** kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec [, idoubles

**massign** ichnl, insnum

**massign** ichnl, "insname"

**maxalloc** insnum, icount

**mclock** ifreq

**mdelay** kstatus, kchan, kd1, kd2, kdelay

`idest midic14 ictrlno1, ictrlno2, imin, imax [, ifn]`

`kdest midic14 ictrlno1, ictrlno2, kmin, kmax [, ifn]`

`idest midic21 ictrlno1, ictrlno2, ictrlno3, imin, imax [, ifn]`

`kdest midic21 ictrlno1, ictrlno2, ictrlno3, kmin, kmax [, ifn]`

`idest midic7 ictrlno, imin, imax [, ifn]`

`kdest midic7 ictrlno, kmin, kmax [, ifn]`

`midichannelaftertouch xchannelaftertouch [, ilow] [, ihigh]`

`ichn midichn`

`midicontrolchange xcontroller, xcontrollervalue [, ilow] [, ihigh]`

`ival midictrl inum [, imin] [, imax]`

`kval midictrl inum [, imin] [, imax]`

`mididefault xdefault, xvalue`

`kstatus, kchan, kdata1, kdata2 midiin`

`midinoteoff xkey, xvelocity`

`midinoteoncps xcps, xvelocity`

`midinoteonkey xkey, xvelocity`

`midinoteonoct xoct, xvelocity`

`midinoteonpch xpch, xvelocity`

`midion2 kchn, knum, kvel, ktrig`

`midion kchn, knum, kvel`

`midiout kstatus, kchan, kdata1, kdata2`

`midipitchbend xpitchbend [, ilow] [, ihigh]`

`midipolyaftertouch xpolyaftertouch, xcontrollervalue [, ilow] [, ihigh]`

**midiprogramchange** xprogram

ar **mirror** asig, klow, khigh

ir **mirror** isig, ilow, ihigh

kr **mirror** ksig, klow, khigh

ar **moog** kamp, kfreq, kfiltq, kfiltrate, kvibf, kvamp, iafn, iwfn, ivfn

ar **moogvcf** asig, xfco, xres [, iscale]

**moscil** kchn, knum, kvel, kdur, kpause

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

**mrtmsg** msgtype

ar **multitap** asig [, itime1] [, igain1] [, itime2] [, igain2] [...]

**mute** insnum [, iswitch]

**mute** "insname" [, iswitch]

ar **mxadsr** iatt, idec, islev, irel [, idel] [, ireltim]

kr **mxadsr** iatt, idec, islev, irel [, idel] [, ireltim]

**nchnls** = iarg

ar **nestedap** asig, imode, imaxdel, idel1, igain1 [, idel2] [, igain2] [, idel3]

ar **nlfilt** ain, ka, kb, kd, kC, kL

ar **noise** xamp, kbeta

**noteoff** ichn, inum, ivel

**noteondur2** ichn, inum, ivel, idur

**noteondur** ichn, inum, ivel, idur

**noteon** ichn, inum, ivel

ival **notnum**

ar **nreverb** asig, ktime, khdif [, iskip] [, inumCombs] [, ifnCombs] [, inumAlpas]

**nrpn** kchan, kparmnum, kparmvalue

**nsamp**(x) (init-rate args only)

insno **nstrnum** "name"

ar **ntrpol** asig1, asig2, kpoint [, imin] [, imax]

ir **ntrpol** isig1, isig2, ipoint [, imin] [, imax]

kr **ntrpol** ksig1, ksig2, kpoint [, imin] [, imax]

**octave**(x)

**octcps** (cps) (init- or control-rate args only)

ioct **octmidib** [irange]

koct **octmidib** [irange]

ioct **octmidi**

**octpch** (pch) (init- or control-rate args only)

**opcode** name, outtypes, intypes

ar **oscbnk** kcps, kamd, kfmd, kpmd, iovrlap, iseed, kllminf, kllmaxf, kl2minf, k

kr **oscilli** idel, kamp, idur, ifn

kr **oscil1** idel, kamp, idur, ifn

ar **oscil3** xamp, xcps, ifn [, iphs]

kr **oscil3** kamp, kcps, ifn [, iphs]

ar **osciliktp** kcps, kfn, kphs [, istor]

ar **oscilikt** xamp, xcps, kfn [, iphs] [, istor]

kr **oscilikt** kamp, kcps, kfn [, iphs] [, istor]

ar **oscilikts** xamp, xcps, kfn, async, kphs [, istor]

ar **oscili** xamp, xcps, ifn [, iphs]

kr **oscili** kamp, kcps, ifn [, iphs]

ar **osciln** kamp, ifrq, ifn, itimes

ar **oscil** xamp, xcps, ifn [, iphs]

kr **oscil** kamp, kcps, ifn [, iphs]

ar **oscils** iamp, icps, iphs [, iflg]

ar **oscilx** kamp, ifrq, ifn, itimes

**out32** asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, asig10, asig11, a

**outch** ksig1, asig1 [, ksig2] [, asig2] [...]

**outc** asig1 [, asig2] [...]

**outh** asig1, asig2, asig3, asig4, asig5, asig6

**outiat** ichn, ivalue, imin, imax

**outic14** ichn, imsb, ilsb, ivalue, imin, imax

**outic** ichn, inum, ivalue, imin, imax

**outipat** ichn, inotenum, ivalue, imin, imax

**outipb** ichn, ivalue, imin, imax

**outipc** ichn, iprog, imin, imax

**outkat** kchn, kvalue, kmin, kmax

**outkc14** kchn, kmsb, klsb, kvalue, kmin, kmax

**outkc** kchn, knum, kvalue, kmin, kmax

**outkpat** kchn, knotenum, kvalue, kmin, kmax

**outkpb** kchn, kvalue, kmin, kmax

**outkpc** kchn, kprog, kmin, kmax

**outo** asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8

**outq1** asig

**outq2** asig

**outq3** asig

**outq4** asig

**outq** asig1, asig2, asig3, asig4

**outs1** asig

**outs2** asig

**out** asig

**outs** asig1, asig2

**outvalue** "channel name", kvalue

**outx** asig1, asig2, asig3, asig4, asig5, asig6, asig7, asig8, asig9, asig10, asig11

**outz** ksig1

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

ar **pareq** asig, kc, kv, kq [, imode]

ar **pcauchy** kalpha

ir **pcauchy** kalpha

kr **pcauchy** kalpha

ibend **pchbend** [imin] [, imax]

kbend **pchbend** [imin] [, imax]

ipch **pchmidib** [irange]

kpch **pchmidib** [irange]

ipch **pchmidi**



**pchoct** (oct) (init- or control-rate args only)

kr **peak** asig

kr **peak** ksig

**pgmassign** ipgm, inst

**pgmassign** ipgm, "insname"

ar **phaser1** asig, kfreq, kord, kfeedback [, iskip]

ar **phaser2** asig, kfreq, kq, kord, kmode, ksep, kfeedback

ar **phasorbnk** xcps, kndx, icnt [, iphs]

kr **phasorbnk** kcps, kndx, icnt [, iphs]

ar **phasor** xcps [, iphs]

kr **phasor** kcps [, iphs]

ar **pinkish** xin [, imethod] [, inumbands] [, iseed] [, iskip]

kcps, krms **pitchamdf** asig, imincps, imaxcps [, icps] [, imedi] [, idowns] [, ie

kcoct, kamp **pitch** asig, iupcte, ilo, ihi, idbthresh [, ifrqs] [, iconf] [, istr

ax, ay, az **planet** kmass1, kmass2, ksep, ix, iy, iz, ivx, ivy, ivz, idelta [, if

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

ar **poisson** klambda

ir **poisson** klambda

kr **poisson** klambda

ir **polyaft** inote [, ilow] [, ihigh]

kr **polyaft** inote [, ilow] [, ihigh]

kr **portk** ksig, khtim [, isig]

kr **port** ksig, ihtim [, isig]

ar **poscil3** kamp, kcps, ifn [, iphs]

kr **poscil3** kamp, kcps, ifn [, iphs]

ar **poscil** aamp, acps, ifn [, iphs]

ar **poscil** aamp, kcps, ifn [, iphs]

ar **poscil** kamp, acps, ifn [, iphs]

ar **poscil** kamp, kcps, ifn [, iphs]

ir **poscil** kamp, kcps, ifn [, iphs]

kr **poscil** kamp, kcps, ifn [, iphs]

**powoftwo**(x) (init-rate or control-rate args only)

ar **pow** aarg, kpow [, inorm]

ir **pow** iarg, ipow [, inorm]

kr **pow** karg, kpow [, inorm]

**prealloc** insnum, icount

**prealloc** "insname", icount

**printk2** kvar [, inumspaces]

**printk** itime, kval [, ispace]

**printks** "string", itime [, kval1] [, kval2] [...]

**print** iarg [, iarg1] [, iarg2] [...]

**prints** "string" [, kval1] [, kval2] [...]

ar **product** asig1, asig2 [, asig3] [...]

**pset** icon1 [, icon2] [...]

**p**(x)

ar **pvadd** ktmpnt, kfmmod, ifilcod, ifn, ibins [, ibinoffset] [, ibinincr] [, iex

```

pvbufread ktmpnt, ifile

ar pvcross ktmpnt, kfmmod, ifile, kampscale1, kampscale2 [, ispecwp]

ar pvinterp ktmpnt, kfmmod, ifile, kfregscale1, kfregscale2, kampscale1, kampscale2

ar pvoc ktmpnt, kfmmod, ifilcod [, ispecwp] [, iextractmode] [, ifreqlim] [, igain]

kfreg, kamp pvread ktmpnt, ifile, ibin

ar pvsadsyn fsrc, inoscs, kfmmod [, ibinoffset] [, ibinincr] [, iinit]

fsig pvsanal ain, ifftsize, ioverlap, iwinsize, iwintype [, iformat] [, iinit]

fsig pvsacross fsrc, fdest, kamp1, kamp2

fsig pvsfread ktmpnt, ifn [, ichan]

pvsftr fsrc, ifna [, ifnf]

kflag pvsftw fsrc, ifna [, ifnf]

ioverlap, inumbins, iwinsize, iformat pvsinfo fsrc

fsig pvsmaska fsrc, ifn, kdepth

ar pvsynth fsrc, [iinit]

ar randh xamp, xcps [, iseed] [, isize] [, ioffset]

kr randh kamp, kcps [, iseed] [, isize] [, ioffset]

ar randi xamp, xcps [, iseed] [, isize] [, ioffset]

kr randi kamp, kcps [, iseed] [, isize] [, ioffset]

ar randomh kmin, kmax, acps

kr randomh kmin, kmax, kcps

ar randomi kmin, kmax, acps

kr randomi kmin, kmax, kcps

ar random kmin, kmax

```

ir **random** imin, imax

kr **random** kmin, kmax

ar **rand** xamp [, iseed] [, isel] [, ibase]

kr **rand** xamp [, iseed] [, isel] [, ibase]

ir **readclock** inum

kr1, kr2 **readk2** ifilename, iformat, ipol [, interp]

kr1, kr2, kr3 **readk3** ifilename, iformat, ipol [, interp]

kr1, kr2, kr3, kr4 **readk4** ifilename, iformat, ipol [, interp]

kr **readk** ifilename, iformat, ipol [, interp]

**reinit** label

kflag **release**

ar **repluck** iplk, kamp, icps, kpick, krefl, axcite

kr **resonk** ksig, kcf, kbw [, iscl] [, iskip]

ar **resonr** asig, kcf, kbw [, iscl] [, iskip]

ar **reson** asig, kcf, kbw [, iscl] [, iskip]

ar **resonx** asig, kcf, kbw [, inumlayer] [, iscl] [, iskip]

ar **resony** asig, kbf, kbw, inum, ksep [, isepmode] [, iscl] [, iskip]

ar **resonz** asig, kcf, kbw [, iscl] [, iskip]

ar **reverb2** asig, ktime, khdif [, iskip] [, inumCombs] [, ifnCombs] [, inumAlpas]

ar **reverb** asig, krvt [, iskip]

ar **rezzy** asig, xfco, xres [, imode]

**rigoto** label

**rireturn**

```

kr rms asig [, ihp] [, iskip]

ax rnd31 kscl, krpow [, iseed]

ix rnd31 iscl, irpow [, iseed]

kx rnd31 kscl, krpow [, iseed]

rnd(x) (init- or control-rate only)

ar rspline xrangeMin, xrangeMax, kcpsMin, kcpsMax

kr rspline krangeMin, krangeMax, kcpsMin, kcpsMax

ir rtclock

kr rtclock

i1,...,i16 s16b14 ichan, ictrlno_msbl, ictrlno_lsbl, imin1, imax1, initvalue1, ifn1

k1,...,k16 s16b14 ichan, ictrlno_msbl, ictrlno_lsbl, imin1, imax1, initvalue1, ifn1

i1,...,i32 s32b14 ichan, ictrlno_msbl, ictrlno_lsbl, imin1, imax1, initvalue1, ifn1

k1,...,k32 s32b14 ichan, ictrlno_msbl, ictrlno_lsbl, imin1, imax1, initvalue1, ifn1

ar samphold asig, agate [, ival] [, ivstor]

kr samphold ksig, kgate [, ival] [, ivstor]

ar sandpaper iamp, idettack [, inum] [, idamp] [, imaxshake]

scanhammer isrc, idst, ipos, imult

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

aout scantable kamp, kpch, ipos, imass, istiff, idamp, ivel

scanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, kstif, kfreq

schedkwhennamed ktrigger, kmintim, kmaxnum, "name", kwhen, kdur [, ip4] [, ip5]

schedkwhen ktrigger, kmintim, kmaxnum, kinsnum, kwhen, kdur [, ip4] [, ip5] [...]

schedkwhen ktrigger, kmintim, kmaxnum, "insname", kwhen, kdur [, ip4] [, ip5] [...]

```

**schedule** insnum, iwhen, idur [, ip4] [, ip5] [...]

**schedule** "insname", iwhen, idur [, ip4] [, ip5] [...]

**schedwhen** ktrigger, kinsnum, kwhen, kdur [, ip4] [, ip5] [...]

**schedwhen** ktrigger, "insname", kwhen, kdur [, ip4] [, ip5] [...]

**seed** ival

ar **sekere** iamp, idettack [, inum] [, idamp] [, imaxshake]

**semitone**(x)

kr **sensekey**

kr **sense**

ktrig\_out **seqtime** ktime\_unit, kstart, kloop, kinitndx, kfn\_times

**setctrl** inum, ival, itype

**setksmps** iksmps

**sfilist** ifilhandle

ar **sfinstr3m** ivel, inotenum, xamp, xfreq, instrnum, ifilhandle [, iflag] [, ioff

ar1, ar2 **sfinstr3** ivel, inotenum, xamp, xfreq, instrnum, ifilhandle [, iflag] [,

ar **sfinstrm** ivel, inotenum, xamp, xfreq, instrnum, ifilhandle [, iflag] [, ioff

ar1, ar2 **sfinstr** ivel, inotenum, xamp, xfreq, instrnum, ifilhandle [, iflag] [,

ir **sfload** "filename"

**sfpassign** istartindex, ifilhandle

ar **sfplay3m** ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset]

ar1, ar2 **sfplay3** ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset]

ar **sfplaym** ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset]

ar1, ar2 **sfplay** ivel, inotenum, xamp, xfreq, ipreindex [, iflag] [, ioffset]

**sfplist** ifilhandle

ir **sfpreset** iprog, ibank, ifilhandle, ipreindex

ar **shaker** kamp, kfreq, kbeans, kdamp, ktimes [, idecay]

**sinh**(x) (no rate restriction)

**sininv**(x) (no rate restriction)

**sin**(x) (no rate restriction)

ar **sleighbells** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifr

k1,...,k16 **slider16f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1,...,

i1,...,i16 **slider16** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum16,

k1,...,k16 **slider16** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum16,

k1,...,k32 **slider32f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1,...,

i1,...,i32 **slider32** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum32,

k1,...,k32 **slider32** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum32,

k1,...,k64 **slider64f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1,...,

i1,...,i64 **slider64** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum64,

k1,...,k64 **slider64** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum64,

k1,...,k8 **slider8f** ichan, ictlnum1, imin1, imax1, init1, ifn1, icutoff1,..., ic

i1,...,i8 **slider8** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum8, imi

k1,...,k8 **slider8** ichan, ictlnum1, imin1, imax1, init1, ifn1,..., ictlnum8, imi

ar [, ac] **sndwarp** xamp, xtimewarp, xresample, ifn1, ibeg, iwsiz, irandw, iover

ar1, ar2 [,ac1] [, ac2] **sndwarpst** xamp, xtimewarp, xresample, ifn1, ibeg, iwsiz

ar1 **soundin** ifilcod [, iskptim] [, iformat]

ar1, ar2 **soundin** ifilcod [, iskptim] [, iformat]

ar1, ar2, ar3 **soundin** ifilcod [, iskptim] [, iformat]

ar1, ar2, ar3, ar4 **soundin** ifilcod [, iskptim] [, iformat]

**soundout** asig1, ifilcod [, iformat]

a1, a2, a3, a4 **space** asig, ifn, ktime, kreverbsend, kx, ky

aW, aX, aY, aZ **spat3di** ain, iX, iY, iZ, idist, ift, imode [, istor]

aW, aX, aY, aZ **spat3d** ain, kX, kY, kZ, idist, ift, imode, imdel, iovr [, istor]

**spat3dt** ioutft, iX, iY, iZ, idist, ift, imode, irlen [, iftnocl]

k1 **spdist** ifn, ktime, kx, ky

wsig **specaddm** wsig1, wsig2 [, imul2]

wsig **specdiff** wsigin

**specdisp** wsig, iprd [, iwtflg]

wsig **specfilt** wsigin, ifhtim

wsig **spechist** wsigin

koct, kamp **specptrk** wsig, kvar, ilo, ihi, istr, idbthresh, inptls, irolloff [,

wsig **specscal** wsigin, ifscale, ifthresh

ksum **specsum** wsig [, interp]

wsig **spectrum** xsig, iprd, iocts, ifrqa [, iq] [, ihann] [, idbout] [, idsprd] [

a1, a2, a3, a4 **spsend**

**sqrt**(x) (no rate restriction)

**sr** = iarg

ar **stix** iamp, idettack [, inum] [, idamp] [, imaxshake]

ar **streson** asig, kfr, ifdbgain

**strset** iarg, istring



**subinstrinit** instrnum [, p4] [, p5] [...]

**subinstrinit** "insname" [, p4] [, p5] [...]

a1, [...] [, a8] **subinstr** instrnum [, p4] [, p5] [...]

a1, [...] [, a8] **subinstr** "insname" [, p4] [, p5] [...]

ar **sum** asig1 [, asig2] [, asig3] [...]

alow, ahigh, aband **svfilter** asig, kcf, kq [, iscl]

ar **table3** andx, ifn [, ixmode] [, ixoff] [, iwrap]

ir **table3** indx, ifn [, ixmode] [, ixoff] [, iwrap]

kr **table3** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

**tablecopy** kdft, ksft

**tablegpw** kfn

**tableicopy** idft, isft

**tableigpw** ifn

ar **tableikt** xndx, kfn [, ixmode] [, ixoff] [, iwrap]

kr **tableikt** kndx, kfn [, ixmode] [, ixoff] [, iwrap]

**tableimix** idft, idoff, ilen, islft, isloff, islg, is2ft, is2off, is2g

ar **tablei** andx, ifn [, ixmode] [, ixoff] [, iwrap]

ir **tablei** indx, ifn [, ixmode] [, ixoff] [, iwrap]

kr **tablei** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

**tableiw** isig, indx, ifn [, ixmode] [, ixoff] [, iwgmde]

ar **tablekt** xndx, kfn [, ixmode] [, ixoff] [, iwrap]

kr **tablekt** kndx, kfn [, ixmode] [, ixoff] [, iwrap]

**tablemix** kdft, kdoff, klen, ks1ft, ksloff, kslg, ks2ft, ks2off, ks2g

ir **tableng** ifn

kr **tableng** kfn

ar **tablera** kfn, kstart, koff

**tableseg** ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]

ar **table** andx, ifn [, ixmode] [, ixoff] [, iwrap]

ir **table** indx, ifn [, ixmode] [, ixoff] [, iwrap]

kr **table** kndx, ifn [, ixmode] [, ixoff] [, iwrap]

kstart **tablewa** kfn, asig, koff

**tablewkt** asig, andx, kfn [, ixmode] [, ixoff] [, iwgmde]

**tablewkt** ksig, kndx, kfn [, ixmode] [, ixoff] [, iwgmde]

**tablew** asig, andx, ifn [, ixmode] [, ixoff] [, iwgmde]

**tablew** isig, indx, ifn [, ixmode] [, ixoff] [, iwgmde]

**tablew** ksig, kndx, ifn [, ixmode] [, ixoff] [, iwgmde]

ar **tablexkt** xndx, kfn, kwrap, iwsiz [, ixmode] [, ixoff] [, iwrap]

**tablexseg** ifn1, idur1, ifn2 [, idur2] [, ifn3] [...]

ar **tambourine** kamp, idettack [, inum] [, idamp] [, imaxshake] [, ifreq] [, ifre

**tanh**(x) (no rate restriction)

ar **taninv2** ay, ax

ir **taninv2** iy, ix

kr **taninv2** ky, kx

**taninv**(x) (no rate restriction)

**tan**(x) (no rate restriction)

ar **tbvcf** asig, xfco, xres, kdist, kasym

ktemp **tempest** kin, iprd, imindur, imemdur, ihp, ithresh, ihtim, ixfdbak, istart

**tempo** ktempo, istartempo

kr **tempoval**

**tigoto** label

kr **timeinstk**

kr **timeinsts**

kr **timeinsts**

ir **timek**

kr **timek**

ir **times**

kr **times**

**timeout** istrtr, idur, label

ir **tival**

kr **tlineto** ksig, ktime, ktrig

kr **tonek** ksig, khp [, iskip]

ar **tone** asig, khp [, iskip]

ar **tonex** asig, khp [, inumlayer] [, iskip]

ar **transeg** ia, idur, itype, ib [, idur2] [, itype] [, ic] ...

kr **transeg** ia, idur, itype, ib [, idur2] [, itype] [, ic] ...

kout **trigger** ksig, kthreshold, kmode

**trigseq** ktrig\_in, kstart, kloop, kinitndx, kfn\_values, kout1 [, kout2] [...]

ar **trirand** krangle

ir **trirand** krangle

kr **trirand** krange

**turnoff**

**turnon** insnum [, itime]

ar **unirand** krange

ir **unirand** krange

kr **unirand** krange

ar **upsamp** ksig

aout = **urd**(ktableNum)

iout = **urd**(itableNum)

kout = **urd**(ktableNum)

ar **valpass** asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]

ar1, ..., ar16 **vbap16move** asig, ispread, ifldnum, ifld1 [, ifld2] [...]

ar1, ..., ar16 **vbap16** asig, iazim [, ielev] [, ispread]

ar1, ar2, ar3, ar4 **vbap4move** asig, ispread, ifldnum, ifld1 [, ifld2] [...]

ar1, ar2, ar3, ar4 **vbap4** asig, iazim [, ielev] [, ispread]

ar1, ..., ar8 **vbap8move** asig, ispread, ifldnum, ifld1 [, ifld2] [...]

ar1, ..., ar8 **vbap8** asig, iazim [, ielev] [, ispread]

**vbaplsinit** idim, ilsnum [, idir1] [, idir2] [...] [, idir32]

**vbapzmove** inumchnls, istartndx, asig, idur, ispread, ifldnum, ifld1, ifld2, [..

**vbapz** inumchnls, istartndx, asig, iazim [, ielev] [, ispread]

kfn **vco2ft** kcps, iwave [, inyx]

ifn **vco2ift** icps, iwave [, inyx]

ifn **vco2init** iwave [, ibasfn] [, ipmul] [, iminsiz] [, imaxsiz] [, isrcft]

```

ar vco2 kamp, kcps [, imodel [, kpw] [, kphs] [, inyx]

ar vcomb asig, krvt, xlpt, imaxlpt [, iskip] [, insmps]

ar vco xamp, xcps, iwave, kpw [, ifn] [, imaxd] [, ileak] [, inyx] [, iphs]

ar vdelay3 asig, adel, imaxdel [, iskip]

ar vdelay asig, adel, imaxdel [, iskip]

aout1, aout2, aout3, aout4 vdelayxq ain1, ain2, ain3, ain4, adl, imd, iws [, is

aout vdelayx ain, adl, imd, iws [, ist]

aout1, aout2 vdelayxs ain1, ain2, adl, imd, iws [, ist]

aout1, aout2, aout3, aout4 vdelayxwq ain1, ain2, ain3, ain4, adl, imd, iws [, i

aout vdelayxw ain, adl, imd, iws [, ist]

aout1, aout2 vdelayxws ain1, ain2, adl, imd, iws [, ist]

ival veloc [ilow] [, ihigh]

ar vibes kamp, kfreq, ihrd, ipos, imp, kvibf, kvamp, ivibfn, idec

kout vibrato kAverageAmp, kAverageFreq, kRandAmountAmp, kRandAmountFreq, kAmpMi

kout vibr kAverageAmp, kAverageFreq, ifn

vincr asig, aincr

ar vlowres asig, kfco, kres, iord, ksep

ar voice kamp, kfreq, kphoneme, kform, kvibf, kvamp, ifn, ivfn

ar vpvoc ktmpnt, kfmod, ifile [, ispecwp] [, ifn]

ar waveset ain, krep [, ilen]

ar weibull ksigma, ktau

ir weibull ksigma, ktau

kr weibull ksigma, ktau

```

```

ar wgbowedbar kamp, kfreq, kpos, kbowpres, kgain [, iconst] [, itvel] [, ibowpo

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

ar wgbrass kamp, kfreq, ktens, iatt, kvibf, kvamp, ifn [, iminfreq]

ar wgclar kamp, kfreq, kstiff, iatt, idetk, kngain, kvibf, kvamp, ifn [, iminfr

ar wgflute kamp, kfreq, kjet, iatt, idetk, kngain, kvibf, kvamp, ifn [, iminfre

ar wgpluck2 iplk, kamp, icps, kpick, krefl

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

ar wguide1 asig, xfreq, kcutoff, kfeedback

ar wguide2 asig, xfreq1, xfreq2, kcutoff1, kcutoff2, kfeedback1, kfeedback2

ar wrap asig, klow, khigh

ir wrap isig, ilow, ihigh

kr wrap ksig, klow, khigh

aout wterrain kamp, kpch, k_xcenter, k_ycenter, k_xradius, k_yradius, itabx, it

ar xadsr iatt, idec, islev, irel [, idel]

kr xadsr iatt, idec, islev, irel [, idel]

xinarg1 [, xinarg2] ... [xinargN] xin

xout xoutarg1 [, xoutarg2] ... [, xoutargN]

kpos, kvel xscanmap iscan, kamp, kvamp [, iwhich]

xscansmap kpos, kvel, iscan, kamp, kvamp [, iwhich]

ar xscans kamp, kfreq, ifntraj, id [, iorder]

xscanu init, irate, ifnvel, ifnmass, ifnstif, ifncentr, ifndamp, kmass, kstif,

xtratim iextradur

kx, ky xyin iprd, ixmin, ixmax, iymmin, iymax [, ixinit] [, iyinit]

```

**zac1** kfirst, klast

**zakinit** isizea, isizek

ar **zamod** asig, kzamod

ar **zarg** kndx, kgain

ar **zar** kndx

**zawm** asig, kndx [, imix]

**zaw** asig, kndx

ar **zfilter2** asig, kdamp, kfreq, iM, iN, ib0, ib1, ..., ibM, ia1, ia2, ..., iaN

ir **zir** indx

**ziwm** isig, indx [, imix]

**ziw** isig, indx

**zkcl** kfirst, klast

kr **zkmod** ksig, kzkmod

kr **zkr** kndx

**zkwm** ksig, kndx [, imix]

**zkw** ksig, kndx