



# **REAKTOR 5**

**Module and Core Reference**

The information in this document is subject to change without notice and does not represent a commitment on the part of NATIVE INSTRUMENTS GmbH. The software described by this document is subject to a License Agreement and may not be copied to other media. No part of this publication may be copied, reproduced or otherwise transmitted or recorded, for any purpose, without prior written permission by NATIVE INSTRUMENTS GmbH, hereinafter referred to as NATIVE INSTRUMENTS. All product and company names are trademarks of their respective owners.

Furthermore, the fact that you are reading this text means you are the owner of legal version rather than an illegal, pirated copy. It is only through the loyalty and honesty of people like yourself that NATIVE INSTRUMENTS GmbH can continue to develop and create innovative audio software. On behalf of the entire company, thank you very much.

Users Guide written by NATIVE INSTRUMENTS

Special thanks to the Beta Test Team, who were invaluable not just in tracking down bugs, but in making this a better product.

© NATIVE INSTRUMENTS GmbH, 2007. All rights reserved.



**NATIVE INSTRUMENTS**

**Germany**

NATIVE INSTRUMENTS GmbH  
Schlesische Str. 28  
D-10997 Berlin  
Germany  
info@native-instruments.de  
www.native-instruments.de

**USA**

NATIVE INSTRUMENTS North America, Inc.  
5631 Hollywood Boulevard  
Los Angeles, CA 90028  
USA  
sales@native-instruments.com  
www.native-instruments.com

# Table Of Contents

<b>Module Reference .....</b>	<b>19</b>
<b>Panel.....</b>	<b>21</b>
Fader.....	21
Knob .....	23
Button .....	24
List.....	25
Switch .....	26
Lamp.....	27
Level .....	28
RGB Lamp .....	29
Meter.....	29
LevelMeter .....	30
Picture.....	30
Multi Picture.....	31
Text .....	32
Multi Text.....	32
XY .....	32
Scope .....	33
Multi Display and Poly Display .....	35
Mouse Area .....	37
Stacked Macro .....	39
IC SendTerminal .....	39
IC Receive Terminal.....	40
<b>MIDI In.....</b>	<b>41</b>
Note Pitch .....	41
Pitchbend.....	41
Gate .....	42
Single Trig. Gate.....	42
Sel. Note Gate .....	42
On Velocity.....	43
Off Velocity .....	43
Controller .....	43
Ch. Aftertouch .....	44
Poly Aftertouch .....	44
Sel. Poly AT.....	44
Program Change.....	45
Start/Stop .....	45
1/96 Clock .....	45

Sync Clock .....	46
Song Pos.....	46
Channel Message .....	47
<b>MIDI Out.....</b>	<b>48</b>
Note Pitch/Gate .....	48
Pitchbend.....	48
Controller .....	48
Ch. Aftertouch .....	49
Poly Aftertouch .....	49
Sel. Poly AT.....	49
Program Change.....	50
Start/Stop .....	50
1/96 Clock .....	50
Song Pos.....	51
Channel Message .....	51
<b>Math .....</b>	<b>53</b>
Constant .....	53
Add .....	53
Subtract.....	54
Invert, -X.....	54
Multiply.....	54
$a * b + c$ .....	55
Reciprocal $1/x$ .....	55
Divide $x/y$ .....	55
Modulo $x \% y$ .....	56
Rectifier .....	56
Rect./Sign .....	56
Compare .....	57
Compare/Equal .....	57
Quantize .....	58
Expon. (A).....	58
Expon. (F).....	58
Log (A) .....	59
Log (F).....	59
Power $x y$ .....	59
Square Root .....	60
$1 / \text{Square Root}$ .....	60
Sine.....	60
Sine/Cos .....	61
Arcsin .....	61
Arccos .....	61

Arctan.....	62
<b>Signal Path .....</b>	<b>63</b>
Selector/Scanner .....	63
Relay 1,2 .....	63
Crossfade.....	64
Distributor/Panner .....	64
Stereo Pan .....	65
Amp/Mixer.....	65
Stereo Amp/Mixer.....	66
<b>Oscillator .....</b>	<b>67</b>
Sawtooth.....	67
Saw FM .....	67
Saw Sync .....	68
Saw Pulse .....	69
Bi-Saw.....	69
Triangle.....	70
Tri FM.....	70
Tri Sync .....	71
Tri/Par Symm.....	71
Parabol .....	72
Par FM .....	72
Par Sync .....	73
Par PWM.....	74
Sine.....	74
Sine FM .....	75
Sine Sync.....	75
Multi-Sine .....	76
Pulse.....	77
Pulse FM.....	78
Pulse Sync .....	78
Pulse 1-ramp.....	79
Pulse 2-ramp.....	80
Bi-Pulse.....	81
Impulse.....	81
Impulse FM .....	82
Impulse Sync.....	82
Multi-Step .....	83
4-Step .....	83
5-Step .....	84
6-Step .....	84
8-Step .....	84

Multi-Ramp .....	84
4-Ramp .....	84
5-Ramp .....	85
6-Ramp .....	85
8-Ramp .....	85
Ramp.....	85
Clock .....	86
Noise.....	86
Random .....	87
Geiger.....	87
<b>Samplers .....</b>	<b>88</b>
Sampler .....	89
Sampler FM.....	90
Sampler Loop .....	91
Grain Resynth .....	93
Grain Pitch Former .....	97
Grain Cloud .....	101
Beat Loop .....	103
Sample Lookup .....	105
<b>Sequencer.....</b>	<b>106</b>
Sequencer.....	106
6-Step .....	106
8-Step .....	107
12-Step .....	107
16-Step .....	107
Multiplex 16 .....	107
<b>LFO, Envelope .....</b>	<b>109</b>
LFO .....	109
Slow Random .....	110
H - Env .....	110
HR - Env.....	111
D - Env .....	112
DR - Env .....	112
DSR - Env .....	113
DBDR - Env.....	113
DBDSR-Env.....	114
AD - Env .....	115
AR - Env .....	115
ADR-Env .....	116
ADSR - Env .....	116
ADBDR - Env.....	117

ADBDJR-Env .....	118
AHDSR - Env .....	119
AHDBDR - Env.....	120
4-Ramp .....	121
5-Ramp .....	122
6-Ramp .....	124
<b>Filter .....</b>	<b>126</b>
HP/LP 1-Pole.....	126
HP/LP 1-Pole FM .....	127
Allpass 1-Pole.....	127
Multi 2-Pole .....	128
Multi 2-Pole FM .....	128
Multi/Notch 2-Pole .....	129
Multi/Notch 2-Pole FM.....	130
Multi/LP 4-Pole.....	131
Multi/LP 4-Pole FM .....	132
Multi/HP 4-Pole .....	133
Multi/HP 4-Pole FM .....	134
Pro-52 Filter.....	135
Ladder Filter.....	135
Ladder Filter FM .....	136
Peak EQ.....	137
Peak EQ FM .....	137
High Shelf EQ.....	138
High Shelf EQ FM .....	138
Low Shelf EQ.....	139
Low Shelf EQ FM .....	140
Differentiator .....	140
Integrator .....	141
<b>Delay .....</b>	<b>142</b>
Single Delay .....	142
Multi-Tap Delay .....	143
Diffuser Delay.....	144
Grain Delay.....	145
Grain Cloud Delay .....	146
Unit Delay .....	148
<b>Audio Modifier .....</b>	<b>149</b>
Saturator.....	149
Saturator 2.....	149
Clipper.....	150
Mod. Clipper.....	151

Mirror 1 Level .....	151
Mirror 2 Levels.....	151
Chopper .....	152
Shaper 1 BP.....	153
Shaper 2 BP.....	153
Shaper 3 BP.....	154
Shaper Parabolic.....	155
Shaper Cubic.....	155
Slew Limiter .....	156
Peak Detector .....	156
Sample & Hold .....	157
Frequency Divider.....	157
Audio Table .....	158
<b>Event Processing.....</b>	<b>160</b>
Accumulator .....	160
Counter.....	160
Randomizer .....	161
Frequency Divider.....	161
Ctrl. Shaper 1 BP.....	162
Ctrl. Shaper 2 BP.....	162
Ctrl. Shaper 3 BP.....	163
Logic AND.....	163
Logic OR.....	164
Logic EXOR.....	164
Logic NOT.....	164
Order .....	165
Iteration .....	165
Separator .....	166
Value .....	166
Merge .....	167
Step Filter .....	167
Router M->1 .....	167
Router 1,2.....	168
Router 1->M .....	168
Timer.....	169
Hold .....	169
Event Table.....	170
<b>Auxiliary.....</b>	<b>172</b>
Tapedeck 1-Ch.....	172
Tapedeck 2-Ch.....	175
Audio Voice Combiner .....	175



Event V.C. All .....	176
Event V.C. Max.....	176
Event V.C. Min .....	177
A to E .....	177
A to E (Trig).....	177
A to E (Perm).....	178
A to Gate.....	178
To Voice .....	179
From Voice .....	179
Voice Shift .....	180
Audio Smoother .....	181
Event Smoother .....	181
Master Tune/Level .....	182
Tempo Info.....	182
Voice Info.....	182
Tuning Info .....	183
System Info .....	183
Note Range Info.....	184
MIDI Channel Info .....	184
Snapshot.....	185
Set Random .....	187
Unison Spread .....	187
Snap Value .....	188
Snap Value Array.....	188
In Port .....	190
Terminal.....	190
Out Port.....	190
Terminal.....	190
Send.....	190
Terminal.....	190
Receive .....	190
Terminal.....	190
IC Send .....	192
Terminal.....	192
IC Receive.....	192
Terminal.....	192
OSC Send .....	193
Terminal.....	193
OSC Receive.....	193
Terminal.....	193

<b>Appendix .....</b>	<b>194</b>
<b>First steps in Reaktor Core .....</b>	<b>195</b>
What is Reaktor Core .....	195
Using core cells .....	196
Using core cells in a real example .....	199
Basic editing of core cells .....	201
<b>Getting into Reaktor Core.....</b>	<b>206</b>
Event and audio core cells .....	206
Creating your first core cell .....	208
Audio and control signals .....	220
Building your first Reaktor Core macros .....	226
Using audio as control signal .....	233
Event signals .....	234
Logic signals.....	238
<b>Reaktor Core fundamentals: the core signal model.....</b>	<b>240</b>
Values.....	240
Events.....	240
Simultaneous events.....	243
Processing order .....	245
Event core cells reviewed .....	246
<b>Structures with internal state .....</b>	<b>252</b>
Clock signals .....	252
Object Bus Connections .....	253
Initialization.....	256
Building an event accumulator .....	259
Event merging.....	260
Event accumulator with reset and initialization.....	262
Fixing the event shaper .....	268
<b>Audio processing at its core .....</b>	<b>271</b>
Audio signals .....	271
Sampling rate clock bus.....	273
Connection feedback .....	274
Feedback around macros.....	277
Denormal values .....	281
Other bad numbers .....	285
Building a 1-pole low pass filter .....	285
Event routing .....	289
Building a signal clipper .....	291
Building a simple sawtooth oscillator .....	293
<b>More signal types.....</b>	<b>294</b>
Float signals .....	294

Integer signals .....	296
Building an event counter .....	299
Building a rising edge counter macro .....	300
<b>Arrays .....</b>	<b>303</b>
Introduction to arrays .....	303
Building an audio signal selector .....	306
Building a delay .....	313
Tables .....	319
<b>Building optimal structures .....</b>	<b>324</b>
Latches and modulation macros .....	324
Routing and merging .....	325
Numerical operations .....	326
Conversions between floats and integers .....	327
<b>Appendix A. Reaktor Core user interface .....</b>	<b>328</b>
A.1. Core cells .....	328
A.2. Core modules/macros .....	328
A.3. Core ports .....	329
A.4. Core structure editing .....	329
<b>Appendix B. Reaktor Core concept .....</b>	<b>330</b>
B.1. Signals and events .....	330
B.2. Initialization .....	330
B.3. OBC connections .....	330
B.4. Routing .....	331
B.5. Latching .....	331
B.6. Clocking .....	331
<b>Appendix C. Core macro ports .....</b>	<b>332</b>
C.1. In .....	332
C.2. Out .....	332
C.3. Latch (input) .....	332
C.4. Latch (output) .....	332
C.5. Bool C (input) .....	332
C.6. Bool C (output) .....	333
<b>Appendix D. Core cell ports .....</b>	<b>333</b>
D.1. In (audio mode) .....	333
D.2. Out (audio mode) .....	333
D.3. In (event mode) .....	333
D.4. Out (event mode) .....	333
<b>Appendix E. Built-in busses .....</b>	<b>334</b>
E.1. SR.C .....	334
E.2. SR.R .....	334

<b>Appendix F. Built-in modules.....</b>	<b>334</b>
F.1. Const.....	334
F.2. Math > + .....	334
F.4. Math > * .....	335
F.5. Math > / .....	335
F.6. Math >  x  .....	335
F.7. Math > -x.....	335
F.8. Math > DN Cancel .....	336
F.9. Math > ~log .....	336
F.10. Math > ~exp.....	336
F.11. Bit > Bit AND .....	336
F.12. Bit > Bit OR .....	337
F.13. Bit > Bit XOR.....	337
F.14. Bit > Bit NOT .....	337
F.15. Bit > Bit <<.....	337
F.16. Bit > Bit >> .....	338
F.17. Flow > Router.....	338
F.18. Flow > Compare .....	338
F.19. Flow > Compare Sign.....	338
F.20. Flow > ES Ctl .....	339
F.21. Flow > ~BoolCtl .....	339
F.22. Flow > Merge .....	339
F.23. Flow > EvtMerge.....	340
F.24. Memory > Read .....	340
F.25. Memory > Write .....	340
F.26. Memory > R/W Order .....	340
F.27. Memory > Array .....	341
F.28. Memory > Size [ ] .....	341
F.29. Memory > Index.....	341
F.30. Memory > Table.....	342
F.31. Macro .....	342
<b>Appendix G. Expert macros.....</b>	<b>343</b>
G.1. Clipping > Clip Max / IClip Max .....	343
G.2. Clipping > Clip Min / IClip Min .....	343
G.3. Clipping > Clip MinMax / IClipMinMax .....	343
G.4. Math > 1 div x .....	343
G.5. Math > 1 wrap.....	343
G.6. Math > lmod .....	344
G.7. Math > Max / IMax .....	344
G.8. Math > Min / IMin .....	344
G.9. Math > round.....	344

G.10. Math > sign +- .....	344
G.11. Math > sqrt (>0).....	344
G.12. Math > sqrt .....	345
G.13. Math > x(>0)^y .....	345
G.14. Math > x^2 / x^3 / x^4 .....	345
G.15. Math > Chain Add / Chain Mult .....	345
G.16. Math > Trig-Hyp > 2 pi wrap.....	345
G.17. Math > Trig-Hyp > arcsin / arccos / arctan .....	345
G.18. Math > Trig-Hyp > sin / cos / tan.....	346
G.19. Math > Trig-Hyp > sin -pi..pi / cos -pi..pi / tan -pi..pi.....	346
G.20. Math > Trig-Hyp > tan -pi4..pi4 .....	346
G.21. Math > Trig-Hyp > sinh / cosh / tanh.....	346
G.22. Memory > Latch / lLatch.....	346
G.23. Memory > z^-1 / z^-1 ndc.....	346
G.24. Memory > Read [] .....	347
G.25. Memory > Write [] .....	347
G.26. Modulation > x + a / Integer > lx + a .....	347
G.27. Modulation > x * a / Integer > lx * a .....	347
G.28. Modulation > x - a / Integer > lx - a .....	348
G.29. Modulation > a - x / Integer > la - x .....	348
G.30. Modulation > x / a .....	348
G.31. Modulation > a / x .....	348
G.32. Modulation > xa + y.....	348
<b>Appendix H. Standard macros .....</b>	<b>349</b>
H.1. Audio Mix-Amp > Amount .....	349
H.2. Audio Mix-Amp > Amp Mod .....	349
H.3. Audio Mix-Amp > Audio Mix .....	349
H.4. Audio Mix-Amp > Audio Relay .....	349
H.5. Audio Mix-Amp > Chain (amount).....	350
H.6. Audio Mix-Amp > Chain (dB) .....	350
H.7. Audio Mix-Amp > Gain (dB) .....	350
H.8. Audio Mix-Amp > Invert.....	351
H.9. Audio Mix-Amp > Mixer 2 ... 4 .....	351
H.10. Audio Mix-Amp > Pan .....	351
H.11. Audio Mix-Amp > Ring-Amp Mod .....	351
H.12. Audio Mix-Amp > Stereo Amp.....	352
H.13. Audio Mix-Amp > Stereo Mixer 2 ... 4 .....	352
H.14. Audio Mix-Amp > VCA.....	352
H.15. Audio Mix-Amp > XFade (lin) .....	353
H.16. Audio Mix-Amp > XFade (par) .....	353
H.17. Audio Shaper > 1+2+3 Shaper .....	353

H.18. Audio Shaper > 3-1-2 Shaper .....	353
H.19. Audio Shaper > Broken Par Sat.....	354
H.20. Audio Shaper > Hyperbol Sat .....	354
H.21. Audio Shaper > Parabol Sat.....	354
H.22. Audio Shaper > Sine Shaper 4 / 8.....	355
H.23. Control > Ctl Amount.....	355
H.24. Control > Ctl Amp Mod.....	355
H.25. Control > Ctl Bi2Uni.....	355
H.26. Control > Ctl Chain.....	356
H.27. Control > Ctl Invert.....	356
H.28. Control > Ctl Mix .....	356
H.29. Control > Ctl Mixer 2.....	356
H.30. Control > Ctl Pan .....	357
H.31. Control > Ctl Relay .....	357
H.32. Control > Ctl XFade .....	357
H.33. Control > Par Ctl Shaper.....	357
H.34. Convert > dB2AF .....	358
H.35. Convert > dP2FF .....	358
H.36. Convert > logT2sec .....	358
H.37. Convert > ms2Hz .....	358
H.38. Convert > ms2sec.....	359
H.39. Convert > P2F .....	359
H.40. Convert > sec2Hz .....	359
H.41. Delay > 2 / 4 Tap Delay 4p.....	359
H.42. Delay > Delay 1p / 2p / 4p .....	359
H.43. Delay > Diff Delay 1p / 2p / 4p.....	360
H.44. Envelope > ADSR .....	360
H.45. Envelope > Env Follower .....	361
H.46. Envelope > Peak Detector .....	361
H.47. EQ > 6dB LP/HP EQ.....	361
H.48. EQ > 6dB LowShelf EQ .....	361
H.49. EQ > 6dB HighShelf EQ.....	362
H.50. EQ > Peak EQ .....	362
H.51. EQ > Static Filter > 1-pole static HP.....	362
H.52. EQ > Static Filter > 1-pole static HS .....	362
H.53. EQ > Static Filter > 1-pole static LP.....	363
H.54. EQ > Static Filter > 1-pole static LS.....	363
H.55. EQ > Static Filter > 2-pole static AP .....	363
H.56. EQ > Static Filter > 2-pole static BP .....	363
H.57. EQ > Static Filter > 2-pole static BP1.....	363
H.58. EQ > Static Filter > 2-pole static HP.....	364

H.59. EQ > Static Filter > 2-pole static HS .....	364
H.60. EQ > Static Filter > 2-pole static LP .....	364
H.61. EQ > Static Filter > 2-pole static LS .....	364
H.62. EQ > Static Filter > 2-pole static N .....	365
H.63. EQ > Static Filter > 2-pole static Pk .....	365
H.64. EQ > Static Filter > Integrator .....	365
H.65. Event Processing > Accumulator .....	365
H.66. Event Processing > Clk Div .....	366
H.67. Event Processing > Clk Gen .....	366
H.68. Event Processing > Clk Rate .....	366
H.69. Event Processing > Counter .....	366
H.70. Event Processing > Ctl2Gate .....	367
H.71. Event Processing > Dup Flt / IDup Flt .....	367
H.72. Event Processing > Impulse .....	367
H.73. Event Processing > Random .....	367
H.74. Event Processing > Separator / ISeparator .....	367
H.75. Event Processing > Thld Crossing .....	368
H.76. Event Processing > Value / IValue .....	368
H.77. LFO > MultiWave LFO .....	368
H.78. LFO > Par LFO .....	368
H.79. LFO > Random LFO .....	369
H.80. LFO > Rect LFO .....	369
H.81. LFO > Saw(down) LFO .....	369
H.82. LFO > Saw(up) LFO .....	369
H.83. LFO > Sine LFO .....	370
H.84. LFO > Tri LFO .....	370
H.85. Logic > AND .....	370
H.86. Logic > Flip Flop .....	370
H.87. Logic > Gate2L .....	370
H.88. Logic > GT / IGT .....	371
H.89. Logic > EQ .....	371
H.90. Logic > GE .....	371
H.91. Logic > L2Clock .....	371
H.92. Logic > L2Gate .....	371
H.93. Logic > NOT .....	372
H.94. Logic > OR .....	372
H.95. Logic > XOR .....	372
H.96. Logic > Schmitt Trigger .....	372
H.97. Oscillators > 4-Wave Mst .....	372
H.98. Oscillators > 4-Wave Slv .....	373
H.99. Oscillators > Binary Noise .....	373

H.100. Oscillators > Digital Noise .....	373
H.101. Oscillators > FM Op .....	374
H.102. Oscillators > Formant Osc .....	374
H.103. Oscillators > MultiWave Osc.....	374
H.104. Oscillators > Par Osc .....	374
H.105. Oscillators > Quad Osc.....	375
H.106. Oscillators > Sin Osc .....	375
H.107. Oscillators > Sub Osc 4 .....	375
H.108. VCF > 2 Pole SV .....	375
H.109. VCF > 2 Pole SV C.....	376
H.110. VCF > 2 Pole SV (x3) S .....	376
H.111. VCF > 2 Pole SV T (S) .....	376
H.112. VCF > Diode Ladder.....	377
H.113. VCF > D/T Ladder.....	377
H.114. VCF > Ladder x3 .....	377
<b>Appendix I. Core cell library .....</b>	<b>378</b>
I.1. Audio Shaper > 3-1-2 Shaper.....	378
I.2. Audio Shaper > Broken Par Sat.....	378
I.3. Audio Shaper > Hyperbol Sat.....	378
I.4. Audio Shaper > Parabol Sat.....	379
I.5. Audio Shaper > Sine Shaper 4/8.....	379
I.6. Control > ADSR .....	379
I.7. Control > Env Follower.....	380
I.8. Control > Flip Flop .....	380
I.9. Control > MultiWave LFO.....	380
I.10. Control > Par Ctl Shaper.....	381
I.11. Control > Schmitt Trigger.....	381
I.12. Control > Sine LFO .....	381
I.13. Delay > 2/4 Tap Delay 4p .....	382
I.14. Delay > Delay 4p .....	382
I.15. Delay > Diff Delay 4p.....	382
I.16. EQ > 6dB LP/HP EQ .....	382
I.17. EQ > HighShelf EQ.....	383
I.18. EQ > LowShelf EQ .....	383
I.19. EQ > Peak EQ .....	383
I.20. EQ > Static Filter > 1-pole static HP.....	383
I.21. EQ > Static Filter > 1-pole static HS.....	384
I.22. EQ > Static Filter > 1-pole static LP .....	384
I.23. EQ > Static Filter > 1-pole static LS .....	384
I.24. EQ > Static Filter > 2-pole static AP.....	384
I.25. EQ > Static Filter > 2-pole static BP.....	385



I.26. EQ > Static Filter > 2-pole static BP1 .....	385
I.27. EQ > Static Filter > 2-pole static HP .....	385
I.28. EQ > Static Filter > 2-pole static HS .....	385
I.29. EQ > Static Filter > 2-pole static LP .....	386
I.30. EQ > Static Filter > 2-pole static LS .....	386
I.31. EQ > Static Filter > 2-pole static N .....	386
I.32. EQ > Static Filter > 2-pole static Pk .....	386
I.33. Oscillator > 4-Wave Mst .....	387
I.34. Oscillator > 4-Wave Slv .....	387
I.35. Oscillator > Digital Noise .....	388
I.36. Oscillator > FM Op .....	388
I.37. Oscillator > Formant Osc .....	388
I.38. Oscillator > Impulse .....	388
I.39. Oscillator > MultiWave Osc .....	389
I.40. Oscillator > Quad Osc .....	389
I.41. Oscillator > Sub Osc .....	389
I.42. VCF > 2 Pole SV C .....	390
I.43. VCF > 2 Pole SV T .....	390
I.44. VCF > 2 Pole SV x3 S .....	391
I.45. VCF > Diode Ladder .....	391
I.46. VCF > D/T Ladder .....	392
I.47. VCF > Ladder x3 .....	392
<b>Index .....</b>	<b>393</b>



# Module Reference

Modules are the most elementary components of a REAKTOR ensemble. This section is primarily intended for REAKTOR users who wish to create their own ensembles from scratch. If you do not wish or feel experienced enough to create your own ensembles from scratch, REAKTOR offers alternative ways of accessing its potential:

- If you are not interested at all in building ensembles on your own, work on the ensemble level.
- If you want to assemble your favourite instruments in one ensemble and use them together, work on the instrument level.
- If you want to build your own Instruments by making use of prebuilt building blocks, work on the macro level.
- If you want to have control over each single parameter of your ensemble, work on module level.

This reference lists all modules available in REAKTOR and provides a basic description for each module. The description includes properties settings if available as well as a description of all input and output ports.

All REAKTOR objects (Modules, Instruments, Macros, and Ensembles) have a Label field in their Properties. This label appears on the object's icon in the structure. By default, this label for modules describes the module's function. Rename modules with care, especially if you want other REAKTOR users to be able to understand your creation's structure.

## Hybrid modules

A lot of modules in REAKTOR are **hybrid** modules. After inserting such a module it appears as an event processing module per default indicated by red port labels. A green dot on a port is indicating that you can connect audio as well as event cables with this port. As soon as you connect an audio cable to one of its inputs the module is converted to an audio processing module indicated by black dots and labels on the ports. If you connect an event cable the port gets a red dot. A hybrid module has the following behaviour:

- If you mix audio and event cables at the inputs or if you solely connect audio cables, the module always works with audio rate.
- If you solely connect event cables to the inputs, it works with event rate.

- If the module's output is connected to an event input of another module, it becomes automatically an event processing module, and it is not possible anymore to add audio cables to the module's inputs.
- If the module's output is connected to an audio input, the module can work either as audio or event module, depending on the cables connected to the module's inputs.
- If the module already works with audio rate due to the input connections, you can not connect the output to an event input of another module.

## Dynamic ports management

Where it makes sense, the modules offer a **dynamic** in- and/or out-port management. You can add inputs to the module by dropping additional cables above the module in the area of an existing port while holding down the **Ctrl**-key. You can see where the new port will be added while holding the mouse pointer above the target module. The vertical position of the mouse pointer determines where the port is added so that it is possible to add a port between existing ports.

# Panel

Panel modules provide onscreen controls for various REAKTOR processes. They can be independently set to appear on the A and B Control Panels and they can be arranged differently on those panels. Some panel modules are for display purposes only. Those include lamps, meters, and scopes for displaying REAKTOR processes as well as graphics and text modules for ornamentation. The remaining modules generate or route data for REAKTOR processing. Those include faders, knobs, buttons, switches, menus, and an XY controller (which can be used for both display and control).

---

## Fader



## Panel

With the slider control in the panel you adjust the value which is output (as event and audio) by the corresponding module in the structure. The signal is mono – when connected to a poly input all voices receive the same value.

## Properties - Function page

**Range:** The output value corresponds to the position of the knob which refers to a range between the limits **Min** and **Max** set in the properties dialog window. With the parameter **Stepsize**, the display and output values can be quantized to coarser steps, e.g. to display fewer decimal digits. Alternatively it is possible to use the **Num Steps** field to set the step resolution of the fader. The values of both fields, **Stepsize** and **Num Steps**, are dependent on each other. Changing the value in one of the fields causes an automatic value change in the other. While you enter the value range per step in the **Stepsize** field, the **Num Steps** field represents the total number of steps across the whole value range of the fader.

The highest possible resolution for a fader is 127.000 steps which is available if you enter 0 in the **Stepsize** field or 127.000 in the **Num Steps** field.

---

**Note:** You are able to enter a step resolution in the **Num Steps** field which is higher than the standardized MIDI resolution of 128. Be aware that REAKTOR can use a higher resolution internally, but it can only exchange MIDI data with external hard- or software within the MIDI range of 128. Under certain circumstances this might effect the functionality or the sound of an ensemble when used in different production environments. Normally this is not a problem since the MIDI resolution is high enough for controlling parameters. Nevertheless in certain situations (using a filter with high resonance while moving the cutoff frequency for example) you might wish to have a higher resolution which then is possible inside REAKTOR as well.

---

**Mouse Res** specifies the distance in pixels the mouse pointer has to cover to get from the lowest value to the highest. If you enter a value in the **Mouse Res** field which is double as high as the one in the **Pixel in Y** field inside the **Appearance** tab, you will have to draw the mouse twice the way of the fader size to change from the lowest to the highest value.

If **Num Steps** is bigger than **Mouse Res**, not every step is reached by moving the mouse. On the other side, if **Mouse Res** is bigger than **Num Steps** the number of pixels per step can be set to a higher value than 1.

The **Default** value is used whenever an initialization of the control happens. The value will be used in the following situations:

- Pressing the **Default** button in the snapshot window will reset all controls in the ensemble/instrument.
- If “default” appears in one of the morph targets in the snapshot window, you can morph between one snapshot and the default values of all controls..
- If you add a control to an Instrument which already contains a snapshot list, the default value is used for the new control until you overwrite the snapshot with a new value for that control or you activate **Snap Isolate**.

If the **Snap Isolate** switch is activated, the fader will not respond to snapshot recall.

Activating **Random Isolate** avoids that the fader reacts on executing the snapshot randomization function.

**ID for Snapshot Files:** The ID number is used for the snapshot management of REAKTOR. The numbers are entered automatically whenever you insert a panel controller (fader, knob, button, switch....). If you change this ID (if

you want to import snapshots from another instrument with similar controls (for instance) already existing snapshots do not recognize the panel element anymore and ignores it. Only modify this number if you know exactly what you are doing.

## Properties - Info page

An info text to explain the fader's function can be entered under **Info** in the properties. The text will appear as a hint (Tooltip) while the mouse pointer rests on the fader in the panel, if hints are switched on.

## Properties - Appearance page

The label you enter here will only be shown above the fader in the panel if **Label** is activated in the **Visible** section in the properties. Likewise, the currently set value will only be shown as a number below the fader if **Value** is activated. It is also possible to hide the fader bitmap itself, so that you only see the value box. In this case you can still use the fader in the panel by clicking and dragging up and down onto the value box.

Faders can appear **vertical** or **horizontal** in the panel. Tick the appropriate radio button in the **Form** section to achieve the desired look.

The length of the fader can be set freely in the **Pixel in Y** field in the **Size** section. You can also define the width of the fader with the three radio buttons **Big**, **Medium** and **Small**.

## Properties - Connection page

The Connection page of the Properties for Panel Controls is described in the section *Connection Properties of Panel Controls* on page 133.

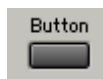
---

### Knob



### Panel

Just like the fader but with a different appearance in the panel.



With the push button control in the panel you select the value which is output (as event and audio) by the corresponding module in the structure. The output values in the on and off state are set with **On Value** and **Off Value** in the button's properties. The signal is mono – when connected to a poly input all voices receive the same value.

## Properties - Function page

**Range:** The values which are sent by the Button when it is pressed and when it is released can be defined in the fields **On Value** and **Off Value**.

**Mode:** There is a choice of three operating modes: **Trigger**, **Gate** and **Toggle**. In trigger mode, events are only generated when the button is pressed, nothing happens when it is released. In gate mode, an event with value zero is sent when the button is released. In toggle mode, the button changes state every time it is pressed.

**Default = On:** Sets the default value which is used at several actions in REAKTOR to On.

Activating **Snap Isolate** avoids that the button reacts on snapshot recalls.

Activating **Random Isolate** avoids that the button reacts on executing the snapshot randomization function.

**ID for Snapshot Files:** See fader description.

## Properties - Info page

An info text to explain the button's function can be entered under **Info** in the properties. The text will appear as a hint (Tooltip) while the mouse pointer rests on the button in the panel, if hints are switched on.

## Properties - Appearance page

The label will only be shown above the button in the panel if **Label** is activated in the properties. Likewise, the currently set value will only be shown as a number below the button if **Label** is activated.

The button can be displayed in three sizes: **Small**, **Medium** and **Big**.

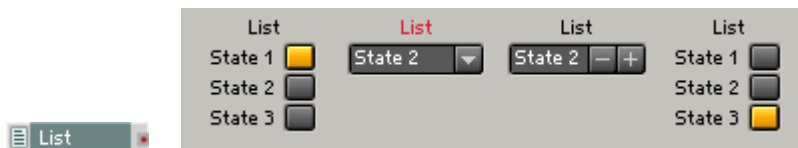


## Properties - Connection page

The Connection page of the Properties for Panel Controls is described in the section *Connection Properties of Panel Controls* on page 133.

## List

## Panel



The List module is for constructing onscreen lists, drop-down menus, button selectors, and scrolling text displays.

## Properties - Function page

The **Function** Properties contains a list into which you can **Append**, **Insert**, and **Delete** individual items. You can also specify the number of items directly using a numerical (**NUM Entries**). The items in the list will be automatically displayed on the control panel in the selected display format. For each item, you can type in a value to be sent to the module output when that item is selected.

The Function page also contains a value generator. With this little tool you can generate values for multiple entries in the entries list. Example: you want to have an increment of 4 instead of 1 for each next entry. Therefore enter “4” as **Stepsize** in the Value Generator and press the **Apply** button. Now the values in the **Value** column of the entry list are modified according to the settings in the Value Generator.

The **Mouse Resolution** only applies to the panel control if you choose the **Spin** style on the **Appearance** page, where you can click on the control entry and drag up or down to change the entry.

## Properties - Appearance page

The panel representation of this module changes depending on the chosen style on the properties' **Appearance** page. The following styles are available:

- **Button:** Each module in-port creates a button. All buttons will be arrayed vertically in the instrument panel. The currently activated button will be displayed in the Indicator color of the Instrument.

- **Menu:** Each module in-port creates a new entry in a drop down list.
- **Text Panel:** Each module in-port creates a new entry in a list which displays multiple entries at the same time. If you have created more entries than fit into the text panel display specified by the **Size X** and **Size Y** fields on the Appearance tab, you will get scrollbars in the panel.
- **Spin:** Each module in-port creates a new entry in a list. You can switch through the list using a + and a - button right hand of the list entry panel display.

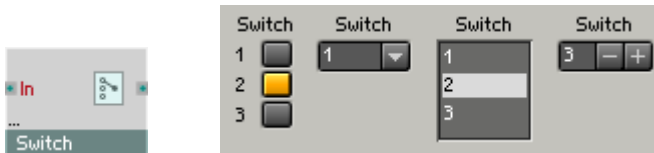
The **Size X** and **Size Y** fields are controlling the display size of the control element in the panel.

## Ports

- **Out:** Event output for the value associated with the selected item.

## Switch

## Panel



Switches are used for changing the signal flow. They do not contain any signal processing of their own but establish a switchable connection between other modules. The output is connected to the selected input by pushing the corresponding button or by selecting a list entry. All the other unselected inputs are disconnected.

Modules whose outputs are disconnected, through the action of a switch or otherwise, are automatically deactivated to reduce unnecessary load to the CPU. A module's status lamp remains dark while the module is deactivated.

This module is a hybrid module , so it can process event signals as well as audio signals, depending on its connections, and it contains a dynamic management of its in-ports.

## Properties - Function page

**Enable Switch Off:** If this option is enabled, the switch can have a state where no input is selected. If your display is set to button style on the **Appearance** page of the Properties, it is possible to switch off all buttons by clicking a

second time on the currently selected button. Using any other of the display styles you get an additional entry called “Off”.

**Min Num Port Groups:** You can set the minimum number of module ports here regardless of the number of cables you have connected to the module.

The **Mouse Resolution** only applies to the panel control if you choose the **Spin** style on the **Appearance** page, where you can click on the control entry and drag up or down to change the entry.

## Properties - Appearance page

The label will only be shown above the button in the panel if **Label** is activated on the **Appearance** page of the properties.

The panel control can be displayed in three sizes: **Small**, **Medium** and **Big**.

If you use a switch with two in-ports, only one button (the upper button) is shown when **1 Toggle Button** is selected: When the button is on, input 1 is connected, when the button is off, input 2 is connected.

The panel representation of this module changes depending on the chosen style on the properties' **Appearance** page. The following styles are available:

- **Button:** Each module in-port creates a button. All buttons will be arrayed vertically in the instrument panel. The currently activated button will be displayed in the Indicator color of the Instrument.
- **Menu:** Each module in-port creates a new entry in a drop-down list.
- **Text Panel:** Each module in-port creates a new entry in a list which displays multiple entries at the same time. If you have created more entries than fit into the text panel display specified by the **Size X** and **Size Y** fields on the Appearance tab, you will get scrollbars in the panel.
- **Spin:** Each module in-port creates a new entry in a list. You can switch through the list using a **+** and a **-** button on the right side of the list entry panel display.

---

## Lamp

## Panel



Indicator lamp for a monophonic signal.

The lamp in the panel lights up as long as the input signal (sampled at 25 Hz) is within the range set with **Min** and **Max** on the properties' function tab,

i.e. the lamp is on when the signal's value is larger than **Min** and less than or equal to **Max**.

If the **Continuous** mode is checked in the properties, the lamp color will fade in and out between the **Min** and **Max** values.

The color of the lamp (red, green, blue, yellow or indicator) can be chosen in the properties. If you check **Indicator Color**, the lamp will use the indicator color chosen in the instrument properties.

It is also possible to define custom colors for the On and Off position of the lamp using the buttons **Set On Color** and **Set Off Color**.

If you uncheck the **Has Frame** option, you are able to place lamps pixel accurate side by side in the panel without any gaps between them.

The label will only be shown above the lamp in the panel if **Label** is activated in the properties.

---

## Level Lamp

## Panel



Indicator lamp for a monophonic signal with logarithmic settings.

The lamp in the panel lights as long as the input level is within the range set with **Min** and **Max** (in dB) in the properties, i.e. the lamp is on when the signal's value is larger than **Min** and less than or equal to **Max**.

If the **Continuous** mode is checked in the properties, the lamp color will fade in and out between the **Min** and **Max** values.

The color of the lamp (red, green, blue, yellow or indicator) can be chosen in the properties. If you check **Indicator Color**, the lamp will use the indicator color chosen in the instrument properties.

It is also possible to define custom colors for the On and Off position of the lamp using the buttons **Set On Color** and **Set Off Color**.

If you uncheck the **Has Frame** option, you are able to place lamps pixel accurate side by side in the panel without any gaps between them.

The label will only be shown above the lamp in the panel if **Label** is activated in the properties.



The RGB Lamp module is a resizable, colored display. Its color is controlled by the values appearing at its three color inputs. Its horizontal and vertical size can be set in pixels in the **Appearance** Properties.

- **R**: Audio input for the intensity of the red component. Range 0 to 1.
- **G**: Audio input for the intensity of the green component. Range 0 to 1.
- **B**: Audio input for the intensity of the blue component. Range 0 to 1.



Value indicator for a monophonic signal.

The value of arriving signal is sampled (at 25 Hz) and displayed on a linear scale. The displayed range is set with **Min** and **Max** in the properties.

The color of the meter (red, green, blue, yellow or indicator) can be chosen in the properties. If you check **Indicator Color**, the lamp will use the indicator color chosen in the instrument properties.

It is also possible to define custom colors for the upper and lower part of the meter using the buttons **Set On Color** and **Set Off Color**.

**Number Of Segments** defines, of how many singular elements the meter is composed.

Under **Size X (Segment)** and **Size Y (Segment)** you can define the size of one meter element. The overall height of the meter will be the result of **Size Y (Segment)** multiplied by **Number of Segments**.

The label will only be shown above the meter in the panel if **Label** is activated in the properties.

---

## LevelMeter

## Panel



Level indicator for a monophonic audio signal.

The amplitude of the connected audio signal is displayed on a logarithmic scale. The displayed range is set in dB with **Min** and **Max** in the properties.

The color of the meter (red, green, blue, yellow or indicator) can be chosen in the properties. If you check **Indicator Color**, the lamp will use the indicator color chosen in the instrument properties.

It is also possible to define custom colors for the upper and lower part of the meter using the buttons **Set On Color** and **Set Off Color**.

**Number Of Segments** defines, of how many singular elements the meter is composed.

Under **Size X (Segment)** and **Size Y (Segment)** you can define the size of one meter element. The overall height of the meter will be the result of **Size Y (Segment)** multiplied by **Number of Segments**.

The label will only be shown above the level meter in the panel if **Label** is activated in the properties.

---

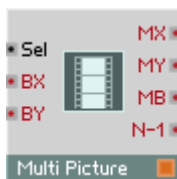
## Picture

## Panel



Allows loading a panel decoration bitmap from a picture file in TGA format (File extension \*.tga). The bitmap module has no in- or outputs. The display size will be set to the size of the image. It is possible to modify the visible frame in pixels under Appearance/Size in the properties. The image cannot be resized inside REAKTOR, for this you need to use external picture editing software.

Select the option **Save bitmap with ensemble** in the Properties to have the image data stored as part of REAKTOR's file.



The Multi Picture module is a 2-dimensional controller (like the XY module) that reports the mouse position and mouse button status at its outputs. It supports multi-frame animation when pictures are set up that way in the Picture Properties window by indicating the number of frames (animations) and their orientation (horizontal or vertical).

24-bit BMP and 32-bit Targa (uncompressed) formatted graphics may be used in REAKTOR in several places: as Instrument and Macro control panel backgrounds, as Instrument and macro structure icons, and in the Picture and Multi Picture control panel modules. The advantage of Targa is that it supports an alpha channel, which can be used as a mask for the visible portion of the graphic-the unmasked portion will then be transparent. That's useful for round knobs on a square background, for example.

You can load pictures using the drop-down Select-Picture menu in the Properties of any object that can display pictures. Opening a picture automatically brings up the Picture Properties window where you make all relevant picture settings. All pictures are automatically shared and available to all appropriate modules.

- **Sel**: Audio input for selecting the frame by number. Range 0 to number of the last frame.
- **MX**: Event output for the mouse horizontal (X) position when the mouse is within the picture.
- **MY**: Event output for the mouse vertical (Y) position when the mouse is within the picture.
- **MB**: Event output for mouse button status (0 when up, 1 when down).
- **N - 1**: Event output for the number of animations you have entered in the Picture Properties -1.


 Text

The Text module does not process any signals, its purpose is only to add text to the structure. For example it can be used for noting the author and creation date of an instrument and to explain how it works.

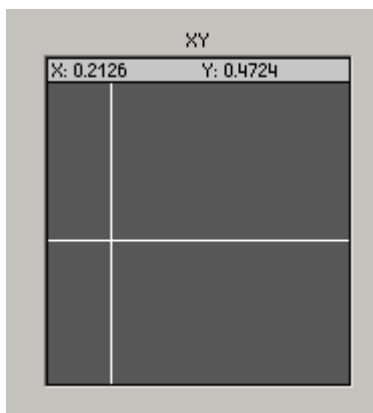
## Multi Text


 Multi Text

The Multi Text module provides a changeable text display for control panels. Any number of text items can be added in the module properties, where items can also be edited or deleted.

**In:** Audio input for the number of the text item to be displayed.

## XY



The XY control field has two functions: It displays audio input signals and acts as a 2-dimensional controller used with the mouse.

## Properties - Function page

The **Always Active** option enables the ability of the module to activate a signal branch connected to one of the in-ports of the module.

In **Incremental Mouse Mode** the control reacts similar to a knob. In this mode you can click anywhere within the panel display of the control and move the mouse without resetting the value to the position of the mouse pointer directly. Instead, the value will follow the mouse pointer with a fixed offset.



## Properties - Appearance page

In the module Properties you can set the display type for the audio signals to be visualized. The inputs **X1** and **Y1** control the position of the visual object (**Pixel** or **Cross**). When the object is set to **Bar** or **Rectangle** then **X1** and **Y1** control one corner and **X2** and **Y2** the opposite corner of the object.

To display fast moving audio data select **Scope** mode. All other modes evaluate the input only at the lower graphics update rate.

You can also set the size of the crosshair appearing at the mouse position.

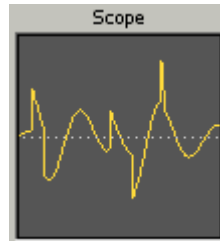
Everything drawn in the display fades out more or less slowly, depending on the value set for **Fade Time** in the Properties (maximum value is 99).

- **X1**: Audio input for the X1 coordinate of the visual object.
- **Y1**: Audio input for the Y1 coordinate of the visual object.
- **X2**: Audio input for the X2 coordinate of the visual object.
- **Y2**: Audio input for the Y2 coordinate of the visual object.
- **MX**: Event output for the X position of the mouse cursor when the mouse button is pressed on the display.
- **MY**: Event output for the Y position of the mouse cursor when the mouse button is pressed on the display.
- **MB**: Left mouse button state (1=pressed, 0=released)

---

## Scope

## Panel



Oscilloscope for displaying a time-varying signal.

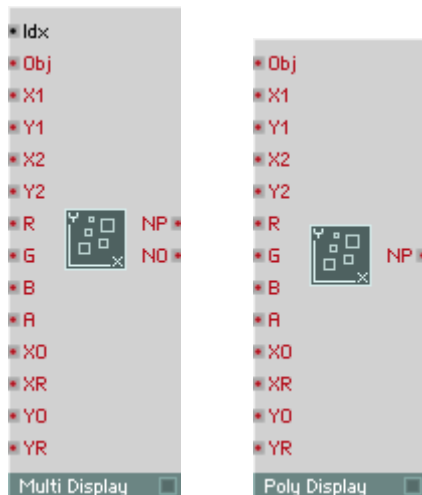
Every time an Event is received at the Trigger input (**Trg**), the module starts recording the audio signal at the input (**In**) and displays it on the panel. When no trigger events are received, the display continues showing the stored signal and it can be shown at varying scales and positions by adjusting the relevant control inputs.

The size of the graph in the panel can be set in the module's properties **Appearance** page with **Pixel in X** and **Pixel in Y**.

On the **Function** page you can set the buffer used by the scope in ms.

You would typically connect an **A to E Trig** module to the **Trg** input for triggering the Scope from an audio signal.

- **Trg**: Mono event input for the trigger signal that synchronises the trace.
- **TP**: Mono event input for controlling the offset in time (Time Position) of the trace in milliseconds. The trace starts at the left edge of the display **TP** ms after the trigger event.
- **TS**: Mono event input for controlling the time scale of the displayed trace. From left to right edge, the display shows **TS** ms of the signal.
- **YP**: Mono event input for controlling the amplitude offset (Y Position) of the trace. **YP** = -1 corresponds to the bottom edge of the display, +1 is the top edge.
- **YS**: Mono event input for controlling the amplitude scaling (Y Scale) of the trace. The difference between the signal values at the top and at the bottom of the display is 2 **YS**. When **YP** = 0, the display shows values between +**YS** and -**YS**.
- **In**: Mono audio input for the signal to be displayed.



Both the Multi Display and Poly Display modules enable the display and manipulation of multiple graphical objects (crosses, bars, pictures, animations, etc.). A range of parameters (including type, position, size and color) can be defined individually for each graphical object.

The major difference between two modules is that for the Multi Display module, the number of graphical objects is specified by the Number of Objects field in the properties, whereas for the Poly Display module, the number of graphical objects is specified by the number of voices in the instrument.

The advantage of Multi Display is that it can display any number of graphical objects, regardless of the number of polyphonic voices. Each object can then be addressed independently using the Idx input. In contrast, the Poly Display could be considered easier to program, as it doesn't require any index-processing code, and all objects can be addressed simultaneously by virtue of parallel polyphonic processing. However, the number of objects is limited to the number of voices of the instrument.

The properties include a variety of options for customisation of the display, including background color and picture.

When used in conjunction with the Mouse Area and Snap Value Array modules, the Multi Display and Poly Display allow sophisticated custom interface elements to be constructed (sequencers for example).

All Multi Display inputs accept monophonic event signals. Idx also accepts monophonic audio signals.

- **Idx:** Index of the graphical element to address. The index is 1-based; i.e. the ID of the first element is 1 (not 0). Fractional values are rounded to the nearest integer. Where Idx values are outside the range of 1 to N, the behaviour depends on the Index Behaviour option in the properties. The stacking order of graphical objects is determined by their Idx values. Objects with lower Idx's appear on top of objects with higher Idx's. It is essential to set the Idx value before transmitting events to the other ports..
- **Obj:** Graphical object type, or picture frame selection.  
4: Cross.  
3: Line from X1, Y1 to X1, Y1 of the next object of the same type.  
-2: Line from X1, Y1 to X2, Y2.  
-1: Bar.  
0: Rectangle.  
1 ... NP: Picture index that specifies a single picture (1) or a series of pictures in an animation (1, 2, 3, ..., NP), where NP is the total number of pictures in the animation.

Fractional values are rounded to the nearest integer.

If Ignore Index Obj (in Properties) is on, all graphical objects are set to the same type.

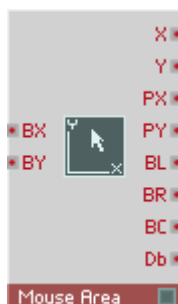
- **X1, Y1:** Coordinates for the first corner of the graphical object area, or if the 'Center to X1, Y1' option is selected in the properties, then these values define the coordinates of the centre of the object.
- **X2, Y2:** Coordinates for the second corner of the graphical object area.
- **R, G, B:** These inputs define the amount of Red, Green and Blue components of the object colour (range 0 to 1). If Ignore Index RGB (in Properties) is on, all graphical objects are set to the same color.
- **A:** Object transparency (0=fully transparent, 1=fully opaque).
- **X0, Y0:** Lowest visible X and Y values, (i.e. for scrolling). Inputs to these ports override entries to the X Origin and Y Origin fields in the properties.

- **XR, YR:** Horizontal/vertical view range, (i.e. for zooming). Inputs to these ports override entries to the X Range and Y Range fields in the properties.
- **NP:** Output reporting the number of pictures in the animation.
- **NO:** Output reporting the number of graphical objects.

The Poly Display inputs and outputs are the same as those of Multi Display, except that Poly Display has no Idx input and no NO output. All Poly Display inputs accept polyphonic event signals, except for X0, XR, Y0 and YR, which accept monophonic event signals.

## Mouse Area

## Panel



The Mouse Area module detects and outputs mouse actions, including button clicks, mouse drags, and changes in position. The Mouse Area can have its own outline and fill color, and can therefore be used as a panel interface element by itself. However, it is most typically used as an invisible (transparent) overlay on top of other modules, especially the Multi Display and Poly Display modules, in order to build highly customised interface elements.

The Mouse Area X and Y outputs can either operate in absolute or incremental mode (as selected in the properties). When Incremental Mode is enabled, the X and Y positions are adjusted incrementally by multiple drags actions, in the same way in which the built-in REAKTOR knobs and faders work. Specifically, when the user begins a new drag, the X and Y outputs transmit values relative to where the previous drag ended, as opposed to the absolute position of the mouse cursor. The BX and BY inputs are only effective in incremental mode, and are used to set the incremental ‘base’ for subsequent drags (overriding the last mouse drag). Typically, these are connected to Snap Value modules to ensure that the incremental base is set according to the last mouse movement saved in a snapshot.

In the properties, Outline Style specifies the outline appearance of the Mouse Area box. If 'Rectangle' is selected, then a 1-pixel outline is drawn around the Mouse Area box, whereas if 'Bar' is selected the area is filled with solid colour.

The Active State option specifies the action that causes the box outline to change from 'inactive' to 'active' state. Choose from either 'Selection' (the active state occurs whenever the Mouse Area box is selected), or the Left/Right/Center button options (active state occurs whenever the relevant mouse button is pressed). The Outline Color properties allow the color and transparency settings to be defined independently for both inactive and active states.

Also on the properties are values for X and Y offset, which allow the position of the Mouse Area on the instrument panel to be offset from the REAKTOR 4 x 4 grid.

**BX:** Input to set the base value for incremental changes at the X output. Legal values include those within the Range X, as defined in the properties.

**BY:** Input to set the base value for incremental changes at the Y output. Legal values include those within the Range Y, as defined in the properties.

Note that BX and BY inputs only influence the X and Y outputs when Incremental Mode option is enabled in the properties.

**X:** Output for the horizontal mouse position, scaled and limited to the Range X values (in the properties). The X output only reports mouse positions that are within the Mouse Area box (as defined by Size X and Size Y in the properties).

**Y:** Output for the horizontal mouse position, scaled and limited to the Range Y values (in the properties). The Y output only reports mouse positions that are within the Mouse Area box (as defined by Size X and Size Y in the properties).

**PX:** Horizontal mouse position in pixels relative to the X origin line (left edge of the Mouse Area box). Moving the mouse to the left of the X origin line outputs negative pixel values, moving it to the right outputs positive values. In contrast to the X output, which is limited to movements within the Mouse Area box, PX reports mouse positions all the way to the left and right edges of the screen.

**PY:** Vertical mouse position in pixels relative to the Y origin line (left edge of the Mouse Area box). Moving the mouse above the Y origin line outputs negative pixel values, moving it below outputs positive values. In contrast to the Y output, which is limited to movements within the Mouse Area box, PY reports mouse positions all the way to the left and right edges of the screen.

- BL:** Left mouse button state: 1 when pressed, or 0 when not pressed.
- BR:** Right mouse button state: 1 when pressed, or 0 when not pressed.
- BC:** Center mouse button state: 1 when pressed, or 0 when not pressed.
- Db:** Outputs a single event with value 1 every time a double-click occurs. The Db value at remains 1.0 after a double-click, so you must test for Db events, not static values.

---

## Stacked Macro

## Panel



Stacked Macros enables multiple graphical objects to share the same area of the instrument panel. Which objects are displayed within the area at any one time can then be controlled using the Panel Index module.

After inserting a Stacked Macro into your structure, place it in the correct position on the panel, and set the desired size in the properties. Then insert 2 or more macros (normal macros, not additional Stacked Macros) inside the Stacked Macro, along with a Panel Index module. Only one of the ‘normal’ macros (along with any panel elements inside it) will be visible at any one time. Which macro is visible depends on the input value to the Panel Index module. (To discover the index number of a macro, right click on that macro on the instrument panel - the index number will be displayed on the context menu).

---

## IC Send

## Terminal



Transmits monophonic event signals to any module capable of receiving IC (Internal Connection) transmission. Such modules include IC Receive modules, but also various panel elements (such as knobs and switches). As internal connections work globally (i.e. at the ensemble level), this module can be used to make wireless connections between different instruments within the ensemble.

The IC Send module has a panel display allowing connections to be configured from the instrument panel. All modules in the ensemble capable of receiving IC transmission will appear here, except those with the 'No Entry in IC Menu' properties option enabled. Connections can also be established in the properties dialog.

---

## IC Receive

## Terminal



Receives and outputs monophonic event signals to modules which connected via the Internal Connection (IC) protocol. Typically the IC Receive module is used with IC Send modules, but can be connected to any module capable of connection via IC (such as knobs and switches).

IC connections can be established in the properties, and when connecting to IC Send modules, connections can also be established using the IC Send panel interface.



# MIDI In

MIDI In modules are for routing MIDI messages in to REAKTOR from external MIDI devices. There are separate modules for various MIDI data types including notes, Velocity, pitchbend, mono and poly aftertouch, controllers, program changes, and MIDI clock. REAKTOR also generates separate gate messages for MIDI note events and there are modules for that. Some MIDI In Modules can also be used in an internal or OSC connection.

---

## Note Pitch

## MIDI In



Polyphonic event source for the pitch of MIDI Note On events.

A Note On event sets the output to a value determined by the key number (note pitch). The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps. A Note Off event has no effect here.

When using the default range of **Min** = 0 to **Max** = 127 and controlling an oscillator's **P**-input (for logarithmic pitch control) you will play in the common equal tempered tuning. One unit corresponds to one semitone and middle C is at 60. By setting **Min** and **Max** to different values, pitch can be skewed or scaled, e.g. for playing in quarter tones.

---

## Pitchbend

## MIDI In



Monophonic event source for MIDI Pitchbend (pitchwheel change) events. The resolution is 16384 steps. When the pitch bend controller is in its neutral position, the output value is always 0. The range of the output value for upward or downward pitchbend can be set independently in the properties dialog window. For pitchbend down the range is set with **Min** and for pitchbend up with **Max**.

When controlling an oscillator's **P**-input (for logarithmic pitch control), e.g. after adding to a note pitch value, and with a range of **Min** = -1 to **Max** = 1 you can change the pitch up or down one semitone. A range of -12 to 12 means pitchbend within  $\pm 12$  semitones which is  $\pm 1$  octave.



Polyphonic event source for MIDI Note On and Note Off events.

A Note On event sets the output to a value determined by the key velocity. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps. A Note Off event sets the output to zero. If velocity sensitivity is to be disabled, **Min** and **Max** should be set to the same value.

---

## Single Trig. Gate



Monophonic event source for MIDI Note On and Note Off events with single trigger characteristic.

A Note On event sets the output to a value determined by the key velocity. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps. A Note Off event sets the output to zero. If velocity sensitivity is to be disabled, **Min** and **Max** should be set to the same value.

Only the first note produces an event and thus triggers (or retriggers) a connected envelope. A new note that is started while the previous one is still held (legato play) does not generate an event and therefore does not retrigger any envelope.

---

## Sel. Note Gate



Monophonic event source for selected Note On and Note Off events.

A Note On event which has the selected note number sets the output to a value determined by the key velocity. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps. A Note Off event which has the selected note number sets the output to zero. If velocity sensitivity is to be disabled, **Min** and **Max** should be set to the same value.

---

## On Velocity

MIDI In



Polyphonic event source for velocity of MIDI Note On events. A Note On event sets the output to a value determined by the key velocity. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps.

---

## Off Velocity

MIDI In



Polyphonic event source for the velocity of MIDI Note Off events. A Note Off event sets the output to a value determined by the key release velocity. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps.

There are very few keyboards that generate a value other than zero for this event.

---

## Controller

MIDI In



Monophonic event source for MIDI Controller events. An event sets the output to a value determined by the position of the controller (e.g. modulation wheel). The number of the controller is set in the properties dialog window with **Controller No** and the range of the output value is set with **Min** and **Max**. The resolution is 128 steps.

The current position of all MIDI controllers (and faders) of an instrument can be stored in a snapshot from where it can be recalled at a later time. If the **Snap Isolate** switch is activated, the controller module's output value will not respond to snapshot recall.

The output of a controller is monophonic and can be used as an event or audio signal.

The numbers of some standard MIDI controllers are:

- 1 Modulation Wheel
- 2 Breath Controller
- 7 Volume
- 10 Panpot

- 64 Sustain Switch
- 65 Portamento Switch
- 66 Hold Switch (Sostenuto)

See your keyboard's MIDI implementation chart to find out which controllers it can transmit.

---

## Ch. Aftertouch

MIDI In



Monophonic event source for MIDI Channel Aftertouch events. An event sets the output to a value determined by the pressure on all keys. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps.

---

## Poly Aftertouch

MIDI In



Polyphonic event source for MIDI Poly Aftertouch events. An event sets the output to a value determined by the pressure on the key. The value is only set for the particular voice in the REAKTOR instrument which is playing the note associated with the key. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps.

There are very few keyboards that generate poly aftertouch events.

---

## Sel. Poly AT

MIDI In



Monophonic event source for selected MIDI Poly Aftertouch events.

An event which has the selected note number sets the output to a value determined by the pressure on the key. The number of the key (note number) is set in the properties dialog window with **Controller No** and the range of the output value is set with **Min** and **Max**. The resolution is 128 steps.

There are very few keyboards that generate poly aftertouch events.

The current value can be stored (together with the MIDI controllers and faders) in a snapshot from where it can be recalled at a later time. If the **Snap Isolate** switch is activated, the module's output value will not change on snapshot recall.

---

## Program Change

MIDI In



Monophonic event source for MIDI Program Change events. A MIDI event sets the module's output to the value determined by the transmitted program number. The range of the output value is set with **Min** and **Max** in the properties dialog window. The resolution is 128 steps.

When using this module, you probably want to turn off **Prog. Change Enable** in the Instrument Properties so that the MIDI Program Change Events do not also recall snapshots.

---

## Start/Stop

MIDI In



Source for start and stop events for the synchronization with external MIDI devices or the internal master clock.

The output of the **Start/Stop** module is a monophonic gate signal. Its value can be set with **Output Value** in the properties. The signal jumps to the set value when the start button in the toolbar is pressed or when a MIDI Start event is received, as appropriate. The signal jumps back to zero when the stop button is pressed or when a MIDI Stop event is received.

The module is typically connected to the reset input of sequencers and event dividers which are synchronized with the MIDI Clock to force a synchronous start.

---

## 1/96 Clock

MIDI In



Source of a clock signal which corresponds to the external MIDI Clock or the internal master clock.

For each 96<sup>th</sup> note an event is sent at the output. The value can be set as **Output Value** in the properties.

In contrast to the **Sync Clock** source, the events of the **1/96 Clock** have to be processed by an **Event Freq. Divider**. This has the advantage that you can experiment with arbitrary division factors.



Source for a clock signal which is derived from an external MIDI Clock or the internal master clock.

The output of the **Sync Clock** module is a monophonic gate signal. The value can be set with **Output Value** in the properties. At each beat, the gate jumps to the respective value and back to zero after a certain time interval. The module is activated and deactivated by start-stop events.

The rate of the clock signal (quarter, eighth, sixteenth notes, etc.) can be adjusted in properties as **Rate**.

The duration of the gate signal (eighth, sixteenth note, etc.) can be adjusted in properties as **Duration**.

---

## Song Pos



Source for the Song Position, counted in 96th notes from the song start. Typically connect this to input (A) of the Modulo module and a Constant of 6 to the (B) input, to get a 16th note count at the (Div) output.

- **96:** Event output for the integer count of 96th notes (that's 24 ppq). Use the Modulo module to get counts of other denominations.
- **96a:** Audio output for the sample-accurate fractional song position measured in 96th notes (24 per quarter note).



Monophonic source for receiving MIDI channel messages from an external MIDI device (keyboard or sequencer etc.), or internally from other instruments within the ensemble. The output order of the ports (see below) is strictly defined, from top to bottom. This ensures that the type (e.g. control change) and source (e.g. controller 7 on channel 1) of the message is always reported before the actual value.

**St:** Output port which reports the type of message received.

- 0 = Note Off
- 1 = Note On
- 2 = Poly Aftertouch
- 3 = Control Change
- 4 = Program Change
- 5 = Channel Aftertouch
- 6 = Pitchbend

**Ch:** Number of the MIDI channel (1-16).

**Nr:** Number of a Note, Control Change, or Program Change (0-127).

**Val:** Velocity of a Note, pressure of Aftertouch, or value of Control Change and Pitchbend. With external MIDI, the values are 7-bit quantized (14-bit for Pitchbend). With internal MIDI, the values are unlimited 32-bit floating-points. All values are scaled according to the Min and Max settings in the properties, which is 0 to 1 by default. Another common setting is 0 to 127, which can aid interpretability of values in some situations (Program Change for example).

# MIDI Out

REAKTOR can generate as well as process MIDI messages. MIDI Out modules are for sending those to external MIDI devices. There are modules here corresponding to each of the MIDI In modules. Some MIDI Out Modules can also be used in an internal or OSC connection.

---

## Note Pitch/Gate

## MIDI Out



Converts a monophonic or polyphonic event signal to MIDI Note events. Each event at the gate input **G** generates a MIDI message at the MIDI port which REAKTOR uses for output. The value of the event specifies the velocity. An event value of 1 produces a velocity of 127. An event with value zero produces a Note Off event.

The pitch of the Note On and Note Off events is the current value at the pitch input **P** in the range set with **Min** and **Max**. An event with value equal to **Min** sets the MIDI Note Pitch to 0, an event with value equal to **Max** sets the MIDI Note Pitch to 127.

---

## Pitchbend

## MIDI Out



Converts a monophonic event signal to MIDI Pitchbend events. Each event generates a MIDI message at the MIDI port which REAKTOR uses for output. The range of the input value is set with **Min** and **Max**. The output resolution is 16384 steps. An event with value equal to **Min** sets the MIDI Pitchbend value to -8192, an event with value equal to **Max** sets the MIDI Pitchbend value to +8191.

---

## Controller

## MIDI Out



Converts a monophonic event signal to MIDI Controller events. Each event generates a MIDI message at the MIDI port which REAKTOR uses for out-



put. The number of the controller is set in the properties dialog window with **Controller No** and the range of the input value is set with **Min** and **Max**. The output resolution is 128 steps. An event with value equal to **Min** sets the MIDI Controller value to 0, an event with value equal to **Max** sets the MIDI Controller value to 127.

---

## Ch. Aftertouch

## MIDI Out



Converts a monophonic event signal to MIDI Channel Aftertouch events. Each event generates a MIDI message at the MIDI port which REAKTOR uses for output. The range of the input value is set with **Min** and **Max**. The output resolution is 128 steps. An event with value equal to **Min** sets the MIDI Aftertouch value to 0, an event with value equal to **Max** sets the MIDI Aftertouch value to 127.

---

## Poly Aftertouch

## MIDI Out



The Aftertouch (**AT**) events are converted to MIDI Poly Aftertouch events. The momentary value at the Pitch (**P**) input is converted to the note number. Values between **Min** and **Max** result in note numbers between 0 and 127. Aftertouch values between 0 and 1 are converted to values between 0 and 127.

- **P**: Input for the Pitch converted to the MIDI note number. Values between **Min** and **Max** result in note numbers between 0 and 127.
- **AT**: Input for the Aftertouch signal. Values between 0 and 1 are converted to MIDI aftertouch values between 0 and 127.

---

## Sel. Poly AT

## MIDI Out



Converts a monophonic event signal to MIDI Poly Aftertouch events. Each event generates a MIDI message at the MIDI port which REAKTOR uses for output. The number of the key for which to set the pressure (note number) is set in the properties dialog window with **Note No.** and the range of the input

value is set with **Min** and **Max**. The output resolution is 128 steps. An event with value equal to **Min** sets the aftertouch value to 0, an event with value equal to **Max** sets the aftertouch value to 127.

There are very few MIDI devices which handle poly aftertouch events.

---

## Program Change

## MIDI Out



Converts a monophonic event signal to MIDI Program Change events. Each event generates a MIDI message at the MIDI port which REAKTOR uses for output. The range of the input value is set with **Min** and **Max**. The output resolution is 128 steps. An event with value equal to **Min** sets the MIDI Program Change value to 0, an event with value equal to **Max** sets the MIDI Program Change value to 127.

---

## Start/Stop

## MIDI Out



Input events create Start/Continue/Stop events at the MIDI output.

- **G**: A positive event sends a Start or a Continue message. Negative or zero events send a Stop message.
- **Rst**: After a positive event at this Reset input, the next positive event at the Gate sends a Start (at zero) message.

---

## 1/96 Clock

## MIDI Out



Input events create (1/96th) clock events at the MIDI output.

- **In**: An event with a positive value creates a MIDI clock event.



The value at the **Pos** input is sent with an event at the **Trig** input as MIDI song position.

- **Trg:** Every event with a positive value creates a MIDI song position event.
- **Pos:** The value at this input (as a multiple of 1/96th notes) is taken with a Trigger event and sent as MIDI song position.



Monophonic source for sending MIDI channel messages to an external MIDI device (sequencer etc.), or internally to other instruments within the ensemble. Channel Messages are transmitted every time an event arrives at the **St** port. Thus, the desired message value and destination must be correctly defined with appropriate values at the other inputs before events arrive at the **St** input. All inputs accept monophonic signals only.

**St:** Event input defining the type of channel message to be sent. A channel message is transmitted everytime an event arrives at this input.

- 0 = Note Off
- 1 = Note On
- 2 = Poly Aftertouch
- 3 = Control Change
- 4 = Program Change
- 5 = Channel Aftertouch
- 6 = Pitchbend

- Ch:** Audio input for the number of the MIDI channel (1-16). Fractional values arriving at Ch are rounded up to the nearest integer; e.g. a value of 0.5 would be rounded up to 1.
- Nr:** Audio input for the number of the Note, Control Change, or Program Change (0-127).
- Val:** Audio input for the velocity of a Note, pressure of Aftertouch, or value of Control Change and Pitchbend.

# Math

You figure out the equation and REAKTOR does the math. There are modules here for common operations such as addition, subtraction, multiplication, and division as well as for less-familiar mathematical functions such as absolute value, arc-tangent, and reciprocal square root. There are also modules here for exponential and logarithmic scale conversion to match REAKTOR's module-input scaling. For example, some REAKTOR modules have linear frequency inputs (measured in Hertz) while others have exponential frequency inputs (measured in semitones and adjusted for MIDI note numbering). There are modules here for converting between those two formats.

All modules in this section are hybrid modules, meaning they can be used as either audio or event modules depending on their inputs. Many of the modules (**Add** and **Mult**, for example) also have dynamic inputs as indicated by three small dots at the bottom-left of the module icon. When a wire is dragged to an empty part of the in-port region (the left edge of the module icon) of a dynamic-input module while holding down the **Ctrl** key, a new input is created automatically.

---

## Constant

## Math



Monophonic source for a constant value. The value is set with **Constant Value** in the properties dialog window. The module only sends an event once, that is for initialization when it is activated.

---

## Add

## Math



Adder for two or more audio or event signals. The output signal is the sum of the input signals ( **Out = In1 + In2 + In3** etc.).

Can also be used as a simple multi-channel mixer where the level of all channels is fixed at 0 dB.

---

## Subtract

Math



Subtractor for two audio or event signals. The output signal is the subtraction of the second input signal from the first (**Out** = **In1** - **In2**).

---

## Invert, -X

Math



Inverter for an audio or event signal. The output signal is the inverse of the input signal ( **Out** = **-In** ).

Simply inverting a sound has no audible effect, except when combining in some way with the uninverted sound. But inverting control signals generally has a very obvious effect.

---

## Multiply

Math



Multiplier for two or more audio or event signals. The output signal is the product of the input signals ( **Out** = **In1** \* **In2** \* **In3** etc).

The typical application is as an amplifier controlled by a signal (corresponds to VCA in analog synthesizers) when an audio signal is fed to one input and an amplification factor to the other.

When a zero is connected to one of the inputs the output value is always zero.

Can also be used to compute the square of a signal when the same signal is fed to two inputs (**Out** = **In** \* **In** = **In<sup>2</sup>** ).

When two different sounds are connected to the inputs, the output is the ring modulation of the two sounds.

---

$$a * b + c$$

**Math**



Combined multiplier-adder: Output is the result of  $(A * B) + C$ .

---

## Reciprocal 1/x

**Math**



The output is one divided by the input.

Be careful with small input values (near zero) because the output values become very large. If the input signal is exactly zero the output is also set to zero.

Processing sound signals does not normally yield anything useful. The module is rather meant for processing control signals (which don't come near zero).

---

## Divide x/y

**Math**



The output is the upper input divided by the lower input.

Be careful with small values (near zero) at the lower input because the output values become very large. If the input signal is exactly zero the output is also set to zero.

Dividing by sound signals does not normally yield anything useful.

Whenever possible use a multiplier instead of a divider for audio signals because the CPU load will be significantly less. For example, rather than dividing by a constant or an event signal, invert the event signal (1/x) and multiply audio with the result.

---

## Modulo x % y

Math



Modulo and Div. Computes the integer division of the two input values and also the remainder.

- **A:** Hybrid input for signal **A** to be divided by **B**. Typ. Range: [ 0 ... 100 ].
- **B:** Hybrid input for signal **B** used for dividing **A**. **B** is normally an integer, but doesn't have to be. Typ. Range: [ 1 ... 100 ].
- **Div:** Hybrid output for the integer division of **A** and **B**. This is the largest integer smaller than or equal to **A/B**. Typ. Range: [ 0 ... 100 ].
- **Mod:** Hybrid output for the modulo of **A** and **B**. This is the remainder of the integer division of **A** and **B**. Range: [ 0 ... B ].

---

## Rectifier

Math



The input signal is rectified, i.e. negative values are inverted and become positive.

- **In:** Hybrid input for signal to be rectified. Negative values become positive with equal magnitude.
- **Out:** Hybrid output for the rectified signal.

---

## Rect./Sign

Math



The input signal is rectified, i.e. negative values are inverted and become positive. The sign of the input signal is available at the **Sign** output.

- **In:** Hybrid input for signal to be rectified and analyzed for the sign.
- **Sign:** Hybrid output for the sign of the input signal. 1 for positive signals, -1 for negative signals.



- **Out:** Hybrid output for the rectified signal. Always positive.

## Compare

## Math

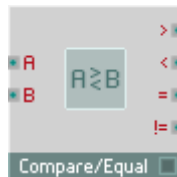


Comparing Logic Function. Compares the two input values and sets the two outputs according to the result of the comparison.

- **A:** Hybrid input for the first of two values to be compared.
- **B:** Hybrid input for the second of two values to be compared.
- **>:** Hybrid output for the result of the comparison “**A** greater than **B**”. (0 = FALSE, 1 = TRUE)
- **<=:** Hybrid output for the result of the comparison “**A** less than **B**”. (0 = FALSE, 1 = TRUE)

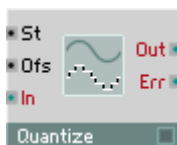
## Compare/Equal

## Math



Comparing Logic Function. Compares the two input values and sets the four outputs according to the result of the comparison.

- **A:** Hybrid input for the first of two values to be compared.
- **B:** Hybrid input for the second of two values to be compared.
- **>:** Hybrid output for the result of the comparison “**A** greater than **B**”. (0 = FALSE, 1 = TRUE)
- **<=:** Hybrid output for the result of the comparison “**A** less than **B**”. (0 = FALSE, 1 = TRUE)
- **=:** Hybrid output for the result of the comparison “**A** equal to **B**”. (0 = FALSE, 1 = TRUE)
- **!=:** Hybrid output for the result of the comparison “**A** not equal to **B**”. (0 = FALSE, 1 = TRUE)



Quantizer for audio and event signals, with adjustable step size.

The input signal is rounded to the nearest quantization step before being output.

- **St:** Hybrid input for controlling the size of the quantization step. When set to zero, the signal is not quantized.
- **In:** Hybrid input for the signal to be quantized.
- **Out:** Hybrid output for the quantized signal.
- **Err:** Hybrid output for the quantization error that is generated by rounding:  $\text{Err} = \text{Out} - \text{In}$ .

---

## Expon. (A)



Exponentiator for converting logarithmic level values in dB to linear amplitude values.

- **Lvl:** Hybrid input for logarithmic level values in dB to be converted to linear amplitude values. Typ. range: [-50 ... 10 ].
- **A:** Hybrid output for a linear amplitude control signal.

---

## Expon. (F)



Exponentiator for converting logarithmic pitch values in semitones to linear frequency values in Hz.

- **P:** Hybrid input for logarithmic pitch values in semitones to be converted to linear frequency values in Hz. Typ. range: [ 0 ... 127 ].
- **F:** Hybrid output for a frequency control signal in Hz.

---

## Log (A)

Math



Logarithm for converting linear amplitude values to logarithmic level values in dB. Also for driving the logarithmic time inputs of envelopes etc.

- **A:** Hybrid input for linear amplitude value to be converted to logarithmic level value in dB. Typ. range: [0 ... 1000].
- **Lvl:** Hybrid output for level in dB. Typ. range: [-60 ... 0].

---

## Log (F)

Math



Logarithm for converting linear frequency values in Hz to logarithmic pitch values in semitones.

- **F:** Hybrid input for linear frequency value in Hz to be converted to logarithmic pitch value in semitones. Typ. range: [ 0 ... 5000 ].
- **P:** Hybrid output for pitch in semitones. Typ. range: [ 0 ... 100 ].

---

## Power x y

Math



Power Function. The output delivers the result of X to the power of Y (usually denoted  $X^Y$  or  $X^Y$ ).

**X:** Hybrid input for the basis.

**Y:** Hybrid input for the exponent.

**$X^Y$ :** Hybrid output for the result of the calculation.

---

## Square Root

Math



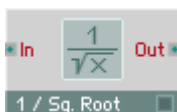
Computes the square root of the input values.

- **In:** Hybrid input for the argument to the square root function. For negative input values, the output is zero. Typ. Range: [ 0 ... 100 ].
- **Out:** Hybrid output for the square root of the input value. Typ. Range: [ 0 ... 10 ].

---

## 1 / Square Root

Math



The reciprocal square root module computes the reciprocal of the square root of the input. It is more efficient than using the Square Root module followed by the Reciprocal module. For negative inputs the output is zero.

- **In:** Hybrid input for the argument.
- **Out:** Hybrid output for the value.

---

## Sine

Math



The Sine module calculates the trigonometric sine function. Both the input and output are scaled to a range of -1 to 1. To calculate an input based on degrees, first divide by 360. To calculate an input based on radians, divide by  $2\pi$  (approximately 6.283). The output ranges from 1 (for input .25) to -1 (input .75).

- **In:** Hybrid input for the argument.
- **Out:** Hybrid output for the value.



For each input event, the sine and cosine are computed. An input value of 1.0 corresponds to a full period of the sin and cos functions (i.e. 360 degrees).

- **In:** Hybrid input for the argument to the sine function. A value of 1.0 corresponds to one period of the sine function (360 degrees). Typ. Range: [ -1 ... 1 ].
- **Sin:** Hybrid output for the sine of the input value. Range: [ -1 ... 1 ].
- **Cos:** Hybrid output for the cosine of the input value. Range: [ -1 ... 1 ].



The ArcSin module calculates the inverse sine function. (The arc sine of  $x$  is the number between 0 and 1 whose sine is  $x$ .) Since the sine function's output range is -1 to 1, the arc sine function is only valid for inputs within that range. For arguments  $< -1$  the ArcCos returns -0.25 and for arguments  $> 1$ , it returns 0.25.

- **In:** Hybrid input for the argument.
- **Out:** Hybrid output for the arc sine of the input.



The ArcCos module calculates the inverse cosine function. (The arc cosine of  $x$  is the number between 0 and 1 whose cosine is  $x$ .) Since the cosine function's output range is -1 to 1, the arc cosine function is only valid for inputs within that range. For arguments  $< -1$  the ArcCos returns 0.5 and for arguments  $> 1$ , it returns 0.

- **In:** Hybrid input for the argument.
- **Out:** Hybrid output for the arc cosine of the input.

---

## ArcTan

## Math



The ArcTan module calculates the inverse tangent function. (The tangent is the sine divided by the cosine.) The output of ArcTan module ranges from -0.25 to 0.25, which corresponds to -90 to 90 degrees.

- **In:** Hybrid input for the argument.
- **Out:** Hybrid output for the arc tangent of the input.

# Signal Path

Signal Path modules allow both control and audio data to be flexibly routed in REAKTOR structures. These include mixers, input and output selectors, remote-control switches (called Relays), crossfaders, and panners.

---

## Selector/Scanner

## Signal Path



Selector/Scanner. The inputs are scanned by sweeping the value at the **Pos** input. When **Pos** is an integer, you get just one input signal, otherwise a mix of two inputs. The output signal is obtained by crossfading between the two inputs whose index is closest to the Value at the **Pos** input. The crossfading is done according to crossfading mode specified in the Properties.

When **Wrap** mode is selected in the Properties, **Pos** wraps around so that Max+1 is the same as 0, Max+2 is 1 etc.

The Selector/Scanner makes nice effects when the inputs are fed from the Multitap Delay and the scan position is controlled by a Ramp Oscillator.

The module has a dynamic in-port management. The number of in-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **Pos:** Hybrid input for selecting the input(s) to scan. **Pos** = 0 selects **In0**, **Pos** = 1 selects **In1**, **Pos** = 0.5 gives a mix of **In0** and **In1**. Typ. Range: [ 0 ... Max ]
- **In 0...Max:** Hybrid inputs for the signals to be scanned.
- **Out:** Output for the scanned signal.

---

## Relay 1,2

## Signal Path



Relay. The upper input is connected to the output if Ctl > 0. Otherwise the lower input is connected to the output. If the lower input is not present a zero signal is produced at the output.

The module has can have one or two in-ports. The number of in-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **Ctl**: Control input for selecting a signal input.
- **In**: Signals input(s).
- **Out**: Output for the selected signal.

---

## Crossfade

## Signal Path



Crossfade module for two signals. The output signal is mixed from the both input signals **In0** and **In1**. The mix ratio is controlled by the **X** input.

The module has a dynamic port management. The number of in-port pairs (**In0** and **In1**) and out-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **X**: Hybrid control input for the mixing ratio. Value between 0 ( **Out** = **In1** ) and 1 ( **Out** = **In2** ).
- **In1, In2**: Hybrid inputs for the two signals to be mixed.
- **Out**: Hybrid output for the mixed signal ( **Out** = (1 – **X**) **In1** + **X** **In2**).

---

## Distributor/Panner

## Signal Path



Distributor/Panner. The signal at the input will be connected/panned to the output(s) selected by the Pos input. When Pos is an integer, the signal is connected to just one input signal, otherwise you get a ‘panning’ between two inputs. The panning is done according to panning mode specified in the Properties. When Wrap mode is selected in the Properties, Pos wraps around so that Max+1 is the same as 0, Max+2 is 1 etc..

The module has a dynamic out-port management. The number of out-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.



- **Pos:** Input for selecting the thru output.
- **In:** Signal input.
- **Out:** Output for the input signal.

---

## Stereo Pan

## Signal Path



Left-right panner. By changing the signal level at the two outputs the input signal is positioned in the stereo field. The sum of the **L** and **R** output values is always exactly twice the input value, i.e. the input is split across the two outputs at a variable ratio.

The module has a dynamic port management. The number of in-ports and out-port pairs (**L** and **R**) can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **Pan:** Control input for the left-right position. Value -1 = left, 0 = center, 1 = right.
- **In:** Hybrid signal input for the signal to be positioned in stereo.
- **L:** Hybrid signal output for the left channel signal.
- **R:** Hybrid signal output for the right channel signal.

---

## Amp/Mixer

## Signal Path



Amp/Mixer for an adjustable number of input signals. The input signals are amplified (or attenuated) by their respective amounts at the **Lvl**-inputs (in dB) and then summed to form the output.

The module has a dynamic in-port management. The number of in-port pairs (**Lvl** and **In**) can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **Lvl:** Logarithmic control input for controlling the gain, value in dB.

- **In:** Audio input for the signal to be amplified.
- **Out:** Output for the mixed signal (  $\text{Out} = \text{In1} \cdot 10^{\text{Lvl1}/20} + \text{In2} \cdot 10^{\text{Lvl2}/20}$  etc.).

## Stereo Amp/Mixer

## Signal Path



Amplifier for an adjustable number of audio signals with integrated left-right panner. The input signals are amplified (or attenuated) by the set amount at the **Lvl** inputs (in dB). By changing the signal level at the two outputs the input signals are positioned in the stereo field.

The module has a dynamic in-port management. The number of in-port groups (**Lvl**, **Pan** and **In**) can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **Lvl:** Logarithmic input for controlling the gain, value in dB. When the value is negative, the output signal is smaller than the input signal.
- **Pan:** Control input for the left-right position. Value -1 = left, 0 = center, 1 = right.
- **In:** Audio signal input for the signal to be amplified and positioned in stereo.
- **L:** Audio signal output for the amplified left channel signal.
- **R:** Audio signal output for the amplified right channel signal .

# Oscillator

REAKTOR uses the term oscillator for a broad range of signal generators. In fact, everything signal generating process that doesn't involve playing samples is called an oscillator. That includes the usual, single-cycle waveform generators (sine, pulse, sawtooth, and so on) as well as impulse, step, noise, and table-driven generators.

All of REAKTOR's oscillators can run at any frequency, from 0 Hz (standstill) through the entire audio range right up to the limit set by the sample rate. Like this they are all equally suited for use as LFOs or for producing sound waves. When using an oscillator as an LFO to modulate another module's input which is of the type that only accepts events (e.g. P as opposed to F) you need to insert an A to E (perm) module for conversion.

---

## Sawtooth

## Oscillator



Oscillator for sawtooth waveform with logarithmic pitch control and linear amplitude modulation.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between +A and -A.
- **Out:** Audio signal output for the sawtooth waveform.

---

## Saw FM

## Oscillator



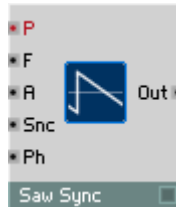
Oscillator for sawtooth waveform with logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).

- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between **+A** and **-A**.
- **Out:** Audio signal output for the sawtooth waveform.

## Saw Sync

## Oscillator



Oscillator for sawtooth waveform with synchronization, logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation.

Whenever the synchronization signal goes from zero to positive values (rising edge) the phase of the oscillator is reset to a position specified by the phase input.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between **+A** and **-A**.
- **Snc:** Audio input for controlling synchronization of the waveform. A rising edge restarts the oscillator.
- **Ph:** Input for specifying the phase (position in the waveform) to which the oscillator is reset when synchronization occurs.
- **Out:** Audio signal output for the sawtooth waveform.



Oscillator for variable sawtooth/needle-pulse waveform with logarithmic pitch control and linear amplitude modulation. The slope of the ramp is variable and with it the shape of the waveform can be changed from a normal sawtooth to a short triangular pulse.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Slp:** Audio input for controlling the slope of the waveform. A value of 0 produces a normal sawtooth, a larger value shortens the ramp to a short spike (needle).
- **Out:** Audio signal output for the sawtooth/pulse waveform.



Oscillator for bipolar sawtooth waveform with zero-phase. With logarithmic pitch control, pulse width modulation (PWM) and linear amplitude modulation.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **W:** Audio input for controlling the pulse width (PWM). Range of values is 0 to about 6. Normal sawtooth-wave at  $W = 0$ , larger  $W$  results in a shorter pulse and a longer zero-phase.
- **Out:** Audio signal output for the bipolar sawtooth waveform with zero-phase.



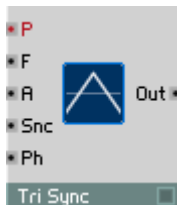
Oscillator for symmetric triangle waveform with logarithmic pitch control and linear amplitude modulation.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Out:** Audio signal output for the triangle waveform.



Oscillator for symmetric triangle waveform with logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Out:** Audio signal output for the triangle waveform.



Oscillator for symmetric triangle waveform with synchronization, logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation.

Whenever the synchronization signal goes from zero to positive values (rising edge) the phase of the oscillator is reset to a position specified by the phase input.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Snc:** Audio input for controlling synchronization of the waveform. A rising edge restarts the oscillator.
- **Ph:** Input for specifying the phase (position in the waveform) to which the oscillator is reset when synchronization occurs.
- **Out:** Audio signal output for the triangle waveform.



Oscillator for variable triangle and parabolic (near to sine) signals, with logarithmic pitch (P) control and linear amplitude (A) modulation. The symmetry is adjustable by (W). Setting the symmetry (with W) allows a range of wave-

forms from symmetric with few harmonics to asymmetric with a broader spectrum.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between +A and -A.
- **W:** Event input for controlling the symmetry of the waveform. A value of -1 produces a rising-ramp sawtooth, 0 produces a symmetrical triangle and +1 a falling-ramp sawtooth.
- **Tri:** Audio signal output for the triangle signal.
- **Par:** Audio signal output for the parabolic signal.

---

## Parabol

## Oscillator



Oscillator for parabolic waveform with logarithmic pitch control and linear amplitude modulation. The waveform consists of two halves that are each a section of a parabola. The oscillator sounds like a sine wave with some added odd numbered overtones at very low level. In many cases it can be used as replacement for a sine generator with less computational load.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between +A and -A.
- **Out:** Audio signal output for the parabolic waveform.

---

## Par FM

## Oscillator



Oscillator for parabolic waveform with logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation. The oscillator



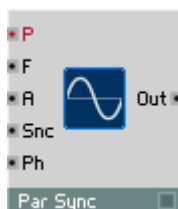
sounds like a sine wave with some added odd numbered overtones at very low level. In many cases it can be used as replacement for a sine generator with less computational load.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Out:** Audio signal output for the parabolic waveform.

---

## Par Sync

## Oscillator



Oscillator for parabolic waveform with synchronization, logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation.

Whenever the synchronization signal goes from zero to positive values (rising edge) the phase of the oscillator is reset to a position specified by the phase input.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Snc:** Audio input for controlling synchronization of the waveform. A rising edge restarts the oscillator.
- **Ph:** Input for specifying the phase (position in the waveform) to which the oscillator is reset when synchronization occurs.
- **Out:** Audio signal output for the parabolic waveform.



Oscillator for variable parabolic waveform with logarithmic pitch control and linear amplitude modulation. The ratio between the length of the upper and lower parts of the curve can be controlled, and with it the waveform can be changed from a normal symmetric parabolic wave to a simple parabola.

This variable waveform is also particularly effective when used as an LFO.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **W:** Event input for controlling the symmetry of the waveform. A value of  $-1$  produces parabolas open downward,  $0$  a normal symmetric parabolic wave and  $+1$  parabolas open upward.
- **Out:** Audio signal output for the variable parabolic waveform.



Oscillator for pure sine waveform with logarithmic pitch control and linear amplitude modulation.

If the sine tone does not need to be completely pure, the parabolic oscillator makes a good replacement with less computational load.

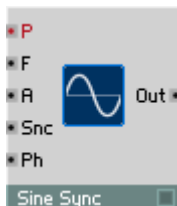
- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Out:** Audio signal output for the sine waveform.



Oscillator for pure sine waveform with logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation.

If the sine tone does not need to be completely pure, the parabolic oscillator makes a good replacement with less computational load.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **Out:** Audio signal output for the sine waveform.



Oscillator for pure sine waveform with synchronization, logarithmic pitch control, linear frequency modulation (FM) and linear amplitude modulation.

Whenever the synchronization signal goes from zero to positive values (rising edge) the phase of the oscillator is reset to a position specified by the phase input.

If the sine tone does not need to be completely pure, the parabolic oscillator makes a good replacement with less computational load.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440$  Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.

- **A:** Audio input for controlling the amplitude. The output signal moves between +A and -A.
- **Snc:** Audio input for controlling synchronization of the waveform. A rising edge restarts the oscillator.
- **Ph:** Input for specifying the phase (position in the waveform) to which the oscillator is reset when synchronization occurs. **Ph** = 0: Phase = 0 deg (middle of rising slope), **Ph** = 0.5: Phase = 180 deg (middle of falling slope), **Ph** = 1: Phase = 360 deg (same as 0 deg).
- **Out:** Audio signal output for the sine waveform.

## Multi-Sine

## Oscillator



Oscillator for Additive Synthesis. A waveform is built up in its harmonics by layering individual sine waves. For each component, the amplitude **A** and the frequency multiple **F** can be set.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **F1:** Input for the frequency factor (harmonic number) of the first sine wave. Scale: multiple of fundamental frequency. Typ. Range: [ 0 ... 20 ].
- **A1:** Input for linear amplitude control of the first sine wave. Can also be used for tremolo and ring modulation. Typ. Range: [ 0 ... 1 ].
- **F2:** Input for the frequency factor (harmonic number) of the second sine wave. Scale: multiple of fundamental frequency. Typ. Range: [ 0 ... 20 ].
- **A2:** Input for linear amplitude control of the second sine wave. Can also be used for tremolo and ring modulation. Typ. Range: [ 0 ... 1 ].

- **F3**: Input for the frequency factor (harmonic number) of the third sine wave. Scale: multiple of fundamental frequency. Typ. Range: [ 0 ... 20 ].
- **A3**: Input for linear amplitude control of the third sine wave. Can also be used for tremolo and ring modulation. Typ. Range: [ 0 ... 1 ].
- **F4**: Input for the frequency factor (harmonic number) of the fourth sine wave. Scale: multiple of fundamental frequency. Typ. Range: [ 0 ... 20 ].
- **A4**: Input for linear amplitude control of the fourth sine wave. Can also be used for tremolo and ring modulation. Typ. Range: [ 0...1].
- **S1**: Output for the first component sine wave.
- **S2**: Output for the second component sine wave.
- **S3**: Output for the third component sine wave.
- **S4**: Output for the fourth component sine wave.
- **Out**: Output for the signal generated by the oscillator by adding all the sine waves.

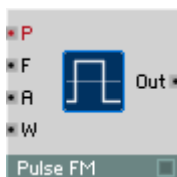
## Pulse

## Oscillator



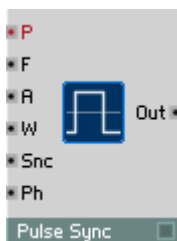
Oscillator for pulse wave with logarithmic pitch control, pulse width modulation (PWM) and linear amplitude modulation.

- **P**: Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **A**: Audio input for controlling the amplitude. The output signal moves between +A and -A.
- **W**: Audio input for controlling the pulse width (PWM). Range of values is -1 to 1. Symmetric waveform (square wave) 50:50 at **W** = 0, Lo:Hi-ratio 33:66 at -0.33, 66:33 at 0.33, 75:25 at 0.5, 90:10 at 0.8. Lo: Hi =  $(1 + \mathbf{W}) / (1 - \mathbf{W})$ .
- **Out**: Audio signal output for the pulse waveform.



Oscillator for pulse wave with logarithmic pitch control, linear frequency modulation (FM), pulse width modulation (PWM) and linear amplitude modulation.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between +A and -A.
- **W:** Audio input for controlling the pulse width (PWM). Range of values is -1 to 1. Symmetric waveform (square wave) 50:50 at **W** = 0, Lo:Hi-ratio 33:66 at -0.33, 66:33 at 0.33, 75:25 at 0.5, 90:10 at 0.8. Lo:Hi =  $(1 + \mathbf{W}) / (1 - \mathbf{W})$ .
- **Out:** Audio signal output for the pulse waveform.



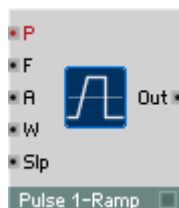
Oscillator for pulse wave with synchronization, logarithmic pitch control, linear frequency modulation (FM), pulse width modulation (PWM) and linear amplitude modulation.

Whenever the synchronization signal goes from zero to positive values (rising edge) the phase of the oscillator is reset to a position specified by the phase input.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **W:** Audio input for controlling the pulse width (PWM). Range of values is -1 to 1. Symmetric waveform (square wave) 50:50 at  $W = 0$ , Lo:Hi-ratio 33:66 at  $-0.33$ , 66:33 at  $0.33$ , 75:25 at  $0.5$ , 90:10 at  $0.8$ . Lo: Hi =  $(1 + W) / (1 - W)$ .
- **Snc:** Audio input for controlling synchronization of the waveform. A rising edge restarts the oscillator.
- **Ph:** Input for specifying the phase (position in the waveform) to which the oscillator is reset when synchronization occurs.
- **Out:** Audio signal output for the pulse waveform.

## Pulse 1-ramp

## Oscillator



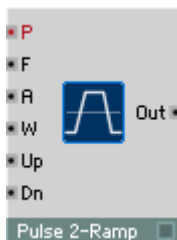
Oscillator for rectangular trapezoid waveform with logarithmic pitch control, linear frequency modulation (FM), pulse width modulation (PWM) and linear amplitude modulation. The waveform is a mixture of pulse and sawtooth: The rising edge of a pulse wave can be flattened and its slope adjusted. The falling edge is vertical.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **W:** Audio input for controlling the pulse width (PWM). Range of values is -1 to 1.

- **Slp**: Audio input for controlling the slope of the rising edge of the waveform. When **Slp** is zero (or when the input is not connected) the waveform does not rise and the output is always zero, i.e. there is no sound. Typical range of values: 1 ... 20
- **Out**: Audio signal output for the trapezoid waveform.

## Pulse 2-ramp

## Oscillator



Oscillator for trapezoid waveform with logarithmic pitch control, linear frequency modulation (FM), pulse width modulation (PWM) and linear amplitude modulation. The waveform is a mixture of pulse, sawtooth and triangle: Both edges of a pulse wave can be flattened and their slope adjusted.

- **P**: Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F**: Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A**: Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **W**: Audio input for controlling the pulse width (PWM). Range of values is -1 to 1.
- **Up**: Audio input for controlling the slope of the rising edge of the waveform.
- **Dn**: Audio input for controlling the slope of the falling edge of the waveform.
- **Out**: Audio signal output for the trapezoid waveform.



---

## Bi-Pulse

## Oscillator



Oscillator for bipolar pulse wave with zero-phase. With logarithmic pitch control, pulse width modulation (PWM) and linear amplitude modulation.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **W:** Audio input for controlling the pulse width (PWM). Range of values is 0 to 1. Symmetric waveform (square wave) 50:50 at  $W = 0$ , larger **W** results in a shorter pulse and longer zero-phase.
- **Out:** Audio signal output for the bipolar pulse waveform.

---

## Impulse

## Oscillator



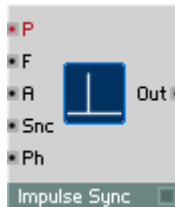
Oscillator for impulse train signal with logarithmic pitch control (P) and linear amplitude modulation (A).

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **A:** Audio input for controlling the amplitude.
- **Out:** Audio signal output for the impulse waveform.



Oscillator for impulse train signal with logarithmic pitch control (P), linear frequency modulation (F) and linear amplitude modulation (A).

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude.
- **Out:** Audio signal output for the impulse waveform.



Oscillator for synchronizable impulse train signal with logarithmic pitch control (P), linear frequency modulation (F), linear amplitude modulation (A) and Sync input (Snc).

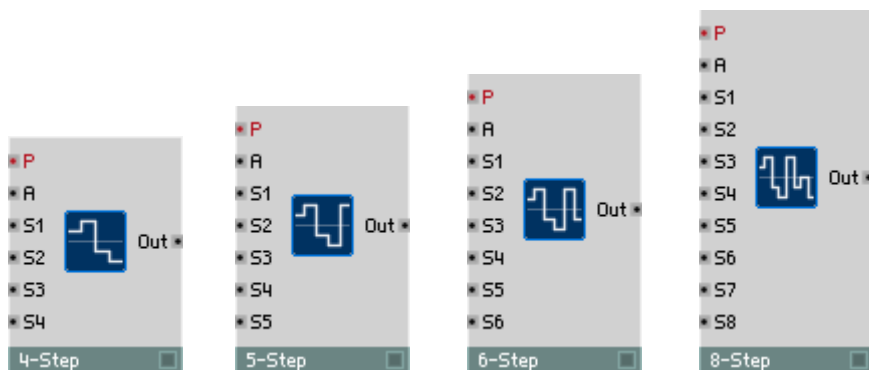
Whenever the synchronization signal goes from zero to positive values (rising edge) the phase of the oscillator is reset to a position specified by the phase input.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **F:** Audio input for linear frequency modulation. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **A:** Audio input for controlling the amplitude.
- **Snc:** Audio input for controlling synchronization of the waveform. A rising edge restarts the oscillator.

- **Ph:** Input for specifying the phase (position in the waveform) to which the oscillator is reset when synchronization occurs.
- **Out:** Audio signal output for the impulse waveform.

## Multi-Step

## Oscillator



## 4-Step

## Oscillator

Oscillator for 4-step waveform with logarithmic pitch control and linear amplitude modulation. The level of each step can be set independent of the others.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **A:** Audio input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **S1:** Audio input for controlling the level of the first step.
- **S2:** Audio input for controlling the level of the second step.
- **S3:** Audio input for controlling the level of the third step.
- **S4:** Audio input for controlling the level of the fourth step.
- **Out:** Audio signal output for the step waveform.

---

## 5-Step

## Oscillator

Like 4-Step but with 5 steps.

---

## 6-Step

## Oscillator

Like 4-Step but with 6 steps.

---

## 8-Step

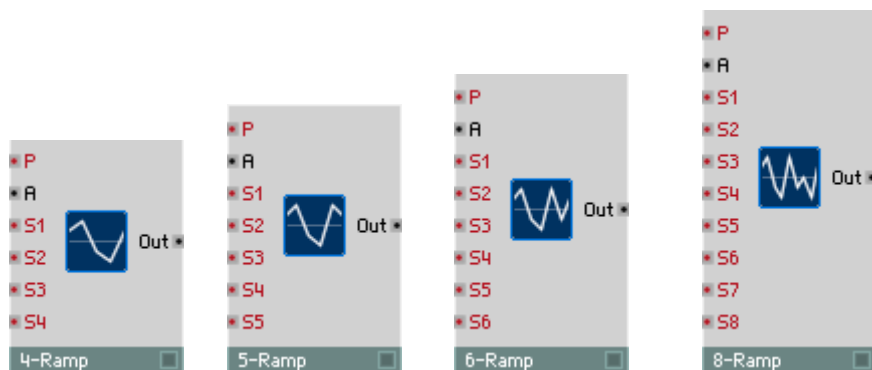
## Oscillator

Like 4-Step but with 8 steps.

---

## Multi-Ramp

## Oscillator



---

## 4-Ramp

## Oscillator

Oscillator for 4-ramp waveform with logarithmic pitch control and linear amplitude modulation. The level of each of the breakpoints which are connected by ramps can be set independent of the others.

- **P:** Logarithmic event input for controlling the pitch (oscillator frequency). Value in semitones (69 = 440 Hz).
- **A:** Audio input for controlling the amplitude. The output signal moves between +A and -A.
- **S1:** Event input for controlling the level of the first breakpoint.
- **S2:** Event input for controlling the level of the second breakpoint.

- **S3:** Event input for controlling the level of the third breakpoint.
- **S4:** Event input for controlling the level of the fourth breakpoint.
- **Out:** Audio signal output for the ramp waveform.

---

## 5-Ramp

## Oscillator

Like 4-Ramp but with 5 ramps.

---

## 6-Ramp

## Oscillator

Like 4-Ramp but with 6 ramps.

---

## 8-Ramp

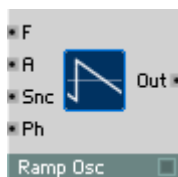
## Oscillator

Like 4-Ramp but with 8 ramps.

---

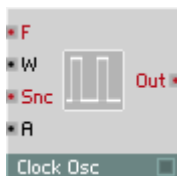
## Ramp

## Oscillator



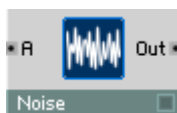
Oscillator to produce a ramp waveform, typically used as a control signal, for example to use an Audio Table module as a waveform oscillator. The signal ramps up from 0 to A and then resets instantly.

- **F:** Audio input for control of the oscillator frequency in Hz. To control the pitch in semitones, use an Event Expon. (F) module.
- **A:** Input for controlling the amplitude of the ramp signal. Typ. Value: 1
- **Snc:** Audio input for controlling synchronization of the waveform. A rising edge resets the oscillator to **Ph**.
- **Ph:** Audio input for the synchronization phase. When the oscillator is synchronised, its output jumps to this value times A. Typ. Range: 0...1
- **Out:** Audio output for the ramp signal. Range 0...A.



Free running clock source. An internal oscillator (monophonic) produces regular clock on/off pulses in the form of events, e.g. for driving a sequencer. The value of the on-events can be set in the module properties.

- **F:** Event input for control of clock frequency in Hz. For control of Tempo in BPM, compute the value in Hz as follows:  $\text{clocks-per-beat} \times \text{BPM}/60$ . So, for 16<sup>th</sup> note clocks at 4 beats to the bar (that is four 16ths per beat), you get  $F = 4/60 \times \text{BPM}$  or  $0.0667 \times \text{BPM}$ .
- **W:** Event input for control of pulse width, i.e. the ratio of the duration of the on-period to the off-period. Range of values is -1 to 1. Symmetric waveform (equal duration on and off) at  $W = 0$ ; Lo:Hi =  $(1 + W) / (1 - W)$ .
- **Snc:** Event input for synchronization of the waveform. A positive event synchronizes the clock source.
- **A:** Input for controlling the amplitude of the clock signal. You must connect something here, otherwise only zero-value events will be produced. Typ. Value:1.
- **Out:** Event output for the clock signal, alternating between on-value and zero.



Noise generator. Produces white noise, i.e. a random signal containing equal amounts of all possible frequency components. The signal consists of only two values,  $A/2$  and  $-A/2$ , but which appear in a random sequence.

- **A:** Audio input for controlling the amplitude of the output signal.
- **Out:** Audio signal output for the noise waveform.



Random value generator. Produces step waveform where the level of each step is random in the given interval with equal distribution. It works like a noise generator with uniform distribution followed by a sample & hold circuit clocked at a regular frequency.

- **P:** Logarithmic event input for controlling the step rate (sample & hold frequency). Value in semitones (69 = 440 Hz).
- **A:** Audio input for controlling the amplitude. The output signal is a random value between **+A** and **-A**.
- **Out:** Audio signal output for the random step waveform.



Generates events at random intervals, much like a Geiger Counter radiation particle detector. The average rate of events can be controlled at the **P** input. **Rnd** controls the randomness of the event timing.

- **P:** Control input for logarithmic control of the average rate or density of the randomly occurring output events. Typ. range: [ -50 ... 50 ]
- **Rnd:** Control input for the randomness of the event distribution: 0 = completely random, 1 = completely regular. Typ. range: [ 0 ... 1 ]
- **Out:** Audio output for randomly timed clicks.
- **Out:** Output for the randomly timed events. Use to trigger an envelope (**G**), for example.

# Samplers

If it generates audio and it's not an oscillator, it's a sampler. REAKTOR's samplers include basic sample players as well as sophisticated sample processors for granular synthesis, pitch and time shifting, and beat slicing. There's even one for picking out individual samples by number.

Depending on the selected Sampler module the following settings are available in the Properties.

## Properties - Function page

- **Waveform-Button:** A click on the button with the waveform icon opens the Sample Map Editor.
- **Embed Samples In Ensemble** allows to store your samples with the saved Ensemble.
- **No Stereo** stops stereo playback (Only the left channel is used). Activation reduces the processor load.
- The **Quality** drop-down menu sets the playback quality of the sample in three options (**Poor**, **Good** and **Excellent**). Higher quality increases the processor load. The term "quality" refers to the absence of noise. Naturally, noise may actually improve the musical "quality" of a sample.
- **Oscil. Mode** places the module into oscillator mode.

Samplers in oscillator mode assume that the samples used contain waveforms or WaveSets. A waveform is a representation of a single vibration. Through repeated playback it is possible to recreate periodic vibration. The module then becomes a digital oscillator. Sampler modules in oscillator mode interpret the values at the **P** input in the same way as oscillators in REAKTOR – as MIDI note numbers – and produce periodical vibrations at the corresponding pitch.

In oscillator mode the **Sampler** and **Sampler FM** modules interpret the entire sample as one vibration. For example to produce a sound at 440 Hz, the entire sample is played 440 times per second, independent of the duration of the sample.

In waveform mode **Sampler Loop** interprets the loop length as vibrations. The loop length is read from the sound file, but can be changed at the **LL** input of the module (see the Module Reference). Therefore a sample can contain numerous waveforms, also known as WaveSets. Assuming a waveform contains 100 cycles, then 100 such waveforms fit into a sample the size of 10,000 cycles. The first waveform covers cycles 0-99, the second 100-199 etc. If



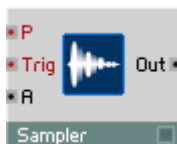
the loop length sets the vibration of a waveform, the position of the loop logically sets the choice of waveforms from the WaveSet. **Sampler Loop** uses the **LS** input to control the loop start point. In waveform mode the values at this input are quantized, so that glitch free playback between waveforms is achieved. The position in a WaveSet can be easily controlled by velocity, etc. Obviously such WaveSet samples need to be either created or generated from a sample. The library contains many WaveSet examples. **Sampler Loop** integrates WaveSet Synthesis with REAKTOR's existing synthesis capabilities. In this way, WaveSet Synthesis is now available for the first time in combination with FM synthesis.

## Properties - Appearance page

- **Picture:** Tick this checkbox to see a waveform display for the sampler module in the control panel.
- **Size X/Size Y:** Allows setting the size of the waveform display for the sampler module in pixel.
- **Scroll Bar:** Tick this checkbox to see a scroll bar under the waveform display for navigation purposes. The Scroll bar allows moving (drag the scroll bar in the middle) and zooming (drag the scroll bar at one of its ends to the left or right).

## Sampler

## Sampler



This is a player used for the polyphonic and transposed playback of a sample or sample map.

Sample management is carried out in the **Sample Map Editor**.

If **Loop** is activated, the whole sample is repeated continuously and is re-started by a positive event at the trigger input. If **Loop** is deactivated and a positive trigger event arrives at the trigger input, the sample will run from start to finish or, if the direction parameter is set to **Backward**, will run from the end to the beginning.

If **Oscillator Mode** is active, the playback rate will be adjusted to suit the length of the current sample in order to produce the correct pitch when operating as a waveform oscillator.

- **P:** Logarithmic control input for the playback rate (pitch) and for selecting a sample from the loaded sample map. If **P** is equal to the **Root Key** of the current sample, the sample will be played back at its original pitch.
- **Trig:** A positive event at this input triggers the sample to be played back from the beginning again.
- **A:** Control input for the output amplitude.
- **Out:** The sample player's output.

## Sampler FM

## Sampler



This is a player used for the polyphonic and transposed playback of a sample or a sample map. The **F** input and the starting point control input (**St**) allow you to manipulate sample playback.

Sample management is carried out in the Sample **Map Editor**.

If **Loop** is activated, the whole sample is repeated continuously; a positive event at the trigger input will make playback jump back to the starting point that has been set via the **St** input. If **Loop** is deactivated and a positive trigger event arrives at the trigger input, the sample will run from the starting point to the end or, if the direction parameter is set to **Backward**, will run from the starting point to the beginning.

If **Oscillator Mode** is active, the playback rate will be adjusted to suit the length of the current sample in order to produce the correct pitch when operating as a waveform oscillator.

- **P:** Logarithmic control input for the playback rate (pitch) and for selecting a sample from the loaded sample map. If **P** is equal to the **Root Key** of the current sample, the sample will be played back at its original pitch.

- **F:** Linear control input for modulating the playback rate. The effect achieved is frequency modulation – the same as for the oscillator modules in REAKTOR. If a large negative value is present, the sample is played backwards.
- **St:** Control input for the starting point to be used when the next trigger occurs. The position is set in milliseconds from the start of the sample.
- **Trig:** A positive event at this input triggers the sample to be played from the beginning again.
- **A:** Control input for the output amplitude.
- **Out:** The sample player's output.
- **Lng:** Polyphonic event output for the length of the current sample in milliseconds.

## Sampler Loop

## Sampler



This is a universal player for the polyphonic and transposed playback of mono or stereo samples, sample maps and WaveSets.

Sample management is carried out in the **Sample Map Editor**.

After a positive **Gate** event, sample playback starts from the starting point (configurable via the **St** input). If **Loop** is deactivated, the sample will run once from the starting point to the end or, if the direction parameter is set to **Backward**, will run from the starting point to the beginning. If **Loop** is activated, the sample will be continuously repeated within the loop range as soon as playback enters this range. The loop range is preset using data from the sound file from which the sample was loaded. If the sound file does not contain any loop data, the beginning of the sample is taken to be the begin-

ning of the loop and the sample length is taken as the loop length. The loop data from the sound file are the default settings. These defaults are always used if the Loop Start (**LS**) or Loop Length (**LL**) inputs are not connected to other modules.

If the **Loop in Release** option is deactivated, loop playback will fade or fade out completely upon a **Gate** event occurring; depending on the direction set, the sample will run to its beginning or end and will then fade out. If the **Loop in Release** option is activated or if loop playback is switched off, the **Gate** events will only have a slight effect or none at all.

If the **No Stereo** option is activated (you can set this in the properties dialog box), stereo samples will be treated as mono samples, i.e. the same signal is sent to the **L** and **R** outputs. If this is the case, **Sampler Loop** only uses the left channel of stereo samples. This option has been made available because mono playback requires less processing capacity.

- **G:** A positive event at this input starts output from the position that is set at the **St** input. If negative values are present at the input, loop playback is interrupted unless the **Loop in Release** option has been activated.
- **P:** Logarithmic control input for the playback rate (pitch). If **P** is equal to the **Root Key** of the current sample, the sample will be played back at its original pitch. Furthermore, if the **Sel** input is not connected, **P** determines the selection of samples from the sample map. In **Oscillator Mode**, the **Sampler Loop** functions as a digital oscillator. In the same way as in the oscillator modules in REAKTOR, **P** determines the fundamental pitch of the generated oscillation.
- **Sel:** Control input for selecting a sample from the sample map. If this input is not connected, the values present at the **P** input are used instead.
- **F:** Linear control input for modulating the playback rate. The effect achieved is frequency modulation – the same as for the oscillator modules in REAKTOR.
- **St:** Control input for the starting point to be used when the next trigger occurs. The position is set in milliseconds from the start of the sample.
- **LS:** Control input for the loop starting point in milliseconds from the start of the sample. If this input is not connected, the loop data stored in the sound file will be used. If no loop data are stored in the sound file, the default is used: **LS** = 0. Changes at this input take effect when samples are retriggered and when a loop limit is reached.

- **LL:** Control input for the loop length in milliseconds. If this input is not connected, the loop data stored in the sound file will be used. If no loop data are stored in the sound file, the default is used: **LL** = length of the sample. Changes at this input take effect when samples are retriggered and when a loop limit is reached.
- **A:** Control input for the output amplitude.
- **L:** Audio output for the left channel of the sample player. When mono samples are being processed or if the **No Stereo** option has been activated, the same signal is present here as at the **R** output.
- **R:** Audio output for the right channel of the sample player. When mono samples are being processed or if the **No Stereo** option has been activated, the same signal is present here as at the **L** output.
- **Lng:** Polyphonic event output for the length of the current sample in milliseconds.

## Grain Resynth



## Sampler

This is a real-time resynthesizer for the polyphonic, transposed playback of mono or stereo samples and sample maps. **Sample Resynth** allows you to control pitch and playback speed independently and in real-time, and also lets you extensively manipulate samples.

“Standard” samplers like the familiar hardware samplers or the **Sampler**, **Sampler FM** and **Sampler Loop** modules in REAKTOR all maintain a pointer for each voice. This pointer points to the current position in the sample. The amplitude at the output of the sampler is always the same as the amplitude of the sample at the pointer position. The pointer moves more quickly or

less quickly through the sample and therefore generates an amplitude at the outputs which varies with time – i.e. an oscillation.

The speed of the pointer movement determines the speed of the sample playback (e.g. the speed of a beat loop). At the same time it also determines the pitch that is heard: the more slowly the signal (which has been recorded in the sample) is sampled, the longer the periods of the oscillations occurring at the output are – the pitch therefore decreases. If the pointer stops moving, the output amplitude stops changing. No audible signal is produced.

In the same way as a conventional sampler, **Sample Resynth** uses a pointer at the current sample position. However, the signal at the outputs is not simply the sample amplitude occurring at the pointer position. What actually happens is that the output signal is generated by a synthesizer inside the module. This synthesizer *resynthesizes* the signal at the pointer position. The pitch of the signal generated by this synthesizer is independent of the pointer's speed. Even if the pointer is not moving at all, the synthesizer continues to produce a sound. While the pitch of the output signal is determined, as is usual, over the **P** input, the **Sp** input determines the pointer speed.

Of course, the slowed down or “frozen” signal does not always correspond to what you might have imagined it to be. What does a hammer hitting a nail sound like if its is slowed down to infinity? The resynthesis algorithm used by **Sample Resynth** is designed in such a way that a broad range of sounds can be processed in a (musically-speaking) sensible, subtle or drastic manner. The algorithm can be adjusted by configuring the **G** (Granularity) and **Sm** (Smoothness) parameters. This means that you can manually adjust it to suit the sample and to produce drastic, strange sound effects. In the properties dialog box you can activate the **Signal-Informed Granulation** option separately for each sample in a sample map. If this option is switched on, the resynthesis algorithm in **Resynth** takes information on the sample into account. This information was collected during the analysis that was carried out when the sample was loaded for the first time. What this means is that the algorithm reacts to the characteristics of the audio material. If this option is deactivated, the results produced are generally quite “electronic”.

Sample management is carried out in the **Sample Map Editor**.

After a positive **Gate** event, sample playback starts from the starting point (configurable via the **St** input). If **Loop** is deactivated, the sample will run once from the starting point to the end or, if the direction parameter is set to **Backward**, will run from the starting point to the beginning. If **Loop** is activated, the sample will be continuously repeated within the loop range as soon as playback enters this range. The loop range is preset using data from

the sound file from which the sample was loaded. If the sound file does not contain any loop data, the beginning of the sample is taken to be the beginning of the loop and the sample length is taken as the loop length. The loop data from the sound file are default settings. These defaults are always used if the Loop Start (**LS**) or Loop Length (**LL**) inputs are not connected to other modules.

If the **Loop in Release** option is deactivated, loop playback will fade or fade out completely upon a **Gate** event occurring. Depending on the direction set, the sample will run to its beginning or end and will then fade out. If the **Loop in Release** option is activated or if loop playback is switched off, the **Gate** events will only have a slight effect or none at all.

If the **No Stereo** option is activated (you can set this in the properties dialog box), stereo samples will be treated as mono samples, i.e. the same signal is sent to the **L** and **R** outputs. In this case, **Resynth** uses only the left channel of stereo samples. This option has been made available because mono playback requires less processing capacity.

All of the **Resynth's** inputs, except **Gate**, are designed as audio inputs. The values at these inputs are applied when samples are (re-triggered) (i.e. during positive **Gate** events). Changes to values during sample playback take effect in the **Granularity** interval (configured at the **Gr** input).

- **G:** A positive event at this input begins output from the position that is set at the **St** input. If negative values are present, loop playback is interrupted unless the **Loop in Release** option has been activated.
- **P:** Logarithmic audio control input for the playback rate (pitch). If **P** is equal to the **Root Key** of the current sample, the sample will be played back at its original pitch. Furthermore, if the **Sel** input is not connected, **P** also determines the selection of samples from the sample map.
- **Sel:** Audio control input for selecting a sample from the sample map. If this input is not connected, the values present at the **P** input are used instead.
- **St:** Audio control input for the starting point to be used when the next trigger occurs. The position is set in milliseconds from the start of the sample.
- **LS:** Audio control input for the loop starting point in milliseconds from the start of the sample. If this input is not connected, the loop data stored in the sound file will be used. If no loop data are stored in the sound file, the default is used: **LS** = 0.

- **LL:** Audio control input for the loop length in milliseconds. If this input is not connected, the loop data stored in the sound file will be used. If no loop data are stored in the sound file, the default is used: **LL** = length of the sample. If **LL** = 0, the movement within the sample stops when the loop starting point is reached; the sound is frozen at this point.
- **Sp:** Audio control input for pitch-independent output speed. Values that are present at the input are interpreted as a factor: When **Sp** = 1, playback occurs at the original speed; **Sp** = 2 corresponds to double the speed; and **Sp** = 0 means stop. If this input is not connected, the transposed original speed is taken as the default so that – as in conventional samplers – the speed reduces as pitch decreases.
- **Gr:** Audio control input for the granularity of the resynthesis process in milliseconds. This parameter determines the size of the sound particles used for resynthesis. If the **Signal-Informed Granulation** option is active, the values at the input will be used as the default; the values that are actually used are adjusted to suit the material.
- **SO:** Audio control input for modulation of the sample position (Sample Offset) in milliseconds. This input is used in order to modulate the position in the sample independent of pitch, e.g. via a noise generator.
- **Sm:** Audio control input for the *Smoothness* of the resynthesis process. The sound particles are adjusted using this. Small values generally lead to a rougher resulting sound.
- **Pan:** Audio control input for the position in the stereo field (-1 = Left, 0 = Center, 1 = Right).
- **A:** Audio control input for the output amplitude.
- **L:** Audio output for the left channel of the resynthesizer.
- **R:** Audio output for the right channel of the resynthesizer.
- **Lng:** Polyphonic event output for the length of the current sample in milliseconds.
- **Pos:** Polyphonic event output for the current position in the sample in milliseconds. Events are generated at time intervals corresponding to the set Granularity (**Gr**).





This is a real-time resynthesizer for the polyphonic playback of mono or stereo samples and sample maps or WaveSets. **Sample Pitch Former** is a WaveSet synthesizer in which not only WaveSets can be loaded but also any samples you like. **Sample Pitch Former** removes the pitch development from a sample and gives the sample any new pitch you wish. Besides independent real-time control over the playback speed, **Sample Pitch Former** also allows you to carry out a spectral transposition, i.e. a pitch-independent transposition of the timbre.

“Standard” samplers like the familiar hardware samplers or the **Sampler**, **Sampler FM** and **Sampler Loop** modules in REAKTOR all maintain a pointer for each voice. This pointer points to the current position in the sample. The amplitude at the output of the sampler is always the same as the amplitude of the sample at the pointer position. The pointer moves more quickly or less quickly through the sample and therefore generates an amplitude at the outputs which varies with time – i.e. an oscillation.

The speed of the pointer movement determines the speed of the sample playback. At the same time it also determines the pitch that is heard: the more slowly the signal (that has been recorded in the sample) is sampled, the longer are the periods of the oscillations occurring at the output – the pitch therefore decreases. If the pointer stops moving, the output amplitude stops changing. No audible signal is produced.

In the same way as a conventional sampler, **Sample Pitch Former** uses a pointer at the current sample position. However, the signal at the outputs is not simply the sample amplitude occurring at the pointer position. What actually happens

is that the output signal is generated by a synthesizer inside the module. This synthesizer resynthesizes the signal at the pointer position. The pitch of the signal generated by this synthesizer is independent of the pointer's speed. Even if the pointer is not moving at all, the synthesizer continues to produce a sound. While the pitch of the output signal is determined, as is usual, via the **P** input, the **Sp** input determines the pointer speed.

While conventional samplers and the **Sample Resynth** module in REAKTOR achieve a *relative* pitch change by *transposing* a sample, **Sample Pitch Former** forces any *absolute and definite pitch* that you wish onto a sample. **Sample Pitch Former** can therefore be used like a REAKTOR oscillator. Depending on the type of processed sound material and the settings used, the result can sound “electronically” altered to a greater or lesser degree. **Sample Pitch Former** will also generate signals with a certain pitch if the processed sample has no unique pitch of its own (e.g. recordings of cymbals or gongs). **Sample Pitch Former** produces fewer sound alterations the more restricted the fundamental pitch of the processed material is to be defined, and the less the original pitch deviates from the generated pitch.

In conventional samplers and in the **Sample Resynth** module in REAKTOR, the transposing of the fundamental pitch goes hand in hand with the transposing of *all* the spectral components. This is often considered a limitation since the transposition also affects certain spectral components which the listener would not expect to hear changed. The “Mickey Mouse effect” that occurs when speech recordings are detuned can be traced back to the transposition of the formants whose position for a “real” human speaker would be largely independent of the fundamental pitch. Within certain limits, **Sample Pitch Former** decouples pitch and formant position from one another. The formant position can be shifted independent of pitch via the formant shift input (**FS**). Since the human ear uses all the spectral components to identify the fundamental pitch, you can manipulate this parameter to create interesting substitutions of fundamental pitch and timbre, especially at very low pitches. Similar sound effects are often created by synthesizer experts using *oscillator synchronization*.

Sample management is carried out in the **Sample Map Editor**.

After a positive **Gate** event, sample playback starts from the starting point (configurable via the **St** input). If **Loop** is deactivated, the sample will run once from the starting point to the end or, if the direction parameter is set to **Backward**, will run from the starting point to the beginning. If **Loop** is activated, the sample will be continuously repeated within the loop range as soon as playback enters this range. The loop range is preset using data from

the sound file from which the sample was loaded. If the sound file does not contain any loop data, the beginning of the sample is taken to be the beginning of the loop and the sample length is taken as the loop length. The loop data from the sound file are default settings. These defaults are always used if the Loop Start (**LS**) or Loop Length (**LL**) inputs are not connected to other modules.

If the **Loop in Release** option is deactivated, loop playback will fade or fade out completely upon a **Gate** event occurring. Depending on the direction set, the sample will run to its beginning or end and will then fade out. If the **Loop in Release** option is activated or if loop playback is switched off, **Gate** events that are smaller than or equal to zero have no effect.

If the **No Stereo** option is activated (you can set this in the properties dialog box), stereo samples will be treated as mono samples, i.e. the same signal is sent to the **L** and **R** outputs. In this case, **Sample Pitch Former** uses only the left channel of stereo samples. This option has been made available because mono playback requires less processing capacity.

All of **Sample Pitch Former's** inputs, except **Gate**, are designed as audio inputs. The values at the inputs are applied when samples are (re-) triggered (i.e. during positive **Gate** events). During sample playback, changes to values take effect at intervals of the fundamental pitch period (you can set this via the **P** input).

- **G:** A positive event at this input starts output from the position that is set at the **St** input. If negative values are present, loop playback is interrupted unless the **Loop in Release** option has been activated.
- **P:** Logarithmic audio control input for the playback rate (pitch). Furthermore, if the **Sel** input is not connected, **P** also determines the selection of samples from the sample map.
- **Sel:** Audio control input for selecting a sample from the sample map. If this input is not connected, the values present at the **P** input are used instead.
- **St:** Audio control input for the starting point to be used when the next trigger occurs. The position is set in milliseconds from the start of the sample.
- **LS:** Audio control input for the loop starting point in milliseconds from the start of the sample. If this input is not connected, the loop data stored in the sound file will be used. If no loop data are stored in the sound file, the default is used: **LS** = 0.

- **LL:** Audio control input for the loop length in milliseconds. If this input is not connected, the loop data stored in the sound file will be used. If no loop data are stored in the sound file, the default is used: **LL** = length of the sample. If **LL** = 0, the movement within the sample stops when the loop starting point is reached; the sound is frozen at this point.
- **Sp:** Audio control input for pitch-independent output speed. Values that are present at the input are interpreted as a factor: when **Sp** = 1, playback occurs at the original speed; **Sp** = 2 corresponds to double the speed; and **Sp** = 0 means stop. If this input is not connected, the value 1 is taken as the default.
- **FS:** Audio control input for pitch-independent transposing of the formant position in semi-tones.
- **SO:** Audio control input for modulation of the sample position (Sample Offset) in milliseconds. This input is used to modulate the position in the sample independent of pitch, e.g. via a noise generator.
- **Sm:** Audio control input for the *Smoothness* of the resynthesis process. The sound particles are adjusted using this. Small values generally lead to a rougher resulting sound.
- **Pan:** Audio control input for the position in the stereo field (-1 = Left, 0 = Center, 1 = Right).
- **A:** Audio control input for the output amplitude.
- **L:** Audio output for the left channel of the resynthesizer.
- **R:** Audio output for the right channel of the resynthesizer.
- **Lng:** Polyphonic event output for the length of the current sample in milliseconds.
- **Pos:** Polyphonic event output for the current position in the sample in milliseconds. Events at this input are generated at time intervals corresponding to the current fundamental pitch period.



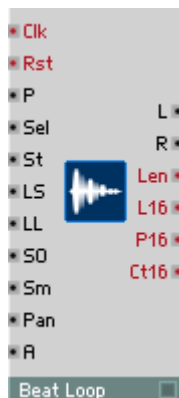
Stereo-multi-sample granular-synthesizer with independent control over pitch **P**, pitch-slide **PS**, sample selection **Sel**, sample-position **Pos** and length **Len** of each grain. The envelope of each grain can be controlled with attack **Att** and decay **Dec**.

For each grain the delta time to the start of the next grain can be set with **dt**. The maximum number of simultaneous grains can be set in the Properties. There are several jitter-inputs which set a range for the respective input.

Sample management is carried out in the **Sample Map Editor**.

- **Trig:** Event input for the Gate signal. (G) > 0 starts immediately next gain.
- **P:** Audio input for logarithmic pitch control (in semitones). The pitch is independent of the speed traversal. The sample plays at the original pitch when P=Root key. Typ. range: [0...127], Default:60.
- **D/F:** Audio input for direction control if P is connected. Otherwise it is a frequency control. The sample plays at the original pitch when F=1, in reverse direction when F = -1. Typ. range: [-4...4], Default: 1.
- **PJ:** Audio input for pitch jitter control (in semitones). Typ. range: [0...3].

- **PS:** Audio input for logarithmic pitch shift control of current grain in semitones. Typ. range: [-3...3].
- **Sel:** Audio input for multi-sample selection (in semitones). When unconnected, the values at the (P) input are used. Typ. range: [0...127].
- **Pos:** Audio input for setting the sound file position in ms. Range [0...<length of sample in ms>].
- **PsJ:** Audio input for sound file position jitter control in ms. Typ. range: [0...<length of sample in ms>].
- **Len:** Audio input for setting the grain length in ms. Typ range: [10...100]. Default: 20 ms.
- **LnJ:** Audio input for grain length jitter control in ms. Typ range: [10...100]. Default: 0 ms.
- **Att:** Audio input for setting the attack time. Range: [0...1]. Default: 0.2.
- **Dec:** Audio input for setting the decay time. Range: [0...1]. Default: 0.2.
- **Dist:** Audio input for setting the delta time before starting the next grain in ms. Typ range: [5...100]. Default: 20.
- **DisJ:** Audio input for delta time jitter control in ms. Typ range: [10...100]. Default: 20 ms.
- **Pan:** Audio input for setting the position in the stereo field. Range: [-1(Left)...1(Right)].
- **PnJ:** Audio input for pan jitter control. Range: [0...1].
- **A:** Audio input for amplitude control. Typ range: [0...1]. Default: 1.
- **L:** Polyphonic left channel audio output. Typ range: [-1...1].
- **R:** Polyphonic right channel audio output. Typ range: [-1...1].
- **Lng:** Event output for the length of samples in ms. Typ range: [0...<length of samples in ms>].
- **GTr:** Event output for grain trigger. Outputs 1 when a new grain starts, 0 when it stops.



This is a real-time resynthesizer for the synchronized playback of beat-loop samples. The transposing of beat-loops can be set via the **P** input independent of playback speed. **Beat Loop** synchronizes itself to a 96<sup>th</sup> note clock source (24 ppq) that is connected to the **C** input or, if the **C** input is not connected, it can sync with the global REAKTOR clock. Therefore, by default all **Beat Loops** in REAKTOR run in sync with one another independent of the sample's internal speed. Furthermore, **Beat Loop** also allows you to easily link internal REAKTOR sequencer modules and MIDI clocks to rhythmic sample material.

Sample management is carried out in the **Sample Map Editor**.

**Beat Loop** requires samples that have been cut exactly, and which contain 2, 4, 8, 16, or 32, etc. beats. The speed of the beat loops used should be between 87 and 174 BPM. If the samples being used fulfill these conditions, **Beat Loop** can output them in good quality even if the playback speed is changed. By activating the sample-related **Pitched Sound** option (accessible in the properties dialog box), possible pitch falsification of basslines (and similar) can be avoided. However, in doing so you will have to accept a deterioration in rhythmic precision.

The loop range of the sample is configured using the Loop Start (**LS**) and Loop Length (**LL**) inputs. If **LS** and **LL** are not connected, the whole sample will be played back as a loop. Using positive **Rst** events, sample playback will be continued from the starting point (you can set this via the **St** input).

If the **No Stereo** option is activated (you can set this in the properties dialog box), stereo samples will be treated as mono samples, i.e. the same signal is sent to the **L** and **R** outputs. In this case, **Beat Loop** uses only the left channel of stereo samples. This option has been made available because mono playback requires less processing capacity.

All of **Beat Loop**'s inputs, except **C** and **Rst**, are designed as audio inputs. During sample playback, changes to values take effect at intervals of sixteenths of a note value.

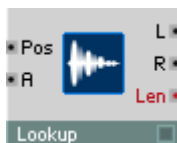
- **C:** A positive event at this input switches the **Beat Loop** module by one 96<sup>th</sup> note value further. If this input is not connected, the module synchronizes itself with the global REAKTOR clock.
- **Rst:** A positive event at this input resets playback to the position set at the **St** input.
- **P:** Logarithmic audio control input for the playback rate (pitch). Furthermore, if the **Sel** input is not connected, **P** determines the selection of samples from the sample map.
- **Sel:** Audio control input for selecting a sample from the sample map. If this input is not connected, the values present at the **P** input are used instead.
- **St:** Audio control input for the starting point to be used when the next trigger occurs. The position is set in sixteenths of a note value from the start of the sample.
- **LS:** Audio control input for the loop starting point in sixteenths of a note value from the start of the sample. If this input is not connected, the default is used: **LS** = 0.
- **LL:** Audio control input for the loop length in sixteenths of a note value. If this input is not connected, the default is used: **LL** = length of the sample.
- **SO:** Audio control input for modulation of the sample position (Sample Offset) in sixteenths of a note value. This input is used to reach steps in the sample which, for instance, can be controlled by a sequencer module.
- **Sm:** Audio control input for the *Smoothness* of the resynthesis process. The sound particles are adjusted using this. Very small values generally lead to a “cracking” sound every sixteenth interval.
- **Pan:** Audio control input for the position in the stereo field (-1 = Left, 0 = Center, 1 = Right).
- **A:** Audio control input for the output amplitude.
- **L:** Audio output for the left channel of the resynthesizer.
- **R:** Audio output for the right channel of the resynthesizer.
- **Len:** Polyphonic event output for the length of the current sample in milliseconds.



- **L16:** Polyphonic event output for the length of the current sample in sixteenths of a note value.
- **P16:** Polyphonic event output for the current position in the sample in sixteenths of a note value.
- **Ct16:** Polyphonic event output for counting the sixteenths of a note value that have passed since the start / reset.

## Sample Lookup

## Sampler



This module makes samples available as function value look-up tables. A position within the sample in milliseconds is set via the **Pos** input. The value of the sample at this point is sent to the outputs.

You can load sound files using the **Load Sound...** entry in the context menu.

Sample management is carried out in the **Sample Map Editor**.

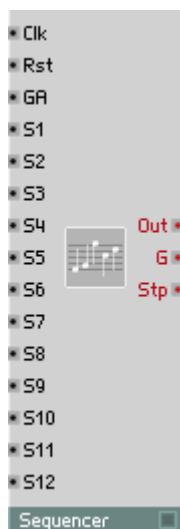
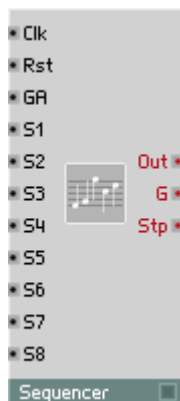
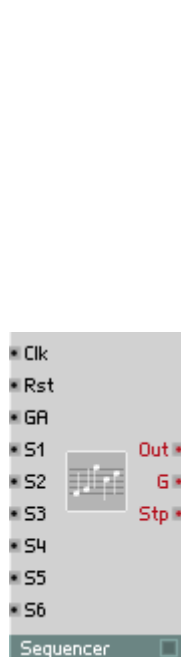
The properties dialog box allows you to select one of three levels of quality that is to be used for interpolation during transposed sample playback (**Poor**, **Good**, **Excellent**). If set to **Poor**, the sample values are not interpolated during output. Higher playback quality is achieved at the expense of processing capacity.

- **Pos:** Audio input for the position in the sample in milliseconds.
- **A:** Audio input for amplitude modulation.
- **L:** Audio output for the left channel of the sample. When processing mono samples, the same signal is output here as is sent to the **R** output.
- **R:** Audio output for the right channel of the sample. When processing mono samples, the same signal is output here as is sent to the **L** output.
- **Lng:** Polyphonic event output for the length of the current sample in milliseconds.

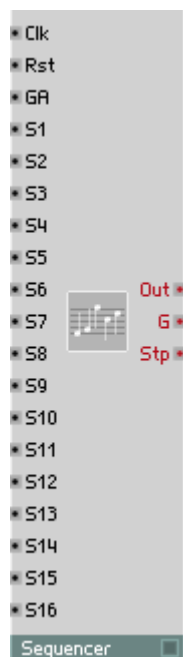
# Sequencer

REAKTOR's sequencers include gated step-sequencers in four sizes as well as a selectable-position sequencer. You'll also find clocks to drive them here.

## Sequencer



## Sequencer



## 6-Step

## Sequencer

Sequencer with 6 steps. The output value for each step (e.g. to control oscillator pitch) can be set independently. In addition a gate signal is output with the amplitude given by the current value of the gate amplitude input at the time the step advances.

- **C:** Audio input for clock control. A positive zero crossing switches to the next step. Typically you would connect a Pulse Oscillator or MIDI Sync module here.
- **Rst:** Audio input for a reset signal. A positive zero crossing puts the sequencer back to the first step. Typically you would connect a button or MIDI Start module here.

- **GA:** Audio input for controlling the amplitude of the gate output. When the value is zero or the input is not connected, no signal appears on the gate output.
- **S1:** Audio input for controlling the value of the first step.
- **S2:** Audio input for controlling the value of the second step.
- **S3:** Audio input for controlling the value of the third step.
- **S4:** Audio input for controlling the value of the fourth step.
- **S5:** Audio input for controlling the value of the fifth step.
- **S6:** Audio input for controlling the value of the sixth step.
- **Out:** Event output for the sequencer step signal.
- **G:** Event output for the gate signal.
- **St:** Event output for the current step number.

---

## 8-Step

## Sequencer

Like 6-Step Sequencer but with 8 steps.

---

## 12-Step

## Sequencer

Like 6-Step Sequencer but with 12 steps.

---

## 16-Step

## Sequencer

Like 6-Step Sequencer but with 16 steps.

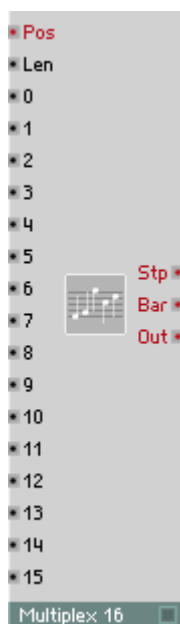
---

## Multiplex 16

## Sequencer

Value Selector, useful as a sequencer. An event at **Pos** addresses with its value one of the 16 inputs and the current value at this input is forwarded to **Out**.

If the **Pos** input is driven by a signal which is incremented stepwise, the output **Out** provides a sequence of the values at the inputs **0...15**. The values at the **Pos** input are folded into the number range [0 ... (Len - 1)] to allow a variable sequence length **Len**.



The **Pos** input can also be driven by a control element, a **Beat Loop** module, a random event source, or by the master clock.

- **Pos:** Input for controlling the selection. Each event at this input results in an event at the outputs. To run Select 16 as a sequencer, the position is set to the number of 16th notes that have passed since start or reset.
- **Len:** Input to set the sequence length. Range: [ 1 ... 16 ].
- **0-15:** Audio input for controlling the value of the appropriate step.
- **Stp:** Current sequence step number. The number at this output is calculated as **Pos** modulo **Len**. Range: [ 0 ... <sequence length> ].
- **Bar:** Current bar number. The number at this output is calculated as  $\text{Integer}(\text{Pos} / \text{Len})$ .
- **Out:** Current sequence step output value.

# LFO, Envelope

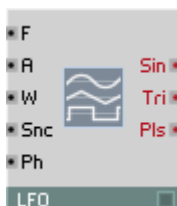
This category includes a fully modulatable, multiwaveform LFO, a random control generator, and envelope generators of just about any description including time/level ramp generators in three sizes.

All Envelopes can optionally display their curve in a panel graphic. This can be achieved with the **Visible** option on the **Appearance** page in the module Properties. The size of the display can be set in the Properties using the **Size X** and **Size Y** options. **The timeline of the curve in the display is not scaled.**

---

## LFO

## LFO, Envelope



Low Frequency Oscillator with Pulse, Triangle and Parabol Wave outputs. It is typically used as a source for modulation signals (for vibrato, tremolo etc.). The output signal is a stream of events at the **Control Rate** selected in the **Settings** menu.

Since the LFO operates at the Control Rate, it is much more CPU efficient than an audio oscillator which could do the same job.

- **F:** Audio input for control of the oscillator frequency in Hz. For control with Tempo in BPM, you can compute the required value in Hz as follows:  $F = \text{oscillations-per-beat} \times \text{BPM}/60$ . So, for example, for 3 oscillations per bar at 4 beats to the bar (that is  $\frac{3}{4}$  oscillations per beat), you get  $F = (\frac{3}{4})/60 \times \text{BPM}$  or  $0.0125 \times \text{BPM}$ .
- **A:** Input for controlling the amplitude. The output signal moves between  $+A$  and  $-A$ .
- **W:** Event input for control of pulse width, i.e. the ratio of the duration of the on-period to the off-period for the pulse output, and appropriate warping of the other waveforms. Range of values is -1 to 1. Symmetric waveforms at  $W = 0$ .
- **Snc:** Event input for synchronization of the LFO waveform. A positive event synchronizes the oscillator, resetting it to the phase given by the **Ph** input.

- **Ph:** Input for specifying the phase (position in the waveform) to which the oscillator is reset when synchronization occurs. **Ph** = 0: Phase = 0 deg (middle of rising slope), **Ph** = 0.5: Phase = 180 deg (middle of falling slope), **Ph** = 1: Phase = 360 deg (same as 0 deg).
- **Sin:** Event output for the sine waveshape signal.
- **Tri:** Event output for the triangle waveshape signal.
- **Pls:** Event output for the pulse waveshape signal.

---

## Slow Random

## LFO, Envelope



Low Frequency Oscillator which produces a random step waveform.

- **F:** Control input for the step frequency in Hz.
- **A:** Control input for the amplitude. The output value is somewhere in the range  $-A$  to  $+A$ .
- **Out:** Event output for the random signal.

---

## H - Env

## LFO, Envelope



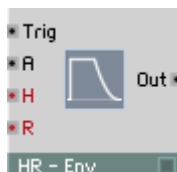
Envelope generator with hold characteristic. When the envelope is triggered the output value jumps to the current value of the amplitude input and stays there until the hold time has passed, after which the output jumps back to zero. The envelope can be triggered at any time, including retriggering during the hold time.

- **T:** Audio input for triggering the envelope on a rising edge (value goes from zero in positive direction).
- **A:** Audio input for the output value during the hold time. The input is sampled at the triggering instant after which the value is held at the output.

- **H:** Logarithmic event input for controlling the hold time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.

## HR - Env

## LFO, Envelope



Envelope generator with hold-release characteristic. When the envelope is triggered the output value jumps to the current value of the amplitude input and stays there until the hold time has passed, after which the output decays exponentially with the release time back to zero.

- **T:** Audio input for triggering the envelope on a rising edge (value goes from zero in positive direction).
- **A:** Audio input for the output value during the hold time. The input is sampled at the triggering instant after which the value is held at the output.
- **H:** Logarithmic event input for controlling the hold time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.



Envelope generator with decay characteristic. When the envelope is triggered the output value jumps to the current value of the amplitude input, after which the output decays exponentially with the decay time back to zero.

- **T:** Audio input for triggering the envelope on a rising edge (value goes from zero in positive direction).
- **A:** Audio input for the initial output value. The input is sampled at the triggering instant.
- **D:** Logarithmic event input for controlling the decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.



Envelope generator with decay-release characteristic. When the envelope is triggered with a gate event the output value jumps to the amplitude value of the gate signal, after which the output decays exponentially with the decay time back to zero. A gate event with amplitude zero (at note off) sets the decaying time to the release time value which is normally set to be shorter.

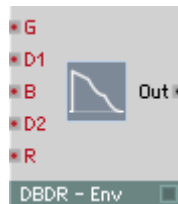
- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the initial output value.
- **D:** Logarithmic event input for controlling the decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.





Envelope generator with decay-sustain-release characteristic. When the envelope is triggered with a gate event the output value jumps to the amplitude value of the gate signal, after which the output decays exponentially with the decay time to the sustain level (multiplied by the amplitude). After a gate event with amplitude zero (at note off) the output decays exponentially with the release time back to zero.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the initial output value.
- **D:** Logarithmic event input for controlling the decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **S:** Event input for controlling the sustain level. Typical range of values is: 0 (decay to zero) to 1 (hold at initial level), but sustain can be greater than 1 or even less than 0.
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.



Envelope generator with decay-breakpoint-release characteristic. When the envelope is triggered with a gate event the output value jumps to the amplitude value of the gate signal, after which the output decays exponentially with the decay-1 time until it reaches the breakpoint level (multiplied by the amplitude). Next it continues decaying exponentially with the decay-2 time back to zero.

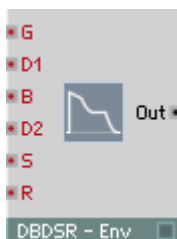
A gate event with amplitude zero (at note off) sets the decaying time to the release time value which is normally set to be shorter.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the initial output value.
- **D1:** Logarithmic event input for controlling the first decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **B:** Event input for controlling the breakpoint level. Range of values: 0 (never use decay-2 time) to 1 (immediately use decay-2 time).
- **D2:** Logarithmic event input for controlling the second decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.

---

## DBDSR-Env

## LFO, Envelope

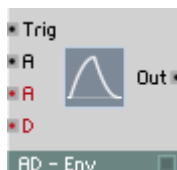


Envelope generator with decay-breakpoint-decay-sustain-release characteristic. When the envelope is triggered with a gate event the output value jumps to the amplitude value of the gate signal, after which the output decays exponentially with the decay-1 time until it reaches the breakpoint level (multiplied by the amplitude). Next it continues decaying with the decay-2 time to the sustain level (multiplied by the amplitude). The output is held at the sustain level until a gate event with amplitude zero (at note off) arrives, after which the output decays exponentially with the release time back to zero.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the initial output value.
- **D1:** Logarithmic event input for controlling the first decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **B:** Event input for controlling the breakpoint level. Range of values: 0 (never use decay-2 time) to 1 (immediately use decay-2 time).

- **D2:** Logarithmic event input for controlling the second decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **S:** Event input for controlling the sustain level. Typical range of values is: 0 (decay to zero) to 1 (hold at end of attack), but sustain can be greater than 1 or even less than 0.
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.

## AD - EnvLFO, Envelope



Attack-Decay envelope, triggered by the positive slope of the T signal. The decay starts immediately, when the attack time is over.

- **T:** Input for the trigger signal. A positive slope starts the envelope.
- **A:** Control input for the output amplitude.
- **A:** Control input for the attack time of the envelope. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **D:** Control input for the decay time of the envelope. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Output for the envelope signal.

## AR - Env

## LFO, Envelope



Envelope generator with attack-release characteristic. When the envelope is triggered with a gate event the output value rises during the attack time with a linear slope to the amplitude value of the gate signal. The output is then held at the maximum level until a gate event with amplitude zero (at note off) arrives, after which the output decays exponentially with the release time back to zero.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the maximum output value.
- **A:** Logarithmic event input for controlling the attack time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{lms}}$ ).
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{lms}}$ ).
- **Out:** Audio output for the envelope signal.

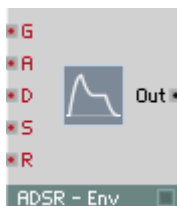
## ADR-Env

## LFO, Envelope



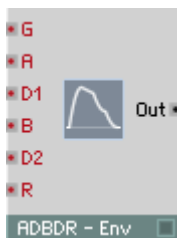
Envelope generator with attack-decay-release characteristic. When the envelope is triggered with a gate event the output value rises during the attack time with a linear slope to the amplitude value of the gate signal, after which it decays with the decay time to zero. When a gate event with amplitude zero (at note off) arrives, the output decays exponentially with the release time back to zero.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the maximum output value.
- **A:** Logarithmic event input for controlling the attack time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{lms}}$ ).
- **D:** Logarithmic event input for controlling the decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{lms}}$ ).
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{lms}}$ ).
- **Out:** Audio output for the envelope signal.



Envelope generator with the classic attack-decay-sustain-release characteristic. When the envelope is triggered with a gate event the output value rises during the attack time with a linear slope to the amplitude value of the gate signal, after which it decays exponentially with the decay time to the sustain level (multiplied by the amplitude). The output is held at the sustain level until a gate event with amplitude zero (at note off) arrives, after which the output decays exponentially with the release time back to zero.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the maximum output value.
- **A:** Logarithmic event input for controlling the attack time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **D:** Logarithmic event input for controlling the decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **S:** Event input for controlling the sustain level. Typical range of values is: 0 (decay to zero) to 1 (hold at end of attack), but sustain can be greater than 1 or even less than 0.
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.

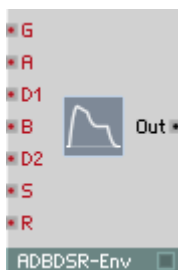


Envelope generator with attack-decay-breakpoint-decay-release characteristic. When the envelope is triggered with a gate event the output value rises during the attack time with a linear slope to the amplitude value of the gate signal, after which it decays exponentially with the decay-1 time until it reaches the breakpoint level (multiplied by the amplitude). Next it continues decaying exponentially with the decay-2 time back to zero. A gate event with amplitude zero (at note off) sets the decaying time to the release time value which is normally set to be shorter.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the maximum output value.
- **A:** Logarithmic event input for controlling the attack time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **D1:** Logarithmic event input for controlling the first decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **B:** Event input for controlling the breakpoint level. Range of values: 0 (never use decay-2 time) to 1 (immediately use decay-2 time).
- **D2:** Logarithmic event input for controlling the second decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.

## ADBD SR-Env

## LFO, Envelope



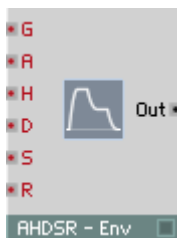
Envelope generator with attack-decay-breakpoint-decay-sustain-release characteristic. When the envelope is triggered with a gate event the output value rises during the attack time with a linear slope to the amplitude value of the gate signal, after which it decays according to the first decay time with a linear slope to the breakpoint level. From there it continues with the second

decay time to the sustain level (the levels are multiplied by the amplitude). The output is held at the sustain level until a gate event with amplitude zero (at note off) arrives, after which the output decays exponentially with the release time back to zero.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the maximum output value.
- **A:** Logarithmic event input for controlling the attack time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **D1:** Logarithmic event input for controlling the first decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **B:** Event input for controlling the break point level. Typical range of values is: 0 to 1.
- **D2:** Logarithmic event input for controlling the second decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **S:** Event input for controlling the sustain level. Typical range of values is: 0 to 1.
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.

## AHDSR - Env

## LFO, Envelope

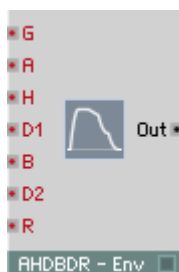


Envelope generator with attack-hold-decay-sustain-release characteristic. When the envelope is triggered with a gate event the output value rises during the attack time with a linear slope to the amplitude value of the gate signal and stays there until the hold time has passed, after which it decays according to the decay time with a linear slope to the sustain level. The output is held at the sustain level until a gate event with amplitude zero (at note off) arrives, after which the output decays exponentially with the release time back to zero.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the maximum output value.
- **A:** Logarithmic event input for controlling the attack time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).
- **H:** Logarithmic event input for controlling the hold time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).
- **D:** Logarithmic event input for controlling the decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).
- **S:** Event input for controlling the sustain level. Typical range of values is: 0 (decay to zero) to 1 (hold at end of attack), but sustain can be greater than 1 or even less than 0.
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).
- **Out:** Audio output for the envelope signal.

## AHDBDR - Env

## LFO, Envelope



Envelope generator with attack-hold-decay-breakpoint-decay-release characteristic. When the envelope is triggered with a gate event the output value rises during the attack time with a linear slope to the amplitude value of the gate signal and stays there until the hold time has passed, after which it decays exponentially with the decay-1 time until it reaches the breakpoint level (multiplied by the amplitude). Next it continues decaying exponentially with the decay-2 time back to zero. A gate event with amplitude zero (at note off) sets the decaying time to the release time value which is normally set to be shorter.

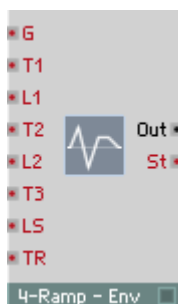
- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal determines the maximum output value.



- **A:** Logarithmic event input for controlling the attack time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **H:** Logarithmic event input for controlling the hold time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **D1:** Logarithmic event input for controlling the first decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **B:** Event input for controlling the breakpoint level. Range of values: 0 (never use decay-2 time) to 1 (immediately use decay-2 time).
- **D2:** Logarithmic event input for controlling the second decay time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **R:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **Out:** Audio output for the envelope signal.

---

## 4-Ramp LFO, Envelope



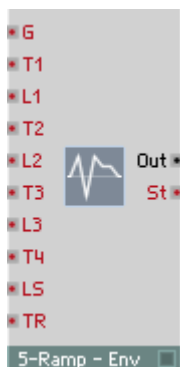
4-stage envelope generator with linear slopes. When the envelope is triggered with a gate event the output value rises to the first level, reaching it within the time set for the first stage. Then it continues to the second level within the time set for the second stage, and so on. The third level is the sustain level, at which the output is held until a gate event with amplitude zero (at note off) arrives, after which the output returns to zero within the last time-period.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal combines with all the level values to determine the actual levels reached.
- **T1:** Logarithmic event input for controlling the time for the first stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).

- **L1:** Input for controlling the level which is reached at the end of the first stage.
- **T2:** Logarithmic event input for controlling the time for the second stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **L2:** Input for controlling the level which is reached at the end of the second stage.
- **T3:** Logarithmic event input for controlling the time for the third stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **LS:** Input for controlling the level which is reached at the end of the third stage. This is the sustain level.
- **TR:** Logarithmic event input for controlling the time for the last stage which is the release. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **St:** Event output for the envelope's current stage (1, 2 ...). After the end of the release stage and before a new trigger, the value is 0. With appropriate processing this value can be used to chain other envelopes, or for the envelope to retrigger itself.
- **Out:** Audio output for the envelope signal.

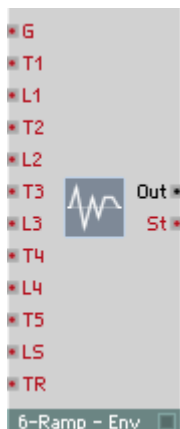
---

## 5-Ramp LFO, Envelope



5-stage envelope generator with linear slopes. When the envelope is triggered with a gate event the output value rises to the first level, reaching it within the time set for the first stage. Then it continues to the second level within the time set for the second stage, and so on. The fourth level is the sustain level, at which the output is held until a gate event with amplitude zero (at note off) arrives, after which the output returns to zero within the last time-period.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal combines with all the level values to determine the actual levels reached.
- **T1:** Logarithmic event input for controlling the time for the first stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **L1:** Input for controlling the level which is reached at the end of the first stage.
- **T2:** Logarithmic event input for controlling the time for the second stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **L2:** Input for controlling the level which is reached at the end of the second stage.
- **T3:** Logarithmic event input for controlling the time for the third stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **L3:** Input for controlling the level which is reached at the end of the third stage.
- **T4:** Logarithmic event input for controlling the time for the fourth stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **LS:** Input for controlling the level which is reached at the end of the fourth stage. This is the sustain level.
- **TR:** Logarithmic event input for controlling the time for the last stage, which is the release. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **St:** Event output for the envelope's current stage (1, 2 ...). After the end of the release stage and before a new trigger, the value is 0. With appropriate processing this value can be used to chain other envelopes, or for the envelope to retrigger itself.
- **Out:** Audio output for the envelope signal.



6-stage envelope generator with linear slopes. When the envelope is triggered with a gate event the output value rises to the first level, reaching it within the time set for the first stage. Then it continues to the second level within the time set for the second stage, and so on. The fifth level is the sustain level, at which the output is held until a gate event with amplitude zero (at note off) arrives, after which the output returns to zero within the last time-period.

- **G:** Event input for the gate signal that triggers the envelope. The amplitude of the gate signal combines with all the level values to determine the actual levels reached.
- **T1:** Logarithmic event input for controlling the time for the first stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).
- **L1 :** Input for controlling the level which is reached at the end of the first stage.
- **T2:** Logarithmic event input for controlling the time for the second stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).
- **L2:** Input for controlling the level which is reached at the end of the second stage.
- **T3:** Logarithmic event input for controlling the time for the third stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).
- **L3:** Input for controlling the level which is reached at the end of the third stage.
- **T4:** Logarithmic event input for controlling the time for the fourth stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{\text{1ms}}$ ).

- **L4:** Input for controlling the level which is reached at the end of the fourth stage.
- **T5:** Logarithmic event input for controlling the time for the fifth stage. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **LS:** Input for controlling the level which is reached at the end of the fifth stage. This is the sustain level.
- **TR:** Logarithmic event input for controlling the time for the last stage, which is the release. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **St:** Event output for the envelope's current stage (1, 2 ...). After the end of the release stage and before a new trigger, the value is 0. With appropriate processing this value can be used to chain other envelopes, or for the envelope to retrigger itself.
- **Out:** Audio output for the envelope signal.

# Filter

Filters make up REAKTOR's largest collection of signal processors. You'll find 22 of them here covering everything from standard low-, high-, and bandpass, to emulations of classic synth filters from Sequential and Moog. There's also an allpass filter for reverb and dispersion circuits as well as integrating and differentiating filters.

All of REAKTOR's filters can operate at any frequency, from 0 Hz (constant signal) through the entire audio range right up to the limit set by the sample rate. This means they are all equally suited for audio processing or for smoothing control signals (e.g. portamento). When using a filter to process the input to a port which is of the type that only accepts events (e.g. P as opposed to F) you need to insert an **A to E (perm)** module for conversion.

All the filters and EQs can optionally display their frequency response in a graph on the panel. This can be achieved with the **Visible option** on the **Appearance** page in the module Properties. The size of the display can be set in the Properties using the **Size X** and **Size Y options**. The graph's frequency axis is logarithmic and ranges from 10 Hz to 20 kHz.

---

## HP/LP 1-Pole

## Filter



1-pole filter with high pass and low pass outputs (6 dB/octave falloff) and logarithmic control of cutoff frequency.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **In:** Audio signal input for the signal to be filtered
- **HP:** Audio output for the high pass filtered signal
- **LP:** Audio output for the low pass filtered signal



1-pole filter with high pass and low pass outputs (6 dB/octave falloff), logarithmic and linear control of cutoff frequency.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **In:** Audio signal input for the signal to be filtered
- **HP:** Audio output for the high pass filtered signal
- **LP:** Audio output for the low pass filtered signal

## Allpass 1-Pole

## Filter



First order allpass filter. This type of filter has a flat amplitude response, but the phase shift between input and output grows from 0 degrees at low frequencies to -180 degrees at high frequencies. At the corner frequency controlled by the P input the phase is shifted for -90 degrees.

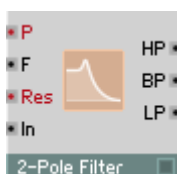
- **P:** Logarithmic control input for the corner frequency, where the phase shift is -90 degrees. Scale: 1 Semitone per unit, 43 = G1 = 98 Hz, 84 = C5 = 1047 Hz.
- **In:** Input for the signal to be allpass filtered (phase-shifted).
- **Out:** Output for the allpass filtered (phased-shifted signal).



2-pole filter with high pass and low pass outputs (12 dB/octave falloff), band pass (above and below each 6 dB/octave falloff), variable resonance, and logarithmic control of cutoff frequency.

The pass band gain is always 1 (0 dB), whereas the gain at the cutoff frequency increases with the resonance. Caution: Very large amplitudes are generated when **Res** is almost 1.

- **P**: Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **Res**: Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!). At high resonance the filter's Q-factor = frequency [Hz] / bandwidth [Hz]  $\cong 1 / (2 - 2 \text{ Res})$ .
- **In**: Audio signal input for the signal to be filtered
- **HP**: Audio output for the high pass filtered signal
- **BP**: Audio output for the band pass filtered signal
- **LP**: Audio output for the low pass filtered signal



2-pole filter with high pass and low pass outputs (12 dB/octave falloff), band pass (above and below each 6 dB/octave falloff), variable resonance, logarithmic and linear control of cutoff frequency.

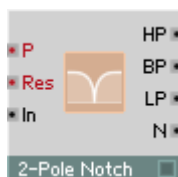
The pass band gain is always 1 (0 dB), whereas the gain at the cutoff frequency increases with the resonance. Caution: Very large amplitudes are generated when **Res** is almost 1.



- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!). At high resonance the filter's Q-factor = frequency [Hz] / bandwidth [Hz]  $\cong 1 / (2 - 2 \text{ Res})$ .
- **In:** Audio signal input for the signal to be filtered
- **HP:** Audio output for the high pass filtered signal
- **BP:** Audio output for the band pass filtered signal
- **LP:** Audio output for the low pass filtered signal

## Multi/Notch 2-Pole

## Filter



2-pole filter with band reject (notch) output, high pass and low pass outputs (12 dB/octave falloff), band pass (above and below each 6 dB/octave falloff), variable resonance and logarithmic control of cutoff frequency.

The gain at the cutoff frequency is always 1 (0 dB), whereas the pass band gain decreases with increasing resonance.

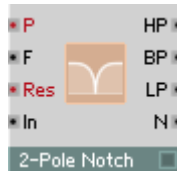
At the notch filter output the selected frequency is completely removed.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!). At high resonance the filter's Q-factor = frequency [Hz] / bandwidth [Hz]  $\cong 1 / (2 - 2 \text{ Res})$ .
- **In:** Audio signal input for the signal to be filtered
- **HP:** Audio output for the high pass filtered signal
- **BP:** Audio output for the band pass filtered signal

- **LP:** Audio output for the low pass filtered signal
- **N:** Audio output for the band reject (notch) filtered signal

## Multi/Notch 2-Pole FM

## Filter

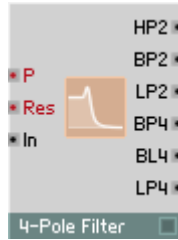


2-pole filter with band reject (notch) output, high pass and low pass outputs (12 dB/octave falloff), band pass (above and below each 6 dB/octave falloff), variable resonance, logarithmic and linear control of cutoff frequency.

The gain at the cutoff frequency is always 1 (0 dB), whereas the pass band gain decreases with increasing resonance.

At the notch filter output the selected frequency is completely removed.

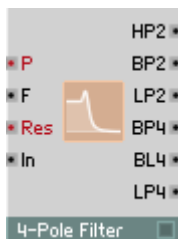
- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!). At high resonance the filter's Q-factor = frequency [Hz] / bandwidth [Hz]  $\cong 1 / (2 - 2 \text{ Res})$ .
- **In:** Audio signal input for the signal to be filtered
- **HP:** Audio output for the high pass filtered signal
- **BP:** Audio output for the band pass filtered signal
- **LP:** Audio output for the low pass filtered signal
- **N:** Audio output for the band reject (notch) filtered signal



4-pole filter with low pass (24 dB/oct falloff), band pass (12/12 dB/oct) and band/low pass (6/18 dB/oct) outputs, 2-pole high and low pass (12 dB/oct) and band pass (6/6 dB/oct) outputs, variable resonance and logarithmic control of cutoff frequency.

The pass band gain is always 1 (0 dB), whereas the gain at the cutoff frequency increases with the resonance. Caution: Very large amplitudes are generated when **Res** is almost 1.

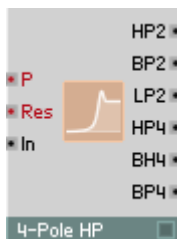
- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!).
- **In:** Audio signal input for the signal to be filtered
- **HP2:** Audio output for the 2-pole high pass filtered signal
- **BP2:** Audio output for the 2-pole band pass filtered signal
- **LP2:** Audio output for the 2-pole low pass filtered signal
- **BP4:** Audio output for the 4-pole band pass filtered signal
- **BL4:** Audio output for the 4-pole band/low pass filtered signal
- **LP4:** Audio output for the 4-pole low pass filtered signal



4-pole filter with low pass (24 dB/oct falloff), band pass (12/12 dB/oct) and band/low pass (6/18 dB/oct) outputs, 2-pole high and low pass (12 dB/oct) and band pass (6/6 dB/oct) outputs, variable resonance, logarithmic and linear control of cutoff frequency.

The pass band gain is always 1 (0 dB), whereas the gain at the cutoff frequency increases with the resonance. Caution: Very large amplitudes are generated when **Res** is almost 1.

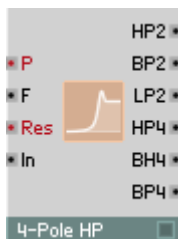
- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!).
- **In:** Audio signal input for the signal to be filtered
- **HP2:** Audio output for the 2-pole high pass filtered signal
- **BP2:** Audio output for the 2-pole band pass filtered signal
- **LP2:** Audio output for the 2-pole low pass filtered signal
- **BP4:** Audio output for the 4-pole band pass filtered signal
- **BL4:** Audio output for the 4-pole band/low pass filtered signal
- **LP4:** Audio output for the 4-pole low pass filtered signal



4-pole filter with high pass (24 dB/oct falloff), band pass (12/12 dB/oct) and band/high pass (18/6 dB/oct) outputs, 2-pole high and low pass (12 dB/oct) and band pass (6/6 dB/oct) outputs, variable resonance and logarithmic control of cutoff frequency.

The pass band gain is always 1 (0 dB), whereas the gain at the cutoff frequency increases with the resonance. Caution: Very large amplitudes are generated when **Res** is almost 1.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!).
- **In:** Audio signal input for the signal to be filtered
- **HP2:** Audio output for the 2-pole high pass filtered signal
- **BP2:** Audio output for the 2-pole band pass filtered signal
- **LP2:** Audio output for the 2-pole low pass filtered signal
- **HP4:** Audio output for the 4-pole high pass filtered signal
- **BH4:** Audio output for the 4-pole band/high pass filtered signal
- **BP4:** Audio output for the 4-pole band pass filtered signal



4-pole filter with high pass (24 dB/oct falloff), band pass (12/12 dB/oct) and band/high pass (18/6 dB/oct) outputs, 2-pole high and low pass (12 dB/oct) and band pass (6/6 dB/oct) outputs, variable resonance, logarithmic and linear control of cutoff frequency.

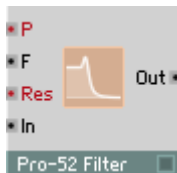
The pass band gain is always 1 (0 dB), whereas the gain at the cutoff frequency increases with the resonance. Caution: Very large amplitudes are generated when **Res** is almost 1.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self oscillation!).
- **In:** Audio signal input for the signal to be filtered
- **HP2:** Audio output for the 2-pole high pass filtered signal
- **BP2:** Audio output for the 2-pole band pass filtered signal
- **LP2:** Audio output for the 2-pole low pass filtered signal
- **HP4:** Audio output for the 4-pole high pass filtered signal
- **BH4:** Audio output for the 4-pole band/high pass filtered signal
- **BP4:** Audio output for the 4-pole band pass filtered signal

---

## Pro-52 Filter

## Filter



Filter taken from the Pro-52 virtual analog synthesizer. It is a 4-pole low pass filter (24 dB/oct falloff) with variable resonance and logarithmic and linear control of cutoff frequency.

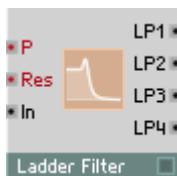
The filter goes into self-oscillation when **Res** approaches the value 1. The amplitude of the self-oscillation is approximately 1.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (maximal damping, no resonance) to 1 (no damping, maximal resonance, self-oscillation).
- **In:** Audio signal input for the signal to be filtered
- **Out:** Audio output for the 4-pole low pass filtered signal

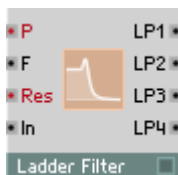
---

## Ladder Filter

## Filter



Same as **Ladder Filter FM** but without the frequency modulation input **F**. See next module description.



Filter modelled on the classic ladder circuit patented by Bob Moog. It is a 4-pole filter with different low pass outputs: 24 dB/oct, 18 dB/oct , 12 dB/oct and 6 dB/oct falloff. It also has variable resonance and logarithmic and linear control of cutoff frequency.

The saturation characteristic of the analog circuit can optionally be simulated by turning on **Distortion** in the properties. When Distortion is enabled, the filter goes into self-oscillation when the value of **Res** is 1 or more. The amplitude of the self-oscillation is approximately 1 when **Res** is just above 1, but can be much larger when driving Res even higher.

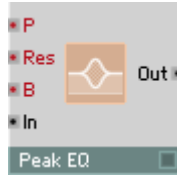
The filter also has options for selecting the quality of the simulation: **Standard**, **High** and **Excellent**. The effect is particularly noticeable when distortion is turned on. Of course, the higher quality settings come at the price of increased CPU usage.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440$  Hz).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **Res:** Event input for controlling the filter's resonance/damping. Range of values 0 (no resonance) to 1 (maximal resonance, self-oscillation). Values above 1 are possible in Distortion mode.
- **In:** Audio signal input for the signal to be filtered
- **LP1:** Audio output for the 6 dB/oct low pass filtered signal
- **LP2:** Audio output for the 12 dB/oct low pass filtered signal
- **LP3:** Audio output for the 18 dB/oct low pass filtered signal
- **LP4:** Audio output for the 24 dB/oct low pass filtered signal



---

## Peak EQ Filter



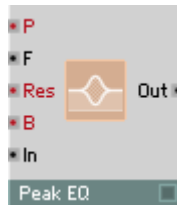
Parametric equalizer with adjustable boost/cut, band width and logarithmic frequency control. With the Peak EQ a specific frequency in the signal and a narrow or wide band of frequencies around it can be amplified or attenuated. More distant frequencies are not affected.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **Res:** Event input for controlling the filter's resonance (Q-factor). Range of values 0 (minimal resonance, maximal band width) to 1 (maximal resonance, minimal band width). At high resonance the filter's Q-factor = frequency [Hz] / bandwidth [Hz]  $\cong 1 / (2 - 2 \text{ Res})$ .
- **B:** Event input for controlling boost/cut in dB. At value **B** = 0 the signal remains unchanged.
- **In:** Audio signal input for the signal to be equalized
- **Out:** Audio output for the equalized signal

---

## Peak EQ FM

## Filter



Parametric equalizer with adjustable boost/cut, band width and logarithmic and linear frequency control. With the Peak EQ a specific frequency in the signal and a narrow or wide band of frequencies around it can be amplified or attenuated. More distant frequencies are not affected.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440 \text{ Hz}$ ).

- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **Res:** Event input for controlling the filter's resonance (Q-factor). Range of values 0 (minimal resonance, maximal band width) to 1 (maximal resonance, minimal band width). At high resonance the filter's Q-factor = frequency [Hz] / bandwidth [Hz]  $\cong 1 / (2 - 2 \text{ Res})$ .
- **B:** Event input for controlling boost/cut in dB. At value **B** = 0 the signal remains unchanged.
- **In:** Audio signal input for the signal to be equalized
- **Out:** Audio output for the equalized signal

## High Shelf EQ

## Filter



Parametric equalizer with high shelving characteristic, adjustable boost/cut, band width and logarithmic frequency control.

The level of frequencies above the corner frequency is raised or reduced by the set amount. Low frequencies are not affected.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones (69 = 440 Hz).
- **B:** Event input for controlling boost/cut in dB. At value **B** = 0 the signal remains unchanged.
- **In:** Audio signal input for the signal to be equalized
- **Out:** Audio output for the equalized signal

## High Shelf EQ FM

## Filter



Parametric equalizer with high shelving characteristic, adjustable boost/cut, band width and logarithmic and linear frequency control.

The level of frequencies above the corner frequency is raised or reduced by the set amount. Low frequencies are not affected.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **B:** Event input for controlling boost/cut in dB. At value **B** = 0 the signal remains unchanged.
- **In:** Audio signal input for the signal to be equalized
- **Out:** Audio output for the equalized signal

---

## Low Shelf EQ

## Filter



Parametric equalizer with low shelving characteristic, adjustable boost/cut, band width and logarithmic frequency control.

The level of frequencies below the corner frequency is raised or reduced by the set amount. High frequencies are not affected.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440 \text{ Hz}$ ).
- **B:** Event input for controlling boost/cut in dB. At value **B** = 0 the signal remains unchanged.
- **In:** Audio signal input for the signal to be equalized
- **Out:** Audio output for the equalized signal



Parametric equalizer with low shelving characteristic, adjustable boost/cut, band width and logarithmic and linear frequency control.

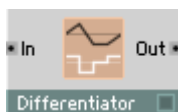
The level of frequencies below the corner frequency is raised or reduced by the set amount. High frequencies are not affected.

- **P:** Logarithmic event input for controlling the cutoff frequency. Value in semitones ( $69 = 440$  Hz).
- **F:** Audio input for linear control of the cutoff frequency. Value in Hz. **P** and **F** together determine the oscillator frequency.
- **B:** Event input for controlling boost/cut in dB. At value **B** = 0 the signal remains unchanged.
- **In:** Audio signal input for the signal to be equalized
- **Out:** Audio output for the equalized signal

---

## Differentiator

## Filter



The differentiator gives the slope of the input signal in change units per millisecond. The effect is like a high pass filter where the amplification is proportional to frequency. Unity gain at 159 Hz.

- **In:** Audio signal input for the signal to be differentiated
- **Out:** Audio output for the differentiated signal



Integrator with Reset input. The output value changes with a slope given by the input in change units per millisecond. The effect is like a low pass filter where amplification is proportional to  $1/\text{frequency}$ . Unity gain at 159 Hz.

- **In:** Audio signal input for the signal to be integrated
- **Set:** Event input for the reset. When an event is received, the output signal is set to the value of the event.
- **Out:** Audio output for the integrated signal

# Delay

REAKTOR provides six types of delay to accomplish most delay functions. Those include two tap delays, two granular delays, a diffuser (handy for re-verb circuits), and a unit delay useful in loop-back processing and physical modeling.

---

## Single Delay

## Delay



Polyphonic delay line for audio and event signals. The input signal appears at the output with a delay corresponding to the set time. The delay time is controlled at the **Dly** input.

The upper limit for the delay time can be adjusted in the module's properties dialog window in the **Max Delay Buffer** field (default: 1 second) and affects the memory usage. How big this value can be set depends on the amount of available RAM storage in the computer. At a sample rate of 44.1 kHz you need 172 kB of RAM for each second of buffer length and for each voice. For one minute you need 10 MB. If the Delay is used as an Event Delay (the **In**-port is red indicating that the module is in Event processing mode) you can set the **Max Count Of Buffered Events** at the same place.

This module replaces several modules of former REAKTOR version:

- It behaves like a REAKTOR 3 **Static Delay** if you connect an audio signal to the **In**-input and an event signal to the **Dly**-input. If the delay time does not correspond to an integer number of samples, the output signal is generated by interpolation. The interpolation method can be chosen in the properties. Linear interpolation may reduce high frequency components in the sound somewhat.
- It behaves like a REAKTOR 3 **Modulation Delay** if you connect an audio signal to the **In**-input and an audio signal to the **Dly**-input. The delay time can be continuously modulated by an audio signal. If the delay time does not correspond to an integer number of samples, the output signal is generated by interpolation. The interpolation method can be chosen in the properties. Linear interpolation may reduce high frequency components in the sound somewhat.

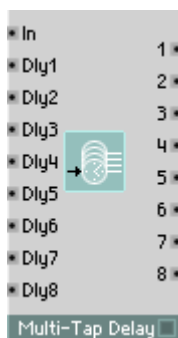
- It behaves like a REAKTOR 3 **Event Delay** if you connect an event signal to the In-input.

## Ports

- **Dly:** Hybrid input for controlling the delay time. Value in milliseconds.
- **In:** Hybrid input for the signal to be delayed.
- **Out:** Hybrid output for the delayed signal.

## Multi-Tap Delay

## Delay



Multi tap delay line for audio signals. If the set delay time corresponds to a non-integer number of sample, interpolation occurs. The interpolation method can be set in the properties.

The outputs are usually connected to a mixer or scanner.

- **In:** Audio input for the signal to be delayed.
- **Dly1...8:** Audio inputs for control of the delay time in milliseconds. Typ range: [0...1000].
- **1...8:** Audio outputs for the delayed input signal. **1** is delayed by time **Dly1**, **2** by **Dly2** etc.



The diffuser is an all-pass filter containing a delay line with feedback. Its function is to smear (decorrelate) the input signal without emphasizing any frequency components. Its typical use is as a building block for reverb type effects, where you would connect several of these modules in series and give them different delay times of a few milliseconds.

The delay time is controlled at the **Dly** input. The delay time can be continuously modulated by an audio signal. If the delay time does not correspond to an integer number of samples, the output signal is generated by interpolation. The interpolation method can be chosen in the properties. Linear interpolation may reduce high frequency components in the sound somewhat.

The amount of internal feedback is controlled at the **Dffs** input.

When the delay time is set to zero, the Diffuser functions as a 1-pole all pass filter. Like this you can construct a phaser, for example, by connecting several diffusers in series and modulating the **Dffs** parameter.

The upper limit for the delay time can be adjusted in the module's properties dialog window (default: 200 ms) and affects the memory usage.

- **Dly**: Hybrid input for controlling the delay time. Value in milliseconds.
- **Dffs**: Event input for controlling the diffusion coefficient. Range of values: -1 ... 1. **Dffs** = 0 : the module is a pure delay, **Dffs** = 1 : output = input, **Dffs** = -1 : output = -input. The most useful values for **Dffs** are near 0.5.
- **In**: Audio input for the signal to be diffused.
- **Out**: Audio output for the diffused signal.





A delay line and pitch-shifter for audio signals. The input signal appears at the outputs with a delay corresponding to the set time at the **Dly** input, transposed by the amount set at the **P** input in semitones.

The input signal is “chopped-up” into sound particles, whose size is controlled by the *Granularity* input (**Gr**). The “roughness” of the resulting sound can be controlled via the *Smoothness* input (**Sm**). The position of the sound particles in the stereo field is set at the **Pan** input.

The delay time of the **Grain Delay** can be varied without affecting the pitch. Interesting effects are possible in combination with random/noise generators.

In the properties dialog window the playback quality and also the upper limit of the delay time can be set. The available maximum delay time can deviate from the set amount by up to 50 %.

- **P**: Logarithmic audio control input for transposing in semitones (Pitch).
- **Dly**: Audio control input for the delay time in milliseconds.
- **Gr**: Audio control input for the granularity of the re-synthesis process, in milliseconds. This parameter sets the size of the sound particles used for the re-synthesis.
- **Sm**: Audio control input for the *Smoothness* of the re-synthesis process. This affects the formation of the sound particles. Low settings result generally in a rougher sound.
- **Pan**: Audio control input for stereo field positioning (-1 = Left, 0 = Middle, 1 = Right).
- **A**: Audio control input for the output amplitude.
- **In**: Input for the audio signal to be delayed.
- **L**: Audio output for the left channel of the delay.

- **R:** Audio output for the right channel of the delay.
- **Dly:** Polyphonic event output, which is set to the current delay time. This output always produces an event whenever a new sound particle is made.

## Grain Cloud Delay

## Delay



The Grain Cloud Delay module is similar to the Grain Cloud module (Samplers section) except that instead of operating on audio files loaded into its memory, it processes a constantly changing audio buffer that is fed by the module's bottom input port (labeled "In").

- **Trig:** Event input for triggering the next grain. Values >0 trigger the next grain. Values =-1 suppress the next grain (see the Dist input).
- **Frz:** Event input to freeze the audio buffer. Values >0 freeze the buffer.
- **P:** Audio input for logarithmic control of pitch-shift in semitones. The pitch is independent of the speed of traversal. Typical range -20 to 20.

- **D/F:** Audio input setting the direction (if the P input is wired) or linear frequency (if the P input is not wired) of grain playback. The audio buffer plays at the original pitch when  $F=1$ , in reverse direction when  $F=-1$ . Typical range -4 to 4. Default 1.
- **PJ:** Audio input for pitch jitter (in semitones). Typical range 0 to 3.
- **PS:** Audio input for logarithmic pitch shift of current grain in semitones. Typical range -3 to 3.
- **Dly:** Audio input for setting the delay time in ms. Allowed range 0 to buffer length.
- **DIJ:** Audio input for delay-time jitter in ms. Allowed range 0 to buffer length.
- **Len:** Audio input for the grain length in ms. Typical range 10 to 100. Default 30.
- **LnJ:** Audio input for grain-length jitter in ms. Typical range 10 to 100. Default 0.
- **Att:** Audio input for setting individual grain playback attack time. Range 0 to 1. Default 0.2.
- **Dec:** Audio input for setting individual grain playback decay time. Range 0 to 1. Default 0.2.
- **Dist:** Audio input for setting the delta-time between grains in ms. Grains are automatically triggered at this rate. Typical range 5 to 100. Default 20.
- **DisJ:** Audio input for delta-time jitter in ms. Typical range 5 to 100. Default 20.
- **Pan:** Audio input for setting the pan position in the stereo field. Range -1 to 1.
- **PnJ:** Audio input for setting the pan jitter. Range 0 to 1.
- **A:** Audio input for amplitude control. Typical range 0 to 1. Default 1.
- **In:** Audio input for the audio signal to be delayed.
- **L:** Polyphonic left channel audio output.
- **R:** Polyphonic right channel audio output.
- **Dly:** Polyphonic delay time output at every grain start.
- **Gtr:** Polyphonic Grain Trigger: 1 when a new grain starts, 0 when it stops.



Delays audio signal by one sample period ( $1/\text{sample rate}$ ). Every structure that uses some kind of feedback must have a unit delay in the loop. If no explicit unit delay is there, the program will insert an invisible unit delay somewhere in the loop.

- **In:** Input for the signal to be delayed by one sample period.
- **Out:** Output for the delayed signal.

# Audio Modifier

REAKTOR's audio processing modules provide various forms of distortion (shaping, clipping, and mirroring, for example). There are also slew limiter, peak detector, sample and hold, and frequency divider modules.

---

## Saturator

## Audio Modifier



Distortion with a smoothly rounded input-output curve for soft transition to saturation. The output value is limited to  $\pm 2$  (reached for input values bigger than  $\pm 4$ ). Very small input values are not changed.

- **In:** Audio input for signal to be saturated
- **Out:** Audio output for saturated signal

---

## Saturator 2

## Audio Modifier



Saturator 2 is a fourth-order asymmetric parabolic saturator and offers control over the saturation curve.

- **LA:** Level asymmetry. At  $LA = 0$  the saturation levels for positive and negative signals are equal. For  $LA > 0$  the positive level is reduced. At  $LA = 1$  it becomes zero. For  $LA < 0$  the negative level is reduced. At  $LA = -1$  it becomes zero.
- **KH:** Knee hardness. At  $KH = 0$  the saturation rises as soft as possible. The full range between zero and the saturation level is used for the rounded curve. With growing values of  $KH$  the curve range is reduced to  $(1 - KH)$  of the saturation level. At  $KH = 1$  the signal is clipped hard at the saturation level.

- **KHA:** Knee hardness asymmetry. At  $KHA = 0$  the knee hardness for positive and for negative signals is the same. For  $KHA > 0$  the knee hardness for positive signals is reduced. At  $KHA = 1$  it becomes zero. For  $KHA < 0$  the knee hardness for negative signals is reduced. At  $KHA = -1$  it becomes zero.
- **Offs:** This input adds an offset to the input signal and shifts it relative to the saturator curve. For a zero signal the offset is fully compensated at the output.
- **In:** Input for the signal to be distorted.
- **Out:** Output for the distorted signal.

---

## Clipper Audio Modifier



Distortion with a hard clipping and adjustable upper and lower limit. When the input signal exceeds the upper limit it is clipped to the limit value, similarly for the lower limit. Signal values between the limits are passed unchanged.

- **Max:** Audio input for controlling the upper limit of the signal
- **Min:** Audio input for controlling the lower limit of the signal
- **In:** Audio input for signal to be clipped
- **Out:** Audio output for clipped signal

---

## Mod. Clipper

## Audio Modifier



Distortion with a hard clipping and variable limit. When the absolute value of the input signal exceeds limit it is clipped to that value. Smaller signal values are passed unchanged.

- **M:** Audio input for controlling the limit on the absolute value of the signal
- **In:** Audio input for signal to be clipped
- **Out:** Audio output for clipped signal

---

## Mirror 1 Level

## Audio Modifier



Distortion by signal value mirroring with adjustable mirror level. Signal values above the mirror level are “reflected” at the level to end up smaller. Signal values below the mirror level are passed unchanged.

- **Max:** Audio input for controlling the mirror level
- **In:** Audio input for signal to be modified
- **Out:** Audio output for the modified signal

---

## Mirror 2 Levels

## Audio Modifier

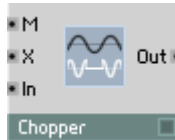


Distortion by double signal value mirroring with adjustable mirror levels. Signal values above the upper mirror level are “reflected” at the level to end up smaller, those below the lower mirror level are “reflected” at that level to end up larger. Signal values in the middle are passed unchanged.

- **Max:** Audio input for controlling the upper mirror level
- **Min:** Audio input for controlling the lower mirror level
- **In:** Audio input for signal to be modified
- **Out:** Audio output for the modified signal

## Chopper

## Audio Modifier



Chopper Modulator that switches amplification of the input signal between a variable factor and one.

When the modulation signal is positive, the input signal is multiplied by the value **X**. When the modulation signal is negative, the input is unchanged.

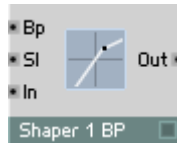
- **M:** Audio input for the modulation signal (only the sign is relevant).
- **X:** Audio input for controlling the amplification factor used when **M** is positive.
- **In:** Audio input for the signal to be chopped.
- **Out:** Audio output for the chopped signal



---

## Shaper 1 BP

## Audio Modifier



Signal shaper with piecewise linear input/output curve and one breakpoint.

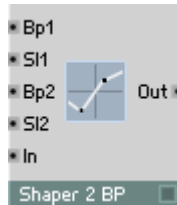
The slope of the curve above the breakpoint can be adjusted. Signal values below the breakpoint are passed unchanged.

- **Bp:** Audio input for controlling the level of the breakpoint
- **SI:** Audio input for controlling the slope of the upper part of the input/output curve (1 = no change in signal).
- **In:** Audio input for the signal to be shaped.
- **Out:** Audio output for the shaped signal.

---

## Shaper 2 BP

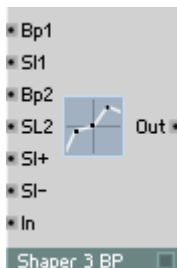
## Audio Modifier



Signal shaper with piecewise linear input/output curve and two breakpoint.

The slope of the curve above the upper and below the lower breakpoint can be adjusted. Signal values between the breakpoints are passed unchanged.

- **Bp1:** Audio input for controlling the level of the upper breakpoint
- **SI1:** Audio input for controlling the slope of the upper part of the input/output curve (1 = no change in signal).
- **Bp2:** Audio input for controlling the level of the lower breakpoint.
- **SI2:** Audio input for controlling the slope of the lower part of the input/output curve (1 = no change in signal).
- **In:** Audio input for the signal to be shaped.
- **Out:** Audio output for the shaped signal.



Signal shaper with piecewise linear input/output curve and three breakpoint.

The slope of the curve above the upper breakpoint, below the lower breakpoint and between the breakpoints and zero can be adjusted. Signal values between the breakpoints are passed unchanged.

- **Bp1:** Audio input for controlling the level of the upper breakpoint
- **Sl1:** Audio input for controlling the slope of the upper part of the input/output curve
- **Bp2:** Audio input for controlling the level of the lower breakpoint
- **SL2:** Audio input for controlling the slope of the lower part of the input/output curve (1 = no change in signal).
- **Sl+:** Audio input for controlling the slope of the input/output curve between zero and the upper breakpoint (1 = no change in signal).
- **Sl-:** Audio input for controlling the slope of the input/output curve between the lower breakpoint and zero (1 = no change in signal).
- **In:** Audio input for the signal to be shaped
- **Out:** Audio output for the shaped signal

---

## Shaper Parabolic

## Audio Modifier



Signal shaper with parabolic (2<sup>nd</sup> order polynomial) input/output curve.

The linear and square parts of the signal can be adjusted ( **Out** = **Lin In** + **Par In<sup>2</sup>** ).

- **Lin:** Audio input for controlling the level of the linear, undistorted part
- **Par:** Audio input for controlling the level of the square, distorted part
- **In:** Audio input for the signal to be shaped
- **Out:** Audio output for the shaped signal

---

## Shaper Cubic

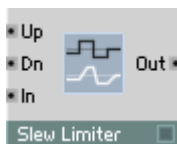
## Audio Modifier



Signal shaper with cubic parabolic (3<sup>rd</sup> order polynomial) input/output curve.

The linear, square and cubic parts of the signal can be adjusted ( **Out** = **Lin In** + **Par In<sup>2</sup>** + **Cub In<sup>3</sup>** ).

- **Lin:** Audio input for controlling the level of the linear, undistorted part
- **Par:** Audio input for controlling the level of the square, distorted part
- **Cub:** Audio input for controlling the level of the cubic, distorted part
- **In:** Audio input for the signal to be shaped
- **Out:** Audio output for the shaped signal



Slew rate limiter with separately adjustable maximum rate for rising and falling signals. The output signal tracks the input signal, but for fast movement and jumps at the input, the output follows with a limited rate of change (ramp) until its value reaches that of the input signal.

- **Up:** Audio input for controlling the maximum slew rate for rising signals. Value in units of 1/sec
- **Dn:** Audio input for controlling the maximum slew rate for falling signals. Value in units of 1/sec
- **In:** Audio input for signal to be slew rate limited
- **Out:** Audio output for slew rate limited signal



Detector for the peak amplitude. The input signal is rectified and smoothed with an adjustable release time. The attack time is zero.

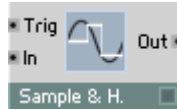
The result of the Peak Detector's action is that the output value follows the amplitude envelope of the input – use this module as an envelope follower.

- **Rel:** Logarithmic event input for controlling the release time. 0 = 1 ms, 20 = 10 ms, 40 = 100 ms (i.e. in  $\text{dB}_{1\text{ms}}$ ).
- **In:** Audio input for signal to be detected
- **Out:** Audio output for the signal

---

## Sample & Hold

## Audio Modifier



Sample & Hold with clock input.

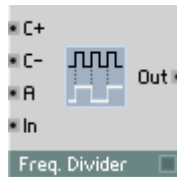
When the clock signal rises above zero, the current input value is passed to the output and held there until the next clock pulse. The result is a step waveform at the output.

- **C:** Audio input for the clock signal. The input is sampled on a rising edge here.
- **In:** Audio input for the signal to be sampled
- **Out:** Audio output for the sampled signal

---

## Frequency Divider

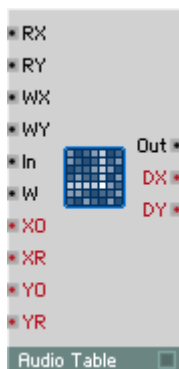
## Audio Modifier



Frequency divider (pulse sub-oscillator) with independently controllable high and low level duration.

A pulse wave is generated by counting the zero crossings of the input signal. The frequency of the output waveform will be a fraction of the frequency of the input waveform. The frequency ratio can be adjusted ( $f_{\text{out}} = 2 f_{\text{in}} / (C+ + C-)$ ). An asymmetric waveform results when **C+** and **C-** have different values.

- **C+:** Audio input for controlling the number of zero crossings of the input signal during the high phase of the output.
- **C-:** Audio input for controlling the number of zero crossings of the input signal during the low phase of the output.
- **A:** Audio input for controlling the amplitude. The output signal moves between **+A** and **-A**.
- **In:** Audio input for the signal to be frequency-divided
- **Out:** Audio output for the frequency divided signal.



Holds a table of data values. The table can be read out as an audio signal, audio can be stored in the table and the table's content can be displayed and edited graphically. The table can be 1-dimensional (a row of values) addressed by X, or 2-dimensional (a matrix of rows and columns, or a set of independent rows) addressed by X and Y.

The value at the output is taken from the table by reading at the position given by the inputs **RX** and **RY**. A signal at the module's **In**-port are stored in individual cells of the table according to the write position given by the inputs **WX** and **WY**.

X is the horizontal position from left to right and Y is the vertical position from top to bottom. The count always starts at 0 for the first element.

The module's panel display can show all the data or a limited region of it. Many options in the properties allow customizing the behaviour.

For full details on properties, menus and keyboard shortcuts, please see the section *Table Modules* on page 165.

- **RX**: Audio input for the X-position of a table cell from which the data is read.
- **RY**: Audio input for the Y-position of a table cell from which the data is read. This is used in 2D-mode or for addressing the row number if more than one row exists.
- **WX**: Audio input for the X-position of a table cell in which the data is written.
- **WY**: Audio input for the Y-position of a table cell in which the data is written.

- **W:** Audio input for activating table write operation.
- **In:** Audio input for the signal to be written into the table. When the value at **W** is bigger than 0, the value at **In** is written into the table at the position given by **WX** and **WY**.
- **XO:** Event input for the horizontal offset of the displayed data region. **XO** controls the data position that appears in the display (according to View Alignment). The value of **XO** is in the units specified in properties.
- **XR:** Event input for the horizontal range of the displayed data region. **XR** controls how many units of data fit in the display, i.e. it lets you zoom into the data.
- **YO:** Event input for the vertical offset of the displayed data region. **YO** controls the data position that appears in the display (according to View Alignment). The value of **YO** is in the units specified in properties.
- **YR:** Event input for the vertical range of the displayed data region in 2D-mode. **YR** controls how many units of data fit in the display, i.e. it lets you zoom into the data.
- **Out:** Audio output for signal read from the table at the position controlled by the **RX** and **RY** inputs.
- **DX:** Event output for the size of the horizontal table rows in units.
- **DY:** Event output for the size of the vertical table columns in units.

# Event Processing

Event modules are used for counting, for logical operations, for splitting and merging control signals, and for timing. You'll also find modules here for shaping and randomizing control signals. Finally there's a full-featured lookup table module-the control equivalent of the audio table in the oscillator section.

---

## Accumulator

## Event Processing

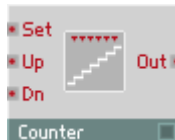


Accumulator (sum) for event values. The value of each event at the input is added to the total value stored in the module. The value of the updated sum is sent as an event at the output.

- **In:** Input for the events to be accumulated.
- **Set:** Event input for (re)setting the internal sum. The value of an event at this input determines the new value of the internal sum.
- **Out:** Event output for the accumulated total value.

---

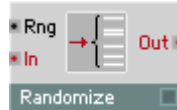
## Counter Event Processing



Counter controlled by events. A positive event at the appropriate input increases or decreases the output value by one.

- **Up:** Input for counting up. Only events with a positive, non-zero value have an effect here, increasing the output by one.
- **Dwn:** Input for counting down. Only events with a positive, non-zero value have an effect here, decreasing the output by one.
- **Set:** Event input for (re)setting the counter value. The value of an event at this input determines the new value of the counter.
- **Out:** Event output for the counter value.

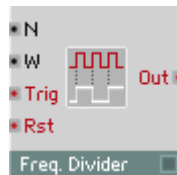




Event randomizer with adjustable spread.

The arriving events are modified with a random positive or negative offset in the given range (**Out** = **In** + X, where  $-\text{Rng} \leq X \leq \text{Rng}$ ).

- **Rng:** Audio input for controlling the range of the random signal modification
- **In:** Event input for signal to be randomized
- **Out:** Event output for randomized signal



The frequency of the output events is the frequency of the input events divided by a number controlled by the input **N**. The output signal returns to zero after a certain number of input events, depending on the pulse length which is set with the input **W**.

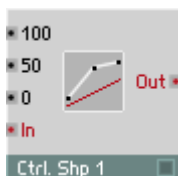
- **N:** Control input for the number of input events per output period. ( $N < 2$  : no division,  $2 \dots 2.99$  : division by 2,  $3 \dots 3.99$  : division by 3, etc.).
- **W:** Control input for the pulse width of the output signal. Range of values:  $0 \dots 1$  (0% ... 100%). **W** = 0 : the gate signal at **Out** returns to zero already at the next event at **In**, **W** = 0.5 : the gate signal at **Out** returns to zero after half the period, **W** = 1 : the gate signal at **Out** stays on all the time.
- **In:** Input for the event (clock) signal to be frequency-divided (e.g. MIDI Clock pulses). Only events with a positive, non-zero value have an effect here.

- **Rst:** Event input for resetting the internal counter in order to force a synchronous start (connect to MIDI Start signal if using the **Event Freq. Divider** for MIDI synchronization). Requires an event with positive non-zero value.
- **Out:** Event output for the frequency divided signal

---

## Ctrl. Shaper 1 BP

## Event Processing



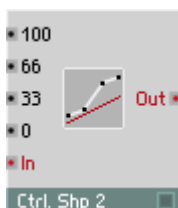
Event signal shaper with piecewise linear input/output curve and one fixed breakpoint. The output produced by linear interpolation between the given points.

- **100:** Output value at **In** = 1 (100%).
- **50:** Output value at the breakpoint at **In** = 0.5 (50%).
- **0:** Output value at **In** = 0 (0%).
- **In:** Event input for the signal to be shaped
- **Out:** Event output for the shaped signal

---

## Ctrl. Shaper 2 BP

## Event Processing



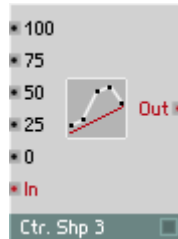
Event signal shaper with piecewise linear input/output curve and two fixed breakpoints. The output produced by linear interpolation between the given points.

- **100:** Output value at **In** = 1 (100%).
- **66:** Output value at the upper breakpoint at **In** = 0.66 (66%).
- **33:** Output value at the lower breakpoint at **In** = 0.33 (33%).

- **0:** Output value at **In** = 0 (0%).
- **In:** Event input for the signal to be shaped
- **Out:** Event output for the shaped signal

## Ctrl. Shaper 3 BP

## Event Processing



Event signal shaper with piecewise linear input/output curve and three fixed breakpoints. The output produced by linear interpolation between the given points.

- **100:** Output value at **In** = 1 (100%).
- **75:** Output value at the upper breakpoint at **In** = 0.75 (75%).
- **50:** Output value at the middle breakpoint at **In** = 0.5 (50%).
- **25:** Output value at the lower breakpoint at **In** = 0.25 (25%).
- **0:** Output value at **In** = 0 (0%).
- **In:** Event input for the signal to be shaped
- **Out:** Event output for the shaped signal

## Logic AND

## Event Processing



Logic gate for event signals. The output is the logic AND of the two input values, i.e. the output is 1 when both inputs are positive, otherwise the output is 0.

The inputs treat values of zero and below as the logic False state, values above zero are logic True.

---

## Logic OR

## Event Processing



Logic gate for event signals. The output is the logic OR of the two input values, i.e. the output is 1 when one or both inputs are positive, otherwise the output is 0.

The inputs treat values of zero and below as the logic False state, values above zero are logic True.

---

## Logic EXOR

## Event Processing



Logic gate for event signals. The output is the logic EXOR of the two input values, i.e. the output is 1 when one but not both inputs are positive, otherwise the output is 0. So in effect one input inverts the logic value of the other input.

The inputs treat values of zero and below as the logic False state, values above zero are logic True.

---

## Logic NOT

## Event Processing



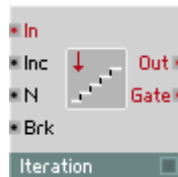
Logic gate for event signals. The output is the logic complement of the input value, i.e. the output is 0 when the input is positive, otherwise the output is 1.

The input treats values of zero and below as the logic False state, values above zero are logic True.



Event-Order. An event arriving at the input is transmitted with the same value on all the outputs, but in a well defined order: first it goes to **1**, then to **2** and finally to **3**. The event travels through ALL the modules in the chain connected to **1**, before going to the first module connected to **2** etc.

- **In:** Input for events to be re-transmitted in a defined order.
- **1:** An event is transmitted on this output first.
- **2:** An event is transmitted on this output second.
- **3:** An event is transmitted on this output third.



An event arriving at the **Trg** input is passed to the output and triggers a series of **N** additional output events. Each subsequent event has the value of its predecessor incremented by the value at the **Inc** input. All events are sent before the next audio sample is processed. This module can be used where an iterative event calculation is needed. It helps to avoid the construction of event loops which can lead to unstable operation of REAKTOR.

- **Trg:** Trigger input. An event at this input triggers N+1 events at the output.
- **Inc:** Increment for the value of each subsequent event.
- **N:** Number of additional output events. The value has to be greater than or equal to the integer number (decimal places will be cut).



All received events are compared to the threshold value and are sent to either one or the other output.

One use for the separator would be to convert a gate signal to a trigger signal by filtering out the zero events (**Thld** = 0, **Hi** = trigger output).

- **Thld:** Audio control input for the threshold value
- **In:** Input for the events to be separated and sent to the two outputs.
- **Hi:** Output for those events whose value is greater than the threshold level.
- **Lo:** Output for those events whose value is less than or equal to the threshold level.

---

## Value Event Processing



Value-changer for events. Events arriving at the input **In** have their value replaced with the current value at the **Val** input, and are then sent with the new value from the output.

This module can also be used as an event-controlled sample&hold module.

- **In:** Input for events to be changed in value.
- **Val:** Input for specifying the new value for the events.
- **Out:** Event output for the value-changed events.

---

## Merge Event Processing



Merger for event signals. When more than one wire is connected at the input, the output value is that of the last received input event, irrespective of which wire it came from.

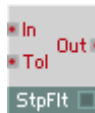
The module has a dynamic in-port management. The number of in-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

Unlike in previous REAKTOR versions subsequent events with the same value will not be filtered out anymore. Therefore use the **Step Filter** module.

---

## Step Filter

## Event Processing



An event is only passed if the input value is larger/smaller than the previous input value +/- tolerance.

- **In**: Input for events to be filtered.
- **Tol**: Input for the tolerance level.
- **Out**: Event output.

---

## Router M->1

## Event Processing



Router multiple to one. The events at the selected input will be let through other events will be filtered. The inputs are selected by the **Pos** input. When **Wrap** mode is selected in the Properties, **Pos** wraps around so that Max+1 is the same as 0, Max+2 is 1 etc..

The module has a dynamic in-port management. The number of in-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **Pos:** Input for selecting the thru input.
- **In:** Signal input.
- **Out:** Output for the selected input signal.

## Router 1,2

## Event Processing



On/off switch for one or toggle switch for two event signals, with control input. The last event passed through is held at the output even while off.

The module has a dynamic in-port management. The number of in-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties between 1 and 2.

- **Ctrl:** Hybrid input for control of switching. While the value is greater than zero all input events are passed to the output.
- **In:** Event input for the signal to be switched
- **Out:** Event output for the switched signal. While **Ctrl** is smaller than zero, the last value is held at the output.

## Router 1->M

## Event Processing



Router one to multiple. The events at the input will be let through to the selected output. The output is selected by the Pos input. When Wrap mode is selected in the Properties, Pos wraps around so that Max+1 is the same as 0, Max+2 is 1 etc..

The module has a dynamic out-port management. The number of out-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

- **Pos:** Input for selecting the thru output.
- **In:** Signal input.
- **Out:** Output for the input signal.



---

## Timer Event Processing



The elapsed time between the two last received events is measured and output as an event. Also, the frequency whose period of oscillation is equal to the measured duration is calculated and output.

- **In:** Polyphonic input for the events whose distance in time is to be measured.
- **F:** Polyphonic event output for the frequency of input events in Hz.
- **T:** Polyphonic event output for the time between events in milliseconds.

---

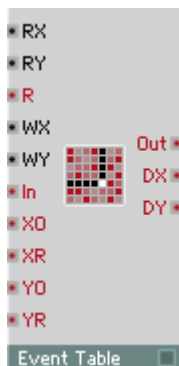
## HoldEvent Processing



Hold envelope.

When the module is triggered by a positive event at the **G** input, the event's value is held as the output value until the hold time has passed, after which the output jumps back to zero. The module can be triggered at any time, including retriggering during the hold time.

- **G:** Event input for triggering the envelope. Only events with a positive values (not zero) have an effect here.
- **H:** Input for controlling the hold time in milliseconds.
- **Out:** Event output for the envelope signal.



Holds a table of data values. Event values can be read from the table, event values can be stored in the table and the table's content can be displayed and edited graphically. The table can be 1-dimensional (a row of values) addressed by X, or 2-dimensional (a matrix of rows and columns, or a set of independent rows) addressed by X and Y.

The value at the output is taken from the table by reading at the position given by the inputs **RX** and **RY**. Values arriving at the module's **W** input are stored in individual cells of the table according to the write position given by the inputs **WX** and **WY**.

X is the horizontal position from left to right and Y is the vertical position from top to bottom. The count always starts at 0 for the first element.

The module's panel display can show all the data or a limited region of it. Many options in the properties allow customizing the behaviour.

For full details on properties, menus and keyboard shortcuts, please see the section *Table Modules* on page 165.

- **RX**: Audio input for the X-position of a table cell from which the data is read.
- **RY**: Audio input for the Y-position of a table cell from which the data is read. This is used in 2D-mode or for addressing the row number if more than one row exists.
- **R**: Event input for triggering table read operation at the position given by **RX** and **RY**. Each event here produces an event at the **Out** output.
- **WX**: Audio input for the X-position of a table cell in which the data is written.

- **WY:** Audio input for the Y-position of a table cell in which the data is written.
- **In:** Event input for writing into the table at the position given by **WX** and **WY**. The value of the data written into the table cell is the value of the event arriving at **In**.
- **XO:** Event input for the horizontal offset of the displayed data region. **XO** controls the data position that appears in the display (according to View Alignment). The value of **XO** is in the units specified in properties.
- **XR:** Event input for the horizontal range of the displayed data region. **XR** controls how many units of data fit in the display, i.e. it lets you zoom into the data.
- **YO:** Event input for the vertical offset of the displayed data region. **YO** controls the data position that appears in the display (according to View Alignment). The value of **YO** is in the units specified in properties.
- **YR:** Event input for the vertical range of the displayed data region in 2D-mode. **YR** controls how many units of data fit in the display, i.e. it lets you zoom into the data.
- **Out:** Event output for data from the cell controlled by the **RX**, **RY** and **R** inputs.
- **DX:** Event output for the size of the horizontal table rows in units.
- **DY:** Event output for the size of the vertical table columns in units.

# Auxiliary

If you can't find it anywhere else, it's probably here. First you'll find tape decks for recording and playing back audio files. Then there are modules for managing polyphonic voices, for converting audio signals to control signals, and for reporting the status of various REAKTOR processes such as tuning and tempo.

---

## Tapedeck 1-Ch

## Auxiliary



1-channel Tapedeck module for recording and playback of audio signals. Audio files can be read from and write to either memory or harddisk depending on the properties setting.

In harddisk mode the files are written into the folder you specify in the REAKTOR preferences. REAKTOR does sample rate converting on the fly and writes the file in the sample rate which is currently used by your audio system.

In memory mode you can display the waveform of a loaded audio file in the panel by ticking the checkbox **Picture** in the **Appearance** tab in the properties. The whole waveform for the audio file will fit automatically to the **Size** of the picture which you enter in the **Appearance** tab.

## Memory mode

By ticking the checkbox **Keep audio only in memory** in the properties the tapedeck module starts working in memory mode and the **Memory** section with the **Save**, **Save as...** and **Reload** buttons becomes active.

The maximum recording time is set with **Max Recording Size (sec)** in seconds. How big this value can be depends on the amount of available RAM storage in the computer. At a sample rate of 44.1 kHz you need 86 kB of RAM for each second of buffer length, for one minute you need 5 MB. If the buffer

memory for the Tapedeck has been chosen too large, virtual memory swaps by the operating system can cause significant hard disk activity during recording. This can prevent REAKTOR from processing audio smoothly.

Audio files can be imported for playback with the button **Select File...** in the properties of the Tapedeck module. A file that is already imported can be loaded again by clicking on the **Reload button**, e.g. if it has been changed using a sample editor.

When importing an audio file, the length of the tapedeck's memory buffer is adjusted to match the loaded data. If you later want to make a recording which is longer than this, you will first need to set a bigger value under **Max Recording Size (sec)** in the module's properties dialog window. The loaded file will automatically set to the current sample rate of REAKTOR.

---

**Note:** Be aware that REAKTOR converts all audio files stored in the tapedecks in memory mode to the new sample rate whenever REAKTOR switches to a new rate. So you should avoid changing the sample rate if your ensemble contains memory based audio samples. If the audio file is stored on hddisk you can use the **Reload** button after changing the sample rate to import the file again.

---

A recording can be exported using **Save** in the properties. If the file has already been exported, you will be asked if you want to overwrite the file, e.g. if you have made a new recording in the Tapedeck. With **Save as...** you can store the file under a new name.

## Harddisk mode

By ticking the checkbox **Stream audio from/to harddisk** in the properties the tapedeck module starts working in harddisk mode. Audio files are written to and read from harddisk directly. You can define an audio file for playback using the **Select File...** button. With the **New File** button you create a new file named "untitled" ( if a file with the name is already existing in your Audio folder you have specified in the REAKTOR Preferences the new name will contain an additional number). You can edit this name directly in the name field inside the Properties.

If **Value** is activated in the properties, a display for the file name will appear in the panel. By opening the context menu on this box files can also be imported and exported.

- **Rec:** Event input for switching recording mode on and off, e.g. with a gate signal. Start of recording on an event with a positive value. End of recording on an event with negative or zero value or when the maximum recording time is reached.
- **Play:** Event input for switching playback mode on and off, e.g. with a gate signal. Start of playback on an event with a positive value, playback stops on an event with negative or zero value.
- **Lp:** Event input for switching loop playback mode on and off. When this input receives a positive value, playback starts from the beginning when the end of playback is reached. The result is an infinite loop while playback is switched on.
- **In:** Monophonic audio input for the signal to be recorded. Maximum value  $\pm 1$ . To record a polyphonic signal a Voice Combiner has to be inserted.
- **Pse:** Pause control input. A value greater than zero pauses, else continues. Typ. range: [0 ... 1].
- **Pos:** Input for setting playback position in ms. Typ. range: [0 ... 20000].
- **Spd:** Variable playback speed of the Tape. Values must be greater than 0. A value of 1 means normal speed. Typ. range: [0.8 ... 1.2].
- **Out:** Audio output for the playback signal or the signal being recorded.
- **Rec:** 1 = Tapedeck is recording, else 0. Typ. range: [0 ... 1].
- **Play:** 1 = Tapedeck is playing back, else 0. Typ. range: [0 ... 1].
- **Wrp:** An Event with value 1 is sent each time the tape is wrapping around the loop.
- **Pos:** 1 = Tapedeck is paused, else 0. Typ. range: [0 ... 1].
- **Time:** current playback/recording position in ms [0 ... <length of sample in ms>].
- **Lng:** Length of recording in ms. Typ. range: [0 ... <length of sample in ms>].



This works just like **Tapedeck 1-Ch** but has two channels for recording and playback of audio signals. It imports and exports stereo files.

- **In L:** Monophonic audio input for the signal to be recorded on the left channel. Maximum value  $\pm 1$ . To record a polyphonic signal a Voice Combiner has to be inserted.
- **In R:** Monophonic audio input for the signal to be recorded on the right channel. Maximum value  $\pm 1$ . To record a polyphonic signal a Voice Combiner has to be inserted.
- **L:** Audio output for the playback signal or the signal being recorded on the left channel.
- **R:** Audio output for the playback signal or the signal being recorded on the right channel.

---

## Audio Voice Combiner

## Auxiliary



Audio Voice Combiner. Converts a polyphonic audio signal to monophonic by summing all the voices.

- **In:** Polyphonic audio input for the signal to be combined to mono
- **Out:** Monophonic audio output for the combined signal

---

## Event V.C. All

Auxiliary



Event Voice Combiner. Converts a polyphonic event signal to monophonic by sending the events of all the voices to the same monophonic voice.

- **In:** Polyphonic event input for the signal to be combined to mono
- **Out:** Monophonic event output for the combined signal

---

## Event V.C. Max

Auxiliary



Event Voice Combiner with maximum value selection. Converts a polyphonic event signal to monophonic by finding the voice with the highest current value and sending it to the mono output. A polyphonic gate signal input is necessary to recognize active voices.

- **In:** Polyphonic event input for the signal whose maximum value is to be output
- **G:** Polyphonic event input for the gate signal of the polyphonic instrument
- **Out:** Monophonic event output for the maximum value signal





Event Voice Combiner with minimum value selection. Converts a polyphonic event signal to monophonic by finding the voice with the lowest current value and sending it to the mono output. A polyphonic gate signal input is necessary to recognize active voices.

- **In:** Polyphonic event input for the signal whose minimum value is to be output
- **G:** Polyphonic event input for the gate signal of the polyphonic instrument
- **Out:** Monophonic event output for the minimum value signal

---

## A to E Auxiliary



Audio to event converter. The audio signal is sampled using the Control Rate specified in the **Settings** menu and the values are sent out as a stream of events.

---

## A to E (Trig)



Audio to event converter with trigger input. When the trigger input signal goes from zero to positive values (rising edge) the audio signal is sampled and output as an event.

- **T:** Audio input for triggering the conversion. Triggers on a rising edge
- **In:** Audio input for signal to be sampled and output as events
- **Out:** Event output for the sampled signal



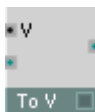
Audio to event converter with permanent conversion at regular intervals and adjustable sampling frequency. The audio signal is sampled with the given frequency and output as a stream of events. When running this module at a high frequency (above 1000 Hz) the CPU-Load caused by event processing can rise significantly. Typically **F** = 200 Hz is sufficient.

- **F:** Audio input for controlling the sample frequency. Value in Hz
- **In:** Audio input for signal to be sampled and output as events
- **Out:** Event output for the sampled signal



Audio to gate event converter with gate amplitude input. When the trigger signal goes from zero to positive values (rising edge), the gate signal is switched on with an amplitude of the current value of the amplitude input. When the trigger input returns to zero or negative values (falling edge) the gate signal is switched off.

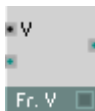
- **T:** Audio input for triggering the gate signal
- **A:** Audio input for controlling the amplitude of the gate events
- **Out:** Event output for the gate event signal.



Monophonic input signals are transmitted to one voice of the polyphonic output signal. The voice number is given by the current value of the **V** input. Signals are discarded when the value at **V** is not a valid voice number.

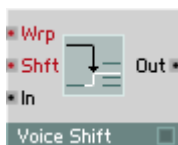
This is useful for making polyphonic sequencers, for example.

- **V**: Monophonic input for defining the number of the voice to which an event is to be sent. Typ. Range: [ 1 ... 10 ].
- **In**: Monophonic hybrid input for signals to be sent to one of the polyphonic voices.
- **Out**: Polyphonic hybrid output. The input signals are transmitted (with their original value) on one voice of this polyphonic signal.



One voice is selected from the polyphonic input signal by the current value of the **V** input. Only events on the selected voice are transmitted to the monophonic output. All events on other voices are discarded.

- **V**: Monophonic input for defining the number of the voice from which events are allowed through. Typ. Range: [ 1 ... 10 ].
- **In**: Hybrid input. One voice of this polyphonic signal is sent to the output.
- **Out**: Monophonic hybrid output. The input signal belonging to one voice are transmitted (with their original value) on this monophonic output.



The Voice Shift module is used to rearrange the polyphonic input values, so that the output values are remapped across the voices as required. For example, you could use Voice Shift to remap the values of voices 1, 2, 3, 4 to output voices 2, 3, 4, 1, or to 3, 4, 2, 1, and so on. If multiple input voices are shifted to the same output voice, they are summed. For example, if input voices 1 and 2 are both shifted to output voice 3, voice 3 will consist of: voice 1 signal + voice 2 signal.

**Wrp:** Monophonic event value that turns voice-shift wrap on and off (off by default). When wrap is off (ie.  $Wrp \leq 0$ ), voices shifted to invalid voice numbers (i.e. less than 1 or greater than the number of polyphonic voices) are discarded. When wrap is on (i.e.  $Wrp > 0$ ), voices shifted to invalid voice numbers are wrapped around to valid voice numbers using modulo math. For example, +1 shifting in a 3-voice instrument will cause voice 3 to be remapped to voice 1.

**Sh:** Polyphonic event value that controls the voice shifting. Positive Sh values shift voices up (e.g.  $Sh = 1$  would shift input voice 1 to output voice 2), and negative Sh values shift voices down ( $Sh = -2$  would shift voice 3 to voice 1). Since Sh is a polyphonic input, each voice can be shifted by its own individual offset. The default Sh value is 1.

**In:** Input for the polyphonic signal to be voice-shifted.

**Out:** Output for the polyphonic voice-shifted signal.



Smoother for monophonic event signals with audio output. Typically, a smoother is connected after a fader or button to get smooth transitions.

The jumps in the value of the input event signal are smoothed to ramps. The **Transition Time** (in milliseconds) can be adjusted in the properties. Exactly after this time has elapsed, the output reaches the same value as the input, assuming that no further jumps occurred at the input. The larger the **Transition Time**, the stronger the smoothing effect.

- **In:** Mono event input for the signal to be smoothed.
- **Out:** Mono audio output for the smoothed signal.



Smoother for monophonic event signals. Typically, a smoother is connected after a fader or button to get smooth transitions.

The jumps in the value of the input event signal are smoothed to ramps. The **Transition Time** (in milliseconds) can be adjusted in the properties. Exactly after this time has elapsed, the output reaches the same value as the input, assuming that no further jumps occurred at the input. The larger the **Transition Time**, the stronger the smoothing effect.

During the transition, events are output with the rate selected as the **Control Rate** in the **Settings** menu. The higher the rate, the higher the resolution of the smoothing transition.

- **In:** Mono event input for the signal to be smoothed.
- **Out:** Mono event output for the smoothed signal.

---

## Master Tune/Level

## Auxiliary



Controls the overall level and tuning.

- **Tun:** Control input for the master tuning. Scale: 1 semitone per unit. At 0.0 the note a3 is tuned to 440 Hz Typ. Range: [ -1 ... 1 ].
- **Lvl:** Control input for the master level at the output converters. Scale: 1 dB per unit 0.0 = unity gain = 0.0 dB Typ. Range: [ -60 ... 0 ].
- **Tun:** Output for for the master tuning.
- **Lvl:** Output for for the master level.

---

## Tempo Info

## Auxiliary



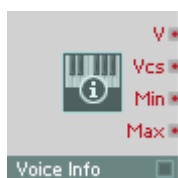
Source for the current Tempo measured in Beats per Second. To get the BPM value, multiply by 60.

- **Out:** Event output for the current tempo in Beats per Second (Hz).

---

## Voice Info

## Auxiliary



Voice Info module.

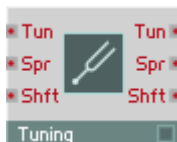
- **V:** Polyphonic output for the ID Number of each Voice [1, 2, 3... Voices ].
- **Vcs:** Output for the current Number of Voices in the instrument.
- **Min:** Output for the Min Unison Voices setting of the Instrument. This is the minimum number of voices assigned to one key for playing in unison.

- **Max:** Output for the Max Unison Voices setting of the Instrument. This is the maximum number of voices assigned to one key for playing in unison.

---

## Tuning Info

## Auxiliary



Control for and information about the settings of the instrument's tuning parameters.

- **Tun:** Control for tuning the instrument in semitones.
- **Spr:** Control for the amount of unison spread in semitones.
- **Shft:** Control for the shifting incoming MIDI notes in semitones.
- **Tun:** Output for the tuning of the instrument in semitones.
- **Spr:** Output for the tuning spread amount in semitones.
- **Shft:** Output for the amount of the note shift of incoming notes in semitones.

---

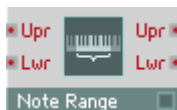
## System Info

## Auxiliary



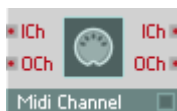
Source for information about the system: Sampling rate (in samples/sec), control rate (in Hz) and CPU load (in %).

- **SR:** Output for the current sampling rate in samples per second.
- **CR:** Output for the current control rate in Hz.
- **DCIk:** Output for the current display rate in frames per second. Event is sent just before each display update.
- **CPU:** Output for the current CPU load in percentages.



Note Range Info module.

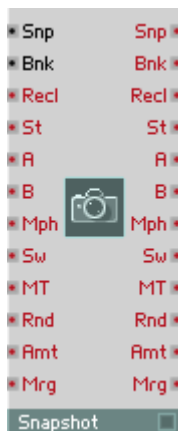
- **Upr:** Input for selecting the upper limit for received midi notes.
- **Lwr:** Input for selecting the lower limit for received midi notes.
- **Upr:** Output for the upper note limit for received midi notes.
- **Lwr:** Output for the lower note limit for received midi notes.



Midi Channel Info module.

- **ICh:** Input for selecting the input midi channel.
- **OCh:** Input for selecting the output midi channel.
- **ICh:** Output for the current input midi channel of the instrument.
- **OCh:** Output for the current output midi channel of the instrument.





The Snapshot module allows you to change snapshots and morph between snapshots from within REAKTOR. The module has a built-in list processor that works exactly like the List module (see the Panel modules section).

## Appearance

The panel representation of this module changes depending on the chosen style on the properties' **Appearance** page. The following styles are available:

- **Button:** Each module in-port creates a button. All buttons will be arrayed vertically in the instrument panel. The currently activated button will be displayed in the Indicator color of the Instrument.
- **Menu:** Each module in-port creates a new entry in a drop down list.
- **Text Panel:** Each module in-port creates a new entry in a list which displays multiple entries at the same time. If you have created more entries than fit into the text panel display specified by the **Size X** and **Size Y** fields on the Appearance tab, you will get scrollbars in the panel.
- **Spin:** Each module in-port creates a new entry in a list. You can switch through the list using a + and a - button at the right side of the list entry panel display.

The **Size X** and **Size Y** fields are controlling the display size of the control element in the panel.

## Ports

- **Snp:** Audio input for selecting the snapshot to be recalled or stored. Range: 1 ... 128.
- **Bnk:** Audio input for selecting the snapshot bank. Range: 1 ... 16.
- **Recl:** A positive event recalls the snapshot selected by the Snp and Bnk inputs.
- **St:** A positive event stores the snapshot to a position selected by the Snp and Bnk inputs.
- **A:** A positive event takes the current values from the Snp and Bnk inputs to select a snapshot for the Morph position A.
- **B:** A positive event takes the current values from the Snp and Bnk inputs to select a snapshot for the Morph position B.
- **Mrph:** This input controls the morph position between the snapshots loaded for A and B. Value  $\leq 0.0$ : A, Value 0.0 -1.0 : morphing between A and B. Value  $\geq 1.0$ : B.
- **Sw:** Events with negative and zero values set the switches/buttons to their position in snapshot A. Positive events set the switches/buttons to their position in snapshot B.
- **MT:** Event input for the morph time in ms.
- **Rnd:** A positive event triggers the snapshot randomize function.
- **Amt:** Input for the randomization amount. Range: 0.0 ... 1.0 (1.0 = 100%)
- **Mrg:** A positive event triggers the random merge function.
- **Snp:** Output for the index of the current snapshot. An event is sent every time a snapshot is recalled or stored.
- **Bnk:** Output for the index of the current snapshot bank. An event is sent every time a snapshot is recalled or stored.
- **Recl:** An event with value = 1.0 is sent when a snapshot was recalled.
- **St:** An event with value = 1.0 is sent when a snapshot was stored.
- **A:** Output for the index of the snapshot of morph position A. An event is sent when a snapshot was recalled or selected for position A.
- **B:** Output for the index of the snapshot of morph position B. An event is sent when a snapshot was recalled or selected for position B.

- **Mrph**: Output for the morph position. Value = 0.0 : A, Value 0.0 - 1.0: morphing between A and B, Value = 1.0: B.
- **Sw**: Output for Switch Position A/B. Value = 0.0 if the switches/buttons are set to their position in snapshot A. Value = 1.0 if the switches/buttons are set to their position in snapshot B.
- **MT**: Output for the morph time in ms.
- **Rnd**: An event (with value = 1.0) is sent when the randomize function was triggered.
- **Amt**: Output for the randomization amount. Range: 0.0 ... 1.0 (1.0 = 100%)
- **Mrg**: An event (with value = 1.0) is sent when the random merge function was triggered.

---

## Set Random

## Auxiliary



Sets the random seed for the pseudo-random number generator used in all **Event Randomize**, **Slow Random** and **Geiger** modules. Only modules in the same instrument are affected. Each unique value starts a different pseudo-random sequence.

---

## Unison Spread

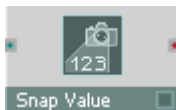
## Auxiliary



Polyphonic event source for a constant value used to spread out parameters for unison voices.

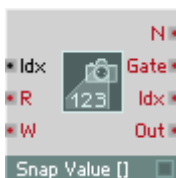
In unison mode, the different voices that play the same note need to have their parameters spread out a little to make them slightly different so that their sum results in a sound that is fatter, and not just louder. For note pitch this happens automatically and is controlled by the **Unison Spread** parameter in the instrument properties. Other parameters can be spread out by adding an offset generated by the Unison Spread module.

The value at the module's input determines the amount of spread. At the output you get a value which is different for each of the voices playing the same note. The value only changes once a new note is played.



This module stores the input value with a snapshot and outputs the value, when the snapshot is recalled.

- **In:** Input for the value to be stored in a snapshot. The input signal is sampled in the moment when the snapshot is stored. If connected to an event source, input events are passed to the output.
- **Out:** Output for the value stored in the snapshot that has been recalled most recently.



Stores and recalls arrays of floating-point (fractional) values to/from the edit buffer and snapshots.

A single Snap Value Array can hold 1-40 arrays, and each array can contain any number of elements, limited only by the availability of system memory. All arrays contain the same number of elements. Both the number of arrays and the number of elements per-array are defined in the properties.

Memory for the Snap Value Array module is allocated dynamically. That is, creating additional snapshots causes additional memory to be allocated, and deleting snapshots causes memory to be released. This keeps memory requirements as low as possible.

When the Snap Isolate option is enabled in the properties, only the most recent value written to each array element is stored (and recalled during ensemble initialisation), and no data is stored or recalled in response to snapshot operations.

A typical application of the Snap Value Array is storing sequencer data in snapshots. For this purpose, it is often used in conjunction with Event Tables or Multi Display modules.

All Snap Value Array inputs accept monophonic signals only.

- Idx:** Index of the array element to address (for both read and write operations). Arrays are 1-based; i.e. the index of the first element is 1 (not 0). Fractional values are rounded to the nearest integer. Where Idx values are outside the range of 1 to N, the behaviour depends on the Index Behaviour option in the properties.
- R:** When an event (of any value) arrives at the R (read) input, the array element specified by the Idx input is read, and its value is transmitted to the output port. In other words, each event at the R input propagates a single event at the array output port. Multiple arrays are addressed in parallel by the same Idx index. Thus, events arriving at the R input propagate output events at every array output port (of the value of the element selected by Idx). It is essential to set the Idx value before events arrive at the R input.
- W:** When an event arrives at W, its value is written to element [Idx] of the array, overwriting any data that was previously there. The Snap Value Array provides a separate W input port for each array it contains (1 array by default, but more can be created in the properties). Each port can be renamed as appropriate. When the Events Thru option is enabled in the properties, events arriving at the W inputs are passed to the corresponding Out outputs.
- N:** The N output reports the number of elements in each array.
- Gate:** Gate sends an event with value 1 before the array output ports send events, and then sends an event of value 0 afterwards.
- Idx:** Output reporting the number of the element currently being read or written to (in response to events arriving at the R and W ports, or from snapshot operations, such as recall and morph). With respect to read operations, the Idx number will be reported before the value event is transmitted at the Out outputs.
- Out:** The value of the currently accessed array element (as defined by Idx) are transmitted here in response to events at the R port and snapshot operations (recall, morph etc.). When the Self-Iteration option is enabled in the properties, all array elements are output in serial fashion (i.e. the first element, then the second element, then the third etc) when any of the following operations occur: initialization, activation, snap recall,

randomize, random merge, and morph. Self-Iteration enables the updating of a complete set of data with snap operations.

**Terminal:** Terminals are used to route audio and control signals in and out of REAKTOR's Instrument and Macro sub-structures. You can create terminals automatically when you drag a cable to an Instrument or Macro within the structure window while holding down the Ctrl key.

---

## In Port

## Terminal



Terminal for audio and event signals to create in-ports for Instruments and Macros.

---

## Out Port

## Terminal



Terminal for audio and event signals to create out-ports for Instruments and Macros.

---

## Send

## Terminal



Send terminal for audio and event signals to create a cableless connection within an Instrument.

Each inserted Send module creates an entry in a list of available signal sources in the Properties of a Receive module.

---

## Receive

## Terminal



Receive terminal for audio and event signals to create a cableless connection within an Instrument.

## Properties - Function page

For each Send module inserted in the same Instrument a list entry is created. The list entry name will be created according to the label of the Send module.

The **Up** and **Down** buttons above the list serves for moving a selected entry up or down.

The list contains the following columns which can be sorted by a click on the appropriate header entry:

- **#**: Indicates the list position of the Send module. The default value refers to this number.
- **Label**: The name of the Send module. If you rename a Send module the label in the list will be updated.
- **State**: Indicates if a connection to this send module is possible. Only establish a connection if this field displays **OK**.
- **Use**: Click on this field to set a connection between to the appropriate Send module. An existing connection is indicated by a cross.

The **Mouse Resolution** only applies to the panel control if you choose the **Spin** style on the **Appearance** page, where you can click on the control entry and drag up or down to change the entry.

The **Default** value is used whenever an initialization of the control happens. In this case the entry **#** in the list will be selected according to the **Default** value entered here.

## Appearance

The panel representation of this module changes depending on the chosen style on the properties' **Appearance** page. The following styles are available:

- **Button**: Each module in-port creates a button. All buttons will be arrayed vertically in the instrument panel. The currently activated button will be displayed in the Indicator color of the Instrument.
- **Menu**: Each module in-port creates a new entry in a drop down list.
- **Text Panel**: Each module in-port creates a new entry in a list which displays multiple entries at the same time. If you have created more entries than fit into the text panel display specified by the **Size X** and **Size Y** fields on the Appearance tab, you will get scrollbars in the panel.
- **Spin**: Each module in-port creates a new entry in a list. You can switch through the list using a **+** and a **-** button right hand of the list entry panel display.

The **Size X** and **Size Y** fields are controlling the display size of the control element in the panel.

---

## IC Send

## Terminal

Transmits monophonic event signals to any module capable of receiving IC (Internal Connection) transmission. Such modules include IC Receive modules, but also various panel elements (such as knobs and switches). As internal connections work globally (i.e. at the ensemble level), this module can be used to make wireless connections between different instruments within the ensemble.

The IC Send module has a panel display allowing connections to be configured from the instrument panel. All modules in the ensemble capable of receiving IC transmission will appear here, except those with the 'No Entry in IC Menu' properties option enabled. Connections can also be established in the properties dialog.

---

## IC Receive

## Terminal

Receives and outputs monophonic event signals to modules which connected via the Internal Connection (IC) protocol. Typically the IC Receive module is used with IC Send modules, but can be connected to any module capable of connection via IC (such as knobs and switches).

IC connections can be established in the properties, and when connecting to IC Send modules, connections can also be established using the IC Send panel interface.





OSC messages can be communicated using the **OSC Send** and **OSC Receive** modules. Both modules are dynamic - new in- or out-ports are added by wiring to blank regions in the port areas while holding down the **Ctrl** key (left edge if the Send and right edge of the Receive Module).

Multiple connections to the **OSC Send** module result in OSC messages with multiple arguments. For example, if you wire the **MX** and **MY** outputs of an **XY** Module to an **OSC Send** module, each OSC message will have both the **X** and **Y** value. You must wire multiple outputs from the **OSC Receive** Module to receive multiple arguments-one for each argument up to a maximum of 10.

If you have created more than one in-port for the **OSC Send** module, always the first in-port of the module will trigger the OSC message.

OSC messages can also be sent and received directly from some REAKTOR Modules, for example panel controls. The send and receive settings are the same as in the OSC terminal modules.

The module has a dynamic in-port management. The number of in-ports can be defined with **Min Num Port Groups** on the **Function** page of the Properties.

---

## OSC Receive

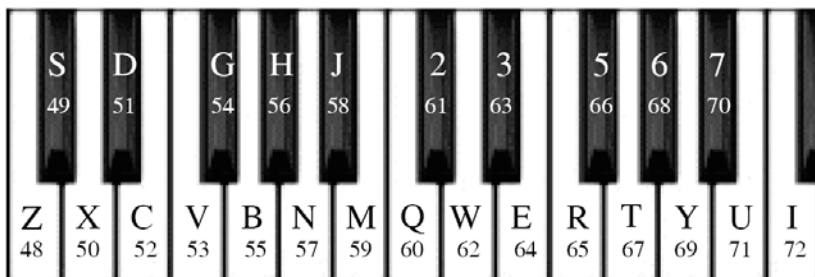
## Terminal



See OSC Send module.

# Appendix

## Transposing Incoming MIDI Notes

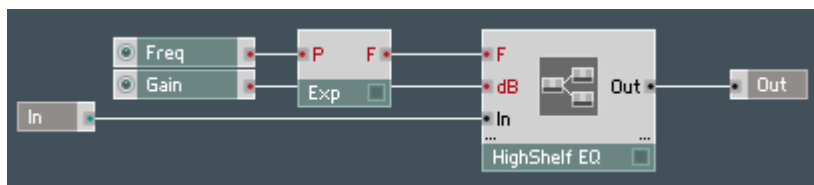


- Holding down the **Shift** key raises all incoming MIDI notes by two octaves (+24).
- Holding down WindowsXP: **Ctrl + Shift** / OS X: **APPLE + Shift** lowers all incoming MIDI notes by two octaves (-24).

# First steps in Reaktor Core

## What is Reaktor Core

*Reaktor Core* is a new level of functionality within Reaktor with a new and different set of features. Because there is also an older level of functionality, we will hereinafter refer to these two levels as the *core level* and the *primary level*, respectively. Also when we say “primary-level structure” we will mean the structure of an instrument or macro, but not the structure of an ensemble. The features of Reaktor Core are not directly compatible with those of the primary level, so some interfacing is required between them, and that comes in the form of *core cells*. Core cells exist inside primary-level structures, and they look similar and behave similarly to primary-level built-in modules. Here is an example structure, using a *HighShelf EQ* core cell, which differs from the primary-level built-in module version in that it has frequency and boost controls:



Inside of core cells are Reaktor Core structures. Those provide an efficient way to implement custom low-level DSP functionality as well as to build larger-scale signal-processing structures using such functionality. We will take a detailed look at these structures later.

Although one of the main purposes of Reaktor Core is to build low level DSP structures, it is not limited to that. For users with little DSP programming experience, we have provided a library of pre-built modules, which you can connect inside core structures, just as you do with ordinary modules and macros in primary-level structures. We have also provided you with a library of pre-built core cells, which are immediately available for you to use in primary-level structures.

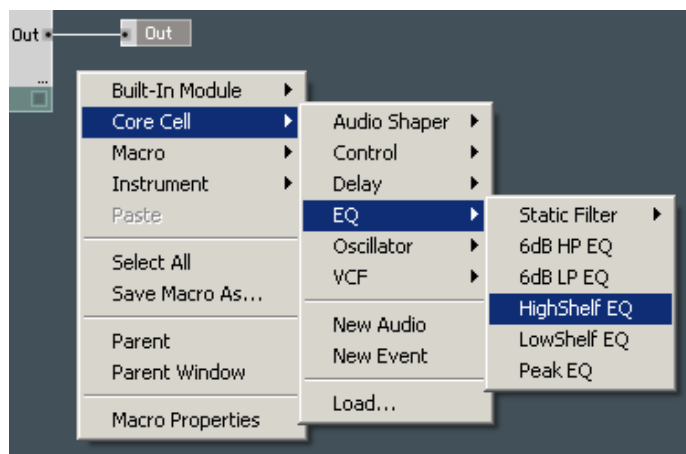
---

In the future, Native Instruments will put less emphasis on creating new primary-level modules. Instead, we will use our new Reaktor Core technology and provide them in the form of core cells. For example, you will already find a set of new filters, envelopes, effects, and so on in the core cell library.

---

## Using core cells

The core cell library can be accessed from primary-level structures by right-clicking on the background and using the *Core Cell* submenu:



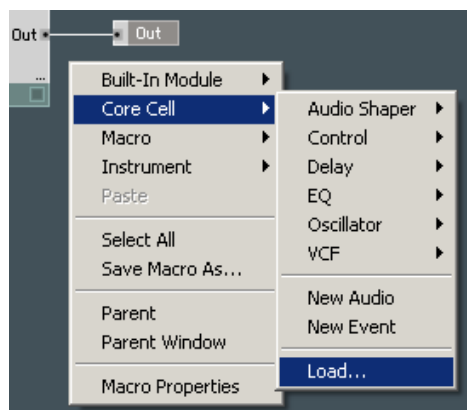
As you can see, there are all different kinds of core cells; they can be used in the same way as primary-level built-in modules.

---

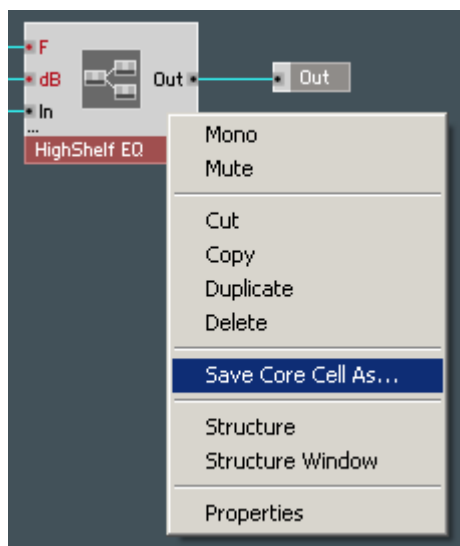
An important limitation of core cells is that you are not allowed to use them inside event loops. Any event loop occurring through a core cell will be blocked by Reaktor.

---

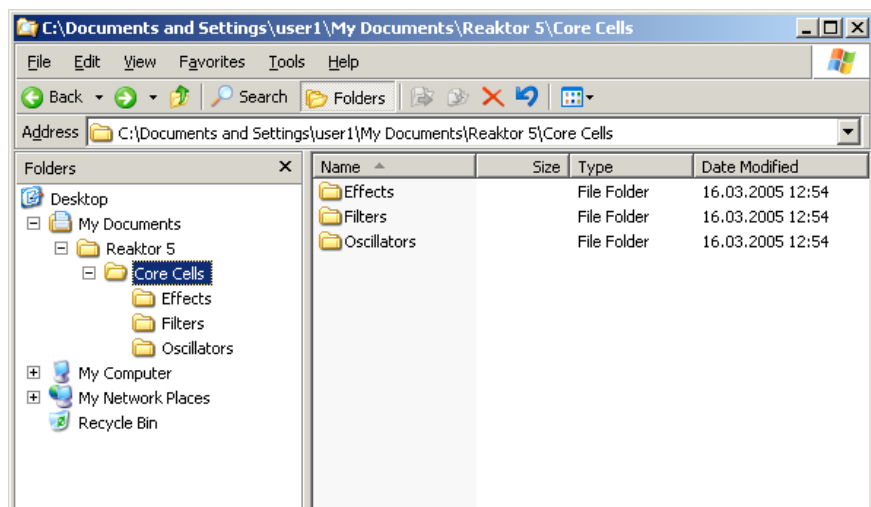
You can also insert core cells that are not in the library. To do that, use the *Load...* command from the *Core Cell* menu:



You may also want to save core cells you've created or modified, so that you can load them into other structures. To save a core cell, right-click on it and select *Save Core Cell As*:

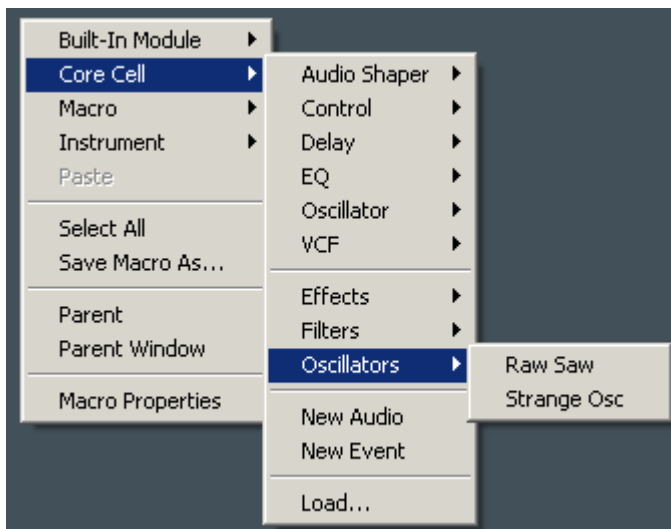


Rather than using the *Load...* command, you can have your core cells appear in the menu by putting them into the *Core Cells* subdirectory of your user library folder. Better still, you can further organize them into subgroups. Here's an example:



“My Documents\Reaktor 5” is the user library folder in this example. On your computer there may be a different path, depending on the choice you’ve made during installation and any changes you’ve made in Reaktor’s preferences. Inside the user library folder there’s a folder named “Core Cells”. (Create it manually if it doesn’t exist.)

Inside the *Core Cells* folder, notice the folder structure consisting of the *Effects*, *Filters*, and *Oscillators* folders. Inside those folders are core cell files that will be displayed in the user part of the *Core Cell* menu:



---

The menu contents are scanned once during Reaktor startup, so after putting new files into these folders, you should restart Reaktor.

---

---

Empty folders are not displayed in the menu; a folder must contain some files to be displayed.

---

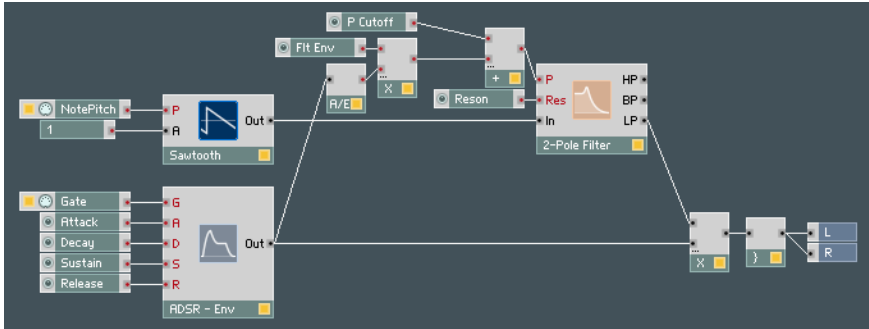
---

Under no circumstances should you put your own files into the system library. The system library may be changed or even completely replaced when installing updates, in which case your files will be lost. The user library is the right place for any content that is not included in the software itself.

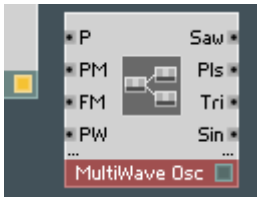
---

## Using core cells in a real example

Here we are going to take a Reaktor instrument built using only primary-level modules and modify it by putting in a few core cells. In the *Core Tutorial Examples* folder in your Reaktor installation, find the *One Osc.ens* ensemble and open it. This ensemble consists of only one instrument, which has the internal structure shown:



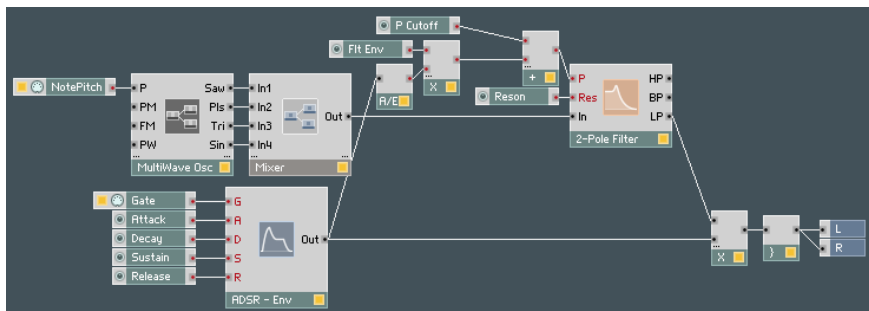
As you can see this is a very simple subtractive synthesizer consisting of one oscillator, one filter and one envelope. We are going to replace the oscillator with a different, more powerful one. Right-click on the background and select *Core Cell > Oscillator > MultiWave Osc*:



The most important feature of this oscillator is that it simultaneously provides different analog waveforms that are locked in phase. We are going to replace the Sawtooth oscillator with the MultiWave Osc and use a mix of its waveforms instead of a single sawtooth waveform. Fortunately, there's already a mixer macro available from *Insert Macro > Classic Modular > O2 - Mixer Amp > Mixer - Simple - Mono*:



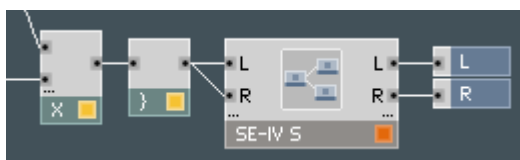
Connect the mixer and the oscillator together and use their combination to replace the sawtooth oscillator:



Switch to the panel view. Now you can use the four faders of the mixer to vary the waveform mix.

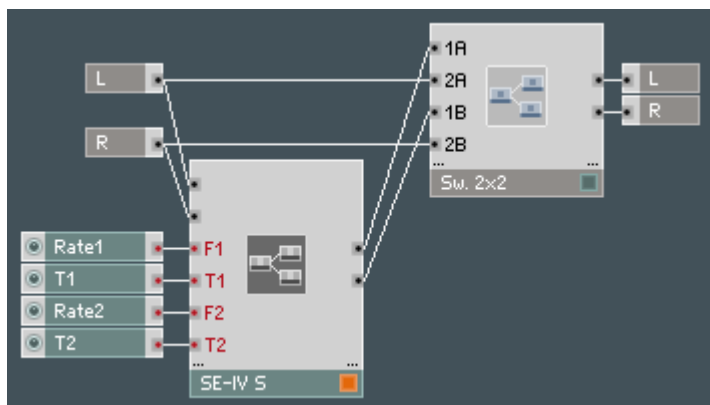
Let's do one more modification to the instrument and add a Reaktor Core-based chorus effect. We say Reaktor Core based, because although the chorus itself is built as a core cell, the part containing panel controls for this chorus is still built using the primary-level features. That's because at this time Reaktor Core structures cannot have their own control panels – the panels have to be built on the primary level.

Select *Insert Macro > Building Blocks > Effects > SE-IV Chorus* and insert it after the Voice Combiner module:



If you look inside the chorus you can see the chorus core cell and the panel controls:

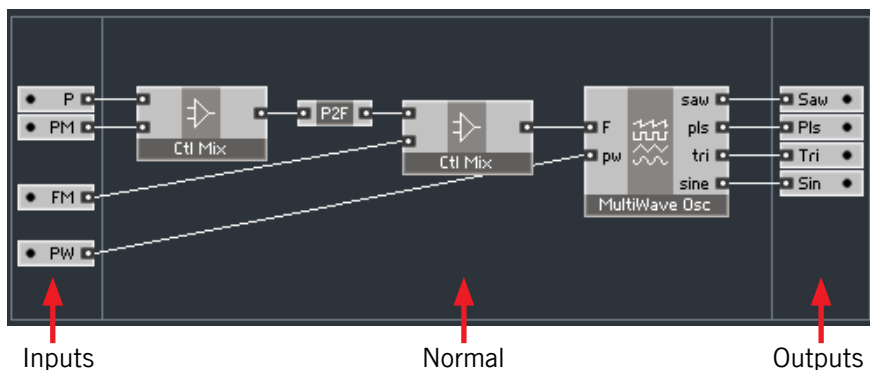




## Basic editing of core cells

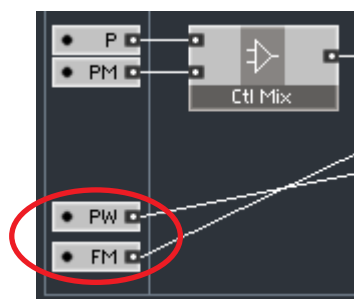
Now we are about to learn a few things about editing core cells. We are going to start with something simple: modifying an existing core cell to your particular needs.

First, double-click the *MultiWave Osc* to go inside:

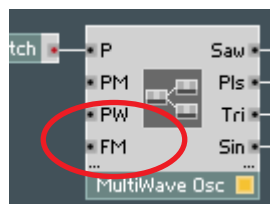


What you see now is a Reaktor Core structure. The three areas separated by vertical lines are for three different kinds of modules: inputs (on the left), outputs (on the right), and normal modules (center).

Whereas normal modules can move in all directions, the inputs and outputs can only be moved vertically, and their relative order matches the order in which they appear outside. So, you can easily rearrange their outside order by moving them around. Try moving the *FM* input below the *PW* input:



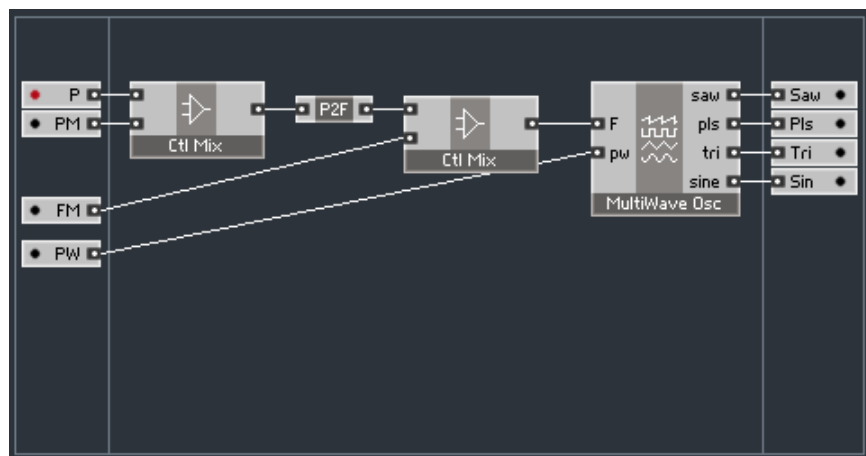
You can double-click the background now to ascend to the outside, primary-level structure and see the changed port order:



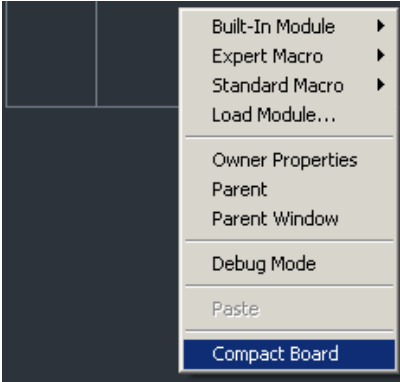
Now go back to the core level and restore the original port order:



As you have probably already noticed, if you move modules around, the three areas of the core structure automatically grow to accommodate all modules inside them. However, they do not automatically shrink, which can lead to these areas sometimes becoming unnecessarily large:



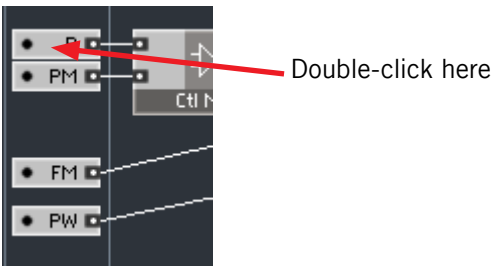
You can shrink them back by right-clicking on the background and selecting *Compact Board* command:




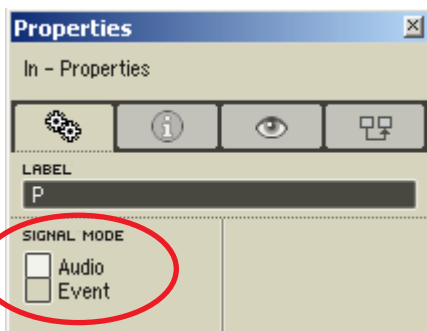
Now that we have learned to move the things around and rearrange the port order of a core cell, let's try a few more options.

For a core cell *that has audio outputs* it's possible to switch the type of its inputs between audio and event (a more detailed explanation can be found later in this manual). In the above example, we used a *MultiWave Osc* module, all of whose inputs and outputs are audio. However, in this example we don't really need them as audio, because the only thing connected to the oscillator is a pitch knob. Wouldn't it be more CPU efficient to have at least some of the ports set to event type? The obvious answer is, "yes, it would." Here's how to do that.

Changing both P and PM inputs to event mode should produce the largest CPU improvement. To do that double-click on the P port module to open its properties window:



Switch the properties window to the function page, if necessary, by clicking on the  tab. You should now see the Signal Mode property:

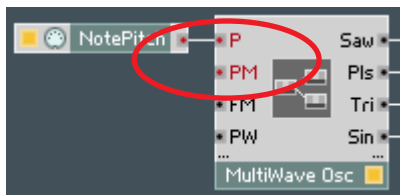


Change it to event. Note how the large dot at the left of the input module changes from black to red indicating that the input is now in event mode (it's more easily visible after you deselect the port – just click elsewhere):

The dot turns red



Now click on the *PM* input to select it, and change it to event mode, too. If you want, you can change the two remaining inputs to event mode as well. Finally, double-click the structure background to return to the primary level and observe that the port colors have changed to red and the CPU usage has gone down.



Sometimes it doesn't make sense to switch a port from one type to another. For example, it doesn't make sense to switch an input that receives a real audio signal (meaning real audio, not just an audio-rate control signal like an envelope) to an event rate. In some cases such switching could even ruin the functionality of the module. Going in the other direction, it doesn't make sense to change an event input that is really event sensitive, such as an envelope's event trigger input (for example, gate inputs of Reaktor primary-level envelopes). If you change such an input to audio, it will no longer work correctly. In addition to cases in which port-type switching obviously does not make sense there may be cases in which it does make sense, but in which the modules will not work correctly if you switch their port types. Such cases are quite special, although they can also result from mistakes in the implementation

or design of the module. Generally, port-type switching should work; hence the following switching rule:

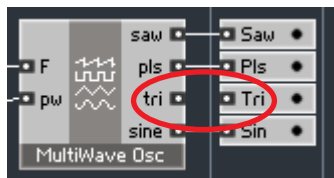
---

In a well designed core cell, an audio-rate control input can typically be switched to event mode without any problem. An event input can be switched to audio only if it doesn't have a trigger (or other event-sensitive) function.

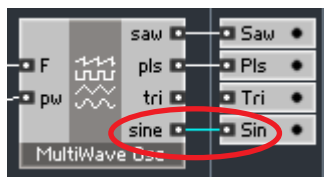
---

Another way to save CPU is to disconnect the outputs that you don't need, thereby deactivating unused parts of the Reaktor Core structure. You have to do that from inside the structure – outside connections do not have any effect on deactivating the core structure elements.

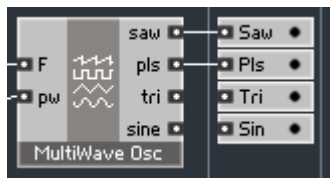
Suppose in our example we decide that we only need the sawtooth and pulse outputs. We can lower the CPU usage by going inside the *MultiWave Osc* and disconnecting the unused outputs. Disconnecting is simple in Reaktor Core, you click on the *input* port of the connection, drag the mouse to the any empty part of the background and release it. For example, click on the input port of the Tri output and drag the mouse into empty space on the background.



There's another way to delete a connection. Click on the wire between the sine output of the *MultiWave Osc* and *Sin* output of the core cell, so that it gets selected (you can tell that it's selected by its blue color):

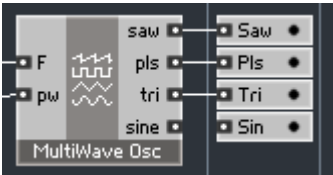


Now you can press the *Delete* key to delete the wire:



After you deleted both wires, the CPU meter should go down a little more.

If you change your mind, you can reactivate the outputs by clicking on either the input or the output that you want to reconnect and dragging the mouse to the other port. For example, click on the *Tri* output of the *MultiWave Osc* and drag to the input of the *Tri* output module. The connection is back:



Of course, numerous fine-tuning adjustments can be made to core cells. You will learn about many more options as you proceed through this manual.

## Getting into Reaktor Core

### Event and audio core cells

Core cells exist in two flavors: *Event* and *Audio*. Event core cells can receive only primary-level event signals at their inputs and produce only primary-level event signals at their outputs in response to such input events. Audio core cells can receive both event and audio signals at their inputs but provide only audio outputs:

Flavor	Inputs	Outputs	Clock Src
Event	Event	Event	Disabled
Audio	Event/Audio	Audio	Enabled

Therefore audio cells can implement oscillators, filters, envelopes, effects and other stuff, while event cells are suitable only for event processing tasks. The *HighShelf EQ* and *MultiWave Osc* modules that you are already familiar with are examples of audio core cells (you can tell that by the fact that they have audio outputs):

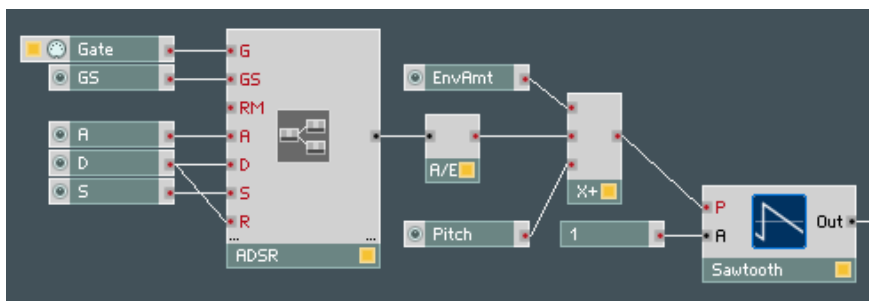


And here is an example of an event core cell:



This module is a parabolic shaper for control signals, which can be used to implement velocity curves or LFO signal shaping, for example.

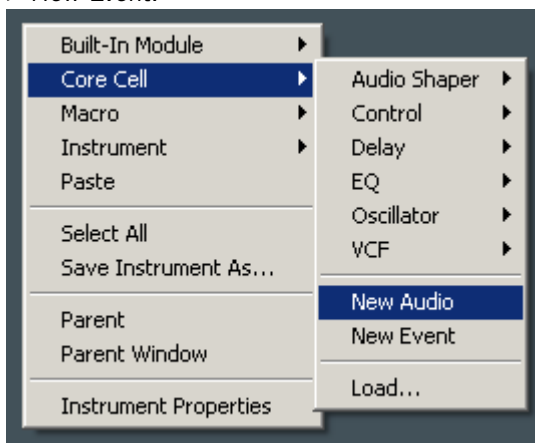
As previously mentioned, event core cells are restricted to event processing tasks. Because clock sources are disabled inside them (see the table above), they cannot generate their own events and, therefore, cannot implement modules such as event-rate LFOs and envelopes. When you need such modules, we suggest that you take an audio cell and convert its output to event rate using one of the primary-level audio to event converters:



The above structure uses an audio core cell implementing an ADSR envelope and converts it to event rate to modulate an oscillator.

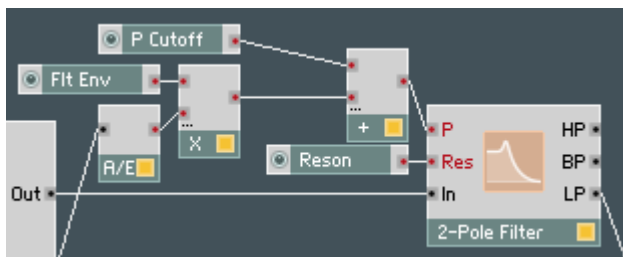
## Creating your first core cell

You create new core cells by right-clicking on the background in a primary-level structure and selecting *Core Cell > New Audio* or, for event cells, *Core Cell > New Event*:



We are going to build a new core cell from scratch inside the same *One Osc.ens* you already played with. We will be using the modified version of that ensemble with the new oscillator and chorus that we built in the last chapter, but if you didn't save it don't worry, you can do the same steps using the original *One Osc.ens*.

As you can see, in this ensemble we are modulating the filter at the *P* input, which accepts only event signals. We are not using the FM version of the same filter because it does not perform as well at higher cutoff frequencies, and because the modulation scale is linear at an *FM* input, which generally gives less musical results when modulated by an envelope. (That phenomenon is typically but incorrectly referred to as “slow envelopes”).:

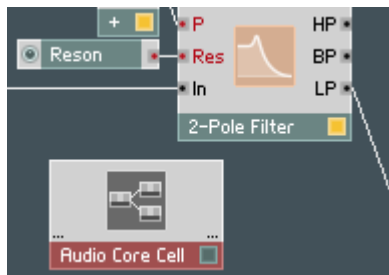


Because we need to apply the modulation at an event input, we also need to convert the envelope's output to an event signal, which we do with an A/E

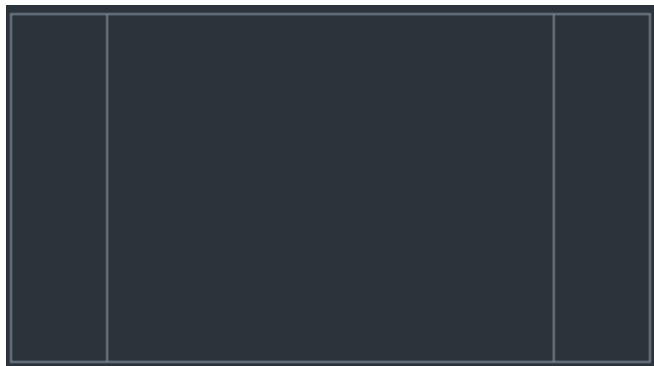


converter. As a result, our control rate is pretty low. Of course we could have used a converter running at a significantly higher rate (and eating up significantly more CPU), but what we are going to do instead is replace this filter with one which we build as a core cell. Alternatively, we could have taken an existing filter from the core-cell library, but then we would miss all the fun of making our first Reaktor Core structure.

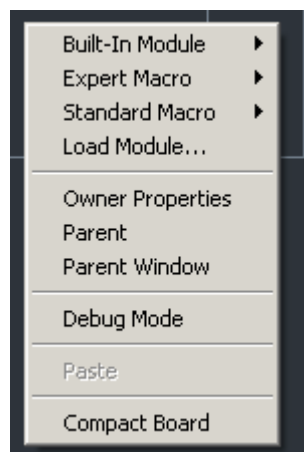
We'll start by creating a new audio core cell. Select *Core Cell > New Audio* and an empty audio core cell will appear:



Double-click it to see its structure, which is obviously empty. As you surely remember, the three areas are meant for input, output, and normal modules:



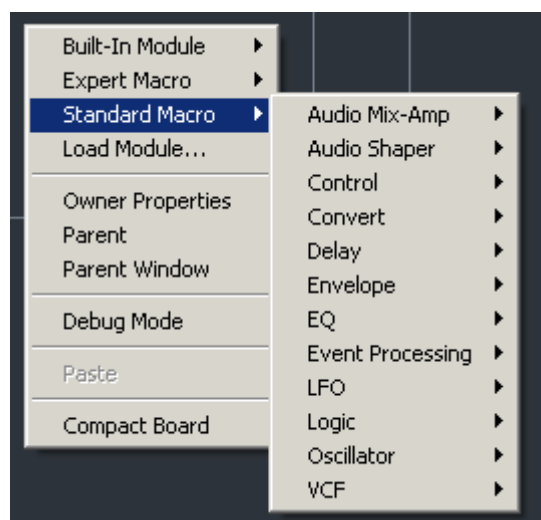
Attention: we are going to insert our first module into a core structure right now! Right-click in the normal area to bring up the module creation menu:



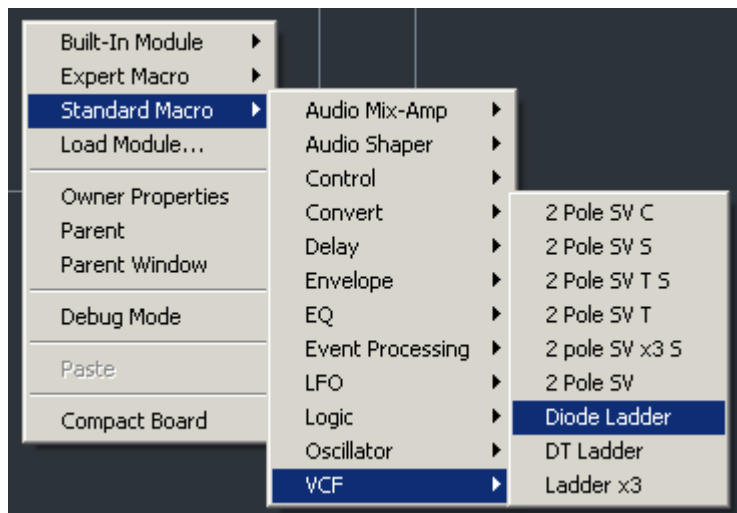
The first submenu is called *Built-In Module* and provides access to the built-in modules of Reaktor Core, which are generally meant to do really low-level stuff and will be discussed later.

The second submenu is called *Expert Macro* and contains macros meant to be used alongside built-in modules for low-level stuff.

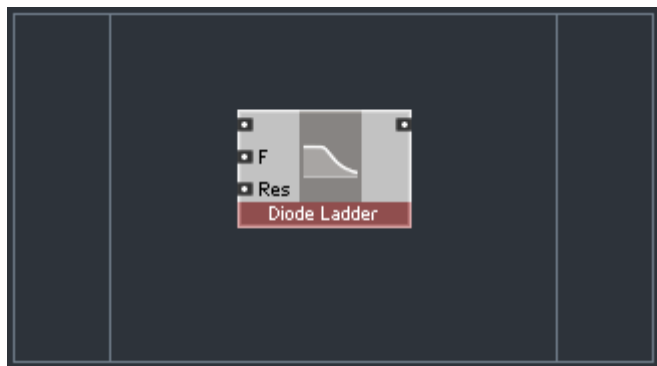
The third submenu, called *Standard Macro*, is the one we want to use:



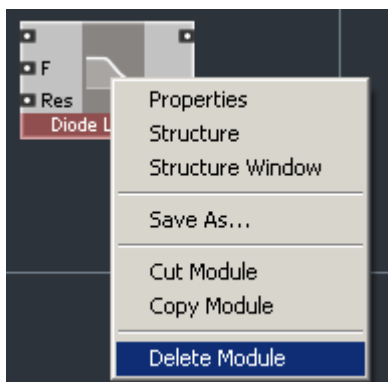
The VCF section could be promising, let's look inside:



Let's try *Diode Ladder*:

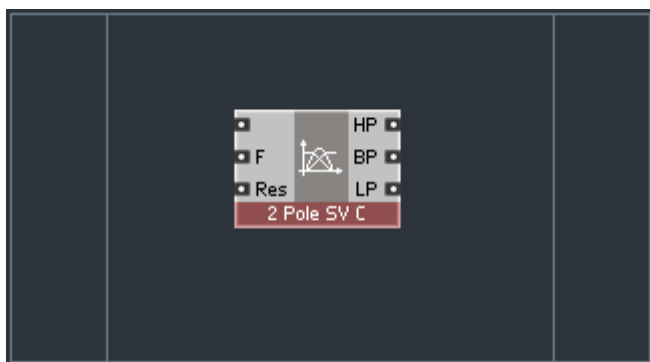


Well, maybe that was not the best idea, because a diode ladder might sound significantly different from the primary-level filter module we are trying to replace. At minimum, *Diode Ladder* is a 4-pole (24dB/octave) filter, and the one we are replacing is a 2-pole filter (12dB/octave). To delete it there are two options. One is to right-click on the module and select *Delete Module*:



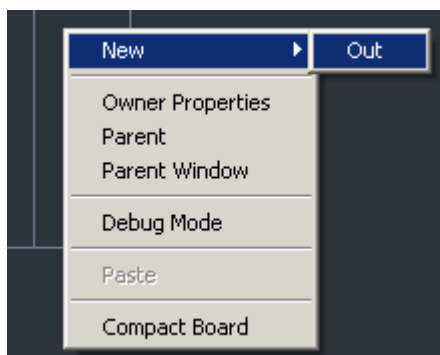
The other option is to select the module by clicking on it and pressing the *Delete* key.

After deleting the *Diode Ladder*, insert a *2 Pole SV C* filter from the same VCF section of the *Standard Macro* submenu:



This is a 2-pole, state-variable filter and is similar to the one we are replacing (there are some differences, but they are quite subtle). What's important is that we can modulate this filter at audio rates.

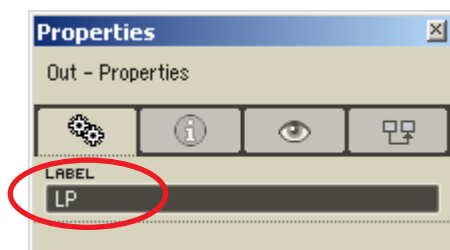
Obviously, we need some inputs and some outputs for our core cell. To be exact we need only one output – for the *LP* signal. To create it right-click in the outputs area:



There's only one kind of module you can create there, so select it. This is what the structure is going to look like:



Double-click the output module to open the Properties window (if it's not already open). Type "LP" in the label field:



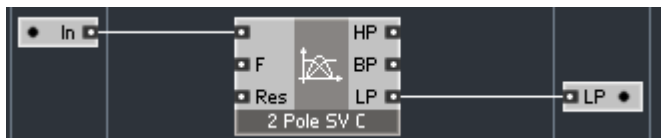
Now connect the *LP* output of the filter to the output module:




Now let's start with the inputs. The first input will be an audio-signal input. Right-click in the background of the inputs area and select *New > In*:

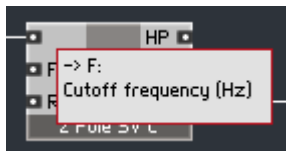


The input is automatically created with the right type – it’s an audio input, as you can tell by the large black dot. Rename this input to “In” in the same way you renamed the output to “LP”, then connect it to the first input of the filter module:

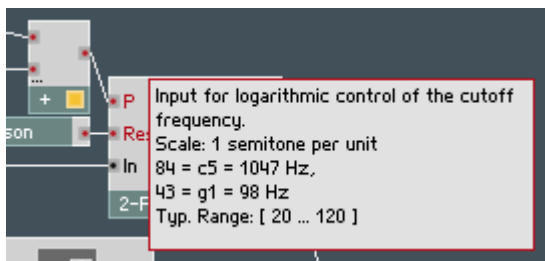


The second input is a little bit more complicated. As you can see, the second input of the filter Reaktor Core module is labeled “F”. That means frequency,

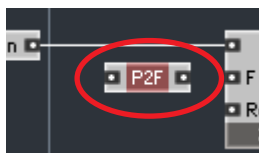
and if you hold your mouse over that input for a while (make sure the  button is active), you’ll see the info text, which says “Cutoff frequency (Hz)”:



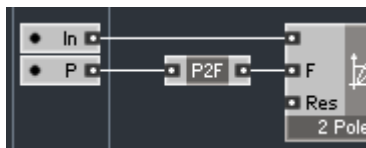
As we know, the cutoff of our primary-level filter module is controlled by an input labeled “P”, and as you can see from the info text, the signal uses a semitone scale.



We obviously need to convert from semitones to Hz. We can do that either on the primary level (using the *Expon. (F)* module) or inside our Reaktor Core structure. Because we are learning to build Reaktor Core structures, let’s go for the latter option. Right-click in the background of the normal area and select *Standard Macro > Convert > P2F*:



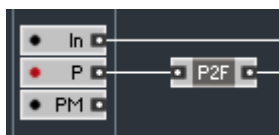
As the name implies (and the info text states), this module converts between P (pitch) and F (frequency) scales – exactly what we need. So let's create a second input labeled “P” and connect it using the *P2F* module:



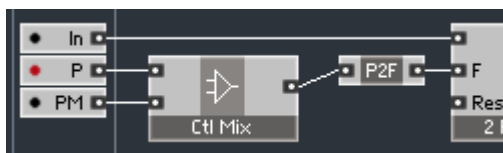
That should do it, but wait! In our instrument we have a “P Cutoff” knob defining the base cutoff of the filter, and to that is added the modulation signal from the envelope, which we have to convert to an event signal on the primary level in order to feed it into the P input of the filter. Now that the conversion is no longer necessary, we can remove the *A/E* module and plug the audio signal directly into the audio P input of our new filter. Although this approach is fine, let's look at another way, just for fun.

We'll start with our P input in event mode and add another modulation input in audio mode. (If you remember our discussion about slow envelopes, you will understand why we decided to call this input “PM”, not “FM”.) We also need to have the modulation input use the semitones (pitch) scale. That's exactly how it was done in our original instrument: we added our envelope signal to the “P Cutoff” signal and plugged the sum into the P input.

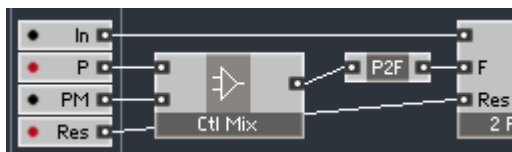
So first change the P input to the event mode (as described previously) and add another PM input, which should be in audio mode:



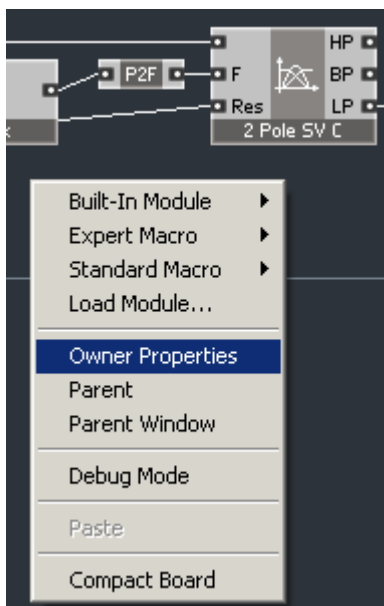
As a user of the Reaktor primary level, you probably expect us to add the two signals together now. In fact, we could do that, but in Reaktor Core the *Add* is considered a low-level module, and using it generally requires some knowledge of fundamental Reaktor Core low-level working principles. They are not that complex and will be described later in this text. For now, you don't need to know them; just use a control signal mixer instead, for example, *Standard Macro > Control > Ctl Mix*:



The last input that we need is a resonance input, it doesn't need to be at audio rate, so let's use an event input:

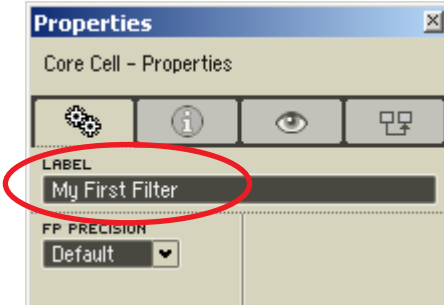


One other thing we need to do is to give our core cell a name. If the Properties window is already open, click on the background to display the core cell's properties. If it's not open, right-click on the background and select the *Owner Properties* command:

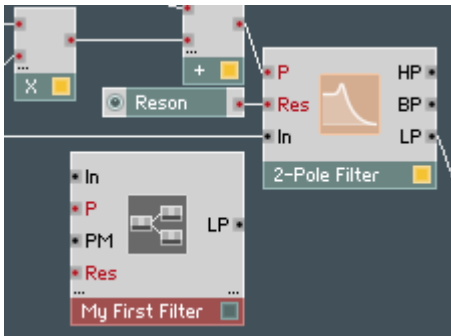




Now you can type some text into the label field:

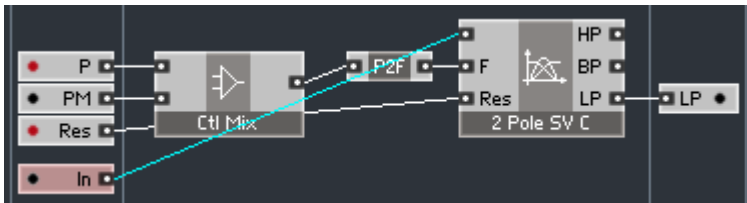


Double-click the background to see your result:



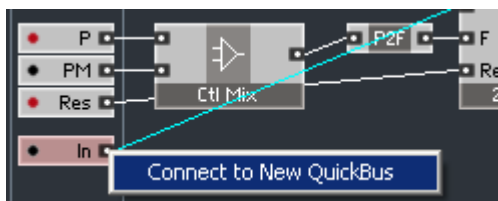
Wow, looks nice except that the audio-signal input is at the top of the core cell, while the primary-level filter module input is on the bottom. We could leave it as is, but it's easy to fix, and you already know how. Let's do it together, and we'll show you a new feature on the way.

The first thing to do is go back inside and drag the audio-signal input all the way to the bottom:

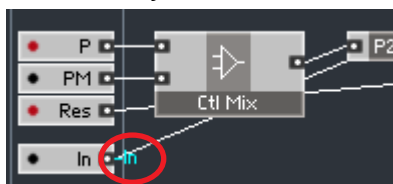


That does the trick, but that diagonal wire over the whole structure doesn't look particularly nice. That's what we are going to fix now.

Right-click on the output of the *In* input module and select the *Connect to New QuickBus* command:



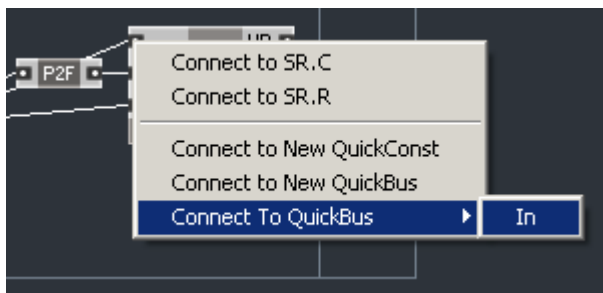
This is what you should see now:



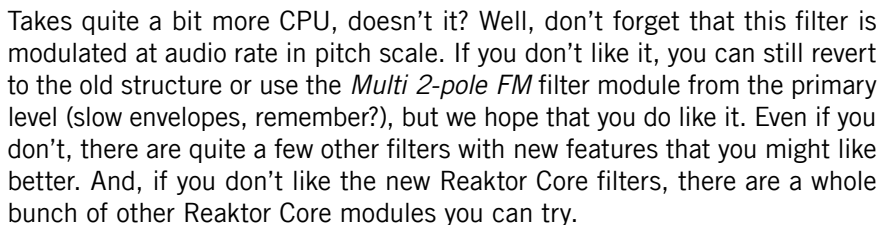
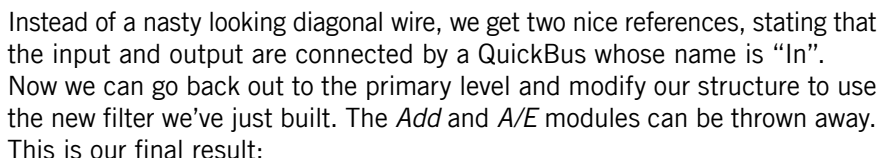
Also, the Properties window should open to display the properties of the QuickBus you've just created. The most useful QuickBus property is the ability to change its name (other properties are quite advanced, so don't touch them for now). You can open the Properties window later by double-clicking on the QuickBus.

Although you can rename this QuickBus, we believe the current name is perfect, because it matches the name of the input connected to the QuickBus. (QuickBusses are local to the given structure, so you don't need to worry about possible name conflicts when a neighboring or nested structure is using a QuickBus with the same name.)

The next thing you should do is right-click on the top input of the *2 Pole SV C* filter module and select *Connect to QuickBus > In*:



In the above menu "In" is the name of the QuickBus you are connecting to. You don't want to create a new QuickBus, you want to connect to one that already exists, and that's what you're doing. This is how your structure should look now:



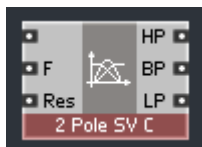
## Audio and control signals

Before we proceed we need to take a look at one particular convention used in the *Standard Macros* of the Reaktor Core library. The modules you find in that area are best described in terms of several different types of signals: audio, control, event, and logic. We will explain event and logic signals a little bit later; for now we'll concentrate on the first two types.

Audio signals are obviously signals which carry audio information. These include signals taken at the outputs of oscillators, filters, amplifiers, delays, and so on. Furthermore, modules such as filters, amplifiers, saturators, delays and the like would normally receive an incoming audio signal to process.

Control signals, on the other hand, do not carry audio, they are used to control other modules. For example, outputs of envelopes and LFOs as well as keyboard pitch and velocity signals do not carry any sound, but can be used to control a filter's cutoff or resonance, or a delay line's delay time, and so on. Correspondingly, a filter's cutoff or resonance input port, or a delay's time input port are intended to receive control signals.

Here is an example of a Reaktor Core filter module which you already know:



The upper input of the filter is for the audio signal to be filtered and, therefore, expects an audio-type signal. The *F* and *Res* inputs are obviously control type. The outputs of the filter carry different kinds of filtered audio, so all those signals are also audio type.

A sine oscillator module, on the other hand, has only a single control input (for the frequency), and a single audio output:



And if we take a look at the *Rect LFO* module, it has two control inputs – for controlling the frequency and pulse width (the third input is of event type) – and one control output (because it would be used to control things like filter cutoff or VCA levels, and so on):



---

Some types of processing, mixing for example, make sense for both audio and control types of signals. In those cases, you will find versions of such macros dedicated to processing audio and versions dedicated to processing control signals. For example, there are audio mixers and control mixers, audio amplifiers and control amplifiers, and so on. Generally it's not a very good idea to misuse a module to process signals of types it was not intended for, unless you *really* know what you're doing.

---

---

Having said that, quite often it's possible to use audio signals for control purposes. The most common example would be to modulate an oscillator's frequency or a filter's cutoff by an audio signal. That is absolutely OK because you are *intending* to use an audio signal as a control signal. We assume that the opposite case, in which you really mean to use a control signal as an audio signal, would be pretty rare.

---

The separation between audio, control, event, and logic signals is not to be confused with event/audio separation on the Reaktor primary level. The primary-level event/audio classification refers to speed of processing, audio signals being processed faster and requiring more CPU. Also as you probably know, primary-level event signals have different propagation rules than audio signals. The difference between audio, control, and event signals in Reaktor Core terminology is purely semantic, defining the meaning of the signal rather than the type of processing. There is not a one-to-one relationship between primary-level event/audio and Reaktor Core audio/control/event/logic terms, but we can still try to explain their relationship:

- a primary-level audio signal normally corresponds to either a Reaktor Core audio signal (for example, an output of an oscillator or an audio filter) or a Reaktor Core control signal (for example, an output of an envelope).
- a primary-level event signal is typically a control signal in terms of Reaktor Core. An example of such signal would be an output of an LFO, a knob, or a MIDI pitch or velocity source.
- sometimes a primary-level event signal corresponds to a Reaktor Core event signal. The most typical example of that is a MIDI gate (Reaktor Core event signals will be described later, as we promised).
- sometimes a primary event signal *resembles* a Reaktor Core logic signal; however, they are not fully compatible, and there must be explicit conversion between them (a topic that also will be covered later). Examples include signals processed by *Logic AND* or similar primary-level modules.

---

It's important to understand that when you select the type for a core-cell input port, you are choosing between the primary-level event and audio signals, not between Reaktor Core event and audio signals. The core-cell ports are the place where both worlds meet, and therefore, they use a bit of the primary-level terminology.

---

We are going to learn a little bit more about this concept while trying to build a tape-echo-effect emulation. We will start by building a simple digital echo, then enhance it to emulate some features of a tape echo.

Start by creating an empty audio core cell; then go inside and set its name to "Echo".

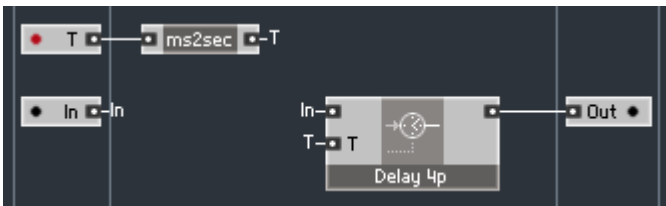
The first module we are going to put into the structure is a delay module. We will pick a 4-point interpolating delay, because it has better quality than a 2-point delay, and a non-interpolating delay would not be suitable for our tape emulation: *Standard Macro > Delay > Delay 4p*:



We obviously need an audio input and an audio output for our delay core cell. We will use a QuickBus connection for the input and a normal connection for the output:

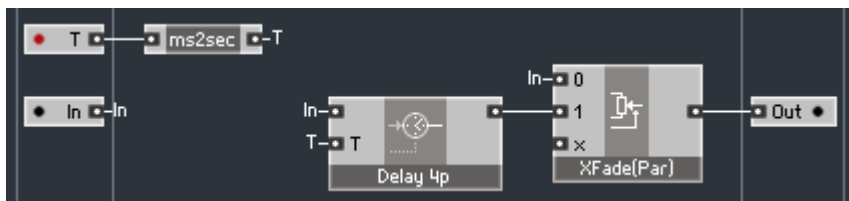


We also need an event input for controlling the delay time. One thing to be aware of here is that, on the primary level, the delay time is usually expressed in milliseconds, while Reaktor Core library delay macros expect it to be in seconds. No problem, there is a conversion module available for that. *Standard Macro > Convert > ms2sec*:

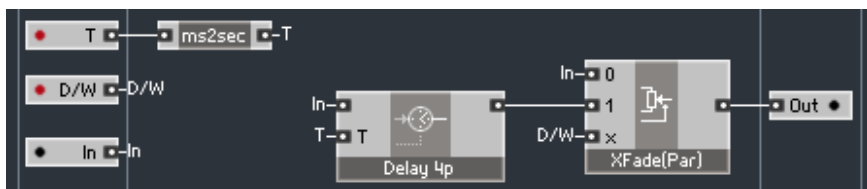


So far, we only have a single echo, and it would also be nice to hear the original signal, not just the echo. To get the original signal at the output we need to

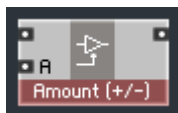
mix it with the delayed signal. *Because we are mixing audio signals here, we need to use an audio mixer* (you may remember we used a control mixer to mix control signals when we were building a filter core cell). Even better, we can use a particular audio mixer type that is specifically designed to crossfade between two signals: *Standard Macro > Audio Mix-Amp > XFade (par)*:



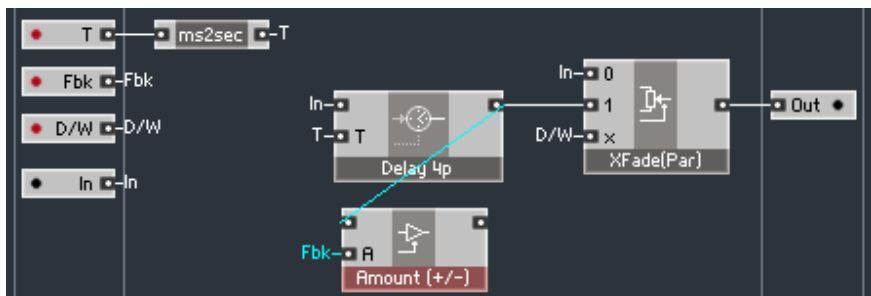
Here “(par)” stands for parabolic, which produces a more natural sounding crossfade than a linear crossfade. We will connect the control (x) input of the crossfade to a new event input to control the mix between the dry (unprocessed) and wet (delayed) signals. When the control signal is 0 we will hear only the original signal, and when it's 1, we will hear only the delayed signal:



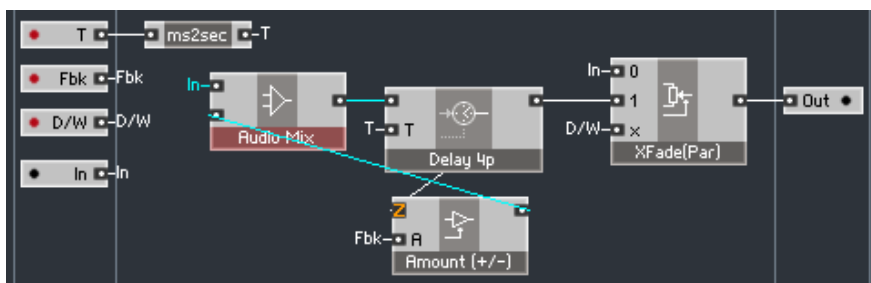
Now we can hear the original signal and the echo, but there's still only one echo. To have multiple echoes we need to feed a fraction of the delayed signal back to the delay input. First we need to attenuate the delayed signal. Following the same guidelines, use an audio amplifier to attenuate an audio signal by choosing *Standard Macro > Audio Mix-Amp > Amount*.



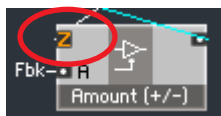
We use the *Amount* amp because we want to control the amount of the signal that is fed back. Also, this amplifier will allow us to invert the signal by using negative amount settings. In contrast, for example *Amp (dB)*, which would be quite suitable to control the signal volume, is not very good here because it doesn't allow us to invert signals. We connect the amplitude control input of the amplifier to an event input controlling the feedback amount:



The reasonable feedback amount range is something like [-0.9..0.9] here. When you try out this delay, be careful with the feedback amount, because you can easily reach excessive signal levels (there is no saturation in our circuitry yet). We could have embedded a safety feedback amount clipper into our delay core cell, but because we are going to have saturation there a little bit later, we didn't think that was necessary. Without it, you will be able to experiment with high feedback levels and hear the delay saturating.



You may wonder what happened to the upper input of the *Amount* module above, which now shows a large orange “Z”:



Actually, depending on the version of the software and other conditions, the Z sign could appear at some other input in the structure, but don't worry you too much about it. The Z sign indicates that a digital feedback has occurred in the structure, and it is meant for advanced structure design, where such information can be an important hint for the structure designer.

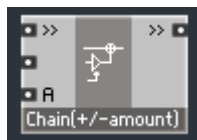
For simple structures like the one above, one normally needn't worry about the Z sign; its presence just shows that there will be a 1-sample delay (about



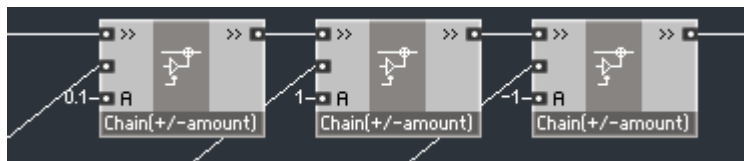
0.02ms at 44.1kHz, even less at higher sampling rates) at that point in the structure. We assume you won't notice if your delay time is 0.02ms off the specified value.

Let's get back to our structure, which now can produce a series of decaying echoes. It is already a decent digital echo, but we want to show you a feature of the library that you can use as a trick to make your structure smaller.

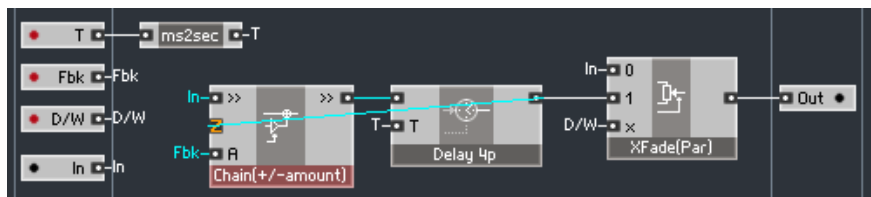
Among the audio amplifiers are amplifiers called "Chain". These amplifiers are capable of amplifying a given signal and mixing it with another, chained signal. One of them is the *Audio Mix-Amp > Chain(amount)* amplifier, which works similarly to the *Amount* amplifier except that it also does chained mixing:



The signal at the second input of this module will be attenuated according to the amount given at the A input and mixed with the signal at the chain (>>) input. The signal at the chain input is not attenuated. Such amplifiers can be used to build mixing chains, where the >> port connections constitute a mixing bus:



In our case we don't need a mixing bus, but we can use this module to replace both our *Audio Mix* and *Amount* modules. The fed back signal will be attenuated by the amount specified by the *Fbk* input and mixed to the input signal exactly as it was before:



Congratulations, you have built a simple digital-echo effect. The next step is to add some tape feel to it.

## Building your first Reaktor Core macros

In the echo effect we just built, we used a *Delay 4p* macro from the library, which gives us a reasonably high-quality digital delay. But, high-quality or not, it still sounds too digital. We will make it sound warmer by adding two features found in tape delays: saturation and flutter.

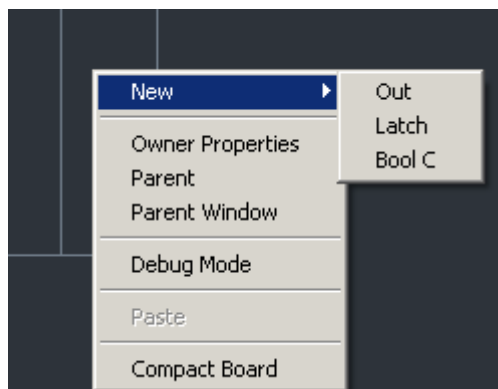
Let's start by deleting the delay macro from the structure and creating an empty macro instead. Right-click on the background as select *Built-In Module* > *Macro*:



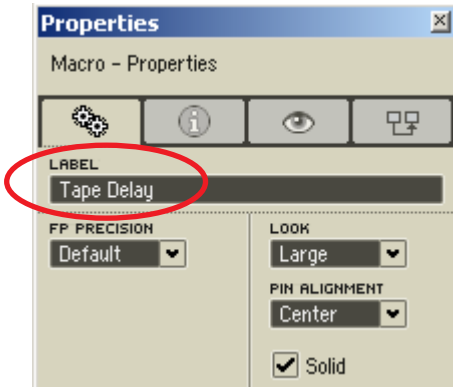
Double-click it to dive inside. You will see an empty structure, similar to the one you are diving from:



It also works similarly, but there are some important differences because the previous one was a structure of a Reaktor Core cell, whereas this one is an internal structure of a Reaktor Core macro. These differences have to do with the available input and output modules, which are different:



The *Latch* and *Bool C* types of ports will be explained much later in this manual and are used for advanced stuff. We are interested now only in the first type, which is called “Out” (or “In” for inputs). It’s a general type of port that can accept audio-, control-, event-, and logic-type signals. In fact, the port doesn’t care whether it’s audio, control, event, or logic; the difference is important only for you as a user, because it describes how the signal is to be used; for Reaktor Core they are all the same. There is also no difference between audio/event inputs/outputs as on the previous structure level, because we don’t have Reaktor primary-level signals on the outside any longer, it is pure Reaktor Core now. The first thing we are going to do is name the macro, which is done in the same way as for core cells, by right-clicking on the background, selecting *Owner Properties*, and typing in the name:



The remaining properties of the macro control various aspects of its appearance and its signal processing.

---

While you are free to experiment with remaining properties as you see fit, we strongly advise against turning the *Solid* parameter off. We also advise changing the FP Precision sparingly. The meaning of these parameters will be described in the advanced topics of this manual.

---

The next thing is to create a set of inputs and outputs for our *Tape Delay* macro:

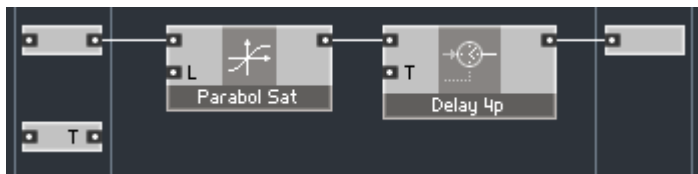


The upper input will receive the audio input, and the lower will receive the time parameter. You may have noticed extra ports on the left side of the input modules; we will explain them a little bit later.

As the central part of our macro we will use the same *Delay 4p* module:




A simple emulation of the saturation effect can be done easily by connecting a saturator module before the delay. Saturator is a kind of signal shaper, so we will look for it among the audio shapers (because it is an *audio* saturator). *Standard Macro > Audio Shaper > Parabol Sat*:

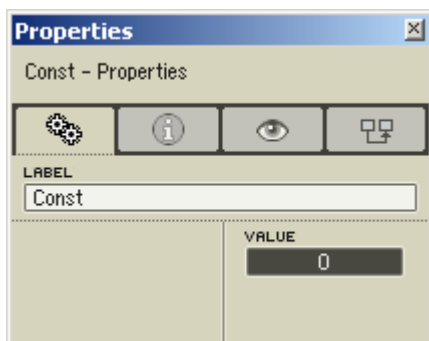


The input signal will now be saturated within the range of  $-1..+1$ . Actually, the range is controlled by the L input of the saturator module, if it is disconnected it defaults to 1. That might be surprising to you because you are probably used to disconnected inputs being treated as if they receive no signal, or put differently, a zero signal. Well, this is not exactly the case in Reaktor Core structures—modules can specify special treatment for disconnected inputs. The saturator, for example, specifies the L input to have a default value of 1.

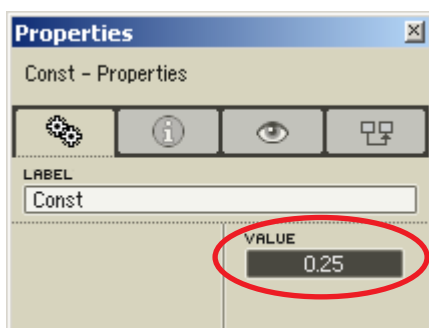
Now we are going to learn to do exactly the same, by specifying a new default value for our T input. Let's say that if our T input is disconnected we would like it to be treated as if the input value was 0.25 sec. Very easy. Right-click on the port on the left of the T input module and select *Connect to New QuickConst*. This is what you should see:



In addition, you should have the properties window displaying the properties of the constant (if it shows a different page, press the  tab):



In the value field type a new value of 0.25:



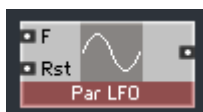
This is how the QuickConst should look now in the structure:



Let's explain what we have just done. The port on the left side of the input module specifies a so-called default signal. That means that if the input is not connected (on the outside of the macro), the default signal will be taken as the input source. In our case, if the T input of the Tape Delay macro is not connected on the outside, it will behave as if you have connected a constant value of 0.25 to it.

Of course, a connection to the QuickConst is not the only possible connection for the default signal input. You can connect it to any other module in the structure, including other input modules.

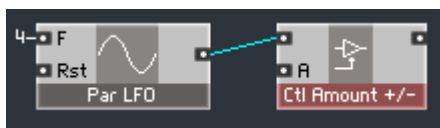
Now that we have saturation and a default value for the T input, let's emulate a tape flutter effect. A simple way to do that is to modulate the delay time with an LFO. You could experiment with different LFO shapes for better flutter effect, but for now, just take one from the library: *Standard Macro > LFO > Par LFO*:



This is a parabolic LFO, which produces a signal similar in shape to a sine, but uses less CPU. Its F input must receive a signal specifying the oscillation rate. We can use a QuickConst again here. A rate of 4 Hz seems reasonable so we can try that:



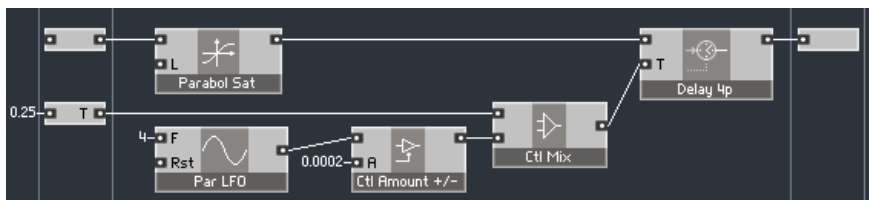
The Rst input is used for restarting the LFO; we won't need it for now. Now we need to specify a modulation amount by scaling the output of the LFO. Currently the LFO output signal varies in the range  $-1 \dots 1$  and that is way too much. Because we are dealing with control signals here, we are going to use a control amount module, which is similar to the *Amount* amplifier we used for audio. *Standard Macro > Control > Ctl Amount*:



A modulation amplitude of 0.0002 should do fine, so we scale the signal to that amount:

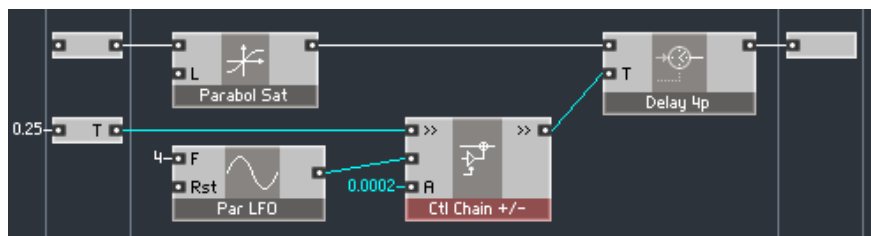


Ultimately, we can mix the two control signals (one from the T input and one from the *Ctl Amount* module) and feed them into the T input of the delay module. The already familiar *Ctl Mix* module can be used for that:




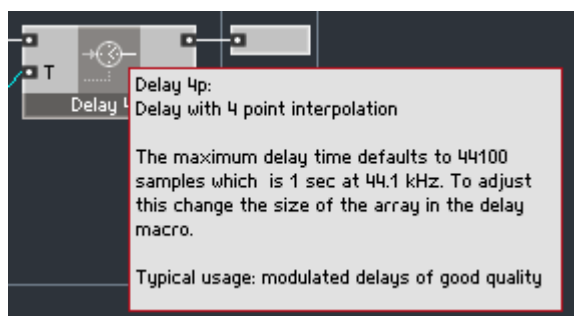
Actually, we have a Chain type of control mixer that is similar to the mixer we have for audio signals. We could use it to replace the *Ctl Amount* and

the *Ctl Mix* modules in the same way we did earlier in the audio path. *Standard Macro > Control > Ctl Chain*:

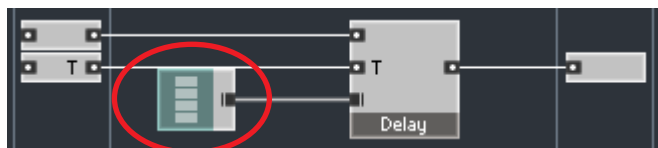



As one last touch for our macro, we are going to change the buffer size for our delay, which defines the maximum possible delay time. If you hold your

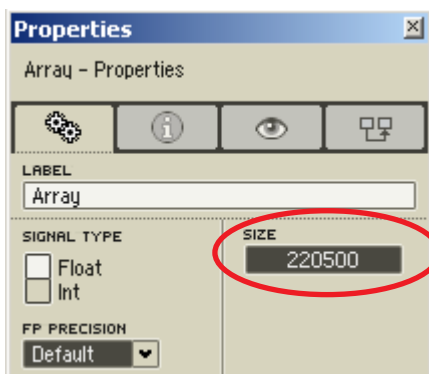
mouse cursor over the *Delay 4p* macro (and provided the  button is active), you can read in the hint text that the default buffer size corresponds to 1 sec of delay at 44.1kHz:



Let's increase the amount to 5 seconds ( $44,100 \times 5 = 220,500$  samples). Because each sample requires 4 bytes, we need  $220,500 \times 4 = 882,000$  bytes, which is almost 1MB). Double-click on the *Delay 4p* macro:

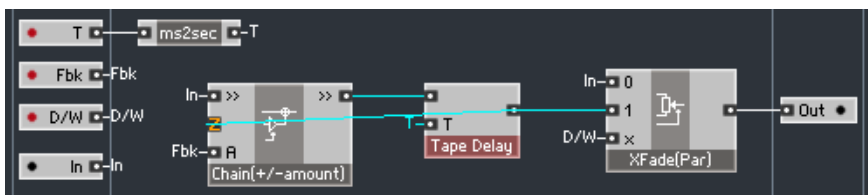


The module on the left is the delay buffer module. Double-click it (or right-click and select *Show Properties*) to edit its properties. Select the  tab and you should see the *Size* property. Change it to 220,500 samples:

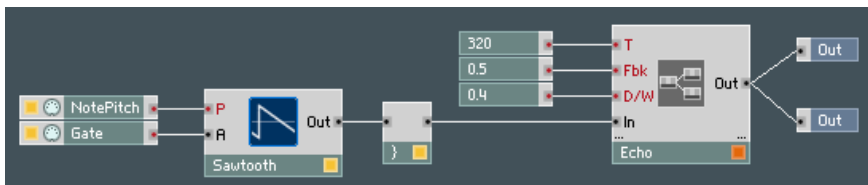


As we have seen, a delay buffer for 5 seconds of audio takes almost 1MB of memory, so be careful when changing delay buffers. That's most important when the delays are used in polyphonic areas of the structure, because the size of the buffer will be multiplied by the number of voices.

Now we can go outside the *Delay 4p* macro and then outside the *Tape Delay* macro we've just created (double-click the background) and make the outside connections:



If you haven't done so yet, try out the echo module now. Here's a Reaktor primary-level test structure, as simple as possible (note that the Echo module is set to *mono*):

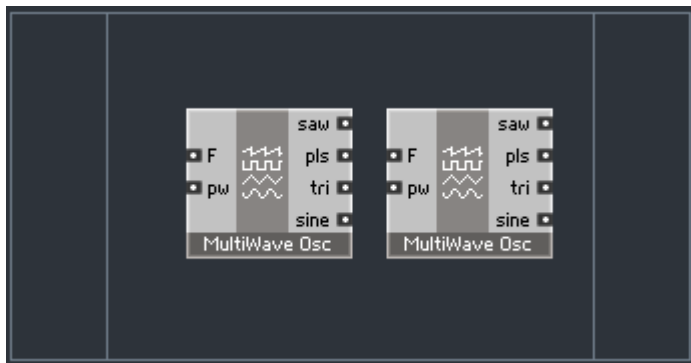


You might want to enhance it in various ways, for example, by providing knobs controlling the echo parameters, by using a real synthesizer as a signal source, and so on.

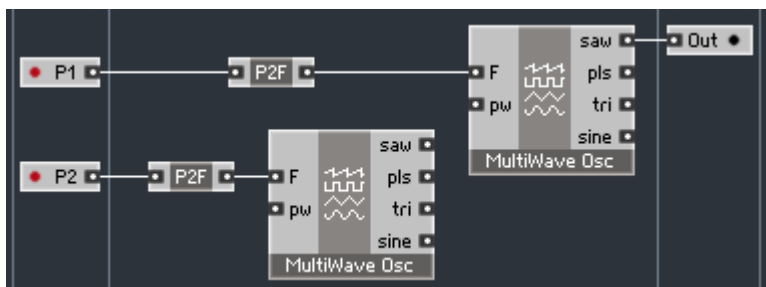


## Using audio as control signal

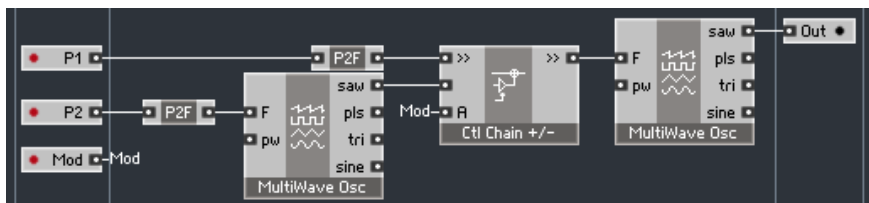
We have mentioned above that it is possible to use an audio signal as a control signal. As an example of that, we are going to create a Reaktor Core cell implementing a pair of oscillators, in which one modulates the other. Start by creating two multiwave oscillators:



We need pitch control for both of the oscillators, and we are going to listen to the output of the second one, so let's create the necessary inputs and outputs:

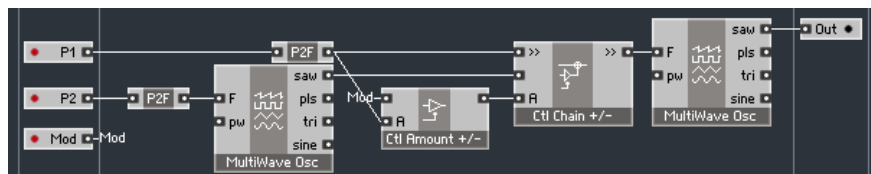


Now we want to take the output of the left oscillator and use it to modulate the frequency of the right oscillator:



The *Mod* input controls the modulation amount.

Notice that we are mixing the modulation signal with the P1 input after a P2F converter so the modulation will take place in frequency scale. (It's also possible to modulate in pitch scale.) It's also a good idea to scale the modulation amount according to the base oscillation frequency:



If you analyze the above structure from the point of view of control and audio signals you will notice that all of the signals in the structure except the outputs of the oscillators are control signals. The outputs of both oscillators are obviously audio signals. Notice, however, that we are misusing the output of the left oscillator as control signal at the point at which we feed it into the *Ctl Chain* mixer.

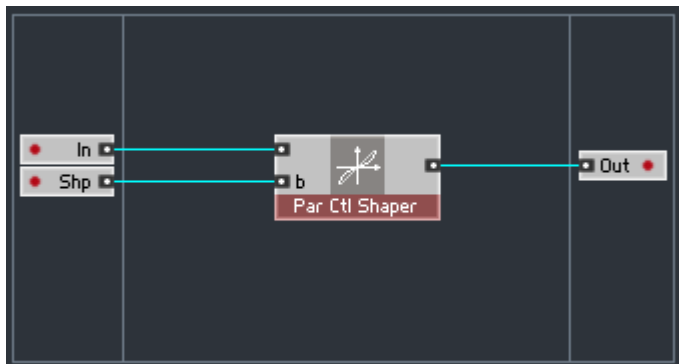
## Event signals

As we said earlier, there are different meanings of the term *event signal*. You should already be familiar with the idea of Reaktor primary-level event signals. There are several ways of using a primary-level event signal. One is as a control signal (for example, LFO output, knob output, and so on), because it uses less CPU than a primary-level audio signal. In that case, you probably could achieve the same effect with an audio signal. But, there are also cases in which an audio signal won't work for control, for instance, when you are interested in both the value of the signal and when the value is sent. A primary-level envelope-gate signal is an example of that, because the envelope will be triggered when the event arrives at the gate input.

When we were talking about audio, control, event, and logic signals in Reaktor Core we were not really talking about different types of signals (technically they are all the same in Reaktor Core). Rather we are talking about different ways of using a signal. As we now know, a Reaktor primary-level event signal can be used as a control, event, or even logic signal, and as we've seen from an earlier example, a Reaktor primary-level audio signal can be used as audio or control.

We have already learned to feed primary-level event signals into Reaktor Core structures and use them there as control signals. Event-mode inputs for an *audio* core cell implementing a filter that we built earlier is a good example of that. There are also cases in which you would use an *event* core cell to process

some primary-level event signals used as control signals. Here's an example in which an event core cell wraps a control shaper core macro:



The control shaper receives an event rate control signal from the primary level (for example, a MIDI velocity signal, or a primary-level LFO signal), bends it according to the Shp parameter, and forwards the result to the output.

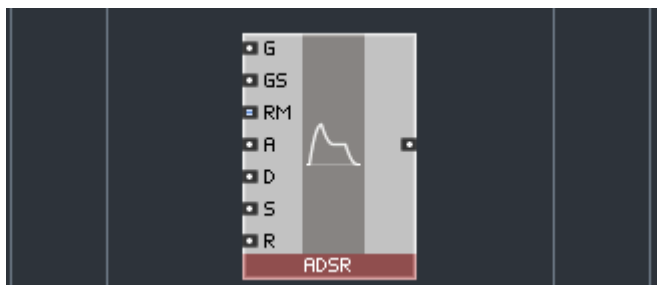
---

An important restriction of event core cells, which we mentioned earlier, is that all clock sources are disabled inside them. That means that not only oscillators and filters, but also envelopes and LFOs do not work inside event core cells. Those modules are restricted to receiving events from the primary level of Reaktor, processing them, and sending them back to the primary level, as in the above example.

---

Alternatively, signals derived from primary-level events can be used as true event signals inside Reaktor Core structures. Let's take a look at a couple of simple cases of using events inside Reaktor Core.

The first case is using an envelope in a core structure. As you can guess from the disabled-clock restriction on event core cells, this has to be an audio core cell. So, create a new audio core cell and choose *Standard Macro > Envelope > ADSR*:

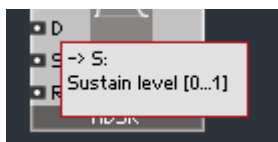


The top input of the envelope is a gate input, which works similarly to the gate inputs of primary-level envelopes—that is, it opens or closes the envelope in response to incoming *events*. For that we create an event input for our core cell:

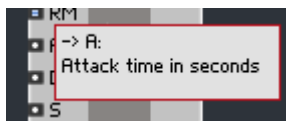


This input will translate the incoming primary-level gate events into the core events.

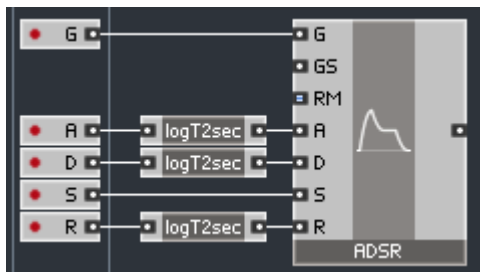
Now let's take a look at the A, D, S, R inputs. The S (sustain level) input works similarly to the primary level; it expects the incoming signal to be in the 0 to 1 range:



The A, D, R inputs are different, however. Unlike primary-level envelopes they expect time to be specified in seconds:



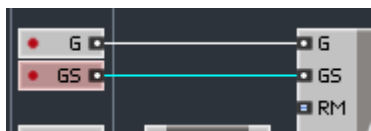
That can be solved by using a *Standard Macro > Convert > logT2sec*, which converts the primary-level envelope times to seconds:



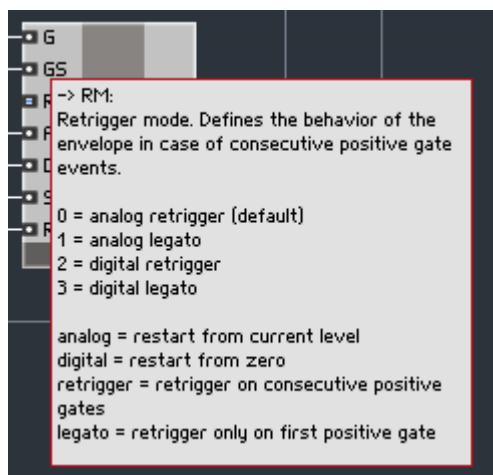
Although all inputs in the above structure are in event mode, the first input produces an event signal, whereas the others produce control signals.

Our envelope still has two unconnected ports. The GS port sets the gate sensitivity amount. At 0 the envelope completely ignores the gate level and

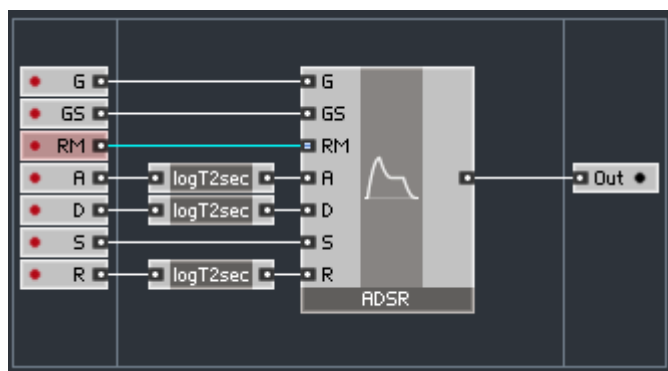
is always at full amplitude. At 1 the gate level has maximum effect, as on the Reaktor primary level. We can control this amount from the outside by adding another input:



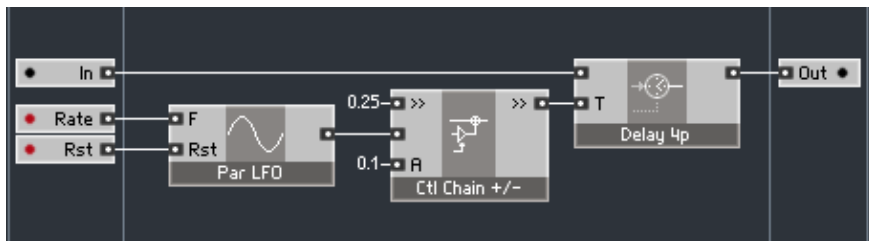
The *RM* port specifies the retrigger mode for the envelope:



The look of this port is different from the others because it *expects* integer values, but that doesn't mean we cannot connect non-integer signals to this port. We can simply use another event input, and the incoming values will be rounded to the nearest integer:



Now let's take a look at another example using a true event signal:



The above structure implements a kind of pitch modulation effect. The effect is produced by a delay whose time varies in the range  $250 \pm 100$  ms. The rate of variation is determined by the *Rate* input, which controls the rate of the modulating LFO (the value is in Hz)—that is a pure control signal. The *Rst* input is a true event signal and can be used for restarting the LFO. The incoming value specifies the restart phase, where 0 would restart the LFO at the beginning of the cycle, 0.5 in the middle, and 1 in the end. You can try it out by connecting a button to send a specific value to this input.

## Logic signals

Now that we have learned about control and event signals, it's time to learn about another way of using signals in Reaktor Core, that would be as *logic signals*. Here's an example of a module that processes logic signals:



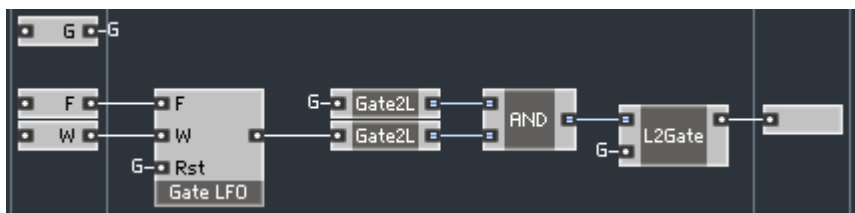
Notice that the ports of this module are integer type, just as was the *RM* input of the envelope. That is because, generally, logic signals carry only integer values; more precisely, they carry only values of 0 and 1.

For logic signals, a value of 1 stands for true, and a value of 0 stands for false. The meaning of “true” and “false” is, of course, up to the user; for instance, it could mean (as in the example here) whether a particular gate is open (true) or closed (false):



Here a *Gate2L* macro checks the incoming gate signal and produces a true (1) output if the gate is open and false (0) output if the gate is closed.

We can use logic signals to do logical processing. For example, here we've built a gate processor that applies a regular clocked gate over a MIDI gate:



The *Gate2L*, *AND*, and *L2Gate* modules are logic modules and can be found in *Standard Macro > Logic* menu. The *Gate LFO* is a macro, which we've built for this processor; it generates an opening and closing gate signal at regular intervals.

The input gate and the output of the LFO are connected to *Gate2L* converters, which convert the gate signals to logic signals, transforming open gates into *true* and closed gates into *false*. The *AND* module outputs a true signal only if both gates are in the open state at the same time. In other words the output of the *AND* module is *true* if and only if the user holds a key and at the same time the LFO outputs an open gate. That means that, as long as the user holds a key, there will be alternating *true* and *false* values at the output of the *AND* module, the speed of the alternation defined by the LFO rate. The output of the *AND* module is converted back to a gate signal, whose amplitude is taken from the gate input, thereby leaving the gate level unchanged.

Here is the structure for our *Gate LFO* macro:



The F input defines the rate of the gate repetitions, and the W input defines the duration of open gates (at 0 they are 50% of the gate period, at -1 it's 0%, and at 1 it's 100%). The Rst input restarts the LFO in response to incoming events (hence the LFO is restarted each time there's a gate event at the main gate input).

The module connected to the Rst input of the *Rect LFO* is called *Value* and can be found in *Standard Macro > Event Processing*. It ensures the LFO is restarted at zero phase by replacing the values of all incoming events by the value at its lower input, which is zero. The LFO output is converted into a gate signal by using a *Ctl2Gate* converter, also found in *Standard Macro > Event Processing*.

---

Remember, LFOs do not work inside event core cells. If you want to try out this structure, you'll need to use an audio core cell.

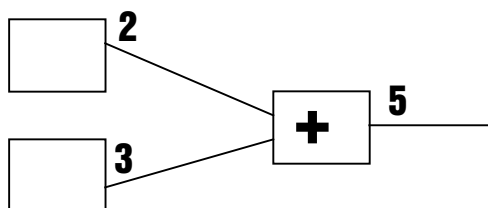
---

# Reaktor Core fundamentals: the core signal model

## Values

Most of the outputs of Reaktor Core modules produce values. (Producing a value means that at any moment in time there is a value associated with the output.) The values are available to all modules whose inputs are connected to those outputs.

In the following example an adder module gets values 2 and 3 from the two modules whose outputs are connected to its inputs, and it produces a value of 5 at its output.



---

If you want to draw an analogy to the hardware world you can think of values as signal levels (voltages), especially with relatively large-scale modules such as oscillators, filters, envelopes, and so on. However, values are not limited to those kinds of processing—they are just values and can be used to implement any processing algorithm, not just voltage-modeling algorithms.

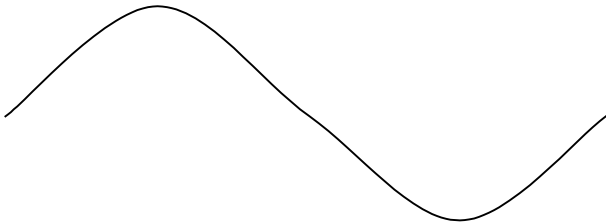
---

## Events

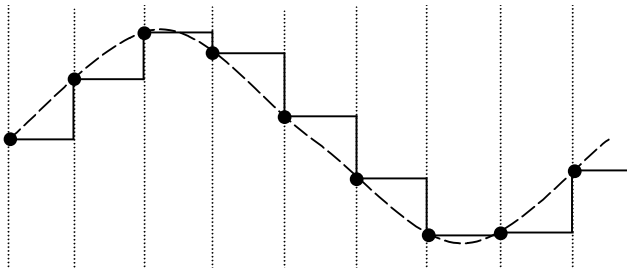
Time is not continuous in the digital world; it is discrete. Probably the most familiar example of this is that a digitally stored recording doesn't store the full information about an audio signal, which is continuously changing over time, but rather stores only information about the signal level at regularly spaced points in time. The number of points per second bears the famous name of *sampling rate*.



Here is a picture of a continuous signal:

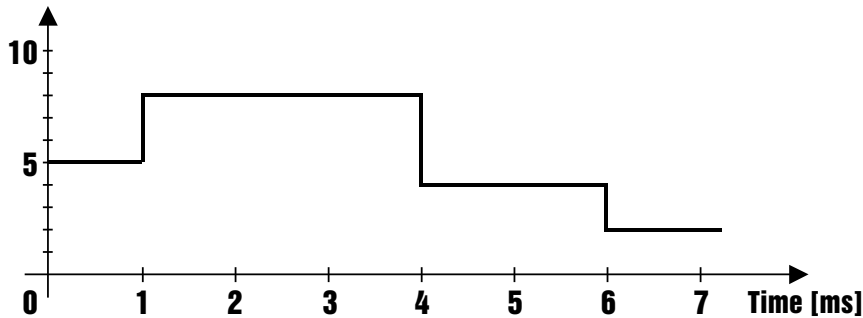


and its digital representation:



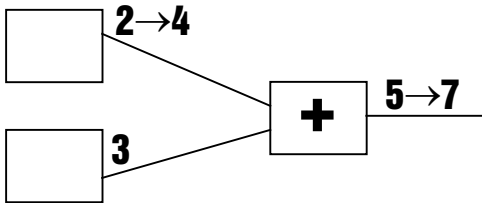
Because we *are* in the digital world, the outputs of our modules cannot change values continuously. On the other hand, we don't have to limit ourselves to changing values at regularly spaced points in time. For one thing, we do not have to maintain a particular sampling rate all over our structures. For another thing, in certain areas of our structures we do not even have to maintain any sampling rate at all; that is, our changes do not have to happen at regular intervals.

For example, at time zero the output of our adder could have a value of 5. The first change could occur at time 1 ms (one millisecond). The second change could occur at 4 ms. The third at 6 ms:

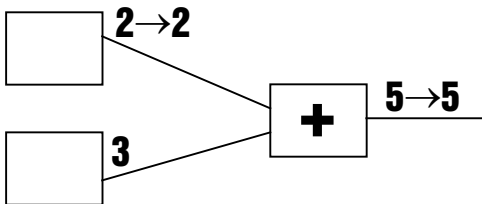


In the picture above we can see changes of the output of our adder occurring during the time from 0 to 7 ms. At the moment in time that the output changes its value, it generates an event. An *event* means that the output *reports* a change of its state, meaning that it has got a new value.

In the following example, the upper left module has changed its output value from 2 to 4, generating an event. In response, the adder module will change its output value and generate an event at its output, too.



Alternatively, the upper left module could have generated a new event with the same value as the old one. The adder would have still responded by generating a new event, but this time, without changing its output value.



---

The new value appearing at the output is not required to be different from the old one. However, the only way an output can change its value is by generating an event.

---

As you have seen from the previous examples, an event occurring at an output of some module will be sensed by *downstream* modules, which would in turn produce further events (remember the adder producing an output event in response to an incoming event). Those new events would be sensed by the modules connected to the corresponding outputs and propagated further downstream, until the propagation stops for one of the reasons discussed later in this text.

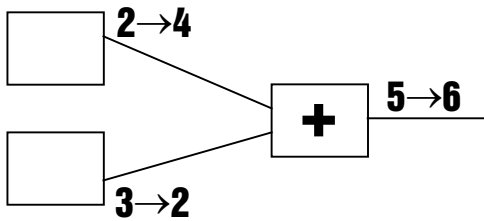
---

Events in Reaktor Core are not the same as events on the Reaktor primary level. They behave according to different rules, which will be explained below.

---

## Simultaneous events

Consider the situation in which the two modules on the left side in the previous examples *simultaneously* produce an event.



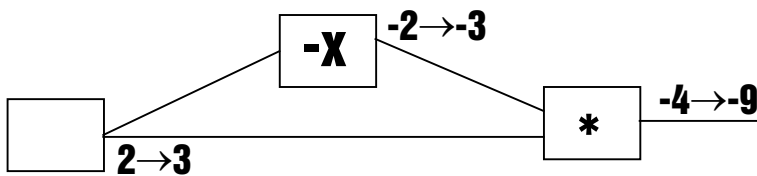
This is one of the key features of the Reaktor Core event model—events can occur simultaneously at several places. In that situation, the events originating at both the left-side modules will arrive at the inputs of the adder simultaneously, and most importantly, the adder will produce exactly *one* output event in response.

---

That is not the same as on the Reaktor primary level, where events cannot happen simultaneously, and the *Add* module (in event mode) would produce two output events in such a situation.

---

Of course, in reality, the events are not produced simultaneously by the upper-left and the lower-left modules, because both modules are being processed by the same CPU, and the CPU can process only one module at a time. But, what is important for us, is that these events are *logically* simultaneous, that is they are treated as simultaneous by the modules receiving them. Here is another example of simultaneous event propagation:



In the example above, the leftmost module is sending an event, changing its output value from 2 to 3. The event is sent *simultaneously* to both the inverter (**-x**) and the multiplier (**\***) modules. In response to the incoming event the inverter will produce a new output value **-3**. It is important to notice that although the output event of the inverter was produced in response to the event sent by the leftmost module, and as such should happen later than the

incoming event, *both events are still logically simultaneous*. That means they simultaneously arrive at the inputs of the multiplier, and the multiplier again produces only one output event, with a value of  $-9$ .

---

Again, on the primary level you would have had two events at the output of the *Event Mult* module. It is also not defined whether the event at the output of the leftmost module would have been sent first to the inverter or to the multiplier (although that is irrelevant for the given structure).

---

In general you can use the following rule to figure out whether two events are simultaneous or not:

---

All events originating from (sent in response to) the same event are simultaneous. All events originating from an arbitrary number of simultaneous events (occurring at different outputs, but known to be simultaneous) are also simultaneous.

---

The last example shows the benefit of having simultaneous events. In that case, we eliminate the redundant processing of the second event by the multiplier, which would have taken extra CPU time. In longer structures, in the absence of simultaneous events, the number of events can grow uncontrollably unless the structure designer pays particular attention to keeping the number of duplicate events low.

In addition to saving CPU time, the concept of simultaneity leads to important differences in one's approach to structure construction, especially for the structures implementing low-level DSP algorithms. You will become more familiar with these differences as you start constructing your own structures.

## Processing order

As you have seen from the previous examples, when a module sends an event, the downstream modules respond to that event. From that, one might conclude that, despite producing logically simultaneous events, the modules are definitely not processed simultaneously. One might further conclude that, for a given connection, it would be reasonable to process the upstream module of the connection before the downstream module of the connection. All those conclusions are, in fact, correct.

The general rule of processing order of the modules is:

---

If two connected modules are processing logically simultaneous events, then the upstream module will be processed first. If the events are not simultaneous, then of course, the order of processing for the modules is the order of the processed events.

---

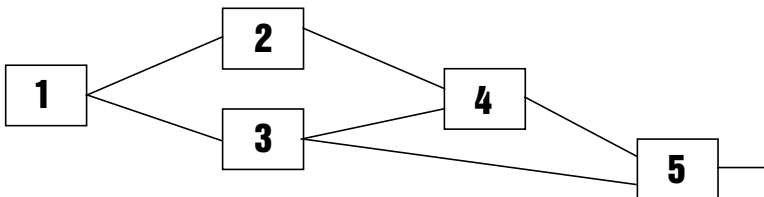
From the above rule it follows that as long as there is a one-direction connection path (always upstream or always downstream) between two modules, then there is a defined processing order for these two modules: the upstream module is processed first.

---

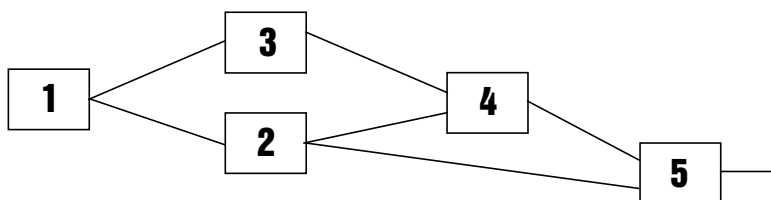
If there is no one-direction connection path between two modules, their processing order relative to each other is undefined for logically simultaneous events. That means that the order is arbitrary and can change as a result of various actions. The structure designer must take care that such situations occur only for modules whose relative processing order is unimportant. That is normally automatically the case as long as no OBC connections (see below) are involved.

---

Here is an example, the digits showing the order of module processing:



For the above structure, there is an alternative valid processing order:

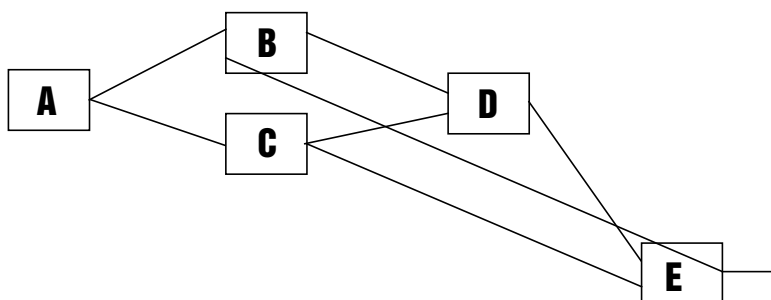


There is no way to tell which one will be taken by the software. Fortunately, as long as you do not use OBC connections, the relative order of modules in such cases is really unimportant.

---

The above rules for processing order cannot be applied if there is feedback in the structures, because in that case, for any pair of modules in the feedback loop we cannot tell which one is upstream to the other. The problem of handling feedback loops, including the processing order, will be addressed later.

---



For the above structure, it is not possible to define whether, for example, module B is upstream to module D or vice versa, because there is an upstream connection going from D to B as well as an upstream connection going from B to D (via E).

## Event core cells reviewed

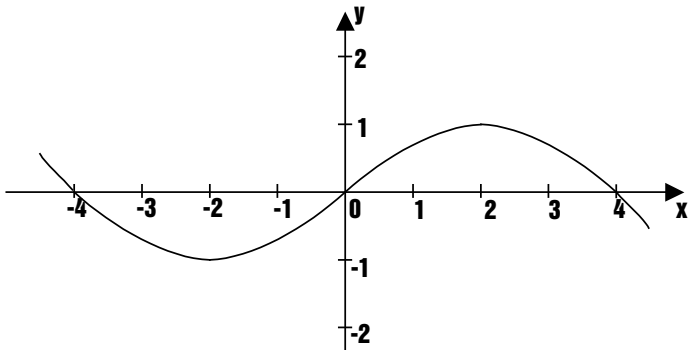
Let's take a look at event core cells from the point of view of the just described event concept of Reaktor Core.

As you'll remember, event core cells have event inputs and event outputs. These inputs and outputs are the interface between Reaktor's primary level and the Reaktor Core level; they perform the conversion between primary-level events and core events, and vice versa. The rules of the conversion are as follows:

**Event Inputs** send core events to the inside of the structure in response to primary-level events coming from outside. Because the outside primary-level events cannot arrive simultaneously at the inputs, the internally produced events also do not occur simultaneously.

**Event Outputs** send primary-level events to the outside of the structure in response to core events coming from the inside. Although core events can occur simultaneously at several outputs, primary-level events cannot be sent simultaneously. Therefore, for simultaneous core events, the corresponding primary-level events will be sent one after another, *with upper outputs always sending before lower ones*.

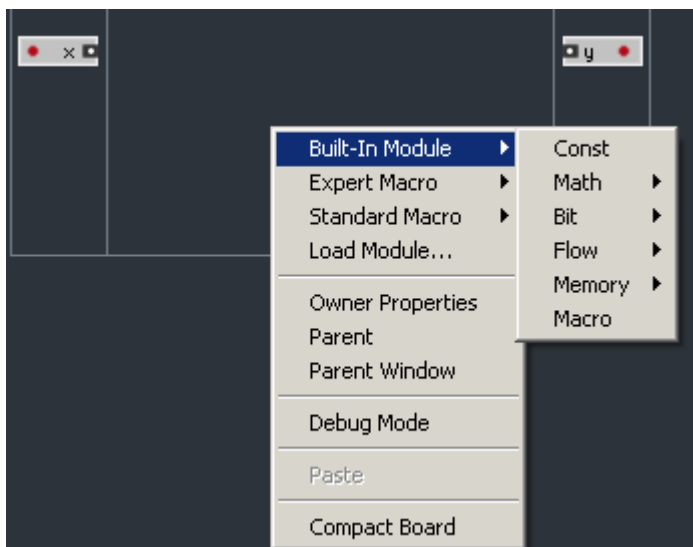
Let's try that in practice by building an event processing module that performs signal shaping according to the formula:  $y = 0.25 * x * (4 - |x|)$   
The graph of this function looks as follows:



Let's start by creating a new event core cell with one input and one output, labeled "x" and "y", respectively.



Now let's create the structure which computes the formula. We need to create **|x|** (absolute value), **-** (subtract), and **two \*** (multiply) modules in the normal area. These are not core macros, but rather true Reaktor Core built-in modules. To insert built-in modules into core structures, right-click in the background of the normal area and select the *Built-In Module* submenu:

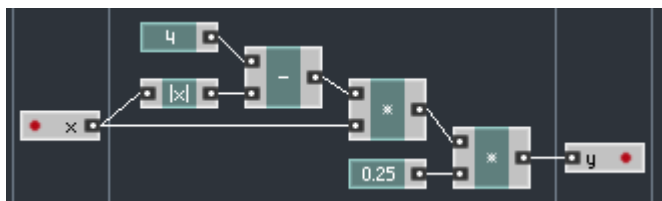


You'll find all the necessary modules in the *Built-In Module* > *Math* sub-menu:



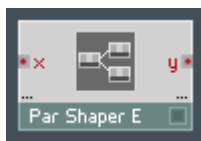
We'll need two constant values: 0.25 and 4. We could use QuickConsts exactly like we did earlier, but we can also insert real constant modules: *Built-In Module* > *Const* (as with the QuickConst, their values can be specified in the Properties window):



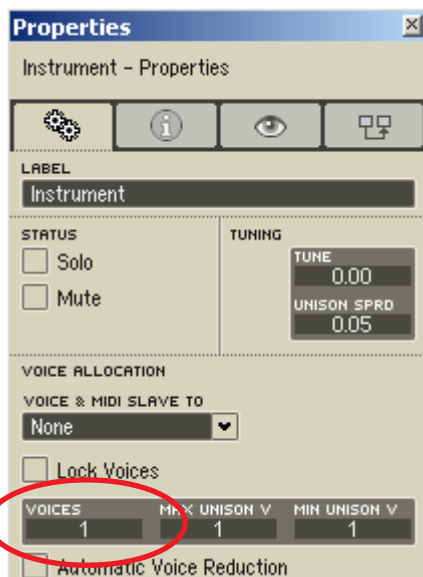


Of course, in this particular case there is no benefit in using *Const* modules instead of QuickConsts, but sometimes you might want to. For example, if the same constant has to be connected to multiple inputs, it may be better to use a *Const* module, because then you need only one of them and you also have a single place to edit the value.

The above structure now shapes the signal in the way described, but as we'll see at the end of this section, the implementation is not perfect. For now, let's give our module a name and go back to the primary level:

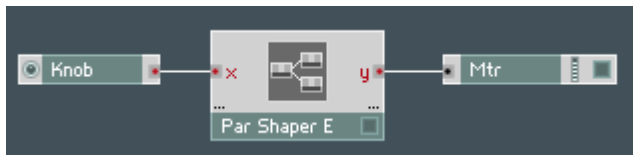


Now let's test it. Set the number of voices for the Reaktor instrument to 1, so that it will be easier to use a Meter module:

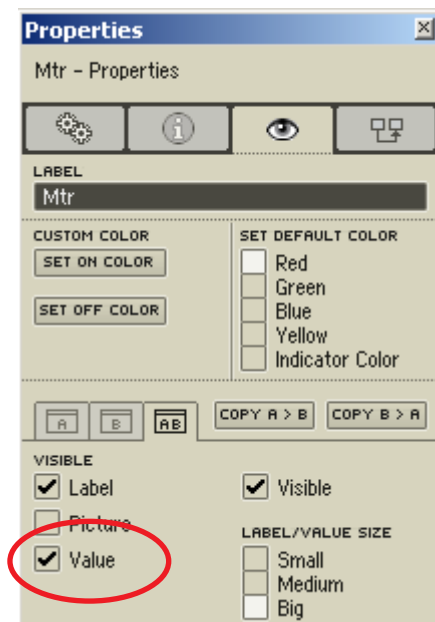


Create a Knob and a Meter and connect them to the input and output of your

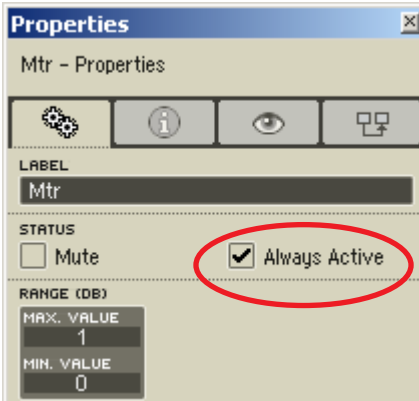
module:



Set up the properties for the knob and the meter. Don't forget to set the meter to display its value:



and to check the Always Active box:



Now move the knob and watch the output value change.



The event-shaper structure we've built should work perfectly for shaping control signals, but it still has one minor flaw in its event-processing behavior. We will return to that problem and fix it a little bit later.

# Structures with internal state

## Clock signals

How a Reaktor Core module processes an incoming event is completely up to the module. Normally a module would process the incoming value in some way, but it can also completely ignore it. The most typical case of such processing is *clock inputs*.

One example of a module with a clock input is a *Latch*. The *Latch* is not a built-in module; it's a macro; nevertheless, it's perfect for demonstrating the clock principle.

The *Latch* has two inputs – one for the value and one for the clock.



The value input (the upper one) will store the incoming value to the internal memory of the latch in response to an incoming event; nothing will be sent to the output. The clock input will send the last stored value to the output in response to an incoming event.

---

Clock inputs (unless otherwise specified) completely ignore the value of the incoming event and respond only to the fact that the event is coming.

---

(Because now we are discussing clock signals, not latches, examples of using the *Latch* module will come later.)

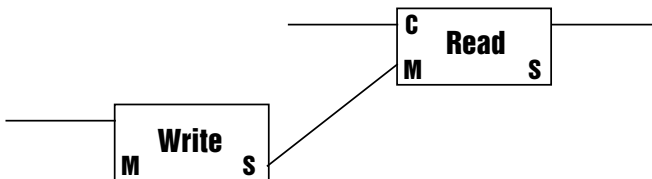
Because there are modules with clock inputs, it should be clear that some of the signals in the structure do not carry any used (or, for that matter, useful) values. Some signals can even be produced for the sole purpose of being used as a clock source. We will call them *clock signals*.

A sampling-rate clock is one example of a clock signal. It produces an event for each new audio sample to be generated, so at 44.1 kHz sampling rate it would tick 44,100 times per second. The value of the signal has no meaning, is not intended to be used in any way, and is (in the current implementation) always zero.

## Object Bus Connections

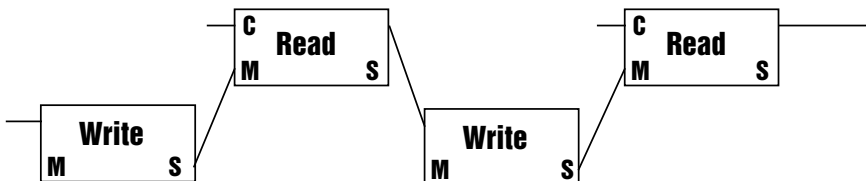
*Object Bus Connections* (OBC) are a special type of connection between modules. An OBC connection between two modules *declares* that they share some internal object. The most typical case of modules using OBC connections are memory *Read* and *Write* modules, which would share a common memory if connected by an OBC.

The functionality of the *Write* module is to write a value that is incoming at its input, to the OBC-shared memory. The functionality of the *Read* module is to read a value from the OBC-shared memory in response to an incoming clock signal (*C* input). The read value is sent to the output of the *Read* module.



The above structure implements the functionality of the Latch macro (in fact, it *is* the internal structure of the Latch macro). The *M* and *S* pins of *Read* and *Write* modules are pins of *Latch OBC type*. The *M* pin is the master connection input, the *S* pin is the slave connection output. The master input of the *Read* module is connected to the slave output of the *Write* module (the other two master and slave pins are unused). Therefore in this structure the *Write* and *Read* modules share the common memory.

In the next structure, there are two pairs of *Write* and *Read* modules. Each pair has its own memory. Notice that the connection in the middle (from the output of *Read* to the input of *Write*) is not an OBC connection.



One could ask what the difference is between master and slave. From the point of view of owning the shared object (in this case memory), there is no difference. However, as you may remember from a previous section of this manual, there is a rule that upstream modules are processed before downstream modules when processing simultaneous events. Therefore, in the two

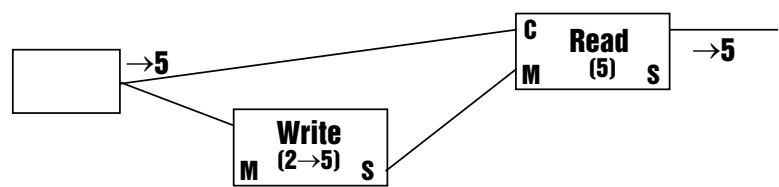
last examples the *Write* modules will be processed before their slave *Read* modules, which is obviously not the same as the reverse.

---

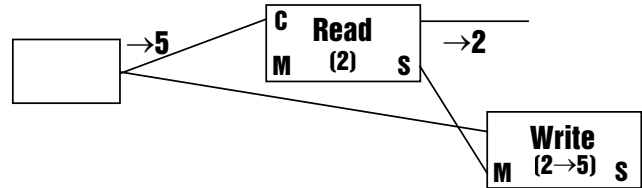
The relative order of processing of OBC connected modules is defined using the same rules as for other modules: upstream modules are processed first.

---

Indeed, let's consider two different cases. In both cases, the original state of the memory will be 2, and the same event of value 5 will be sent to both the *Write* and *Read* modules. In one case, the *Write* module will be the master and in the other case the *Read* will be the master.



Above we have the structure for the first case. The module on the left side sends an event of value 5, which first arrives at the Write module, causing it to write the new value of 5 into the memory shared by the *Write* and *Read* modules. Next, the event arrives at the Read module, working as a clock event and triggering the read operation, which in turn reads the recently stored value of 5 and sends it to the output. That is the functionality provided by the *Latch* macro in the Reaktor Core macro library. Now consider the second structure:



Here we have the opposite situation. First, the clock event arrives at the *Read* module, sending the stored value of 2 to the output. Only after that does the event arrive at the input of the *Write* module, changing the stored value to 5. This structure implements the functionality of a  $Z^{-1}$  block (one sample delay), widely used in DSP theory. Indeed, the output value is always one step behind the input value here.

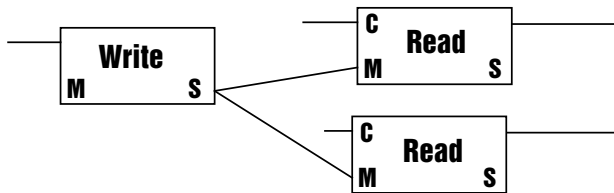
---

As mentioned, the above structure implements the  $Z^{-1}$  functionality. However, before you can really build or use such structures yourself, there are a few other important things you have to know, so please read on.

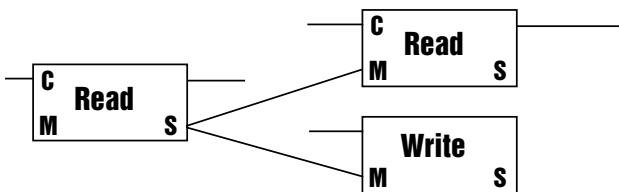
---

When there are more than two modules connected by OBC wires, they all share the same object. Then it becomes very important to know whether the order of specific read and write operations is important, and if so, what that order should be.

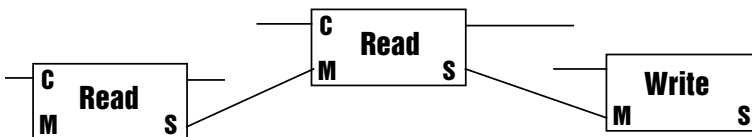
For example, in the following structure the relative order of the two read operations is undefined, but they both happen after the write operation, so it should be completely OK:



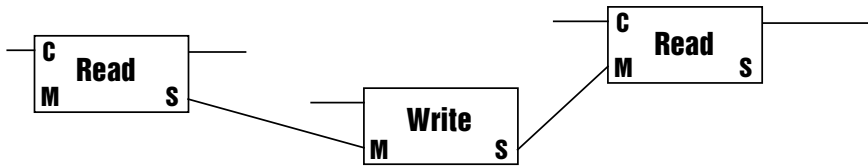
In the next structure, the relative order of the write operation and the second read operation is undefined. That can be a potentially dangerous structure and generally has to be avoided:



A better way to realize the above structure is possibly this one:



Or this one:



Even when it appears that the relative order of read and write operations is irrelevant, it doesn't hurt to impose a particular order, and it's a little bit safer.

---

The relative order of write operations is important. The relative order of read operations does not matter, as long as their order relative to the write operations remains defined.

---



---

OBC connections are not compatible with normal signal connections. Furthermore, OBC connections corresponding to different types of objects (for example, different floating point precision of memory storage) are not compatible with each other. Pins of incompatible types cannot be connected; for example, you cannot connect a normal signal output to an OBC input.

---

## Initialization

As we are starting to work with objects that have an internal state (in case of *Read* and *Write*, the shared memory of the objects is their internal state), it becomes important to understand what the *initial state* of the structure you've built is. For example if we are going to read a value from memory (using a *Read* module) before anything is written to it, what value will be read? And, if we don't like the default value, how can we change it?

Those questions are addressed by the initialization mechanism of Reaktor Core. The initialization of core structures is performed in the following way:

- first, all state elements are initialized to some default values, usually zeroes. Particularly all shared memory and all output values of the modules will be set to zeroes, unless explicitly specified otherwise in the documentation
- second, an *initialization event* is sent *simultaneously* from all *initialization sources*. The initialization sources include most of the modules that do not have an input: Const modules (including QuickConsts), core cell inputs (typically), and some others. The sources would normally send their initial values during an initialization event; for example,



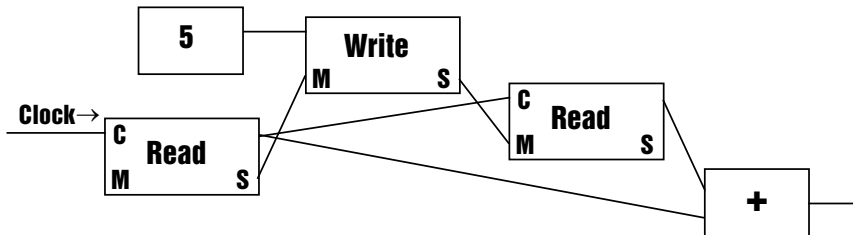
constants would send their values and core cell inputs would send the initial values received from the primary level structure outside.

---

If a module is an initialization event source, you will find information about initialization in the module reference section for the module. If a module is not an initialization source, it treats the initialization event exactly like any other incoming event. *Mostly* initialization sources are those and only those modules that do not have inputs.

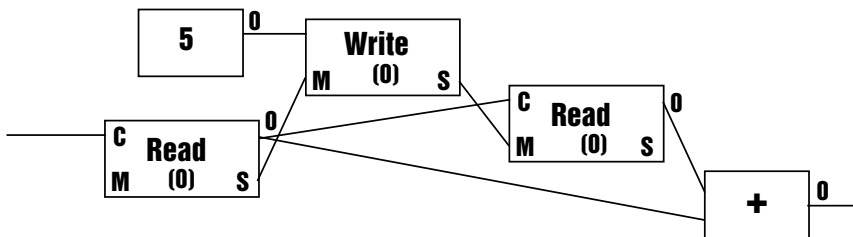
---

Here's a look at how initialization works:

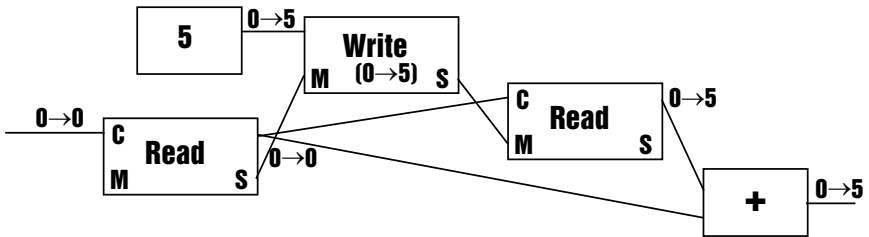


This is a part of the structure; the Read module on the left is connected to some clock source, which also sends an initialization event (as clock sources typically do).

Initially, all signal outputs and the internal state of Read-Write-Read chain are set to zero.



Then an initialization event is sent simultaneously from the clock source and from the constant 5.

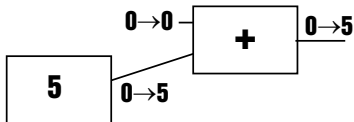


The *Read* module on the left is processed before the *Write* module and therefore the clock event arrives there before the new value is written into the memory, so the output of this module is zero. Then the value is written into the memory by the *Write* module. Now the second *Read* module is triggered, producing a value of 5 at the output. Lastly the adder module is processed, producing a sum of 5.

---

As you remember, disconnected inputs are treated in Reaktor Core as zero values (unless otherwise specified by a particular module). More precisely, they are treated as zero constants. That means that these inputs also receive the initialization event, exactly as if a real constant module with zero value were connected there.

---



Above, an adder with one input disconnected and one connected to a constant module receives two simultaneous initialization events, one from the default zero constant connection and one from a real connection to a constant.

---

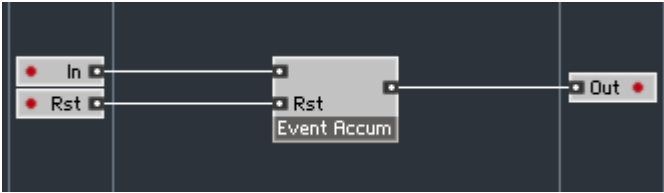
There can also be special meaning for disconnected inputs that are not signal inputs (obviously they cannot be connected to a zero constant). For example a disconnected master input of a *Write* module means that the shared memory chain starts there and continues to the modules connected to the slave output.

---

## Building an event accumulator

The event accumulator module that we want to build now is going to have two inputs: one for the event values to be accumulated, and one for resetting the accumulator to zero. There is also going to be one output, which outputs the sum of the accumulated events.

We are going to build this module in the form of a core macro, which would be easy to use inside an event core cell:



This is what the inside of our macro looks like:

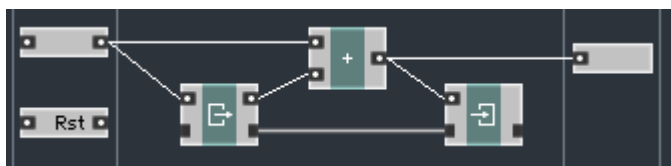


Obviously the accumulator module needs to have an internal state where it's going to store its current accumulated value. We are going to use *Read* and *Write* modules to build the accumulator loop. They can be found in the *Built-In Module > Memory* submenu:



The module which you see on the left (with an arrow pointing outwards) is the *Read* module and the module on the right (arrow pointing inwards) is the *Write* module.

In response to an incoming event, the accumulator loop should take the current value and add the new value to it. Therefore, we have to use a *Read* module to retrieve the current state, use an adder to add the new value, and use a *Write* module to store the sum.



Note that the *Read* module is clocked by the incoming event and, of course, that its OBC-connected *Write* module is located downstream, because we want to write after we read.

The above structure works in the sense that it accumulates incoming values and outputs their total at its output. What is missing is reset functionality and circuitry to ensure the correct initial state.

Let's build the resetting circuitry first. Because we are within the Reaktor Core world, the In input and the Rst input can send events simultaneously, and if we want this to be a generally usable core macro, we need to take that into account. Let's assume that the In and Rst inputs simultaneously produce an event. What do we want to happen? Is the reset logically supposed to happen before the accumulated event is processed or after? (This is very similar to the difference between the *Latch* and the  $Z^{-1}$  functionality, which differ only in relative processing order for the signal and clock inputs).

We suggest taking the Latch approach, because that module is very widely used in Reaktor Core structures, and therefore such behavior would be more intuitive. In a Latch, the clock signal logically arrives later than the value signal. In our case, the reset signal should arrive logically after the accumulated signal (forcing the state and the output to zero). Therefore, we need to somehow override the accumulator output with an initial value. To achieve that we will need to use a new concept, which we are about to discuss.

## Event merging

You have seen various ways of combining two different signals in Reaktor Core, including arithmetic operations and other ways. What has been missing is a way to simply merge two signals.

Merging is not adding. Merging means that the result of the operation is the last incoming value, rather than the sum of all incoming values. To merge signals you need to use the *Merge* module. Let's take a look at how it works.

Imagine we have a *Merge* module with two inputs. The initial output value (before the initialization event) is, as for most of the modules, zero:



Now an event with a value of 4 arrives at the second input of the module:



The event goes through the module and appears at the output. Now the output of the merge has a value of 4.

Then another even with a value of 5 arrives at the first input:



The event goes through the module and appears at the output, which changes its value to 5.

Now two events with values of 2 and 8 arrive simultaneously at both inputs.



Here we have a special rule for the *Merge* module:

---

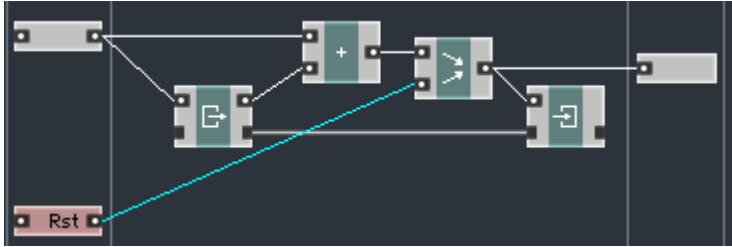
Events arriving simultaneously at the inputs of a Merge module are processed in the order of the input numbering. Still there is only one output event generated, because a Reaktor Core output cannot produce several simultaneous events.

---

In the above case this means that the event at the second input will be processed after the first event, overriding the value of 2 by the value of 8, which then appears at the output.

## Event accumulator with reset and initialization

So, in order to achieve the desired reset functionality we need to override the adder output by some initial value. To do this we can use a *Merge* module (found in the *Built-In Module > Flow* submenu). The simplest way is to connect the second input of the merge module to the Rst input.



Now the reset event will be immediately sent to the Merge module, overriding the adder output, should the accumulated event arrive at the same time. From there it goes to the output and into the internal state of the accumulator.

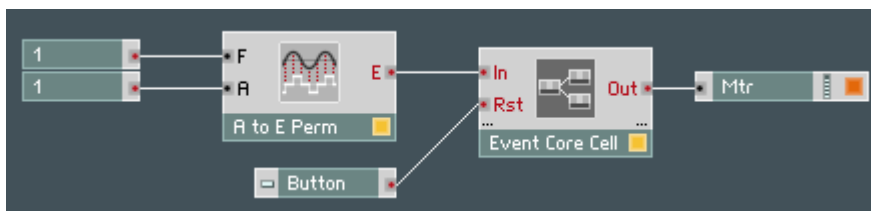
In the above structure, the value occurring at the Rst input will be used as the new value of the accumulator. Maybe it's even not such a bad idea, but then it's not exactly a reset function, but rather a set function, as implemented in the standard Reaktor event accumulator module. If we want to have a true reset function we should write only zero values into the state, regardless of the value appearing at the Rst input. So what we have to do is to send a zero value to the *Write* module each time an event occurs at the Rst input.

Sending an event with a particular value in response to an incoming event is a quite common operation in Reaktor Core, and we suggest using the *Latch* library macro for that. *Expert Macro > Memory > Latch*:

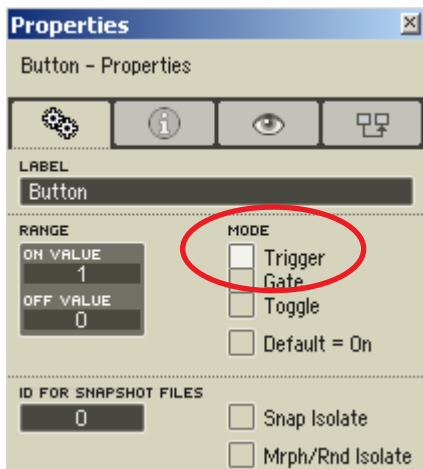


As we have already described, the Latch module has a value input (top) and a clock input (bottom). We need to connect the Rst input to the clock input of the latch to trigger the sending of an event to the output of the latch, and we also need to connect a zero constant to the value input of the latch, because we want the output events to always be zero. Or we can remember that disconnected inputs are considered to be zero constants (unless otherwise specified), and we can leave the value input of the latch disconnected:





The instrument number of voices should be set to 1, and the meter should be set to display a value and to be always active, as in the previous example. The button should be set to the trigger mode.



Now switch to the Panel and see the values incrementing in steps of 1 each second and resetting in response to pressing the button.

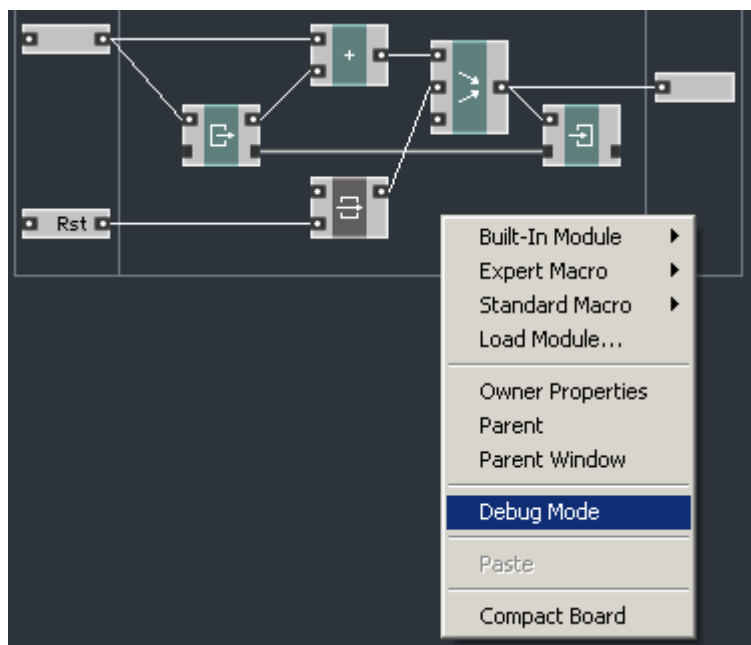



We are going to use this opportunity to introduce Reaktor Core's debug mode. As you've probably already noticed, unlike on Reaktor's primary level, where you can see the value at the output of a module if you keep your mouse cursor over the output, output values don't appear under the cursor in Reaktor Core structures. That is an unfortunate side effect of Reaktor Core's internal optimization—values from Reaktor Core structures are typically unavailable on the outside.

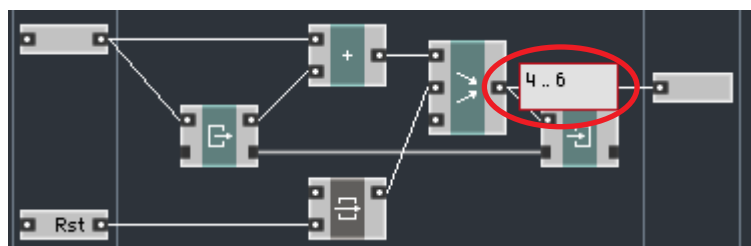
Ok, we already hear you complaining, and we've provided a compromise. You can disable the optimization for a particular core structure in order to see the




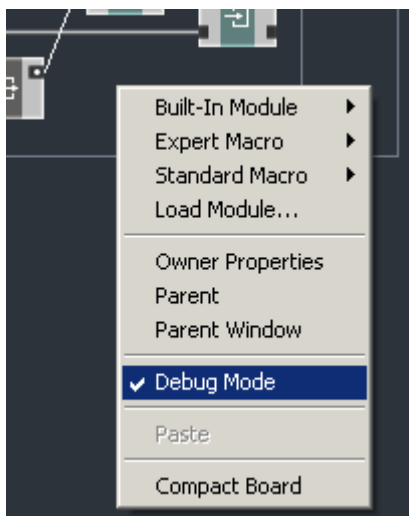
output values. Let's try that with the structure we've just built. Right-click on the background and select *Debug Mode*:



(You can do the same thing using the  button on the toolbar). Now if you keep your cursor over a particular output, you will see its value (or range of values):

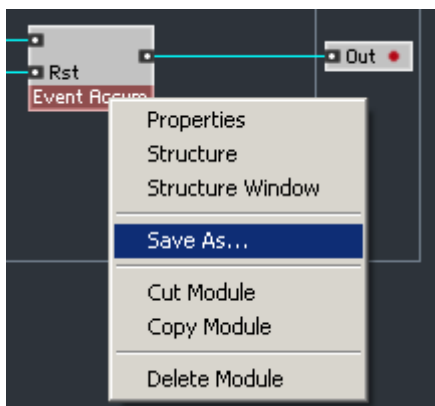


You can disable debug mode by selecting the same command (or pressing the  button) again:

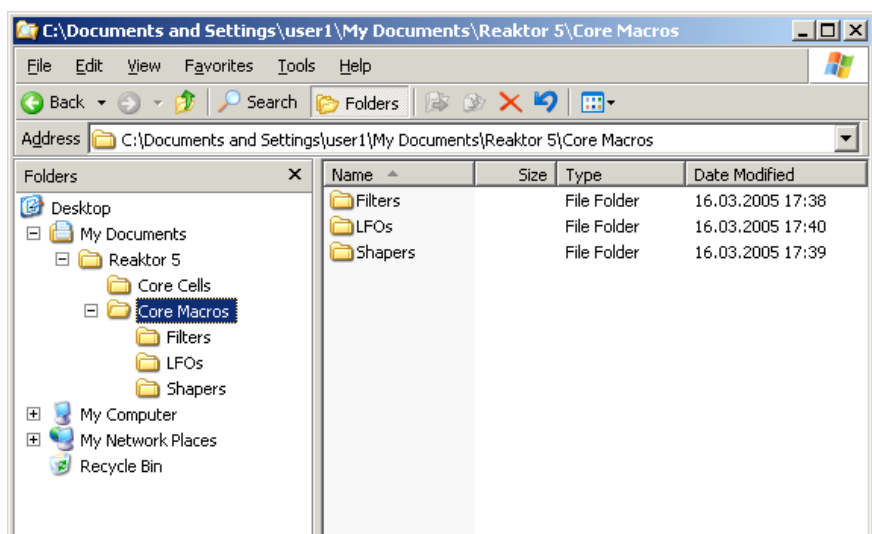


Also, it will be automatically turned off when you leave the structure, so you may need to enable it again for another structure.

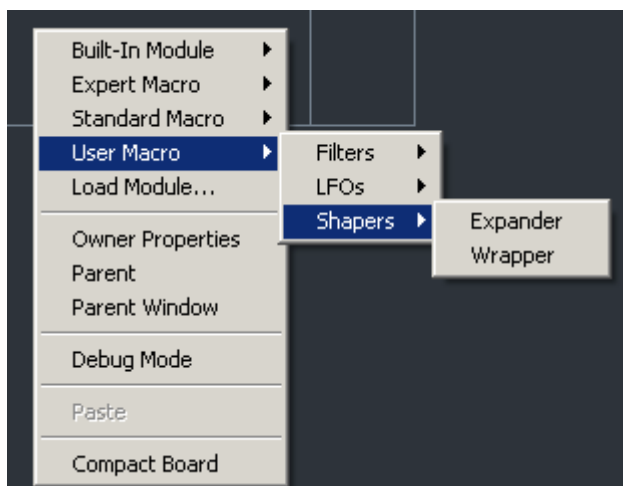
After debugging our core macro we might consider saving it as a separate file for future use. That can be done by right-clicking on the macro and selecting *Save As...*:



As with core cells you have the option of having your own macros in the menu. The macros have to be put into the *Core Macros* subfolder of the Reaktor user library folder:



Should any files be found in this *Core Macros* folder or its subfolders a new submenu appears in the right-click menu:

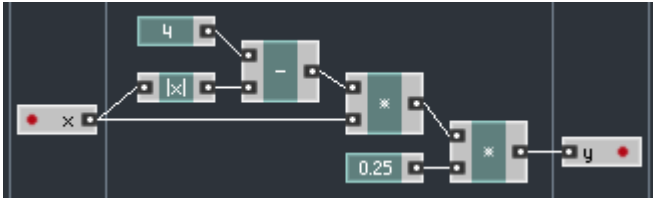


Similar restrictions apply to the *Core Macros* folder as apply to the *Core Cells* folder:

- empty folders are not displayed in the menu
- never put your own files into the system library, put them into your user library folder

## Fixing the event shaper

We can now discuss in more detail exactly what's wrong with the event shaper module structure we built earlier:



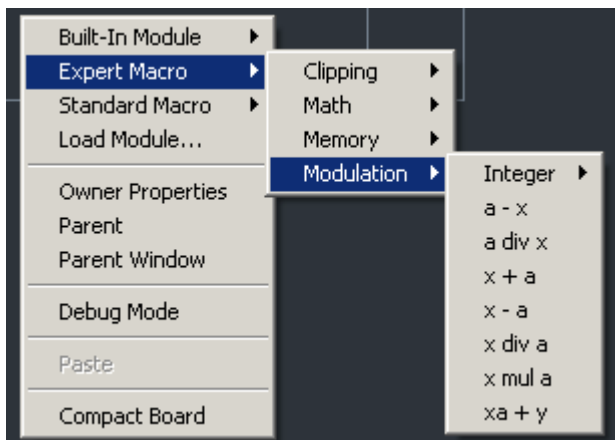
The problem is the initialization event. If you consider how the initialization of the above structure will happen you'll notice the following:

- the  $x$  input is firing or not firing an initialization event depending on whether it receives an initialization event from the outside, primary-level structure (that is the initialization event rule for core-cell event inputs)
- the constants 4 and 0.25 are always firing an initialization event

Thus, in case for whatever reason the initialization event does not occur at the input of the shaper, the output of the shaper will still receive the event from the last multiplier and will forward that event to the outside primary level structure.

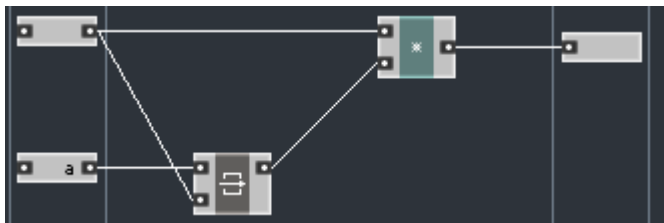
Although for control signal processing purposes, that might be OK (in case of a missing input initialization event, the input is considered zero, and the output initialization event is still fired), it is not exactly what one would intuitively expect from an event processing module. A more intuitive behavior would be for the module to send an output event only in response to an incoming event. So, the problem is that our two constant modules may be sending events at a wrong time (that is when there's no input event). As a solution, we suggest replacing the subtraction and multiplication modules, which have constants at their inputs, with their *Modulation* counterparts.

The Modulation macros is a group of modules in the Reaktor Core library found under *Expert Macro > Modulation*:



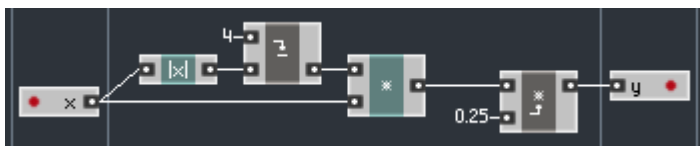
The name “Modulation”, although not 100% correct, still reflects their purpose of using one signal to modulate another. (That will be especially easy to see later, when we use control signals to modulate audio signals in low-level structures). Most of the modulation macros combine two signals, one is carrier and the other is modulator. Unlike built-in arithmetic core modules, the modulation macros generate output events only in response to events at the carrier input. The events at the modulator input do not trigger the recalculation process.

The internal implementation of modulation macros is very simple, they just latch the modulator signal, the latch being clocked by the carrier. Here is an example of a modulation multiplier macro's internal structure:



The latch at the modulator input (a) ensures that the modulator value will be sent to the multiplier only when the event at the carrier input arrives.

Here we replace the subtraction module by the  $a - x$  modulation macro and the second multiplication module by  $x \text{ mul } a$  modulation macro. This is how our structure is going to look after the replacement (we also replaced the Const modules with QuickConst, but that's unimportant):



You can normally tell the modulator inputs of modulation macros by their icons (pictures on the modules). A modulator input is indicated by an arrow icon. In case of the subtraction module, the arrow is on top, therefore the modulation input is on top. In case of the multiplication module, it's the other way around. You may also notice that the output of these modules is located against the carrier input, which is an additional source of information. You can move your mouse cursor over the modules and their inputs and read the corresponding hint texts.

In the above structure no events will be sent unless there's an event at the input of the core cell:

- the  $|x|$  module is triggered by the core cell input event directly
- the subsequent subtraction module will be triggered only by the output of the  $|x|$  module, which sends an event only in response to the input event, the QuickConst has no triggering effect
- the first multiplier is triggered by either the output of the subtraction module or the core cell input event, but we have already seen that both occur only simultaneously
- the second multiplier is triggered only by the incoming event and not by the QuickConst

So, now our structure's behavior is a little more intuitive.

# Audio processing at its core

## Audio signals

There is no special type for audio signals in Reaktor Core; audio signals are represented by events that, from the structure point of view, do not differ from any other event. What is different is that along the audio signal path the events are normally produced at regularly spaced time intervals, where the time between events is determined by the sampling rate.

To produce regularly spaced events (or for that matter, any events) we need some event source. As with *event* core cells, where the inputs of the module are the event sources, in *audio* core cells the inputs are also event sources. However, we now have an extra input type available:

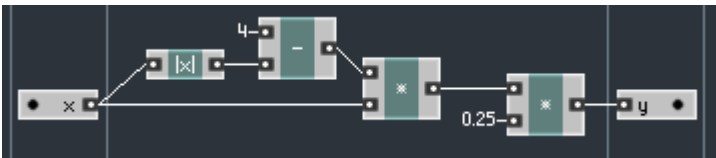
**Audio Inputs** repeatedly send core events to the inside of the structure at the rate determined by the sampling rate setting of the outside primary-level structure. The events are sent simultaneously from all audio inputs of a core-cell structure.

The audio inputs also send the initialization event to the core-cell structure. This event is sent regardless of what happens in the primary-level structure outside. However the value sent by these inputs during the initialization is dependent on the outside initialization process.

There is also a new output type which has to be used *instead* of event outputs.

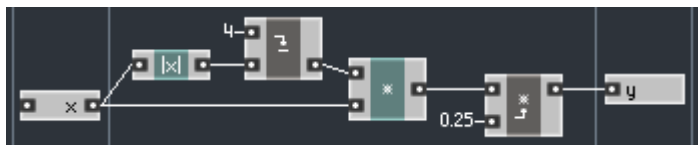
**Audio Outputs** deliver the last value received from the inside core structure to the outside primary-level structure. Because the audio outputs in the primary level do not send events, no events are sent to the outside.

Now we are going to rebuild the same shaper we built for events in audio mode. Therefore we create a new audio core cell. Generally we can use exactly the same structure, except instead of event inputs and outputs we will have audio inputs and outputs:

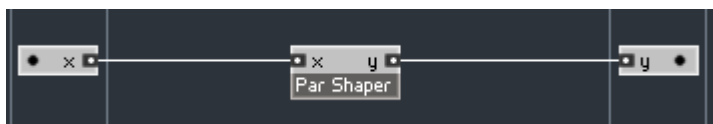


You may wonder why didn't we use the modulation macros in this case? The reason is that we are processing audio signals here, and audio signals always send the initialization event, so it's safe to do it this way. (You could use modulation macros if you prefer; it doesn't really matter.)

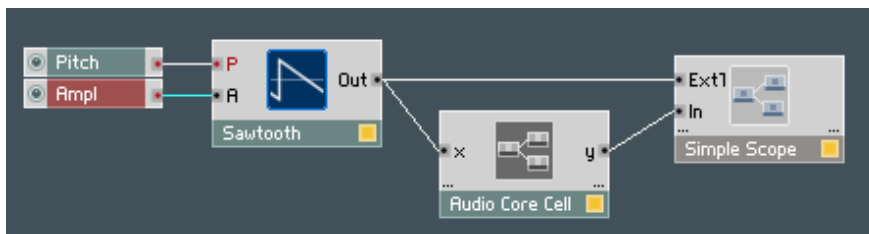
We could also pack the above structure into a macro, which could be used inside other Reaktor Core structures for both audio and event processing. In that case, we better use modulation macros inside, because we don't know in advance what kind of signal will be processed by the macro:



This is the inside structure of the audio core cell in that case:



To test it we are going to connect a sawtooth oscillator and an oscilloscope to it. An oscilloscope can be found under *Insert Macro > Classic Modular > OO Classic Modular – Display > Simple Scope* (from a primary-level structure). Also don't forget to make sure the number of voices for the instrument is 1.



We are using the external trigger for the oscilloscope for better synchronization at high distortion levels (the *Ext* button on the oscilloscope panel must be on for that to work). Change the range of the *Ampl* knob to something like 0 to 5 to be able to see the shaping.





## Sampling rate clock bus

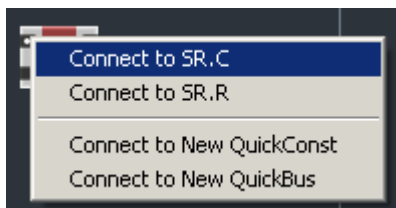
A couple more features are needed for building audio structures. One is to be able to create audio core cells with no audio inputs. (More precisely, we can create them, but what do we use as an audio event source?) Because many DSP algorithms need to know the current sampling rate, the other feature we need is to be able to access that. Of course, we have included both features:

There is a special connection possibility available in every Reaktor Core structure called the “sampling rate clock bus”. The bus carries two signals: the clock and the rate.

**Clock** is a signal source that sends regularly spaced events at the audio sampling rate. As do all standard audio signals, it also sends an initialization event. The values of all events are currently zero, but generally any structure using the clock signal should ignore the values, because it may be changed in the future.

**Rate** is a signal source whose value is always equal to the current audio sampling rate in Hz. The events are sent from this source during the initialization and whenever the sampling rate is changed.

You can access the sampling-rate bus by right-clicking on any signal input and selecting “Connect to SR.C” for clock signal or “Connect to SR.R” for rate signal.



The connection will be displayed next to the input:



---

The sample rate clock bus doesn't work inside event core cells.

---

## Connection feedback

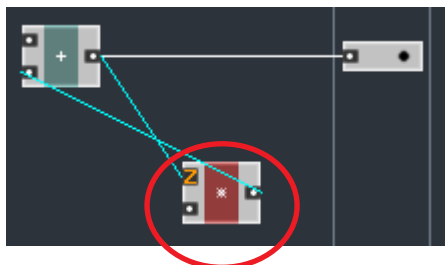
As we have already seen, the processing order rules cannot be applied if there are feedback connections within a structure. Therefore we need to provide additional rules, defining how feedback is handled.

The main rule is: Reaktor Core structures cannot handle feedback.

Well, not exactly so. You can make feedback connections in Reaktor Core; but because the Reaktor Core engine cannot handle structures with feedback, it will resolve them. Resolving the feedback means that the structure will be modified internally (you won't see it on the screen) in a way that results in no feedback.

The reason that is necessary is that, in the digital world, feedback without delay is not possible. Normally there is a one-sample (minimum) delay in the digital audio feedback path, and that is what the Reaktor Core engine will do during feedback resolution—it will introduce a one-sample delay module ( $Z^{-1}$ ) into the feedback path.

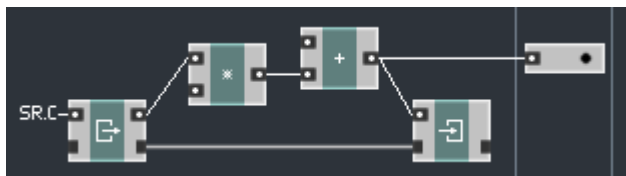
As you already know, places where implicit  $Z^{-1}$ 's have been introduced are indicated by the large orange Z displayed in place of the normal port icon:



We have already seen a structure built using a *Read* and a *Write* module that implements  $Z^{-1}$  functionality. Let's try putting that construction into our structure. We will put it on the wire where the automatic feedback resolution took place:



So, first we write, then we read (note that the *Read* module is clocked by SR.C to make sure that the reading is happening once per audio tick). That makes the read value always one audio sample behind the written one. Now there is no feedback in the structure. Don't see it? OK, let's move the modules around a little bit (we won't change a single connection):



Do you see it now? Of course.

So, inserting an explicit  $Z^{-1}$  module formally removes the feedback from the structure, while keeping it there logically (with a one audio sample delay).

---

Actually, the inside structure of a  $Z^{-1}$  macro is a little bit more complicated than a pair of *Read* and *Write* modules. We will learn how and why in the next section.

---

You don't have any control over the place where automatic feedback resolution will occur. It occurs on an arbitrary signal wire in the feedback loop. It is not even guaranteed that the resolution will always occur on a particular wire—it could change in the next version of the software, it could change in response to a change elsewhere in the structure, and it could be different the next time you load the structure from disk.

Hence, automatic feedback resolution is meant for the structures for which it's not important where exactly the resolution occurs. For example, such structures might be built by users who are not deep enough into DSP to understand these problems. Automatic feedback resolution allows them to still get reasonable results.

---

If you need to have precise control over feedback resolution points you can achieve that by explicitly inserting  $Z^{-1}$  modules in your structures. These modules will formally explicitly eliminate the feedback and automatic resolution will not be needed.

---

Here is a version of the above structure with a  $Z^{-1}$  macro inserted (it can be found in *Expert Macro > Memory* submenu):



As you can see, the big orange Z mark is gone now. Also note that the 1-sample delay point is different from the one which was automatically inserted (the automatic one was on the wire going from the Adder output to the Multiplier input and now it's on the wire going from the Multiplier output to the Adder input).

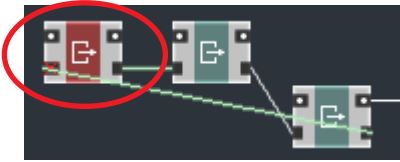
---

The meaning of the second input of the  $Z^{-1}$  module will be explained later. Typically you would just leave it disconnected.

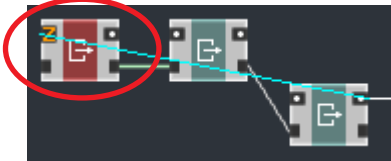
---

Feedback on OBC and other types of non-signal connections (which will be introduced later) does not make any sense and, therefore, is not allowed. Should feedback loops occur that do not have any signal wires in them, one of the

connections will be marked as invalid and considered not to exist. The *Invalid* mark is displayed as a big red X-shaped cross in the place of the port:



On the other hand, feedback loops with mixed types of the connections, are perfectly OK as long as they contain some normal signal wires in them; in that case they will be resolved in the normal way, with the resolution occurring on one of the normal signal wires:



---

In essence, this means that non-signal connections are never affected by feedback resolution, unless you make a completely non-signal feedback, which doesn't make any sense.

---

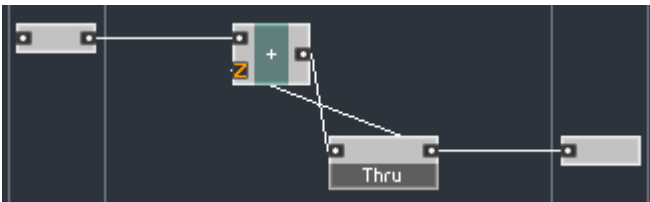
## Feedback around macros

In terms of feedback resolution, macros are generally treated in the same way as built-in modules.

Let's consider a macro which just passes the incoming signal through to the output. Here is the internal structure of such a macro:



Now assume we build a feedback structure using this macro:



The feedback loop goes through two wires in the above structure and through

another wire *inside* of the macro. Now where is the resolution going to occur? (OK, you can see in the above picture that is occurring at the adder input *in this particular case*, but we know it might as well have occurred at another point.)

Imagine for a moment that *Thru* was not a macro but a built-in module. In that case, it's obvious that the feedback resolution could not occur within the module, it must occur outside.

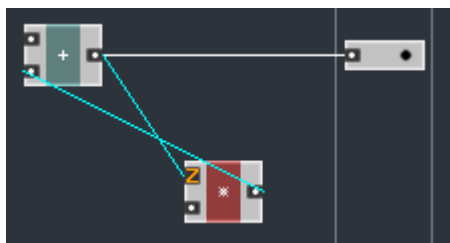
Well, we are trying our best to make macros look and behave as if they were built-in modules. For that reason *by default*, the resolution of feedback loops will occur outside the macro. It's not specified exactly where it will take place, but it will take place outside of the macro.

---

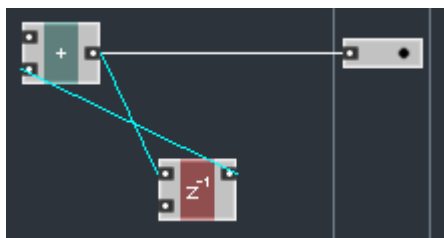
As a general rule, feedback resolution occurs on the highest structure level of the feedback loop.

---

However, you can change that behavior and allow feedback resolution to happen inside the macros. In fact, you should have wondered, if macros are treated the same as built-in modules, how can a  $Z^{-1}$  macro resolve the feedback. Consider the following structure:

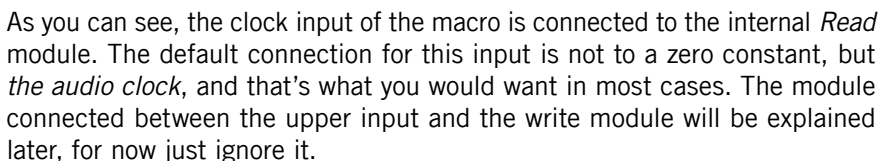


If macros and built-in modules are the same then nothing should change when we replace the multiplier by a  $Z^{-1}$  macro:



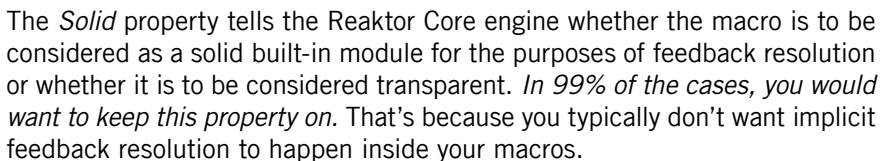
But it *is* different, because the implicit feedback is now gone. There must be something special about the  $Z^{-1}$  macro. And, in fact, there is.

If we look inside this macro we'll see almost the same structure as the one we mentioned earlier to implement the  $Z^{-1}$  functionality:



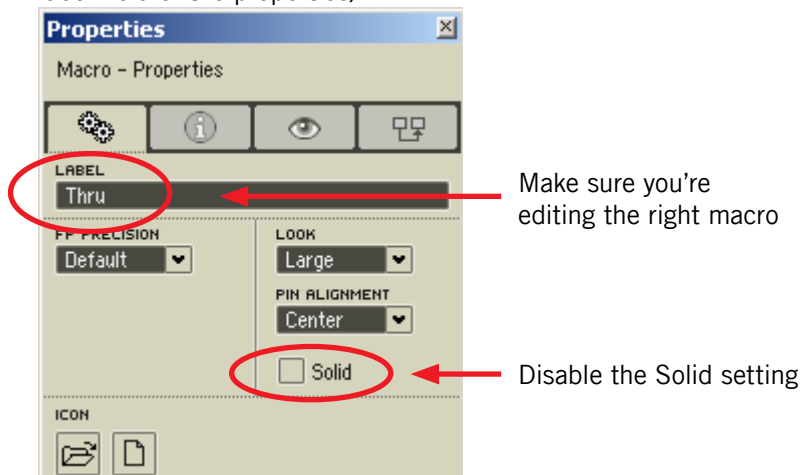
So far, there's nothing special about this macro, except that it seems to implement the  $Z^{-1}$  structure we have discussed earlier. So how does the Reaktor Core engine know that this structure is meant to resolve feedback loops? Obviously, the engine can know that it *can* resolve feedback loops, but how does it know that it's intended to?

This is controlled by the *Solid* setting in the macro properties:

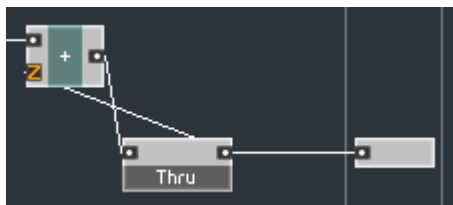


One reason for that is that the resolution happening inside a macro won't be visible unless you go into the macro, so that some of the implicit feedback

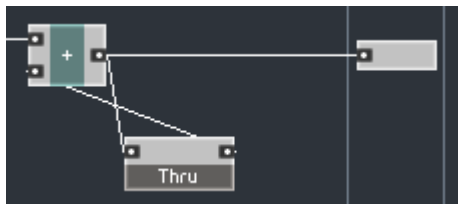
delays can go unnoticed. For example, we can take our previous structure with the Thru macro and disable the *Solid* setting (make sure you are editing the *Solid* setting for the right macro, you can see it by the Thru text in the label field of the properties):



Now your outside structure probably still looks the same (we say probably because you never can be sure where exactly the automatic feedback resolution will happen):



But if you change your structure a little, connecting the output to another module, it could look like this:



Our feedback resolution delay seems gone. So in a larger and more complicated structure we could easily miss the fact that there's an implicit delay. Where's this delay gone? Of course, it's now inside the Thru macro—the only place left which we cannot see from the outside:





Another reason for keeping the *Solid* property on is that with it off, in some cases the macro's internal operation could change once it's put in the feedback path. So please do yourself a favor and turn the property off only if you build macros which are meant to resolve feedback. There won't be many.

Now let's return to the  $Z^{-1}$  module. Because the *Solid* property is turned off for this macro, the boundary of this macro is completely transparent for feedback resolution. Thus the  $Z^{-1}$  macro is not really treated as a built-in module and is capable of resolving feedback in the way described earlier in this text.

## Denormal values

The signal values in the structures that we have been building in the previous sections are represented inside the computer by a binary data type called *floating point numbers* or *floats* for short. Floats are an efficient representation for a wide range of values.

The term *floating point numbers* does not exactly specify how the numbers are represented. It just describes the approach taken to represent them, still leaving lots of freedom for implementation details.

The CPUs of today's personal computer use the IEEE floating point standard. This standard defines exactly how the floating point numbers should be represented and what should be the results of operations on them (for example, how to handle limited precision issues, and so on.) In particular, this standard says that, for a group of particularly small floating point values, which because of limited floating point precision cannot be represented in the normal way, a special representation form is to be used. This other form is called "denormal" representation.

---

Denormal representation for 32 bit float values is used roughly in the range from  $10^{-38}$  to  $10^{-45}$  and from  $-10^{-38}$  to  $-10^{-45}$ . Values less than  $10^{-45}$  in absolute magnitude cannot be represented at all and are considered to be zero.

---

Because their representation is somewhat different from that of normal numbers, some CPUs have certain problems with handling these numbers. In particular, operations on these numbers can be performed much much more slowly (as much as 10 times or more) on some processors.

---

A typical situation in which denormal numbers appear for *prolonged periods* of time is in calculating exponentially decaying values, as in filters, some envelopes, and feedback structures. In such structures, after the input signal reaches zero level, the output signal *asymptotically* decays to zero. *Asymptotically* means that the signal gets closer and closer to zero without ever reaching it. In that situation, denormal numbers can appear and stay in the structure for relatively long time (until their absolute value falls below  $10^{-45}$ ), and that can cause a significant increase in CPU load.

---

---

Another situation in which denormal numbers may occur is when you change the precision of a floating point value from a higher precision (64 bit) to a lower precision (32 bit), because a value  $10^{-41}$  is not a denormal in a 64 bit precision float but it is a denormal in a 32 bit precision float (changing the precision of floats is discussed later).

---

Let's consider modeling an analog 1-pole lowpass filter with its cutoff set to 20 Hz. Our digital signal values will correspond to analog voltages (measured in volts). Let's imagine that the input signal level was equal to 1V (volt) over a long enough period of time. Then the voltage at the filter output is also equal to 1V. Now we abruptly change the input voltage to zero. The output voltage will decay according to the law:

$$V_{out} = V_0 e^{-2\pi f_c t}$$

where  $f_c$  is the filter cutoff in Hz,  $t$  is time in seconds and  $V_0 = 1\text{ V}$  (initial voltage).

Then the output voltage will change as follows:

after 0.5 sec	$V_{out} \approx 10^{-29}$ volt
after 0.6 sec	$V_{out} \approx 10^{-33}$ volt
after 0.7 sec	$V_{out} \approx 10^{-38}$ volt
after 0.8 sec	$V_{out} \approx 10^{-44}$ volt

Oops, the numbers between  $10^{-38}$  and  $10^{-45}$  are in the denormal range. So in the time period from approximately 0.7 to 0.8 seconds, our voltage is represented by a denormal value. And it's not only inside the filter. The filter output is probably further processed by the downstream structure, causing at least the few following modules also to deal with denormal values.

At a sampling rate of 44.1 kHz, the time interval of 0.1 second corresponds to 4,410 samples. Assuming that the typical ASIO buffer size is a few hundred

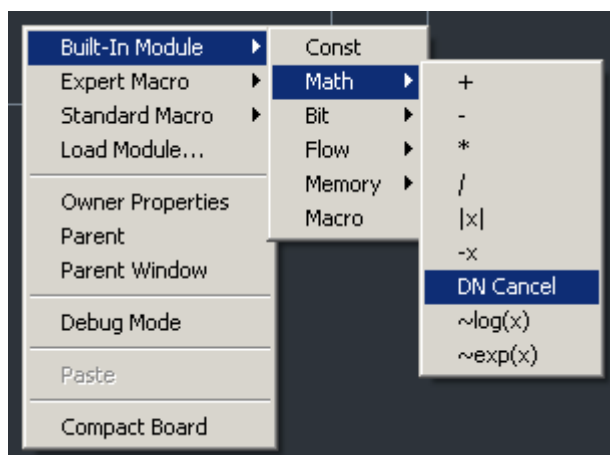
samples, we have to produce several buffers at a significantly higher CPU load. Should the CPU load (per buffer computation) get close enough to or exceed 100%, it will cause audio dropouts.

---

From the above text you need to draw one conclusion: denormal values are bad in real-time audio.

---

Reaktor primary-level modules are programmed in a way that generally prevents denormals from occurring inside them. Specifically, the DSP algorithms have been modified in a way that they generally shouldn't produce any denormal values. If you are designing your own low-level DSP structures in Reaktor Core you also have to take care of denormals. To help you with that job we have introduced the *Denormal Cancel* module, available in *Built-In Module* > *Math* submenu:



The *Denormal Cancel* module has one input and one output, and it tries to slightly modify the incoming value in a way that prevents denormals from occurring at the output:



The way this module modifies the signal is not fixed and may change from one software version to another, or even from one place in the structure to another. Currently it adds a very small constant to the input value. Because of precision losses, this addition does not modify values that are large enough (a value as large as  $10^{-10}$  will not be modified at all), and because of the same precision losses, it is very unlikely that the result of addition can be a denormal value (in most of the cases it is probably even impossible).

---

If for whatever reason the *Denormal Cancel* module does not work for your structure, you are, of course, free to use your own denormal canceling techniques. But the problem may be that a technique that works on one platform sometimes may not work on another, whereas we are going to adapt the built-in *DN Cancel* algorithm to each supported platform. So whenever possible, try to use the *DN cancel* module. We will even consider building alternative algorithms into this module – feel free to discuss this with us on the support forum.

---

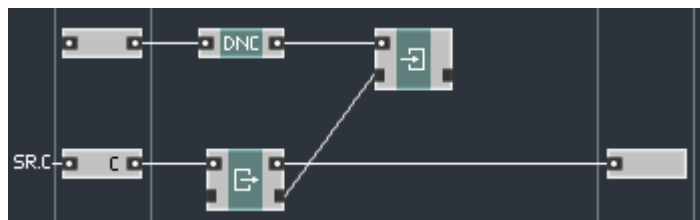
---

Some CPUs offer an option to violate the IEEE standard by disabling the production of denormal numbers, forcing the denormal results to zero. Because Reaktor Core structures are meant to be platform independent, it's strongly advised to always take care of denormal canceling in your structures, even if your particular system does not suffer from them.

---

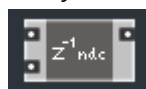
Because one of the most typical situations for the denormals to appear are exponentially decaying feedback loops, and because most of feedback loops in audio processing are exponentially decaying (including but not limited to filters and feedback structures with delays), we decided to build denormal canceling into the standard  $Z^{-1}$  macro.

As you remember, the inside of this macro looks like this:



Now you probably can tell what the Denormal Cancel module is doing in there. Because you would often use the  $Z^{-1}$  macro inside feedback structures, there's a good possibility of denormals occurring. We therefore decided to put the DNC module into the  $Z^{-1}$  macro structure.

There's another version of this macro called  $Z^{-1} \text{ ndc}$  which does not perform denormal canceling (ndc = no denormal cancel). You can use it in the structures that you are sure do not generate denormals (for example, FIR filters):



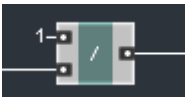
## Other bad numbers

Denormal numbers are not the only kind that can cause problems in Reaktor Core structures with internal states, and particularly in feedback loops. Other examples include INFs, NaNs and QNaNs. We are not going to discuss those in detail here, because that information is available in other places, including the Internet. What's important for us is preventing those kinds of numbers from appearing in our structures.

Generally, such numbers appear as the result of invalid operations. Division by zero is the easiest case. Other cases involve numbers getting too large to fit in the floating point representation (that would be above  $10^{38}$  in absolute magnitude), or outside the reasonable range for a particular operation.

Such numbers tend to get stuck inside structures, and in a way they are more sticky than denormals. For example as soon as you add a denormal value to another value which is not denormal, the result will be non-denormal (unless the other value is also very small and close to being a denormal). On the other hand if you add a normal value to an INF, the result will still be an INF.

Besides having a tendency to stick in structures forever (or better said, until the structure is reset), these numbers also have a bad habit of requiring much larger processing times on some CPUs. Therefore you should do your best to prevent them from being created at all. That means, for example, that whenever you divide two numbers you ensure that the denominator (the bottom part of the fraction) is not zero. The case of initialization requires particular attention here. For example, consider the following structure element:



If for whatever reason, the initialization event does not come on the lower input of the Divider module, a division by zero will happen during initialization processing. In this case you might consider using a modulation delay macro instead, or depending on your particular needs find another solution.

## Building a 1-pole low pass filter

A simple 1-pole low pass filter can be built using a recursive equation:

$$y = b * x + (1 - b) * y_{-1}$$

where

$x$  is the input sample,

$y$  is the new output sample,

$y_{-1}$  is the previous output sample, and

$b$  is the coefficient defining the filter's cutoff.

The value of the coefficient  $b$  can be taken equal to the normalized circular cutoff frequency, which can be computed using the following formula:

$$F_c = 2 * \pi * f_c / f_{SR}$$

where

$f_c$  is the desired cutoff frequency in Hz

$f_{SR}$  is the sampling rate in Hz

$\pi$  is 3.14159...

$F_c$  is normalized circular cutoff (in radians)

---

In fact, the coefficient  $b$  is equal to the normalized cutoff only approximately, the error increasing at high cutoff values, but it should be more or less OK for our purposes, especially if we do not need to have a precise setting of the cutoff frequency for our filter.

---

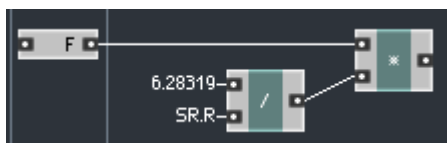
We start by creating an audio core cell with two inputs: one for the audio input and one for cutoff. We are going to use an event input for the cutoff in this version of the module.



Actually, because we think it's a good habit to build Reaktor Core structures as core macros to enhance their reusability, we are going to create our filter as a macro. So we create a new macro inside the structure and create the same inputs for that macro:



Now let's build the circuitry for converting the cutoff frequency into the normalized circular cutoff:



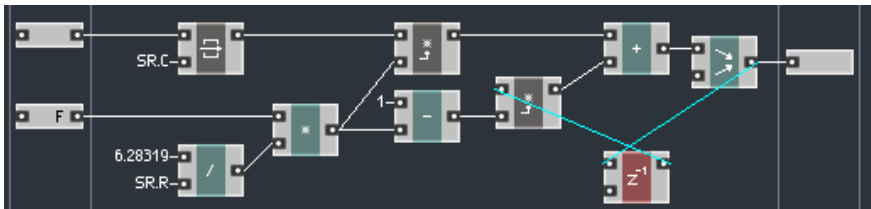
$6.28319$  is  $2\pi$ , which is then divided by the sampling rate, forming the value to be multiplied with the cutoff frequency. We don't need a modulation multiplier, because  $F$  logically is a control signal input, so we might perform the initial multiplication even if there is no initialization event at the  $F$  input.

---

We perform the division before the multiplication, because the division is relatively heavy on the CPU, and the sampling rate doesn't change that often. If only the cutoff frequency changes, there are no events sent to the divisor module, and therefore, the division will not be performed. This is one of the standard optimizations that can be done by the core-structure designer.

---

Let's build the circuitry implementing the filter's equation:



The audio input is latched just in case events at this input arrive asynchronously to the standard audio clock. That wouldn't be necessary in the core-cell structure, where an audio input is known to send events at correct times, but in a general core macro it is a very good practice.

Two modulation multipliers are used to prevent events at the  $F$  input (which generally speaking, can happen at any time) from triggering the computation in the feedback loop. Here it should be more clear why they are called modulation macros, in this case the cutoff-derived signal is used to modulate gains in the feedback path.

---

Latching is a standard Reaktor Core technique to make sure that incoming events do not trigger the computations at improper times. It's also very widely employed in the form of modulation macros and in other similar situations.

---

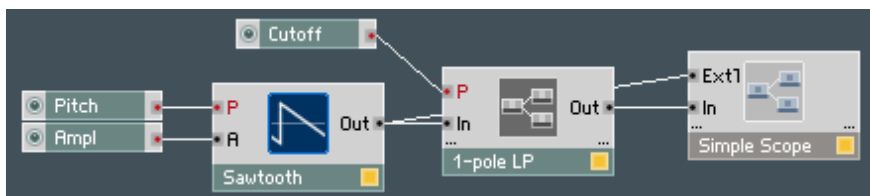
The  $Z^{-1}$  module is used to store the previous output value and will automatically send an event with the previous output value on every audio clock tick. It also takes care of possible denormal values which otherwise could occur. Those familiar with DSP should notice that the structure looks pretty much similar to the standard DSP filter diagrams.

The Merge module at the adder output is making sure that the filter state after the initialization will still be zero, even if the input signal has a non-zero value.

Finally, we put the pitch to frequency converter into the core-cell structure and we are ready to test:



For testing we suggest using the following structure (don't forget about the 1-voice setting for the instrument):



The Cutoff knob should be set to the range 0 to 100 or something similar. Beware of too high cutoff values. Because of the increasing filter coefficient error at high cutoffs, the filter will become unstable with large cutoff values.

---

A better filter design should at least clip the cutoff values to the range where the filter is stable. For our case this could have been achieved by clipping the  $b$  coefficient to the range of 0..0.99 or something similar. Techniques for value clipping will be described later in this text.

---



This is what you should see in the panel now:



Move the cutoff knob and watch the signal shape changing.

## Conditional processing

### Event routing

Events in Reaktor Core do not always have to travel along the same predefined paths. It is possible to dynamically change these paths. You can achieve this by using the *Router* module (*Built-In Module > Flow > Router*):



The *Router* module accepts events at its signal input (bottom) and routes them to either its output *1* (top) or its output *0* (bottom). The routing, i.e. whether the event goes to output *1* or output *2*, depends on the current state of the Router, which is controlled from the *Ctl* input (top)

The *Ctl* input accepts a connection of a new type, which is not compatible with either normal signals or OBC connections. It is a *BoolCtl* (Boolean control) signal type. The BoolCtl signal can be in one of two states: true or false (on or off, 1 or 0). If the control signal is in the true state the events are routed to output 1. If the control signal is in the false state the events are routed to output 0.

---

The control signals have a significant difference from normal signals in Reaktor Core: they do not transmit events and therefore cannot trigger any processing by their own.

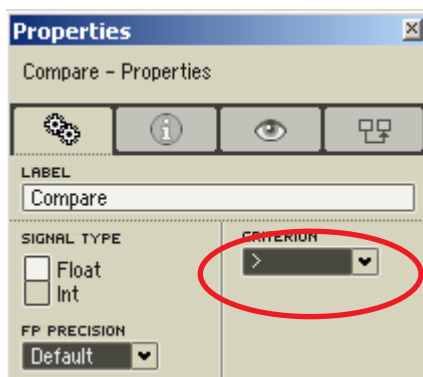
---

To control a Router you obviously need a control signal source, the most common of which is the Comparison module found under *Built-In Module > Flow > Compare*:



This module performs a comparison of the two incoming signals and outputs the result as a BoolCtl signal. The upper input is assumed to be on the left of the comparison sign and the lower input, on the right. So a module reading '>' produces a true control signal if the value at the upper input is greater than the value at the lower input.

You can change the comparison criterion in the properties of the module:



The available criteria are:

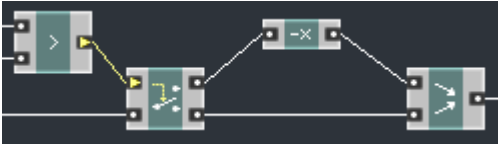
- |    |                             |
|----|-----------------------------|
| =  | equal                       |
| != | not equal ( $\neq$ )        |
| <= | less or equal ( $\leq$ )    |
| <  | less                        |
| >= | greater or equal ( $\geq$ ) |
| >  | greater                     |

---

It is, of course, possible to connect several routers to the same comparison module, in which case they will change their state simultaneously.

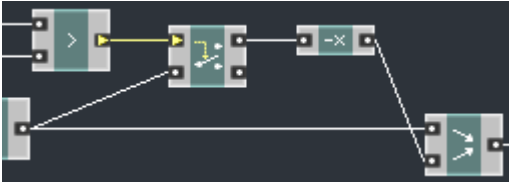
---

The Router module splits the event path into two branches. Quite often these branches will later be merged:



Depending on the result of the comparison the above structure will either invert the input signal or leave it intact.

An alternative implementation of this structure would be:



In this version, the 0 output of the Router is disconnected; therefore, the Router works as a gate, letting the events through only if it's in the 'true' state. The inverted value then arrives at the second input of the *Merge*, thus overriding the non-inverted value, which is always arriving at the first input. If the router is in 'false' state the inverter doesn't receive an event and doesn't send an event to the second input of the *Merge*; therefore, the original unmodified signal goes to the output of the *Merge*.

---

The branches are most often merged with a Merge module. But theoretically speaking you could use many other modules (for example, arithmetic modules like adder, multiplier, and so on) instead.

---



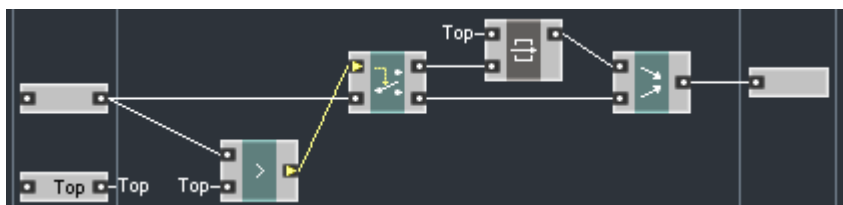
---

Routers treat the initialization event just like any other event. Therefore, one could filter out the initialization event by using routers, thereby ensuring that the initialization event won't appear in particular areas of the structure.

---

## Building a signal clipper

Let's build a Reaktor Core macro structure that would clip the incoming audio signal from the top at a specified level:



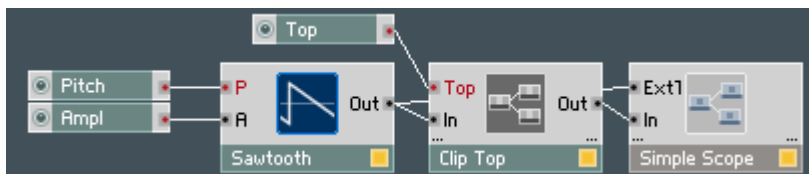
If the input signal is not greater than the threshold it will be routed to output 0 of the Router and, through the Merge, to the output of the structure. Otherwise, the signal will be routed to output 1, where it triggers the latch, sending the threshold value to the Merge instead. The same thing happens during initialization.

---

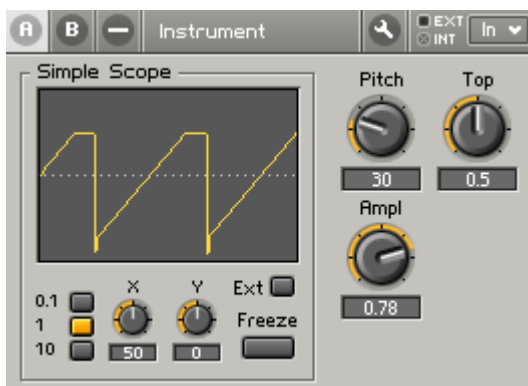
Note that this structure will not change its output in response to changes to the threshold. Rather the new threshold value will be used for the next and all subsequent events at the signal input. This is in a way similar to a modulation macro's behavior, where modulator changes do not result in output events.

---

Here is a testing structure for the clipper module we have built (an audio core cell has been used):



And this is what you should see in the panel:



In fact, there are a number of such “modulation” clipper macros found in the *Expert Macro > Clipping* menu.

## Building a simple sawtooth oscillator

Let’s build a simple sawtooth oscillator, generating a sawtooth waveform with amplitude 1 and a specified frequency. We will use the following algorithm: increment the output signal level at constant speed and at the moment the level becomes greater than 1 drop it by 2.

---

Instead of dropping by 2 we could reset the level to  $-1$ , but that is generally not as good, because we won’t be able to precisely maintain the specified oscillator frequency.

---

The incrementing speed defines the oscillator frequency by the following equation:

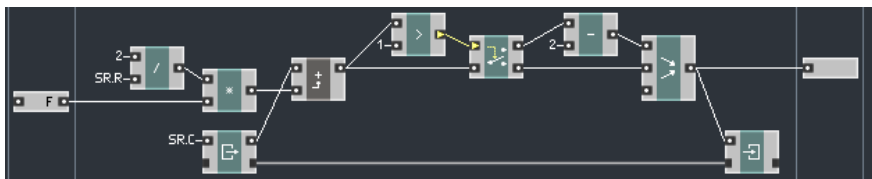
$$d = 2f / f_{SR}$$

where  $d$  is the level increment per one audio sample,  
 $f$  is the oscillator frequency and  $f_{SR}$  is the sampling rate.

First we are going to build the circuitry for computing the incrementing speed:



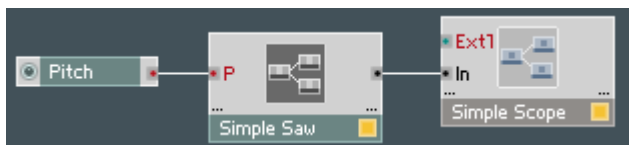
Now we need the increment loop. It’s time to use a pair of *Read* and *Write* modules exactly as we did in the accumulator:



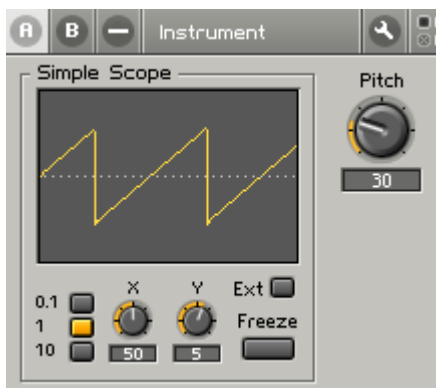
The Read module triggers the level increment at each audio event. The sum of the old level and the increment is then compared against 1 and routed either directly to the result writing or to the wraparound circuitry.

The third input of the Merge module ensures that the oscillator is initialized to zero. Theoretically, the module that subtracts 2 from the signal level should have been a modulation macro, but we didn’t bother, because the Merge overrides the initialization result anyway.

Here’s the suggested test structure (don’t forget the P2F converter inside the core cell):



And here is the panel view:



## More signal types

### Float signals

The most common signal type used for DSP (digital signal processing) on modern personal computers is floating point (*float* for short). Floats can represent a wide range of values, as large as  $10^{38}$  (in 32 bit mode) or even  $10^{308}$  (in 64 bit mode). As useful as they are, floats have a drawback – limited precision. The precision is higher in 64 bit mode, but it is still limited.

---

The precision of float values is limited for technical reasons. If it weren't limited, float values would require an infinite amount of memory to store and processing them would require an infinitely fast CPU. It's similar to the impossibility of writing the full decimal representation of a transcendental number, such as  $\pi$ , on a finite piece of paper. Even if you can somehow compute all the digits (which is not always possible for transcendental numbers), you will eventually run out of paper (and time).

---

The signals and memory storage that we have been discussing so far use 32 bit floating point numbers for their representation. Reaktor Core also offers the possibility of using 64 bit floats, should you need higher precision (or a

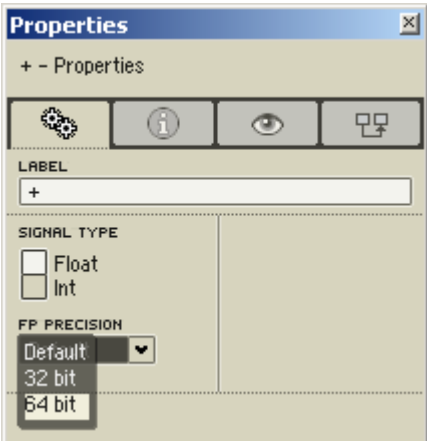
larger value range, although it's difficult to imagine that  $10^{-38}$  to  $10^{38}$  is not a large enough range).

---

By default all processing in Reaktor Core is done in 32 bit floats. This doesn't exactly mean that the signals are really processed as 32 bit floats, but rather that at *minimum*, 32 bit floats will be used for processing (although 64 bit floats may occasionally be used for intermediate results).

---

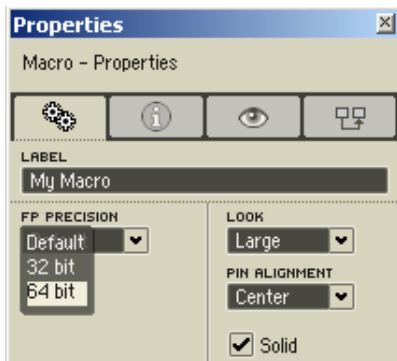
You can change the floating point precision for individual modules as well as for whole macros. For individual modules you do so in the module's *FP Precision* (floating point precision) property:



- |                |   |
|----------------|---|
| <i>default</i> | means use whatever precision is the default for the current structure |
| <i>32 bit</i>  | use minimum 32 bits of precision                                      |
| <i>64 bit</i>  | use minimum 64 bits of precision                                      |

Changing the precision for a module means that the processing within that module will be done using the precision specified and that the output value will be generated using the same precision.

You can also change the default precision for whole structures by right-clicking on the background and selecting *Owner Properties* to open the properties of the owner module:



So changed, the default precision will be effective for all modules inside the current structure, including macros, as long as they do not define their own precision (or in the case of macros, if a new default precision is defined for their respective inside structures).

---

Normal floating point signals of 32 and 64 bit precision are fully compatible with each other and can be freely interconnected. OBC signals of different precision are not compatible with each other (because you cannot have storage that is simultaneously 32 and 64 bit). Also, for OBC signals 'default', '32 bit' and '64 bit' settings are all considered different and incompatible, because the effective default precision can be changed by changing the properties of one of the owning macros.

---

---

The input and output modules of top-level structures of core cells always send and receive 32 bit floats, because that is the type of the signal used for Reaktor primary-level event and audio connections.

---

## Integer signals

There is another data type commonly supported by modern CPUs, and actually this one is more fundamental to the digital world than floats. It is the *integer* type. Integer numbers are represented and processed with *infinite* precision. Although the precision of integers is infinite, the range of representable integer values is limited. For 32 bit integers the values can go up to more than  $10^9$ .

---

Infinite precision for storage and processing of integer values is possible because they don't have any decimal digits after the



period, so you can write them using a finite number of digits. Let's write down the number of seconds in an hour: 3, 6, 0, 0, done. It's that easy. If you try to write down the value of  $\pi$  you cannot do it completely: 3, 1, 4, 1, stop. Not complete, OK let's write a couple more digits: 5, 9, stop. Still not complete, and so on. With an integer number you can do it completely and precisely: 3600, that's it.

---

While floating point is a natural choice for values that are changing continuously, as are audio signals, for discretely changing values (for example, counters) integers may be a more appropriate choice.

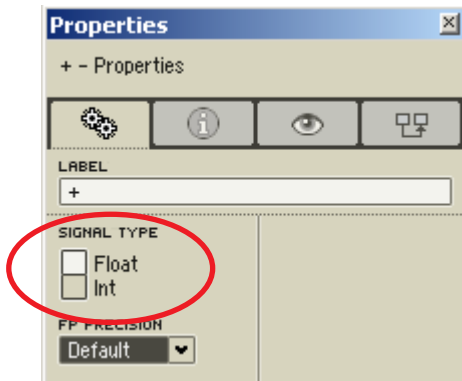
Many Reaktor Core modules can be switched to integer mode, in which case they expect integer signals at their inputs; they process them as integers (that means with infinite precision); and they produce the integer outputs. Examples of such modules include arithmetic modules like adder, multiplier, or subtractor. There are even some modules that can be used only on integers.

---

Minimum 32 bit length is guaranteed for Reaktor Core integer values.

---

Switching between float and integer types (if it's supported by the module) is done in the *Signal Type* property of the module:



A module set to integer type will process the input values as integers and produce integer output values. You can tell that a module is in integer state by the fact that its signal inputs and outputs look different:



There is no such thing as default signal type for macros. The reason is that normally you wouldn't build structures that process integers in exactly the

same way as structures processing floats and vice versa (although you might for some relatively simple structures).

---

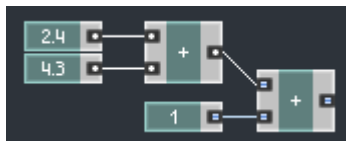
Integer signals can be freely interconnected with floats, but the wires created between different type signals will perform signal conversion, which can use a certain amount of CPU. At the time of this writing, the extra CPU usage is somewhat noticeable on PCs and quite significant on Macs. The OBC connections of float and integer types are not compatible with each other, of course.

There can also be information loss during such conversions. In particular, large integers cannot be precisely represented by floats, and obviously, floats cannot be precisely represented by integers. Large floats (larger than the largest representable integer) cannot be represented as integers at all, in which case the result of the conversion is undefined. During float-to-integer conversion, the values will be rounded *approximately to the nearest integer*. We say approximately because the result of rounding 0.5 can be either 0 or 1, although you can rely on the fact that 0.49 will be rounded to 0, and 0.51 to 1.

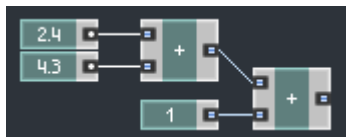
---

---

It is important to understand that turning the processing mode of an operation to integer and converting of a floating point result of the same operation to an integer is not the same. Let's consider an example. Here we are adding two numbers 2.4 and 4.3 as floats. The result is clearly 6.7, which when converted to integer will produce 7. So the output of the following structure is 8:



Now if we change the mode of the first adder to integer, instead of adding 2.4 and 4.3 we will add their rounded versions which are 2 and 4 respectively, producing 6. So the result is 7:



Clock inputs completely ignore their incoming values, therefore they are normally always floats. Furthermore, signal type conversion will not be performed for the signals that are used only as clocks:



Here the clock input of the Read module is still float although the module has been set to integer mode (the OBC ports look the same regardless whether they are float or integer).

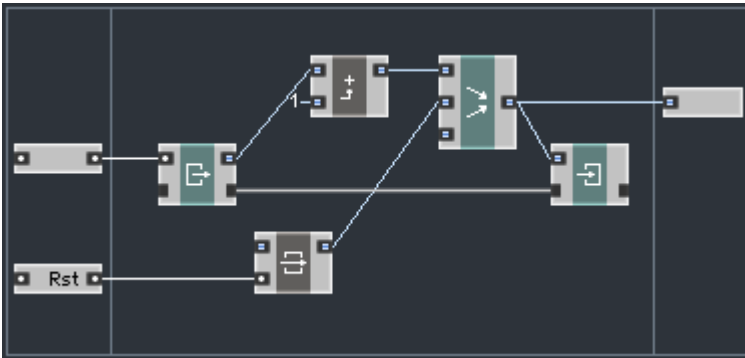
---

Integer feedback is automatically resolved in the same way as float feedback – by inserting an integer mode  $Z^{-1}$  module (of course no denormal canceling is needed here).

---

## Building an event counter

Let's build an event counter macro. The function of this macro is similar to an event accumulator, but instead of summing the values of events, this one will just count them. Integer signal type seems a logical choice for counting:



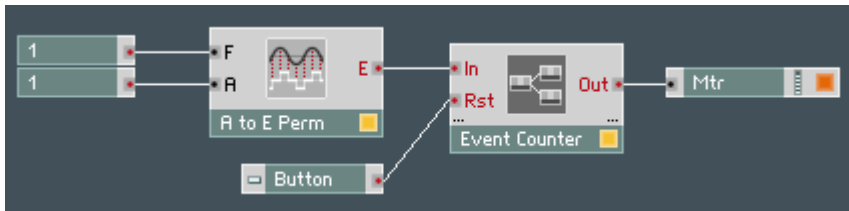
The output and all built-in modules have been set to integer mode here. The ILatch macro is used instead of Latch for resetting the circuitry. It does exactly the same (and can be found in the same menu), but works on integer signals. Also, an integer modulation macro is used (it's found in *Expert Macro > Modulation > Integer* menu). Both inputs do not need to be set to integer mode, because they provide only clock signals.

If we take a look at the structure of the event core cell containing this macro:



we'll see that the output of this module is not set to the integer mode (it's also not possible to set it to integer mode). That's because the core cell being a Reaktor primary level module on the outside must output a normal primary-level event, which is a float value.

Here is a testing structure for the counter module:



And the resulting panel:



## Building a rising edge counter macro

Now we are going to discuss a *sign comparison* technique which you might sometimes need in building Reaktor Core structures. Sign comparison is a special way of comparing two numbers, in which you ignore their values and pay attention only to their signs (plus or minus). Naturally, plus is considered greater than minus. So for example:

3.1 is sign-greater than -1.4

2.1 is sign-equal to 5.0

4.5 is sign-equal to -2.9

---

Beware that the sign of zero is undefined, which means that result of any sign comparison involving a zero value can be arbitrary.

---

Of course you could have implemented the sign comparison using several Comparison modules and several Routers, but there is a more efficient way. The sign comparison can be done in Reaktor Core structures using the *Compare Sign* module (*Built-In Module > Flow > Compare Sign*):



This module produces a BoolCtl signal at the output, so that you can connect it to a Router.

One of the possible uses of such a module is detecting the rising edges of an incoming signal. Below we are going to build a rising edge counter Reaktor Core macro:

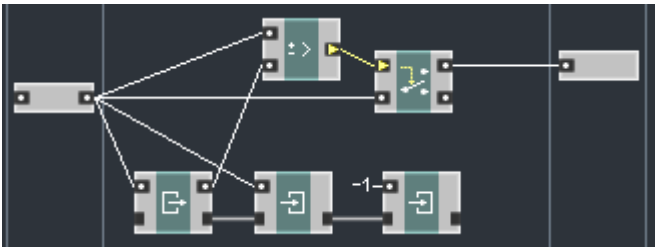


Note that the output is set to integer mode because the count is an integer value.

The first thing we are going to need inside is an edge detector macro to convert the detected edge to an event:



This is how the detector macro can be implemented:



The chain at the bottom keeps the previous input signal value. As you can see the new value is stored after the old one is read. The last Write module in the chain performs the initialization job for the previous value storage. We initialize the storage to -1 so that the first positive value will be counted as a rising edge.

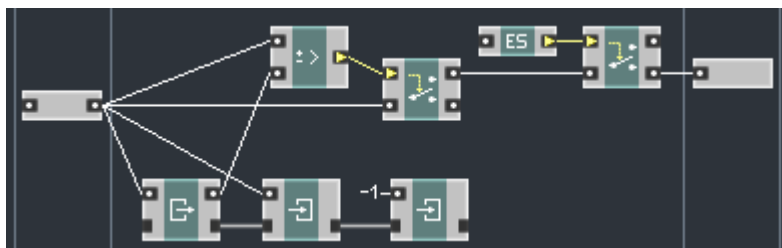
---

Having a Write module at the end of an OBC chain is another way (as opposed to Merge) to initialize the storage. It must be the last Write module in the chain in order to overwrite the results stored by upstream Write modules.

---

The Router controlled by the Sign Comparison module will gate the events, letting through only those where a sign change from negative to positive occurs.

It's not clear whether such a module will send an event during initialization or not, particularly because the storage is still zero at the time of initialization event processing, and the sign of zero is undefined. We can modify this structure in order to avoid sending an event during the initialization:



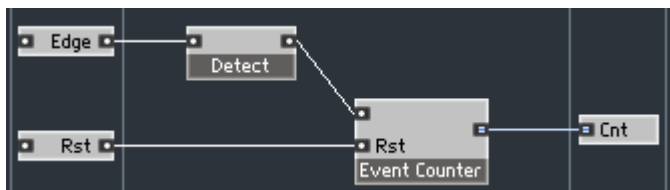
The ES Ctl module is an event sensitive control. The control signal produced by this module is true only if there is an incoming event at the input of this module. Because this input is disconnected in the above structure, which means it's connected to a zero constant, the only time the control signal is true is at initialization. So the second router will block any event occurring during initialization and let all others through.

---

Note that here we have an example of a module that does not send any event from its output during initialization.

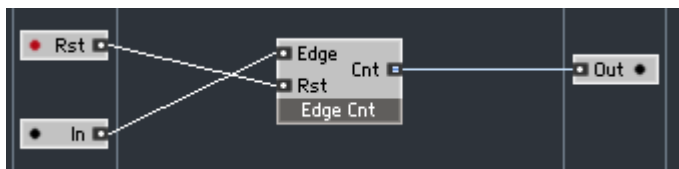
---

Now that we have a detector module we can connect it to the counting circuitry that we already have:

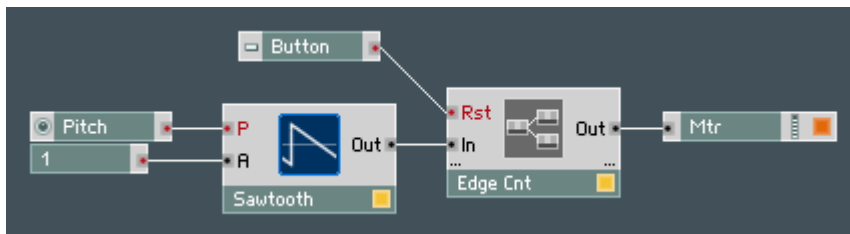


The function of the above circuitry should be clear: The Detect module sends an event each time it detects a rising edge, these events are counted by the Evt Cnt module.

To test it, let's put this macro into an audio core cell, and count the rising edges of a sawtooth waveform. The internal structure of the core cell will look like:



and the primary level test structure:



(Don't forget to set the Meter properties as in previous examples.)

Here is what you should see in the panel:



The speed of the number change in the meter must correspond to the frequency of the oscillator, defined by the pitch knob. At the pitch value of zero the oscillator frequency is approximately 8 Hz, so the numbers should increment at approximately the rate of 8 per second.

## Arrays

### Introduction to arrays

Let's imagine you want to build an audio-signal selector module, which, depending on the value at the control input, picks up the signal from one of four audio-signal inputs:



One approach would be to use Router modules, but there is also another possibility—we can use another feature of Reaktor Core – *arrays*.

A *one-dimensional array* is an *ordered collection* of data items *of the same type* which can be addressed by their rank in this order or *index*. For example, here we have a group of 5 float numbers:

5.2    16.1    -24.0    11.9    -0.5

In Reaktor Core the array element indices are zero-based, which means that the first element of the array has an index of 0. Therefore, the element with an index of 0 is 5.2, an index of 1 gives us 16.1, and indices of 2, 3, and 4 address -24.0, 11.9, and 0.5, respectively.

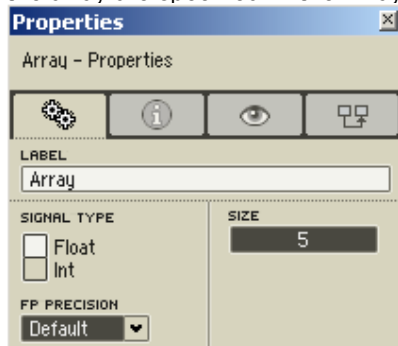
Here is a representation of this array using a table:

Index	0	1	2	3	4
Value	5.2	16.1	-24.0	11.9	0.5

Arrays are created in Reaktor Core using *Array* modules (*Built-In Module > Memory > Array*):



An Array module has a single output which is of *Array OBC* type. The size of the array (number of elements) and the type of data kept in the elements of the array are specified in the Array module's properties:



For example, for the above table of 5 elements we will need to specify the *float* data type and the size of 5.



---

Please note that because array indices in Reaktor Core are zero-based, the index range for an array of size 5 would be 0 to 4 (you can also see it in the table above).

---

---

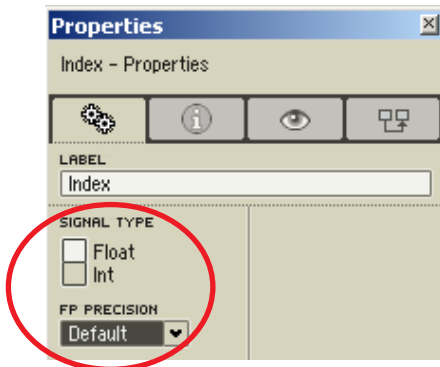
Array OBC signals corresponding to different item data types are, of course, not compatible.

---

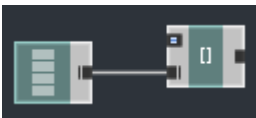
To address an array element you need to specify an index, which you can do using an *Index* module (*Built-In Module > Memory > Index*):



The master OBC input (bottom) of the Index module should be connected to the slave output of an array module. The master input connection type should match the array type, the former can be specified in the properties of the Index module:



And now the connection:

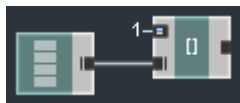


The upper input of the Index module is always integer type and accepts the index value. Here we are addressing the array element with the index of 1:

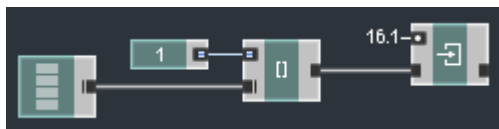


Notice that the constant module has also been set to integer mode (you can tell that by the look of the output port). This is not necessary, because automatic conversion to integer would have been performed anyway; it just looks better.

Alternatively we could have used a QuickConst:



The output of the Index module is a Latch OBC type, which means that you can connect *Read* and *Write* modules (or even several of them) to that output. Of course, you need to take care that the Read and Write modules are set to the same data type as the data type of the Array and Index modules. Here the array element with index of 1 will be initialized to 16.1:



---

If an out-of-range index is sent to the Index module, the result of accessing the array is undefined. The structure will not crash, but it's unspecified which array element will be accessed in this case or whether the access operation will take place at all. If you're unsure of the range of incoming index values, you should clip the input value range using Routers or macro modules from the library.

---

## Building an audio signal selector

Now let's return to building the audio signal selector module:



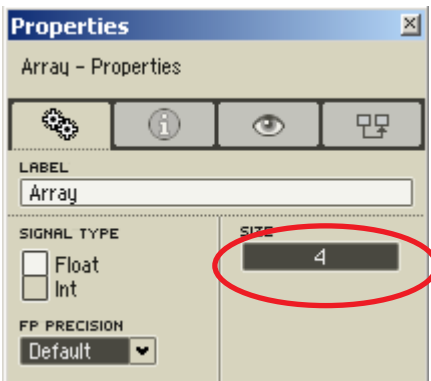
Here is an empty internal structure for this module:



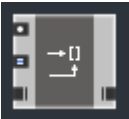
We are going to use an array of 4 float elements for storing our audio signals:



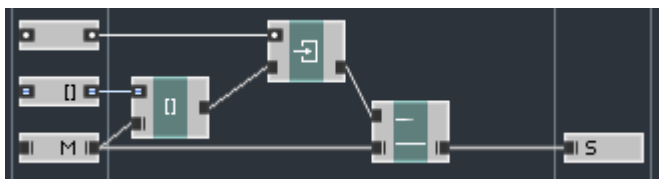
Here are the properties of the array module:



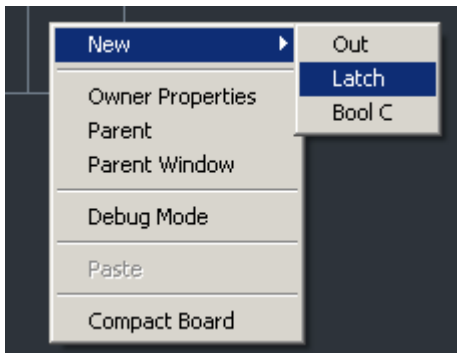
To write the input values into the array we will use the standard macro *Write []* (*Expert Macro > Memory > Write []*):



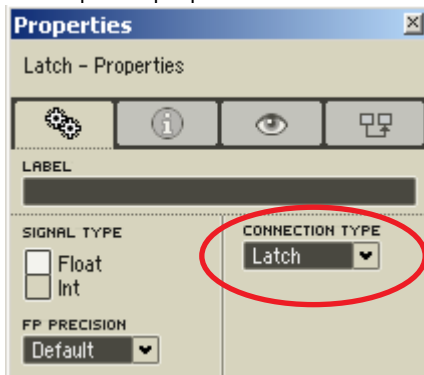
This macro internally has an Index module and a Write module, performing writing into the array element with a specified index:



The upper input, of course, receives the value to be written. The [] input receives the index at which the write operation should take place. The M input receives the OBC connection to a default precision float array, and the S output is a thru connection, similar to other OBC modules like *Read* and *Write*. The M input and the S output are another type of macro port, which differs from the ones we have been using up to now. These ports can be inserted by selecting the Latch entry from the port insertion menu (the third type is the *BoolCtI* macro port type):



Latch ports can be used for latch OBC connections (between Reads and Writes) as well as for array OBC connections. How they are used is controlled in the port's properties:



Setting the connection type to Latch or Array defines the OBC connection type between latch OBC and array OBC, respectively. For the *Write []* macro ports this has obviously been set to Array type.

The module with two parallel horizontal lines is the *R/W Order* module (*Built-In Module > Memory > R/W Order*):



It does nothing except let the connection at its master (bottom) input through to its slave output. The upper input has absolutely no effect; however, because there is a connection at this input, it will affect the processing order of the modules. Therefore, *everything connected to the S output of the macro will be processed after the Write module*, which would not be the case were the R/W Order module missing from the structure.

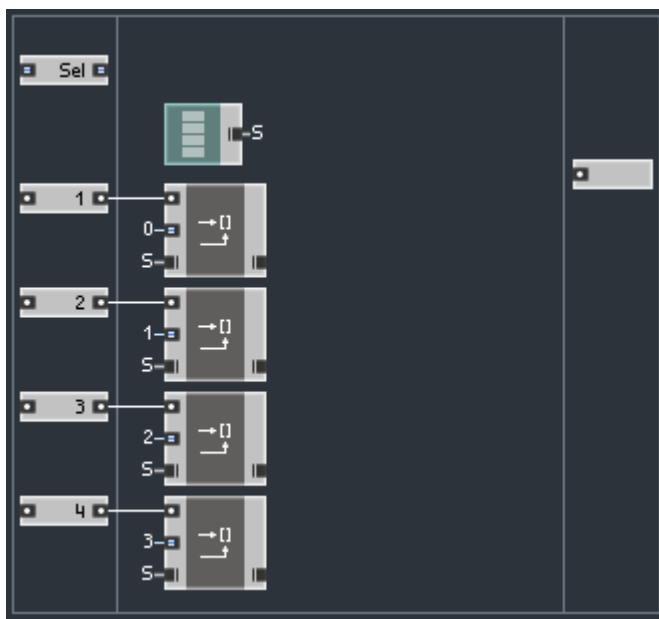
---

In the absence of the R/W Order module, the functionality of the Write [] macro would not be very reliable or intuitive, because the user expects everything connected to the S output of the Write [] macro to be processed after this macro. Generally, such problem arises only with OBC connections, and in those cases, you need to take care to put R/W Order modules into the macros that you design where necessary.

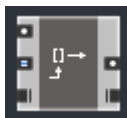
---

Like OBC ports, the R/W Order module has a *Connection Type* property. For this module the *Connection Type* property controls only the type of the M and S ports; the sidechain input is always in latch mode. See the description of R/W Order in the module reference section for details.

Now let's build the circuitry for writing the input signals into the array:

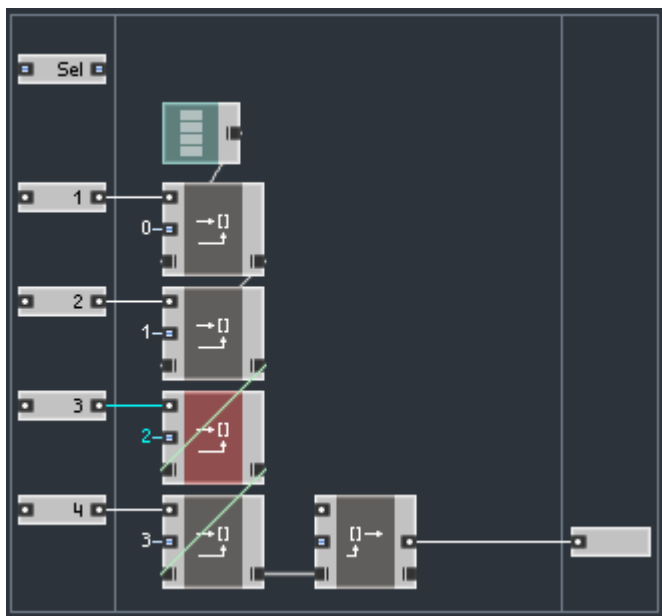


The four Write [] modules will take care of storing the incoming audio values into the array. We now need some circuitry to read one of the 4 values. We suggest using *Read []* macro (*Expert Macro > Memory > Read []*):

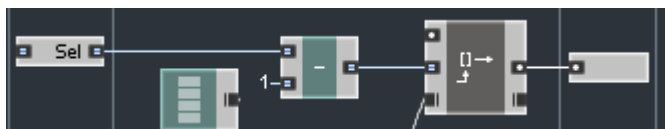


This macro reads from an array element whose index is specified at the integer input in the middle. The top input is the clock input for the read operation – it will send the read value to the upper output of the module in response to an incoming event. The ports at the bottom are, of course, the master and slave array connections.

Now to what do we connect the master input? Obviously we cannot connect it directly to the array module, because we need the read operation to be performed after all write operations (otherwise, there might be an effective one-sample delay, or there might not be, all-in-all not very reliable). We also cannot connect it to any of the Write [] modules, because that wouldn't solve our problem. We suggest that, rather than connecting the Write [] modules to the array module in a fan pattern, you connect them serially; then connect the Read [] module to the output of the last Write [] module.



Now, what do we connect to the index input of the Read [] module? Because we want our selection value to be in the range 1 to 4, we need to subtract 1 from the Sel input value. Notice that we perform integer subtraction (and because Sel is just a control input we don't really need a modulation macro here):

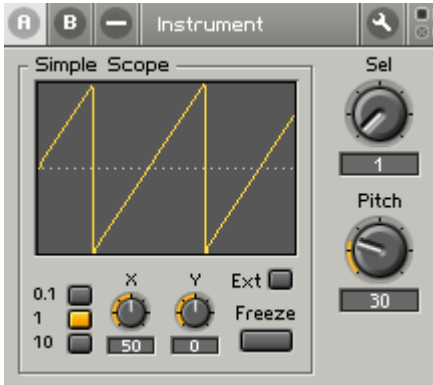


The last step is to clock the read module by the sampling-rate clock (because we are building an audio selector):



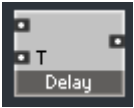


The Sel knob is set to switch between 4 values from 1 to 4.  
Now switch to the panel and look at different waveforms corresponding to the knob setting:



## Building a delay

Now that we have some experience with arrays, let's build a simple audio-delay macro. The module will look like this:

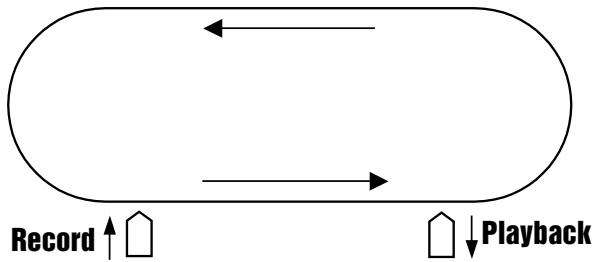


Or even better, like this (to align the output port with the top input port, we need to go into the macro and change its *Port Alignment* property to top):

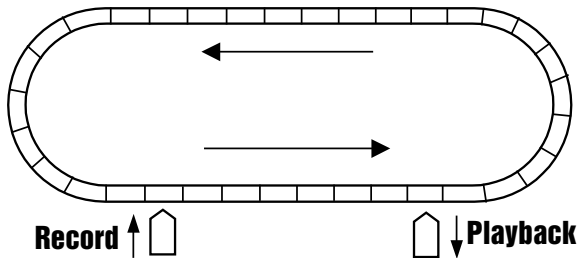


The T input expects the delay time in seconds.

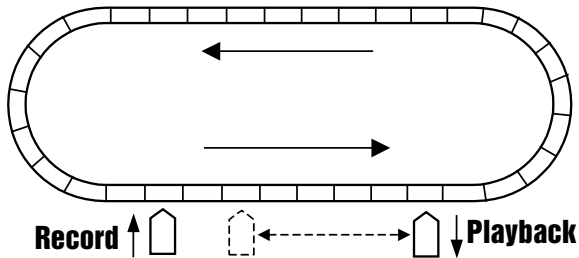
If you take a look at an analog tape delay device, you'll see a tape loop combined with record and playback heads. Strictly speaking there's also an erase head, but for simplicity we can imagine that the record head does both the jobs of erasing and recording.



If we want to simulate this in a digital form, we need some kind of digital tape loop. Because of the discrete nature of digital, the digital tape loop will hold a finite number of audio samples, and these samples will be recorded and read at the audio sampling rate:



A natural choice for a digital tape loop would be an array, the size of the array being equal to the number of samples recorded in the whole loop. In an analog tape delay, the delay time depends on the distance between the record and playback heads and on the tape speed. Usually the distance between the heads is fixed and the tape speed is variable. It is done that way for obvious technical reasons: it's much easier to vary the tape speed than the distance between the heads. In the digital case, it's just the opposite, because varying the tape speed means performing sampling-rate conversion between the digital tape and the output, while varying the distance between the heads is relatively simple, so that is what we are going to do:



There's also another difference: in the analog world the tape is moving. If we want to move our digital tape we would need to copy all array elements to their neighbor positions *at each audio clock*, which is quite CPU intensive. Instead, we will move the heads.

From the preceding, we can conclude that we will need the following:

- array – to simulate our digital tape loop
- write index – this is our record head
- read index – this is our playback head

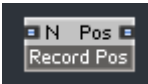
The write and the read indices will be moving through the array sample by sample. When either of them reaches the end of the array, it needs to be reset to the beginning of the array (that corresponds to connecting the open ends of the tape into a loop). The difference between the write and the read position corresponds to the delay time measured in samples.

---

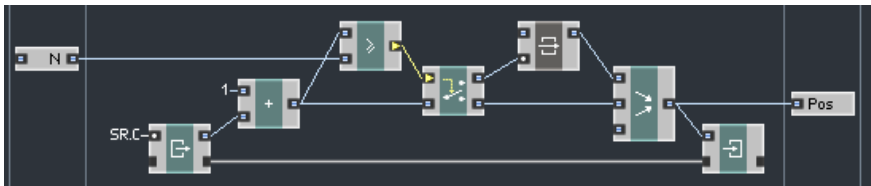
This technique is quite common in programming and is called “circular buffer” or “ring buffer”.

---

We start by programming the record head. It operates similarly to the sawtooth oscillator we programmed earlier, except that the computations are done in integer mode. The value increment is one per audio tick and the output value range is from 0 to  $N-1$ , where  $N$  is the size of the array. Let's put the circuitry for computing the write index into a RecordPos macro:

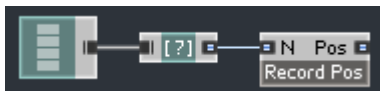


The  $N$  input should receive the number of elements in the array and the Pos output will carry the current writing position (index). Here is how this macro can be implemented (compare this to the sawtooth oscillator implementation):

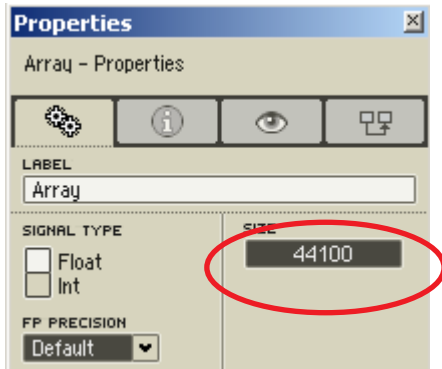


Note that the comparison module is set to  $\geq$ . That was unimportant for the sawtooth oscillator, we could use  $\geq$  or  $>$  there, but in integer computations the difference is in most cases critical. Using  $\geq$  condition ensures that the write index will never reach a value of  $N$  (which would be out of range).

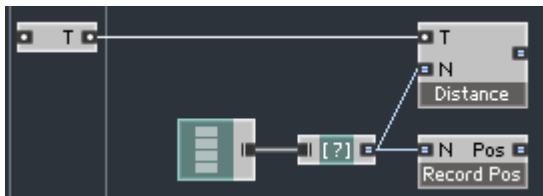
On the top-level, we create an array module and connect it to the RecordPos through the `Size []` module (available in *Built-In Module > Memory > Size []*), which reports the size of the array:



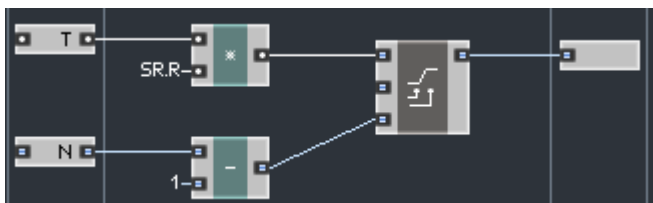
The size property of the array can be set to 44,100. This will allow us as much as 1 second of delay (actually one sample less) at 44.1 kHz sampling rate:



Now we need to compute the read index, which we will do by building two macros. The first macro will convert the requested delay time into the distance in samples:



That can be done by multiplying the time in seconds by the sampling rate in Hz. We also should not forget to clip the result, a clipping macro *Expert Macro* > *Clipping* > *IClipMinMax* should be good for that:



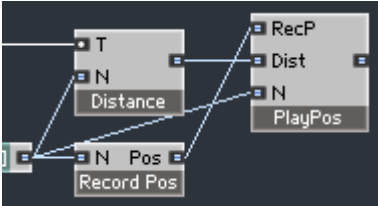
We clip to N-1 because that's the maximum distance between two different array elements. Note the conversion to integers, which is done after the multiplication.

---

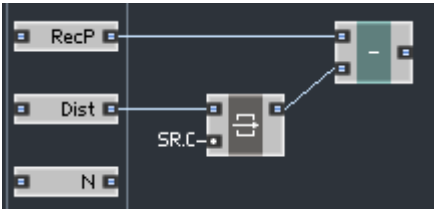
Alternatively, we could have clipped the input value (to a different range, of course), and that is generally a little bit better, because float values which are out of the range of integer representation can produce arbitrary integer values, in which case we would no longer get true clipping.

---

Now, we use another macro to compute the read index from the RecordPos and Distance:



Obviously, the playback position must be the *Distance* in samples behind the record position; therefore, we subtract one from the other:

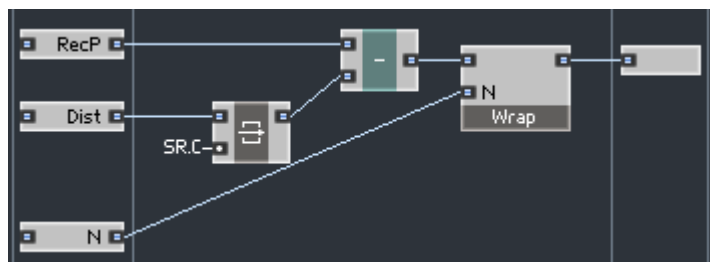


The distance value is latched because it is produced by a control signal input, which potentially can receive events at any time, and we do not want the subtraction happening at times other than at audio-clock events.

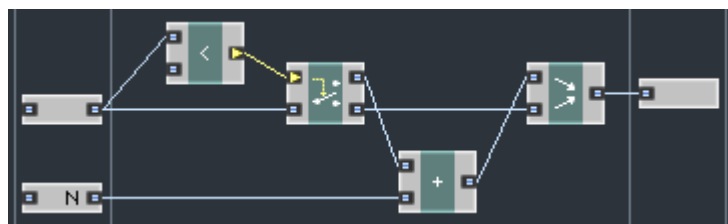
If we just subtract, the difference can turn out to be less than zero because our array is not a loop; its ends are not connected together. So we need to *wrap* the result:

- 1 must become  $N-1$ ,
- 2 must become  $N-2$ ,
- 3 must become  $N-3$ ,
- and so on.

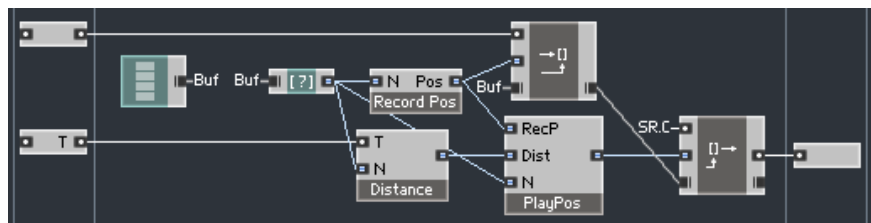
So we include another macro for wrapping:



Because we know that the difference cannot be smaller than  $-N+1$  (because RecordPos is between 0 and  $N-1$  and Distance is between 0 and  $N-1$ ), wrapping can be implemented as simple addition of  $N$ :

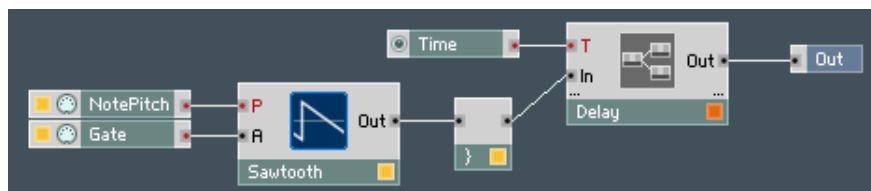


Let's get back to our top level structure. Now that we have the write and read indices, we just need to perform reading and writing:



Note that reading is happening after writing and that it's clocked by the sampling-rate clock.

Here's a proposed test structure. Don't forget to put an ms2sec converter into the Delay core cell and to set the Delay core cell to monophonic mode:



---

Actually it's a good idea to switch the delay to monophonic mode as soon as possible, because each voice will consume about 200K of memory. 44,100 samples, using 4 bytes (32 bit) each:  
 $44,100 * 4 = 176,400$  bytes, which is a little bit more than 172K (a kilobyte has 1024 bytes).

---

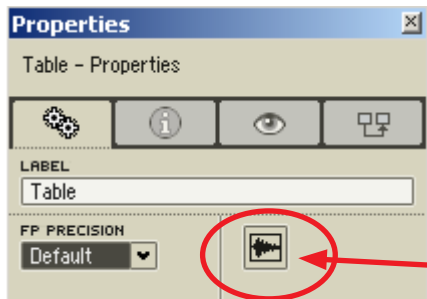
To test the above structure play notes on your midi keyboard and hear them delayed by the amount of time specified by the *Time* knob.

## Tables

There's another module similar to *Array*. The name of this module is *Table* and it can be found in the *Built-In Module > Memory* submenu:

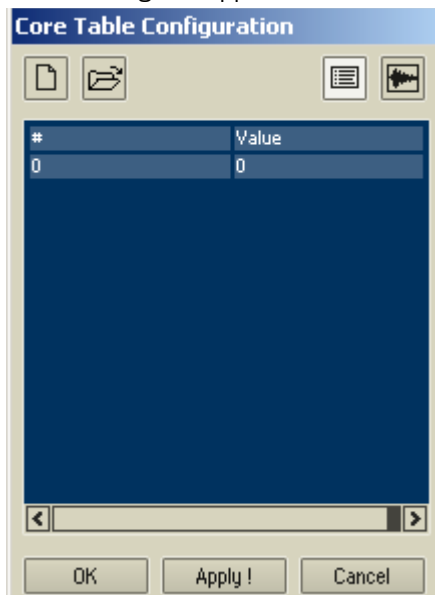


The difference between a table and an array is that you can only read from a table; you cannot write to it. The values in a table are pre-initialized using the module's properties. To get access to the list of the values press the button in the properties window:




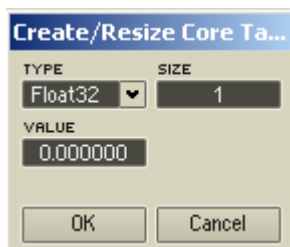
Click here to edit the values

A new dialog will appear:



What you currently see is an empty table. It consists of a single element with zero value. You can type in new values manually, or you can import them

from a file. If you go for the manual option you have to click the  button. The following dialog appears:



There you need to select the type of values stored in the table, the table size (number of elements in the table), and a value to initialize all elements of the table.

Alternatively, you can import the table from a file. The file can be an audio file (WAV/AIFF), a text file (TXT/ASC), or a Native Table File (NTF). To import

from a file press the  button. A file dialog will appear asking you to select



a file. After that another dialog will appear asking you to select the data type for the table values.

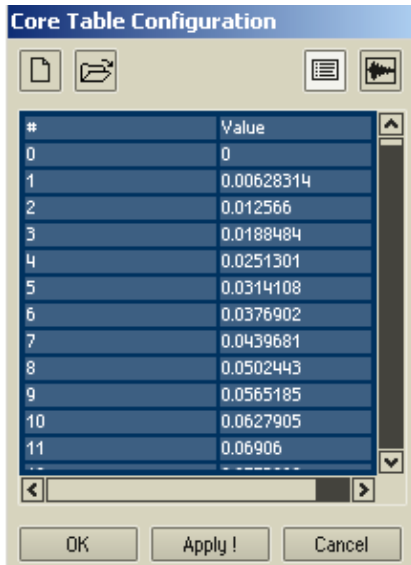
Let's use a table. We are going to build a sine-oscillator macro using a table lookup approach:





Inside this macro we are going to create a table module:



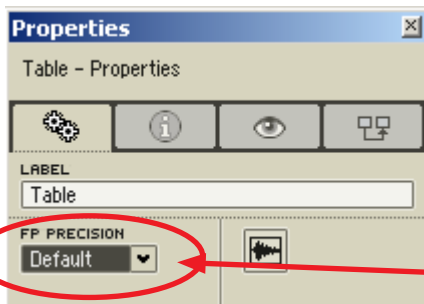
We initialize the table from the file *sinetable.txt*, which we've prepared for you in the *Core Tutorial Examples* folder in your Reaktor installation. It's a text file containing values for one period and one sample of sine function. Import it as Float32 type values:



You can also view the loaded values as a waveform display. The  and  buttons switch between the list and waveform view respectively.

Press OK to close the dialog and commit the loaded values to the table.

There's also an *FP Precision* property setting in the properties window for the table. It doesn't really control the precision of the values in the table (that you should have selected when importing the file or manually creating a list of values), but rather it sets the formal precision type of the table module's output. Generally, you would keep it set to default:

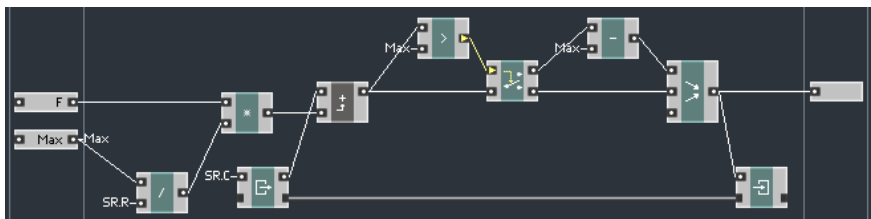


Formal output precision

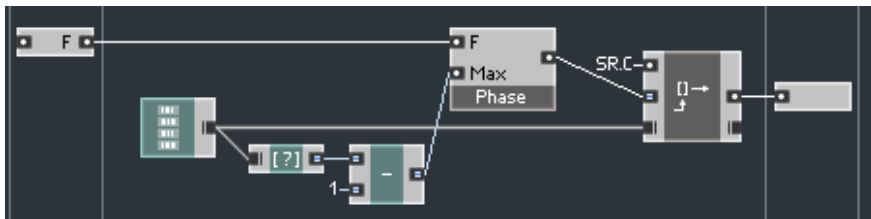
Now that we have the table, we can continue building the oscillator. At its core there will be a phase oscillator generating a rising sawtooth ramp signal from 0 to the size of the table minus one:



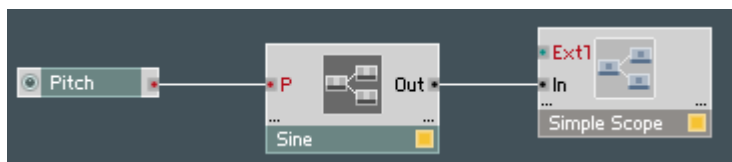
The phase oscillator implementation is similar to the sawtooth oscillator and to the recording position in the delay:



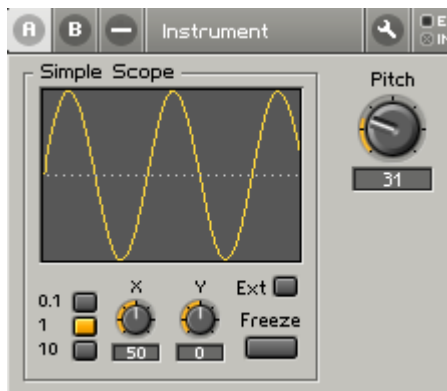
A *Read []* module connected to the phase oscillator and clocked by the sampling-rate clock will access the corresponding table element and output its value.



Here's the suggested test structure (don't forget a P2F converter in the core cell):



And this is the panel view:



Of course, this is not a very clean sounding sine because we don't have any interpolation in there. We leave it up to you to build an interpolating version if you wish.

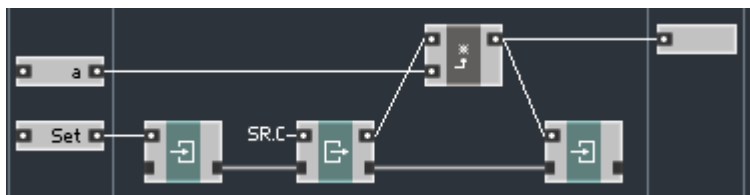
# Building optimal structures

As a rule, no tool is ideal. The Reaktor Core technology is no exception. Although this technology is quite powerful on its own, you do need to know a few things to get the most out of it. So here are some essential tips and tricks to get you going.

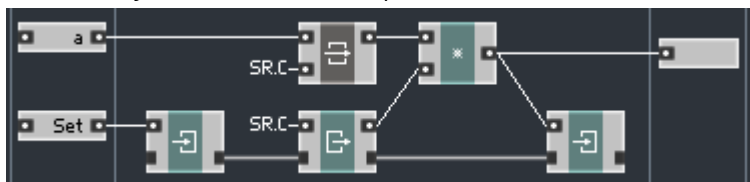
## Latches and modulation macros

Use Latches and/or modulation macros at all necessary places to ensure that events are delayed until the values they carry actually need to be processed.

Here is a structure which uses a modulation macro for multiplication in an audio iteration loop. Using the modulation macro prevents the processing from being triggered by events at the a input:



Alternatively one could use an explicit latch in the structure:



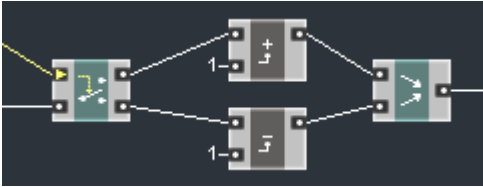
There were multiple examples of this technique throughout this tutorial. Using latches has to do both with performance optimization and the *correctness* of your structures. Some typical mistakes in structure programming have to do with sending events to certain modules at improper times. Don't be afraid that the latches will slow down the performance of your structures. Latches do not require much computation, and in many cases, they use *absolutely* no CPU time.

Latches are generally preferable to routing for event filtering because of their lower CPU cost. Try to use routers only where the processing logic dictates routing.

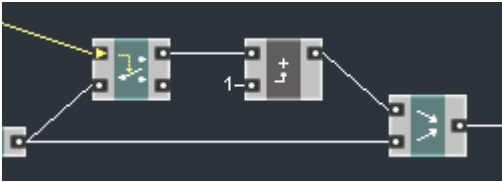
## Routing and merging

Routing can be more or less CPU intensive depending on the situation and the platform. If you can avoid routing without adding other CPU-intensive operations to your structure, do so.

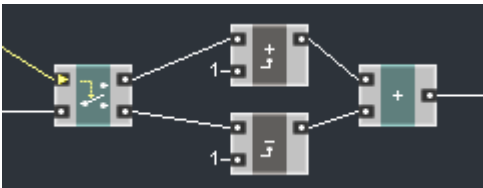
Sometimes ES Ctl routing can be replaced by using Latches. If possible, do so. If you split the event path into two branches using a Router, it's a good idea to merge the branches generated by the outputs of the Router:



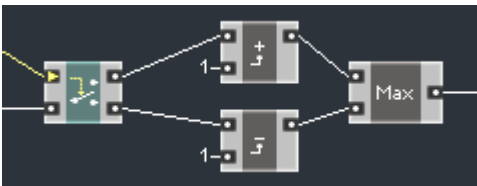
It's also a good idea to merge to the incoming (unsplit) event to the Router:



Merging is not necessarily done by using a *Merge* module. Any arithmetic or a similar module will do the job:



Merging can also happen inside a macro (depending on its internal structure):



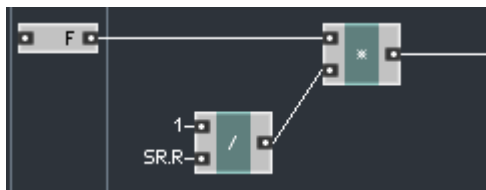
It may be reasonable or necessary to merge the branches generated by different Routers, but beware of higher CPU loads in that case.

## Numerical operations

Floating point addition, multiplication, subtraction, absolute value, and negation are generally the least CPU-intensive float operations. Integer addition, subtraction and negation are the least CPU-intensive integer operations. Integer absolute value is also more or less OK. *DN Cancel* currently uses plain addition, as you may remember.

Float division, and integer multiplication and division are significantly more CPU intensive on average.

It is advisable to group your operations in a way that the most CPU intensive ones get evaluated as rarely as possible. For example, if you need to compute normalized frequency by dividing the frequency in Hz by the sampling rate it could be reasonable to compute the reciprocal of the sampling rate first and multiply the frequency by the result:



In the above structure, the division will be performed only when the sample rate changes, which should be pretty rare. Changes to the frequency will trigger only multiplication.

Compare that to the more straightforward implementation of the same formula:

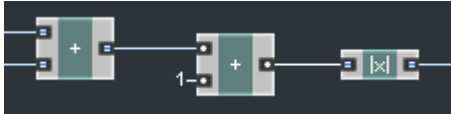


where the division would be executed in response to every change of frequency.

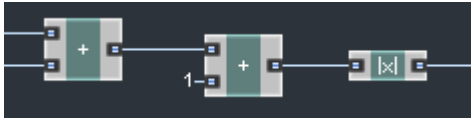
## Conversions between floats and integers

Generally, avoid all unnecessary conversions between float and integer numbers. Depending on the platform such conversions could use significant amounts of CPU. The conversions that are necessary to do are OK, of course.

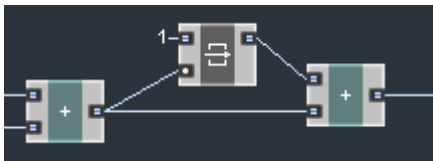
Although the following structure might work as expected, in fact, there are two unnecessary conversions between integer and float types:



The first conversion happens at the input of the adder module in the middle. This module is set to the float mode, but it receives an integer input signal. Therefore an integer to float conversion will be done. The second conversion is at the input of the absolute value module, which is set to integer mode, but receives a float input. There, a conversion from float to integer will be done. This would be a much better way to do it:



All modules are set to integer modes, therefore no conversions will be done. Clock signals generally should have float type, but if an integer signal is used, that's no problem:



Even though the clock input of the ILatch is float, it's clocked by an integer signal, but because the clock value is unimportant, no conversion is done.

# Appendix A. Reaktor Core user interface

## A.1. Core cells

A core cell can be created from a Reaktor primary-level structure (except the ensemble structure) by right-clicking on the background and selecting *Core Cell > New Audio* or *Core Cell > New Event*.

Library core cells (from both the system and user libraries) are found in the same “*Core Cell*” menu. You can also load core cells using *Core Cell > Load...* command.

To delete a core cell, select it and press the *Delete* key, or right-click on the core cell and select the *Delete* command. Deleting multiple selections is also possible.

To save a core cell to a file, right-click on the core cell and select *Save Core Cell As...* command.

To edit the internal structure of a core cell, double-click on the core cell. To ascend back to the previous level, double-click on the background.

To edit the outside properties of the core cell, right-click on the cell and select *Properties*. If the properties window is already open, it's enough to just click on the cell.

To edit its inside properties you have to go to the inside structure, right-click on the background and select *Owner Properties*. If the properties window is already open, it's enough to just click on the background.

## A.2. Core modules/macros

To create a normal core module or a macro, right-click in the central (the largest) area of the core structure and select one from the modules/macros from *Built-In Module*, *Expert Macro*, *Standard Macro*, or *User Macro* menu. You can also load a module/macro by right-clicking on the background and selecting *Load Module...* command.

An empty macro can be created from the *Built-In Module* menu.

To save a core module/macro to a file, right-click on it and select *Save As...* command.

To delete a core module/macro, select it and press the *Delete* key, or right-click on it and select the *Delete* command. Deleting multiple selections is also possible.

To edit the internal structure of the core macro, double-click on the macro. To ascend back to the previous level double-click on the background.



To edit the properties of the core module/macro, you have to go to the inside structure, right-click on the background, and select *Owner Properties*. If the properties window is already open, it's enough to just click on the background.

You can also access the properties of the module/macro from the outside, by right-clicking on it and selecting *Properties*. If the properties window is already open, it's enough to just click on the module/macro.

### A.3. Core ports

To create a core port right-click in the left (inputs) or the right (outputs) area of the core structure and select one from the available types from the *New* submenu.

To delete a core port, select it and press the *Delete* key, or right-click on it and select the *Delete* command. Deleting multiple selections (including mixed module/port selections) is also possible.

### A.4. Core structure editing

To move a core module, click on it and drag to the desired location. The ports can only be dragged vertically; their vertical order defines the order of their outside appearance.

To create a connection between an input of one module and an output of another module, click on one of them and drag to the other.



To remove a connection click on the connection wire to select it and press the *Delete* key. Alternatively you can drag from the input to the structure background.

To create a QuickConst, right-click on an input of a module and select *Connect to New QuickConst*. To access QuickConst properties, click on the QuickConst.

To create a QuickBus, right-click on an input or an output of a module and select *Connect to New QuickBus*. To connect an input or an output of a module to an existing QuickBus, right-click on the input or output and select one of the available busses in the *Connect to QuickBus* menu.

# Appendix B. Reaktor Core concept

## B.1. Signals and events

There are signals of float and integer types. Float ports look like this:  and integer ports look like this: .

Signals propagate through connections from outputs to connected inputs in the form of events. An event is a basic action that happens at a particular output and usually results in a change of the value at that output; an event can also result in the same value at the output.

All events originating from the same event source are considered simultaneous. Simultaneous means that if two such events arrive at several inputs of the same module, they arrive at the same time.

The same source means the same output, additionally under certain circumstances several outputs can be considered the same event source. For example, all core-cell audio inputs and standard sampling-rate clock connections are considered to be the same event source. During initialization all outputs sending events are considered the same event source. The same source doesn't mean the same value in this context, but rather it means simultaneousness.

Unless a given module is an event source, the only thing that can trigger it to process incoming values is one or more events arriving at its inputs. In case of multiple events only one output event will be generated, since the input events are considered to arrive simultaneously.

## B.2. Initialization

Initialization of the structures is done as follows. First, all values are reset to zeroes. Then an initialization event is sent simultaneously from all initialization sources. Generally, those are constants, core-cell inputs (not always), and clocks. That's it.

## B.3. OBC connections

OBC (Object Bus Connections) are connections between modules which do not send any signals, but declare that the modules share a common state (memory). The most common example of an OBC connection is a connection between *Read* and *Write* modules accessing the same stored value.

## B.4. Routing

You can use *Router* modules to direct the flow of events between two possible paths. In case a *Router* chooses one output path for incoming events, the other output receives no events (in particular, that means the value of the other output cannot change).

The *Router* is controlled by a *BoolCtl* type of input connection. On the other side of the connection you would typically have a *Compare* module (sometimes a few intermediate modules like *BoolCtl* macro ports can be placed between the *Compare* and the *Router*).

By using *Routers* you can dynamically enable or disable evaluations in parts of your structure.

Typically, after splitting the signal path in two using a *Router*, you would merge the two branches using a *Merge* or other module. Often you would merge a branch with the original, unsplit signal. But generally you are free to do whatever you want there (be careful with performance issues though).

## B.5. Latching

Latching is probably the most common technique in Reaktor Core. It means using *Latch* modules to prevent events from being sent at the wrong time. For example you wouldn't want a control signal event to trigger a computation in an audio loop.

Alternatively you can use macros from the *Expert Macros > Modulation* group, which are basically a set of the most typical combinations of *Latches* with arithmetic processing modules.

## B.6. Clocking

Clocks are sources of events. The clock events typically occur at regular time intervals corresponding to the clock rate. You normally need clocks to drive various modules, such as oscillators, filters, and so on. Most of the implementations of such modules do not require an explicit clock connection from the outside, but implicitly use a standard clock source available in core structures. That source is the sample rate clock, which runs at the default audio rate. Note that in event core cells, although the connection to the sample rate clock is available, the clock signal itself is not available. Therefore most oscillators, filters, and similar modules will not run in event core cells.

# Appendix C. Core macro ports

## C.1. In



Accepts an incoming event from the outside and forwards it unmodified to its own inside output.

The inside input connection can be used for overriding the default (disconnected) meaning of this port.

## C.2. Out



Accepts an incoming event at the inside input and forwards it unmodified to the outside.

## C.3. Latch (input)



Forwards an OBC connection from the outside of the macro to the inside of the macro. The inside input connection can be used for overriding the default (disconnected) meaning of this port.

## C.4. Latch (output)



Forwards an OBC connection from the inside of the macro to the outside of the macro.

## C.5. Bool C (input)



Forwards a BoolCtl connection from the outside of the macro to the inside of the macro. The inside input connection can be used for overriding the default (disconnected) meaning of this port.

## C.6. Bool C (output)



Forwards a BoolCtl connection from the inside of the macro to the outside of the macro.

# Appendix D. Core cell ports

## D.1. In (audio mode)



Provides access to the audio signal from the outside of the module. Sends regular events at the sampling rate (synchronous to the SR.C global sampling rate clock).

**INITIALIZATION EVENT:** sends an initialization event. The value is defined by the outside initialization

## D.2. Out (audio mode)



Makes the values received at the input inside available to the outside of the module. For any given time the last received value will be delivered to the outside.

## D.3. In (event mode)



Converts Reaktor primary-level events arriving from the outside into Reaktor Core events and forwards them to the inside.

**INITIALIZATION EVENT:** sends an initialization event if there's an initialization event received on the outside

## D.4. Out (event mode)



Converts Reaktor Core events arriving from the inside into Reaktor primary-level events and forwards them to the outside. If several event mode outputs

simultaneously receive Reaktor Core events, the corresponding primary-level events will be sent in the order from upper outputs to lower ones.

## Appendix E. Built-in busses

### E.1. SR.C

Sends regular clock events at the sampling rate

**INITIALIZATION EVENT:** always sends an initialization event

### E.2. SR.R

Provides the current sampling rate in Hz. Sends events with new values in response to sampling-rate changes.

**INITIALIZATION EVENT:** always sends an initialization event with initial sampling rate

## Appendix F. Built-in modules

### F.1. Const



Produces a signal of a constant value. The value is displayed in the module.

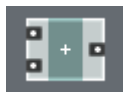
**INITIALIZATION EVENT:** during initialization sends the event of the specified value to the output. This is the only time when this module sends an event.

**PROPERTIES:**

Value

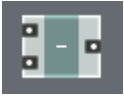
the value to be sent to the output

### F.2. Math > +



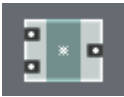
Produces the sum of the incoming signals at the output. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

### F.3. Math > -



Produces the difference of the incoming signals at the output (the signal at the lower input is subtracted from the signal at the upper input). The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

### F.4. Math > \*



Generates the multiplication product of the incoming signals at the output. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

### F.5. Math > /



Produces the quotient of the incoming signals at the output (the signal at the upper input is divided by the signal at the lower input). In integer mode performs a division with remainder, the remainder being discarded. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

### F.6. Math > |x|



Produces the absolute value of the incoming signal at the output. The output event is sent each time there is an event at the input.

### F.7. Math > -x



Produces the inverted value (changes sign) of the incoming signal at the output. The output event is sent each time there is an event at the input.

## F.8. Math > DN Cancel



Modifies the incoming signal in a way to kill denormal numbers. This is currently implemented by adding a very small constant. Works only on floating point numbers. The output event is sent each time there is an event at the input.

## F.9. Math > ~log

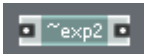


Computes an approximation of a logarithm of the incoming value. The output event is sent each time there is an event at the input.

### PROPERTIES:

<b>Base</b>	the logarithm base
<b>Precision</b>	the approximation precision (better precisions require more CPU)

## F.10. Math > ~exp



Computes an approximation of an exponent of the incoming value. The output event is sent each time there is an event at the input.

### PROPERTIES:

<b>Base</b>	the exponent base
<b>Precision</b>	the approximation precision (better precision requires more CPU)

## F.11. Bit > Bit AND



Performs the bitwise conjunction of the incoming signals. Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.



## F.12. Bit > Bit OR



Performs the bitwise disjunction of the incoming signals. Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## F.13. Bit > Bit XOR



Performs the bitwise exclusive disjunction of the incoming signals. Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## F.14. Bit > Bit NOT



Performs the bitwise inversion of the incoming signal. Works only on integers. The output event is sent each time there is an event at the input.

## F.15. Bit > Bit <<



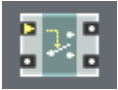
Bit-shifts the value at the upper input to the left (towards more significant bits). The amount of bits to shift by is specified by the lower input. The result for  $N < 0$  and  $N > 31$  is undefined (use it only for  $0 \leq N \leq 31$ ). Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## F.16. Bit > Bit >>



Bit-shifts the value at the upper input to the right (towards less significant bits). No sign extension is performed. The amount of bits to shift by is specified by the lower input. The result for  $N < 0$  and  $N > 31$  is undefined (use it only for  $0 \leq N \leq 31$ ). Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## F.17. Flow > Router



Routes the signal at the signal input (the lower one) to one of the two outputs depending on the state of the control signal (the upper input). If control signal is in the true state it routes to output 1 (the upper one), and if control signal is in the false state it routes to output 0 (the lower one). The output event is sent to exactly one of the outputs each time there is an event at the signal input.

## F.18. Flow > Compare



Produces a BoolCtl signal at the output indicating the result of comparison of the input values. The value at the upper input is placed to the left of the comparison sign and the value at the lower input to the right (so that the module on the picture above checks if upper value is greater than the lower one).

### PROPERTIES:

**Criterion**      the comparison operation to be performed

## F.19. Flow > Compare Sign



Produces a BoolCtl signal at the output indicating the result of the sign comparison of the input values. The value at the upper input is placed to the left of the comparison sign and the value at the lower input to the right (so

that the module on the picture above checks if the sign of the upper value is greater than the sign of the lower one).

The sign comparison is defined as follows:

- + is equal to +
- is equal to –
- + is larger than –

The sign of zero value is undefined, so arbitrary result may be produced should one of the compared values be zero.

**PROPERTIES:**

**Criterion**      the comparison operation to be performed

## F.20. Flow > ES Ctl



Produces a BoolCtl signal at the output indicating the momentary presence of an event at the input (that is, the control signal is true if there is an event at the input of this module at the given moment).

## F.21. Flow > ~BoolCtl



Produces a BoolCtl signal at the output which is an inversion of the input BoolCtl signal (true changes to false and vice versa).

## F.22. Flow > Merge

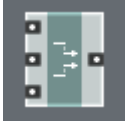


An output event is sent each time there is an event at any of the inputs or at several of them simultaneously. If only one input receives the event at a given time, the value of the output event will be equal to the value of the input event. If several inputs receive an event simultaneously, the value at the lowest input (among those receiving the event) will be selected. For example, if both second and third (counting from top) inputs receive an event, the value at the third input will be taken.

**PROPERTIES:**

**Input Count**    number of inputs of the module

## F.23. Flow > EvtMerge



The functionality is similar to that of the Merge module, except that all input values are ignored. The value of the output event is undefined. This module is intended to be used for generating signals to be used as clocks. Works only in floating point mode, since the value is not meant to be used anyway.

### PROPERTIES:

**Input Count** number of inputs of the module

## F.24. Memory > Read



Reads the stored value from the memory associated with the OBC chain that this module belongs to. The reading occurs in response to an event at the upper (clock) input and is sent to the upper output. The ports at the bottom are OBC master and slave connections, respectively.

## F.25. Memory > Write



Writes the value arriving at the upper input to the memory associated with the OBC chain that this module belongs to. The writing occurs in response to an event at the upper input. The ports at the bottom are OBC master and slave connections, respectively.

## F.26. Memory > R/W Order



This module does not perform any action. It can be inserted into a structure to control the processing order of OBC-connected modules. The OBC ports at the bottom are OBC master and slave connections, which are internally connected in a “thru” way. The OBC input at the top is called the “sidechain” connection and allows you to place this module logically after the module

connected to the sidechain input.

The sidechain connection can be connected only to normal Latch OBC type modules. The master and slave ports on the other hand can be connected to Latch or Array OBC type modules depending on the properties settings for the R/W Order module. In any case, the signal type and the precision must be the same for all connections to this module (e.F. you cannot connect side chain to an integer Read and the master and the slave to float modules at the same time).

**PROPERTIES:**

**Connection Type**      type of the “thru” port connection (latch or array)

## F.27. Memory > Array



Defines an array memory object. The module itself does not perform any action. All operations on the array are to be performed by the modules connected to the array output which is an OBC slave connection of array type.

**PROPERTIES:**

**Size**                      number of elements in the array

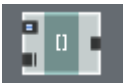
## F.28. Memory > Size [ ]



Reports the size of the array object connected to the input. The size is a constant integer value.

**INITIALIZATION EVENT:**      during initialization sends the event with the value of the array size to the output. This is the only time when this module sends an event.

## F.29. Memory > Index



Provides access to a single array element. The access is provided in a form of a latch OBC connection associated with the array element. The association is established and/or changed by sending an event to the upper (index) input of the Index module, which is *zero-based* and is always in the integer mode. The lower input is the master OBC connection to the array. The output provides

the latch OBC connection to the array element selected by the index input. The base value type and precision are, of course, the same for both input and output OBC connections and are controlled by the module properties.

## F.30. Memory > Table



Defines a pre-initialized read-only array. The module itself does not perform any action. All operations on the table are to be performed by the modules connected to the table output which is an OBC slave connection of array type.

### PROPERTIES:



#### FP Precision

edit the values in the table  
controls the formal precision of the output connection

## F.31. Macro



Provides a container for an internal structure. The number of inputs and outputs is not fixed and is defined by the internal structure.

### PROPERTIES:

#### FP Precision

controls the formal precision of the output connection

#### Look

changes between Large (label and port names visible) and Small (label and port names invisible) looks

#### Pin Alignment


controls the alignment of the ports in the outside view of the macro

#### Solid

controls the treatment of the macro by the core engine. If turned off the macro boundary is transparent for feedback resolution and possibly other things. Leave it ON unless you really, really know what you're doing!

#### Icon



button loads a new icon for the macro,  button clears the icon (no icon assigned)

## Appendix G. Expert macros

### G.1. Clipping > Clip Max / IClip Max



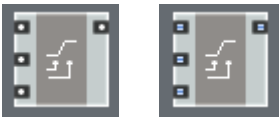
The signal at the upper input is clipped from the top by the threshold value at the lower input. Changes to the threshold do not generate events.

### G.2. Clipping > Clip Min / IClip Min



The signal at the upper input is clipped from the bottom by the threshold value at the lower input. Changes to the threshold do not generate events.

### G.3. Clipping > Clip MinMax / IClipMinMax



The signal at the upper input is clipped from the bottom by the threshold value at the middle input and from the top by the threshold value at the lower input. Changes to the thresholds do not generate events.

### G.4. Math > 1 div x



Computes the reciprocal of the input value

### G.5. Math > 1 wrap



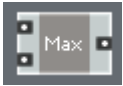
Wraps the incoming value into the range [-0.5..0.5] (the wrapping period is 1).

## G.6. Math > Imod



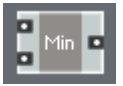
Computes the remainder of the division of upper value by the lower value. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## G.7. Math > Max / IMax



Computes the maximum of the input values. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## G.8. Math > Min / IMin



Computes the minimum of the input values. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## G.9. Math > round



Rounds the incoming value to the nearest integer. The result of rounding values exactly in the middle between two integers is not defined.

E.g. 1.5 could be rounded either to 1 or 2.

## G.10. Math > sign +/-



Outputs either 1 or -1 depending on the sign of the input (positive numbers produce 1, negative -1, the zero is never output).

## G.11. Math > sqrt (>0)



Square root approximation. Works only for inputs greater than 0.

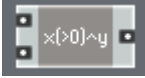


## G.12. Math > sqrt



Square root approximation (zero input is allowed).

## G.13. Math > $x(>0)^y$



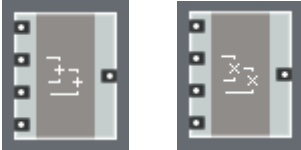
An approximation of  $x^y$ .  $x$  must be  $>0$ . The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

## G.14. Math > $x^2 / x^3 / x^4$



Computes the 2nd/3rd/4th power of  $x$ .

## G.15. Math > Chain Add / Chain Mult



Add/multiply the signals together in a top to bottom order. The output event is generated if there is one or more events at any of the inputs.

## G.16. Math > Trig-Hyp > 2 pi wrap



Wraps the incoming value into the range  $[-\pi.. \pi]$  (the wrapping period is  $2\pi$ ).

## G.17. Math > Trig-Hyp > arcsin / arccos / arctan



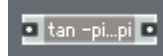
Arcsine/arccosine/arctangent approximation.

## G.18. Math > Trig-Hyp > sin / cos / tan



Sine/cosine/tangent approximation.

## G.19. Math > Trig-Hyp > sin $-\pi..pi$ / cos $-\pi..pi$ / tan $-\pi..pi$



Sine/cosine/tangent approximation (works only in the range  $[-\pi..pi]$ ).

## G.20. Math > Trig-Hyp > tan $-\pi/4..pi/4$



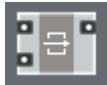
Tangent approximation (works only in the range  $[-\pi/4..pi/4]$ ).

## G.21. Math > Trig-Hyp > sinh / cosh / tanh



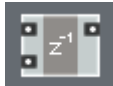
Hyperbolic sine/cosine/tangent approximation.

## G.22. Memory > Latch / lLatch



Latches (delays) the signal at the upper input until a clock event arrives at the lower input. If both events arrive simultaneously, the incoming signal will be let through immediately.

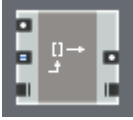
## G.23. Memory > $z^{-1}$ / $z^{-1} \text{ ndc}$



Sends out the last value that has been received at the upper input before a clock event arrives at the lower input in response to that clock event. If the clock input is disconnected the module will use the standard audio clock (SR.C) instead and effectively work as a one sample delay.

Both modules can automatically resolve feedback loops, however only  $z^{-1}$  version provides denormal cancellation. The  $z^{-1} \text{ ndc}$  version is meant to be used only in the places where denormals are not expected.

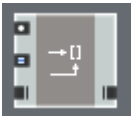
## G.24. Memory > Read []



Reads a value from an array at a given index (specified by the middle input) in response to an incoming clock event (at the upper input). The lower (OBC) input is the array connection.

Use the OBC output of the module to create OBC chains to serialize array access operations!

## G.25. Memory > Write []



Writes a value (received by the upper input) into an array at a given index (specified by the middle input). The writing operation is triggered by an incoming value. The lower (OBC) input is the array connection.

Use the OBC output of the module to create OBC chains to serialize array access operations!

## G.26. Modulation > $x + a$ / Integer > $lx + a$



Adds a parameter (lower input) to the signal (upper input) in response to an incoming signal event. Parameter changes do not generate events.

## G.27. Modulation > $x * a$ / Integer > $lx * a$



Multiplies the signal (upper input) by a parameter (lower input) in response to an incoming signal event. Parameter changes do not generate events.

## G.28. Modulation > $x - a$ / Integer > $lx - a$



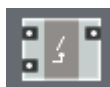
Subtracts a parameter (lower input) from the signal (upper input) in response to an incoming signal event. Parameter changes do not generate events.

## G.29. Modulation > $a - x$ / Integer > $la - x$



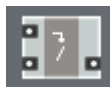
Subtracts the signal (lower input) from a parameter (upper input) in response to an incoming signal event. Parameter changes do not generate events.

## G.30. Modulation > $x / a$



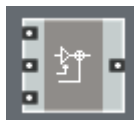
Divides the signal (upper input) by a parameter (lower input) in response to an incoming signal event. Parameter changes do not generate events.

## G.31. Modulation > $a / x$



Divides a parameter (upper input) by the signal (lower input) in response to an incoming signal event. Parameter changes do not generate events.

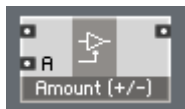
## G.32. Modulation > $xa + y$



Multiplies the signal at the upper input by the gain parameter (middle input) and adds the result to the signal at the lower input. Events at either or both signal inputs generate a new output value, events at the parameter input do not.

# Appendix H. Standard macros

## H.1. Audio Mix-Amp > Amount



Provides linear invertible control of the amount (amplitude) of an audio signal.

$A = 0$	mutes the signal
$A = 1$	leaves the signal intact
$A = -1$	inverts the signal
Typical usage:	controlling audio feedback amount

## H.2. Audio Mix-Amp > Amp Mod



Modulates the audio signal's amplitude by a given amount (AM) in the linear scale.

$AM = 1$	doubles the amplitude
$AM = 0$	no change
$AM = -1$	mutes the signal
Typical usage:	tremelo, AM.

## H.3. Audio Mix-Amp > Audio Mix



Mixes two audio signals together.

## H.4. Audio Mix-Amp > Audio Relay



Switches between two input audio signals. If 'x' is greater than 0, picks up signal 1, otherwise signal 0.

## H.5. Audio Mix-Amp > Chain (amount)



Changes the audio signal's amplitude by a given linear amount (A) and mixes it with the chained audio signal (>>).

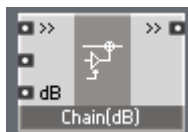
A = 0          signal is muted

A = 1          signal is unchanged

A = -1        signal is inverted

Typical usage: audio mixing chains, audio feedback amount control

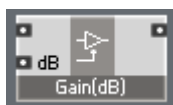
## H.6. Audio Mix-Amp > Chain (dB)



Changes the audio signal's amplitude by a given amount of dB and mixes it with the chained audio signal (>>).

Typical usage: audio mixing chains

## H.7. Audio Mix-Amp > Gain (dB)



Changes the audio signal's amplitude by a given amount in dB.

+6 dB          doubles the amplitude

0 dB          no change

-6 dB          halves the amplitude

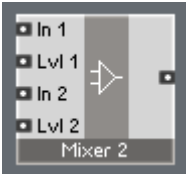
Typical usage: signal volume control in dB scale

## H.8. Audio Mix-Amp > Invert



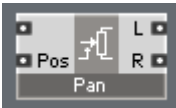
Inverts the polarity of the audio signal

## H.9. Audio Mix-Amp > Mixer 2 ... 4



Mixes the incoming audio signals (*In 1*, *In 2*, ...) attenuating their levels by the specified amounts in dB (*Lvl 1*, *Lvl 2*, ...).

## H.10. Audio Mix-Amp > Pan



Pans the incoming audio signal using a parabolic curve.

- 1 hard left
- 0 center
- 1 hard right

## H.11. Audio Mix-Amp > Ring-Amp Mod

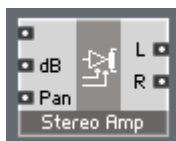


The carrier audio signal (at the upper input) is modulated by the audio signal at the *Mod* input. The type of the modulation is controlled by the *R/A* input, which smoothly morphs between ring and amplitude modulation.

- $R/A = 0$  ring modulation
- $R/A = 1$  amplitude modulation

(For true amplitude modulation the modulator amplitude should not exceed 1)

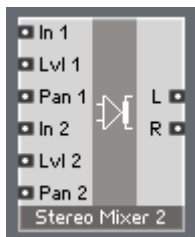
## H.12. Audio Mix-Amp > Stereo Amp



Amplifies a monophonic audio signal by a given amount in dB and pans it to the specified position. The pan position is defined as:

- 1 hard left
- 0 center
- 1 hard right

## H.13. Audio Mix-Amp > Stereo Mixer 2 ... 4



Mixes the input audio signals (*In 1*, *In 2*, ...), attenuating their levels by the specified amounts of dB (*Lvl 1*, *Lvl 2*, ...) and panning them to the specified positions (*Pan 1*, *Pan 2*, ...). The pan positions are defined as:

- 1 hard left
- 0 center
- 1 hard right

## H.14. Audio Mix-Amp > VCA



Audio amplifier with direct linear control for the amplitude.

- A = 0 mutes the signal
- A = 1 leaves the signal unchanged

Typical usage: connect the amplitude envelope to the *A* input.

Note: for invertible amplification use the Audio Amount module.



## H.15. Audio Mix-Amp > XFade (lin)

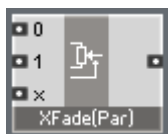


Audio crossfade with linear curve.

- $x = 0$       only signal 0 is heard
- $x = 0.5$     equal mix of both signals
- $x = 1$       only signal 1 is heard

Note: a parabolic crossfade usually gives better sounding results.

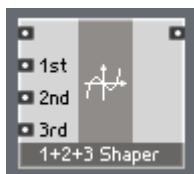
## H.16. Audio Mix-Amp > XFade (par)



Audio crossfade with parabolic curve. Usually gives better sounding results than linear crossfade.

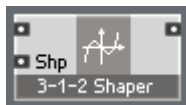
- $x = 0$       only signal 0 is heard
- $x = 0.5$     equal mix of both signals
- $x = 1$       only signal 1 is heard

## H.17. Audio Shaper > 1+2+3 Shaper



Provides audio-signal controllable shaping of 2nd and 3rd order. The *1st* input specifies the amount of the original signal in the output (1=unchanged, 0=none). The *2nd* and *3rd* inputs specify the amounts of the 2nd and 3rd order distortion. respectively.

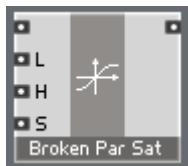
## H.18. Audio Shaper > 3-1-2 Shaper



Audio signal shaper with variable amount of 2nd and 3rd order distortion. The distortion amount and type is controlled by the Shp input:

Shp = 0      no shaping  
Shp > 0      3rd order shaping  
Shp < 0      2nd order shaping

## H.19. Audio Shaper > Broken Par Sat



Broken parabolic saturator. Has a linear segment around the zero level.

L input specifies the output level for the “full saturation” (typical value = 1).

H input specifies the hardness (range 0...1). Larger values correspond to a larger linear segment in the middle.

S input controls the symmetry of the shaping curve (range -1...1).

At 0 the curve is symmetric.

## H.20. Audio Shaper > Hyperbol Sat



Simple hyperbolic saturator. The L input specifies the full saturation output level (default = 1). However the full saturation is never reached with this type of saturator.

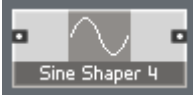
## H.21. Audio Shaper > Parabol Sat



Simple parabolic saturator. The L input specifies the full saturation output level (default = 1).

Note: the full saturation is reached at the input level equal to 2L.

## H.22. Audio Shaper > Sine Shaper 4 / 8



4th / 8th order sine shaper. The 8th order shaper has a better sine approximation but takes more CPU.

## H.23. Control > Ctl Amount



Linear invertible control of the amount (amplitude) of the control signal.

- A = 0      turns off the signal
- A = 1      leaves the signal unchanged
- A = -1     inverts the signal

Typical usage: controlling modulation amount

## H.24. Control > Ctl Amp Mod



Modulates the control signal's amplitude by a given amount (AM) in linear scale.

- AM = 1      doubles the amplitude
- AM = 0      no change
- AM = -1     mutes the signal

## H.25. Control > Ctl Bi2Uni



Changes a  $-1...1$  bipolar signal into a unipolar one. The  $a$  input controls the amount of change, at 0 there's no change, at 1 there is 100% change (default is 1).

Typical usage: connect immediately after an LFO to adjust the polarity of the modulation.

## H.26. Control > Ctl Chain



Changes the control signal's amplitude by a given linear amount  $A$  and mixes it to the chained control signal  $>>$ .

- $A = 0$       signal is turned off
- $A = 1$       signal is unchanged
- $A = -1$      signal is inverted

Typical usage: control mixing chains

## H.27. Control > Ctl Invert



Inverts the control signal's polarity

## H.28. Control > Ctl Mix



Mixes two control signals.

## H.29. Control > Ctl Mixer 2



Mixes two control signals  $In\ 1$ ,  $In\ 2$  together using the specified gain factors  $A\ 1$ ,  $A\ 2$ .

- $A = 0$       no signal
- $A = 1$       unchanged
- $A = -1$      inverted

## H.30. Control > Ctl Pan



“Pans” a control signal using a parabolic curve.

Pos = -1      hard left  
Pos = 0       center  
Pos = 1       hard right

## H.31. Control > Ctl Relay



Switches between two control signals. If  $x > 0$  picks up signal 1, else picks up signal 0.

## H.32. Control > Ctl XFade



Crossfades between two control signals using a linear curve.

$x = 0$             only signal 0 comes through  
 $x = 0.5$         equal mix of both signals  
 $x = 1$             only signal 1 comes through

## H.33. Control > Par Ctl Shaper



Applies a double parabolic curve to a controller signal. The input signal must be in  $-1..0..1$  range. The output signal will also have the range of  $-1..0..1$ . The amount of bending is controlled by the  $b$  input (the range is also  $-1..0..1$ ).

$b = 0$             no bend (linear curve)  
 $b = -1$         max possible bend “towards” X axis  
 $b = 1$             max possible bend “towards” Y axis

You can also use this shaper for signals whose range is 0..1, in which case only half of the curve will be used.

Typical use: velocity and other controllers shaping

## H.34. Convert > dB2AF



Converts a control signal from dB scale to linear amplitude gain factor.

0 dB → 1.0

-6 dB → 0.5

etc.

## H.35. Convert > dP2FF



Converts a control signal from an interval in pitch scale (pitch difference in semitones) to a frequency ratio.

12 semitones → 2

-12 semitones → -2

etc.

## H.36. Convert > logT2sec



Converts Reaktor primary-level logarithmic time (used for envelopes) to seconds.

0 → 0.001 sec

60 → 1 sec

etc.

## H.37. Convert > ms2Hz



Converts time period in milliseconds into corresponding frequency in Hz.

E.g. 100ms → 10 Hz.

### H.38. Convert > ms2sec



Converts the time specified in milliseconds into time specified in seconds.  
E.g. 500ms → 0.5 sec.

### H.39. Convert > P2F



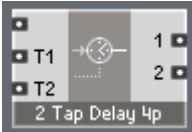
Converts a control signal from pitch scale to frequency scale.  
E.g. pitch 69 → 440 Hz.

### H.40. Convert > sec2Hz



Converts time period in seconds into corresponding frequency in Hz.  
E.g. 0.1sec → 10 Hz.

### H.41. Delay > 2 / 4 Tap Delay 4p



2/4-tap delay with 4 point interpolation. T1...T4 inputs specify the delay time in seconds for each of the taps.

The maximum delay time defaults to 44,100 samples which is 1sec at 44.1kHz.  
To adjust the time change the size of the array in the delay macro.

### H.42. Delay > Delay 1p / 2p / 4p



1-point (non-interpolated)/2-point interpolated/4-point interpolated delay. T input specifies the delay time in seconds.

The maximum delay time defaults to 44100 samples which is 1sec at 44.1kHz.  
To adjust the time change the size of the array in the delay macro.

Use interpolated versions of delays for modulated delays. For non-modulated (fixed time) delays non-interpolated version is normally better.

### H.43. Delay > Diff Delay 1p / 2p / 4p



1-point (non-interpolated)/2-point interpolated/4-point interpolated diffusion delay. T input specifies the delay time in seconds. The Dffs input sets the diffusion factor.

The maximum delay time defaults to 44,100 samples which is 1sec at 44.1kHz. To adjust the time change the size of the array in the delay macro.

### H.44. Envelope > ADSR



Generates an ADSR Envelope.

- A, D, R specify attack, decay and release times in seconds
- S specifies sustain level (range 0..1, at 1 sustain level is equal to the peak level)
- G gate input. Positive incoming events (re-)start the envelope. Zero or negative events close the envelope
- GS gate sensitivity. At zero sensitivity the envelope peak has always amplitude of 1. At sensitivity equal to one, the peak level is equal to the positive gate level.
- RM retrigger mode. Selects between analog/digital mode and between retrigger/legato mode. In “digital” mode the envelope always restarts from zero while in “analog” mode the envelope restarts from its current output level. In “retrigger” mode consecutive positive gate events will restart the envelope, while in “legato” mode it restarts only when the gate changes from negative/zero



to positive. The allowed RM values are following:

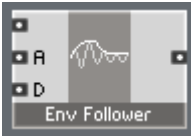
RM = 0 analog retrigger (default)

RM = 1 analog legato

RM = 2 digital retrigger

RM = 3 digital legato

## H.45. Envelope > Env Follower



Outputs a control signal which “follows” the envelope of the incoming audio signal. The A and D inputs specify the follow attack and decay time parameters in seconds.

## H.46. Envelope > Peak Detector



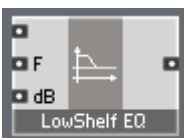
Outputs the “last” peak level of the incoming audio as a control signal. The D input specifies the output level decay time parameter in seconds.

## H.47. EQ > 6dB LP/HP EQ



1-pole (6dB/octave) lowpass/highpass EQ. The F input specifies the cutoff frequency (in Hz) for both LP and HP outputs.

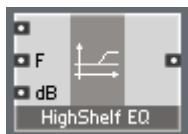
## H.48. EQ > 6dB LowShelf EQ



1-pole low-shelving EQ. The dB input specifies the low frequency boost in dB

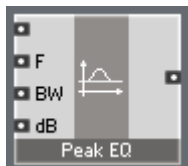
(negative values will cut the frequencies), the F input specifies the transition mid-frequency in Hz.

### H.49. EQ > 6dB HighShelf EQ



1-pole high-shelving EQ. The dB input specifies the high frequencies boost in dB (negative values will cut the frequencies), the F input specifies the transition mid-frequency in Hz.

### H.50. EQ > Peak EQ



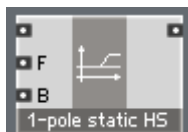
2-pole peak/notch EQ. The F input specifies the center frequency in Hz, the BW input specifies the EQ bandwidth in octaves and dB input specifies the peak height (negative values produce a notch).

### H.51. EQ > Static Filter > 1-pole static HP



1-pole static highpass filter. The F input specifies the cutoff frequency in Hz.

### H.52. EQ > Static Filter > 1-pole static HS



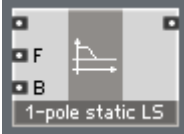
1-pole static high-shelving filter. The F input specifies the cutoff frequency in Hz and the B input specifies the high frequency boost in dB.

### H.53. EQ > Static Filter > 1-pole static LP



1-pole static lowpass filter. The F input specifies the cutoff frequency in Hz.

### H.54. EQ > Static Filter > 1-pole static LS



1-pole static low-shelving filter. The F input specifies the cutoff frequency in Hz and the B input specifies the low frequency boost in dB.

### H.55. EQ > Static Filter > 2-pole static AP



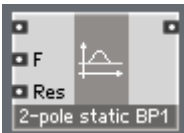
2-pole static allpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

### H.56. EQ > Static Filter > 2-pole static BP



2-pole static bandpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

### H.57. EQ > Static Filter > 2-pole static BP1



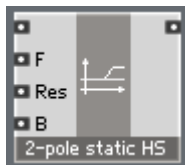
2-pole static bandpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1). The amplification at cutoff frequency is always 1 regardless of the resonance.

## H.58. EQ > Static Filter > 2-pole static HP



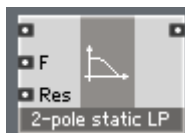
2-pole static highpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

## H.59. EQ > Static Filter > 2-pole static HS



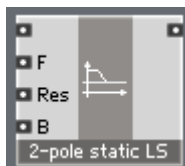
2-pole static high-shelving filter. The F input specifies the cutoff frequency in Hz, the Res input specifies the resonance (0..1), and the B input specifies the high frequency boost in dB.

## H.60. EQ > Static Filter > 2-pole static LP



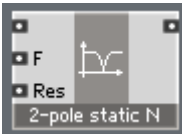
2-pole static lowpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

## H.61. EQ > Static Filter > 2-pole static LS



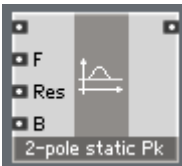
2-pole static low-shelving filter. The F input specifies the cutoff frequency in Hz, the Res input specifies the resonance (0..1), and the B input specifies the low frequency boost in dB.

## H.62. EQ > Static Filter > 2-pole static N



2-pole static notch filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

## H.63. EQ > Static Filter > 2-pole static Pk



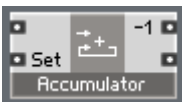
2-pole static peak filter. The F input specifies the cutoff frequency in Hz, the Res input specifies the resonance (0..1), and the B input specifies the center frequency boost in dB.

## H.64. EQ > Static Filter > Integrator



Integrates the incoming audio signal using the rectangular sum method. An event at the Rst input resets the integrator output to the value of this event.

## H.65. Event Processing > Accumulator



Computes the sum of the values at the upper input. An event at the Set input resets the output to the value of this event. The lower output value is the sum of all previous events, the upper output value is the sum of all previous event except the last one.

## H.66. Event Processing > Clk Div



Clock frequency divider. The clock events arriving at the upper input will be filtered, allowing only 1st,  $N+1$ th,  $2N+1$ th etc. events to come through ( $N$  is the value at the lower input and specifies the division ratio).

## H.67. Event Processing > Clk Gen



Generates clock events at the rate specified by the input (in Hz). This module works only inside audio core cells.

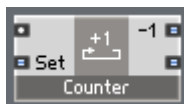
## H.68. Event Processing > Clk Rate



Estimates the rate and the period of the incoming clock events. The  $F$  output is the rate in Hz and the  $T$  output is the period in seconds. This module works only inside audio core cells.

The initial period value is zero and the rate is a very large value. You get reasonable output only after the second clock event.

## H.69. Event Processing > Counter



Counts the number of events at the upper input. An event at the Set input resets the output to the value of this event. The lower output value is the count of all previous events, the upper output value is the count of all previous events except the last one.

## H.70. Event Processing > Ctl2Gate



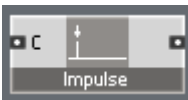
Converts a control (or audio) signal at the upper input to the gate signal with an amplitude defined by the lower input. Positive zero crossings open the gate, negative zero crossings close the gate.

## H.71. Event Processing > Dup Flt / IDup Flt



Filters out events with duplicate values (only events with values different from the previous one will be let through).

## H.72. Event Processing > Impulse



Generates a one sample impulse of amplitude 1 in response to an incoming event. This module works only inside audio core cells.

## H.73. Event Processing > Random



Generates random numbers in response to the incoming clocks. The output range is  $-1..1$ . An event at the Seed input will “reseed” the generator with the value of this event.

## H.74. Event Processing > Separator / ISeparator



Events at the upper input with the values larger than the Thld value will be routed to the Hi output. The rest will be routed to the Lo output.

## H.75. Event Processing > Thld Crossing



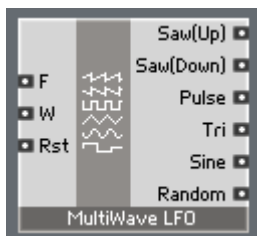
Whenever a rising signal at the upper input crosses the threshold specified by the lower input, an event will be sent from the Up output. Whenever a falling signal crosses the threshold, an event will be sent from the Dn output.

## H.76. Event Processing > Value / IValue



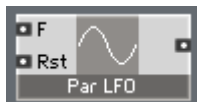
Changes the value of an incoming event at the upper input to the value available at this time at the lower input.

## H.77. LFO > MultiWave LFO



Outputs several phase-locked low-frequency waveforms simultaneously. The F input specifies the rate in Hz, the W input controls the pulse width (range -1..0..1, affects pulse output only), the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

## H.78. LFO > Par LFO



Generates a parabolic low-frequency control signal. The F input specifies the rate in Hz, the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).



## H.79. LFO > Random LFO



Generates a random low-frequency stepped control signal (“random sample-and-hold”). The F input specifies the rate in Hz, the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

## H.80. LFO > Rect LFO



Generates a rectangular low-frequency control signal. The F input specifies the rate in Hz, the W input controls the pulse width (range -1..0..1), the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

## H.81. LFO > Saw(down) LFO



Generates a falling sawtooth low-frequency control signal. The F input specifies the rate in Hz, the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

## H.82. LFO > Saw(up) LFO



Generates a rising sawtooth low-frequency control signal. The F input specifies the rate in Hz, the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

## H.83. LFO > Sine LFO



Generates a sine-shaped low-frequency control signal. The F input specifies the rate in Hz, the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

## H.84. LFO > Tri LFO



Generates a triangular low-frequency control signal. The F input specifies the rate in Hz, the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

## H.85. Logic > AND



Performs a conjunction of two logical signals (the output is 1 only if both inputs are 1). For input values other than 0 or 1 the result is undefined.

## H.86. Logic > Flip Flop



The output is flipped between 0 and 1 each time the clock input receives an event.

## H.87. Logic > Gate2L



Converts gate signal to logic signal. Open gate produces output value 1, closed gate produces output value 0.

## H.88. Logic > GT / IGT



Compares the two incoming float/integer values and outputs 1 if the upper value is greater than the lower value, otherwise outputs 0.

## H.89. Logic > EQ



Compares the two incoming integer values and outputs 1 if both values are equal, otherwise outputs 0.

## H.90. Logic > GE



Compares the two incoming integer values and outputs 1 if the upper value is greater or equal to the lower value, otherwise outputs 0.

## H.91. Logic > L2Clock



Converts a logic signal into the clock signal. Switching the input signal from 0 to 1 sends the clock event. For input values other than 0 or 1 the functionality is undefined.

## H.92. Logic > L2Gate



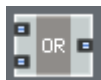
Converts a logic signal to a gate signal. Switching the input signal from 0 to 1 opens the gate, switching back closes the gate. The open gate level is defined by the value at the lower input (default = 1). For input values other than 0 or 1 the functionality is undefined.

## H.93. Logic > NOT



Converts 1 to 0 and vice versa. For input values other than 0 or 1 the result is undefined.

## H.94. Logic > OR



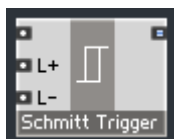
Performs a disjunction of two logical signals (the output is 1 if at least one of the inputs is 1). For input values other than 0 or 1 the result is undefined.

## H.95. Logic > XOR



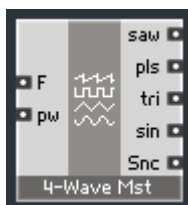
Performs an exclusive disjunction of two logical signals (the output is 1 if one of the inputs is equal to 1 and the other equal to 0). For input values other than 0 or 1 the result is undefined.

## H.96. Logic > Schmitt Trigger



Switches the output to 1 if the input value becomes larger than L+ (default 0.67), switches the output to 0 if the input value becomes less than L- (default 0.33).

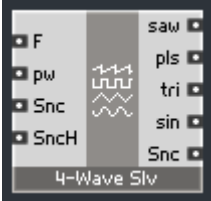
## H.97. Oscillators > 4-Wave Mst



Generates 4 phase-locked audio waveforms. The frequency is specified by the *F* input (in Hz). The pulse width is specified by the *pw* input (range  $-1..0..1$ , affects only the pulse waveform).

This oscillator can oscillate at negative frequencies and additionally offers a synchronization output for *4-Wave S/v* oscillator.

## H.98. Oscillators > 4-Wave S/v



Generates 4 phase-locked audio waveforms. The frequency is specified by the *F* input (in Hz). The pulse width is specified by the *pw* input (range  $-1..0..1$ , affects only the pulse waveform).

This oscillator can oscillate at negative frequencies and can be synchronized to another *4-Wave Mst/S/v* oscillator. The *SncH* input controls the synchronization hardness (0 = no sync, 1 = hard sync,  $0..1$  = various degrees of soft sync). A synchronization output for another *4-Wave S/v* oscillator is also provided.

## H.99. Oscillators > Binary Noise



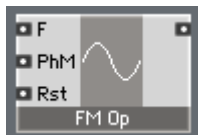
Binary white noise generator. Outputs randomly alternating values of 1 and  $-1$ . An incoming event at the *Seed* input would (re-)initialize the internal random generator with a given seed value.

## H.100. Oscillators > Digital Noise



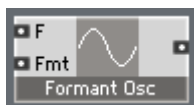
Digital white noise generator. Outputs random values in the range  $-1..1$ . An incoming event at the *Seed* input would (re-)initialize the internal random generator with a given seed value.

## H.101. Oscillators > FM Op



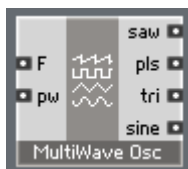
Classical FM operator. Outputs a sine wave whose frequency is defined by the F input (in Hz). The sine can be phase-modulated by the PhM input (in radians). An incoming event at the Rst input would restart the oscillator to the phase specified by the value of this event (range 0..1).

## H.102. Oscillators > Formant Osc



Generates a waveform with a fundamental frequency specified by the F input (in Hz) and the formant frequency specified by the Fmt input (in Hz).

## H.103. Oscillators > MultiWave Osc



Generates 4 phase-locked audio waveforms. The frequency is specified by the F input (in Hz). The pulse width is specified by the 'pw' input (range -1..0..1, affects only the pulse waveform).

This oscillator cannot oscillate at negative frequencies.

## H.104. Oscillators > Par Osc



Generates a parabolic audio waveform. The F input specifies the frequency in Hz.

## H.105. Oscillators > Quad Osc



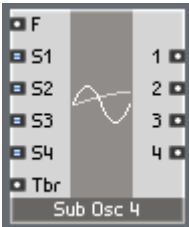
Generates a pair of phase-locked sine waveforms with a phase shift of 90 degrees. The F input specifies the frequency in Hz.

## H.106. Oscillators > Sin Osc



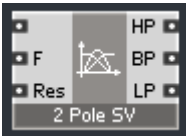
Generates a sine wave. The F input specifies the frequency in Hz.

## H.107. Oscillators > Sub Osc 4



Generates 4 phase-locked subharmonics. The fundamental frequency is specified by the F input (in Hz). The subharmonic numbers are specified by S1..S4 inputs (range 1..120). The Tbr input controls the harmonic content of the output waveform (range 0..1).

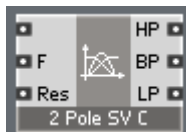
## H.108. VCF > 2 Pole SV



2-pole state-variable filter. The F input specifies the cutoff in Hz and the Res input specifies the resonance (range 0..0.98).

The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

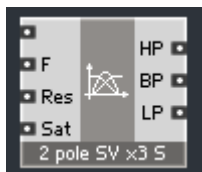
## H.109. VCF > 2 Pole SV C



2-pole state-variable filter (compensated version). Offers an improved behavior at high cutoff settings. The F input specifies the cutoff in Hz and the Res input specifies the resonance (range 0..0.98). You also can use negative resonance values which will smear the slope further.

The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

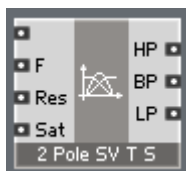
## H.110. VCF > 2 Pole SV (x3) S



2-pole state-variable filter with optional oversampling (x3 version) and saturation. The F input specifies the cutoff in Hz, the Res input specifies the resonance (range 0..1), the Sat input specifies the saturation level (typical range 8..32).

The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

## H.111. VCF > 2 Pole SV T (S)

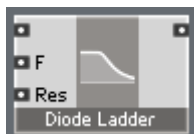


2-pole state-variable filter with table compensation and optional saturation (S version). Offers an improved behavior at high cutoff settings, but slightly different from the 2 Pole SV C version. The F input specifies the cutoff in Hz, the Res input specifies the resonance (range 0..1), the Sat input specifies the saturation level (typical range 8..32).

The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

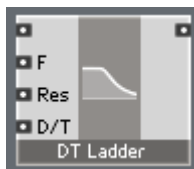


## H.112. VCF > Diode Ladder



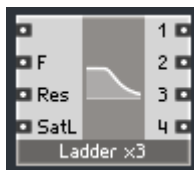
Diode-ladder filter linear emulation. The F input specifies the cutoff in Hz and the Res input specifies the resonance (range 0..0.98).

## H.113. VCF > D/T Ladder



Ladder filter linear emulation, can be morphed between diode and transistor ladder behavior. The F input specifies the cutoff in Hz, the Res input specifies the resonance (range 0..0.98), the D/T input morphs between diode and transistor (0=diode, 1=transistor).

## H.114. VCF > Ladder x3



An emulation of saturating transistor ladder filter (x3 times oversampled). The F input specifies the cutoff in Hz, the Res input specifies the resonance (range 0..1), the SatL input specifies the saturation level (typical range 1..32). The outputs 1-4 are taken from the corresponding taps of the emulated ladder. Take the 4th tap for the 'classic' ladder filter sound.

# Appendix I. Core cell library

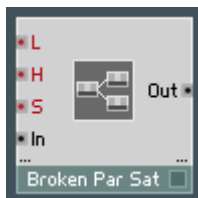
## I.1. Audio Shaper > 3-1-2 Shaper



Audio signal shaper with variable amount of 2nd and 3rd order distortion. The distortion amount and type is controlled by the Shp input:

- Shp = 0      no shaping
- Shp > 0     3rd order shaping
- Shp < 0     2nd order shaping

## I.2. Audio Shaper > Broken Par Sat



Broken parabolic saturator. Has a linear segment around the zero level.

- L*    input specifies the output level for the full saturation (typical value = 1).
- H*    input specifies the hardness (range 0...1). Larger values correspond to a larger linear segment in the middle.
- S*    input controls the symmetry of the shaping curve (range -1...1). At 0 the curve is symmetric.

## I.3. Audio Shaper > Hyperbol Sat



Simple hyperbolic saturator. The *L* input specifies the full saturation output level (typical value = 1). However the full saturation is never reached with this type of saturator.

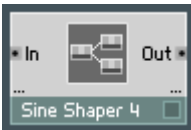
## I.4. Audio Shaper > Parabol Sat



Simple parabolic saturator. The  $L$  input specifies the full saturation output level (typical value = 1).

Note: the full saturation is reached at the input level equal to  $2L$ .

## I.5. Audio Shaper > Sine Shaper 4/8



4th / 8th order sine shaper. The 8th order shaper has a better sine approximation but takes more CPU.

## I.6. Control > ADSR



Generates an ADSR Envelope.

- $A, D, R$  specify attack, decay and release times in seconds
- $S$  specifies sustain level (range 0..1, at 1 sustain level is equal to the peak level)
- $G$  gate input. Positive incoming events (re-)start the envelope. Zero or negative events close the envelope
- $GS$  gate sensitivity. At zero sensitivity the envelope peak has always amplitude of 1. At sensitivity equal to one, the peak level is equal to the positive gate level.
- $RM$  retrigger mode. Selects between analog/digital mode and between retrigger/legato mode. In digital mode the envelope always restarts from zero while in analog mode the envelope restarts from its

current output level. In retrigger mode consecutive positive gate events will restart the envelope, while in legato mode it restarts only when the gate changes from negative/zero to positive. The allowed RM values are following:

RM = 0	analog retrigger (default)
RM = 1	analog legato
RM = 2	digital retrigger
RM = 3	digital legato

## I.7. Control > Env Follower



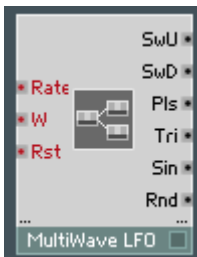
Outputs a control signal which follows the envelope of the incoming audio signal. The *A* and *D* inputs specify the follow attack and decay time parameters in seconds.

## I.8. Control > Flip Flop



The output is flipped between 0 and 1 each time the trigger input receives an event.

## I.9. Control > MultiWave LFO



Outputs several phase-locked low-frequency waveforms simultaneously. The *Rate* input specifies the rate in Hz, the *W* input controls the pulse width (range -1..0..1, affects pulse output only), the events at the *Rst* input restart the LFO at the phase specified by the event value (range 0..1).

## I.10. Control > Par Ctl Shaper



Applies a double parabolic curve to a controller signal. The input signal must be in  $-1..0..1$  range. The output signal will also have the range of  $-1..0..1$ . The amount of bending is controlled by the *b* input (the range is also  $-1..0..1$ ).

$b = 0$  no bend (linear curve)

$b = -1$  max possible bend towards X axis

$b = 1$  max possible bend towards Y axis

You can also use this shaper for signals whose range is  $0..1$ , in which case only half of the curve will be used.

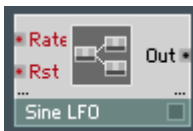
Typical use: velocity and other controllers shaping

## I.11. Control > Schmitt Trigger



Switches the output to 1 if the input value becomes larger than  $L+$  (default 0.67), switches the output to 0 if the input value becomes less than  $L-$  (default 0.33).

## I.12. Control > Sine LFO



Generates a sine-shaped low-frequency control signal. The *Rate* input specifies the rate in Hz, the events at the *Rst* input restart the LFO at the phase specified by the event value (range  $0..1$ ).

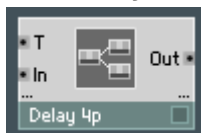
## I.13. Delay > 2/4 Tap Delay 4p



2/4-tap delay with 4 point interpolation.  $T1...T4$  inputs specify the delay time in milliseconds for each of the taps.

The maximum delay time defaults to 44100 samples which is 1sec at 44.1kHz. To adjust the time change the size of the array in the delay macro.

## I.14. Delay > Delay 4p



4-point interpolated delay.  $T$  input specifies the delay time in milliseconds. The maximum delay time defaults to 44100 samples which is 1sec at 44.1kHz. To adjust the time change the size of the array in the delay macro.

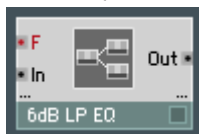
## I.15. Delay > Diff Delay 4p



4-point interpolated diffusion delay.  $T$  input specifies the delay time in milliseconds. The  $Dffs$  input sets the diffusion factor.

The maximum delay time defaults to 44100 samples which is 1sec at 44.1kHz. To adjust the time change the size of the array in the delay macro.

## I.16. EQ > 6dB LP/HP EQ



1-pole (6dB/octave) lowpass/highpass EQ. The  $F$  input specifies the cutoff frequency (in Hz).

## I.17. EQ > HighShelf EQ



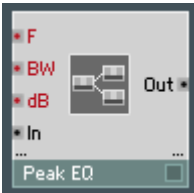
1-pole high-shelving EQ. The  $dB$  input specifies the high frequencies boost in dB (negative values will cut the frequencies), the  $F$  input specifies the transition mid-frequency in Hz.

## I.18. EQ > LowShelf EQ



1-pole low-shelving EQ. The  $dB$  input specifies the low frequencies boost in dB (negative values will cut the frequencies), the  $F$  input specifies the transition mid-frequency in Hz.

## I.19. EQ > Peak EQ



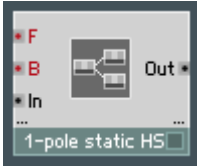
2-pole peak/notch EQ. The  $F$  input specifies the center frequency in Hz, the  $BW$  input specifies the EQ bandwidth in octaves and  $dB$  input specifies the peak height (negative values produce a notch).

## I.20. EQ > Static Filter > 1-pole static HP



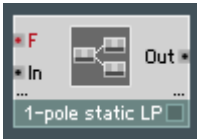
1-pole static highpass filter. The  $F$  input specifies the cutoff frequency in Hz.

### I.21. EQ > Static Filter > 1-pole static HS



1-pole static high-shelving filter. The  $F$  input specifies the cutoff frequency in Hz and the  $B$  input specifies the high frequency boost in dB.

### I.22. EQ > Static Filter > 1-pole static LP



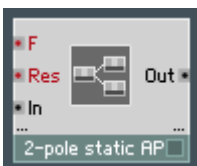
1-pole static lowpass filter. The  $F$  input specifies the cutoff frequency in Hz.

### I.23. EQ > Static Filter > 1-pole static LS



1-pole static low-shelving filter. The  $F$  input specifies the cutoff frequency in Hz and the  $B$  input specifies the low frequency boost in dB.

### I.24. EQ > Static Filter > 2-pole static AP



2-pole static allpass filter. The  $F$  input specifies the cutoff frequency in Hz and the  $Res$  input specifies the resonance (0..1).



### I.25. EQ > Static Filter > 2-pole static BP



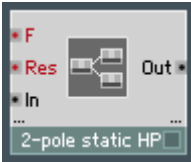
2-pole static bandpass filter. The  $F$  input specifies the cutoff frequency in Hz and the  $Res$  input specifies the resonance (0..1).

### I.26. EQ > Static Filter > 2-pole static BP1



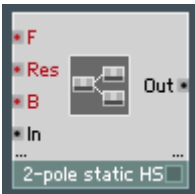
2-pole static bandpass filter. The  $F$  input specifies the cutoff frequency in Hz and the  $Res$  input specifies the resonance (0..1). The amplification at cutoff frequency is always 1 regardless of the resonance.

### I.27. EQ > Static Filter > 2-pole static HP



2-pole static highpass filter. The  $F$  input specifies the cutoff frequency in Hz and the  $Res$  input specifies the resonance (0..1).

### I.28. EQ > Static Filter > 2-pole static HS



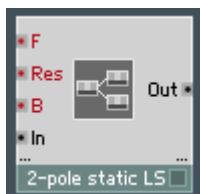
2-pole static high-shelving filter. The  $F$  input specifies the cutoff frequency in Hz, the  $Res$  input specifies the resonance (0..1), and the  $B$  input specifies the high frequency boost in dB.

## I.29. EQ > Static Filter > 2-pole static LP



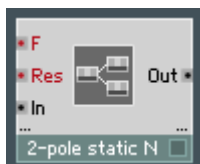
2-pole static lowpass filter. The *F* input specifies the cutoff frequency in Hz and the *Res* input specifies the resonance (0..1).

## I.30. EQ > Static Filter > 2-pole static LS



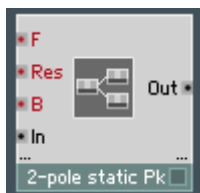
2-pole static low-shelving filter. The *F* input specifies the cutoff frequency in Hz, the *Res* input specifies the resonance (0..1), and the *B* input specifies the low frequency boost in dB.

## I.31. EQ > Static Filter > 2-pole static N



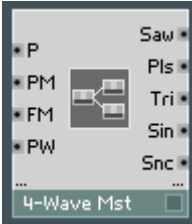
2-pole static notch filter. The *F* input specifies the cutoff frequency in Hz and the *Res* input specifies the resonance (0..1).

## I.32. EQ > Static Filter > 2-pole static Pk



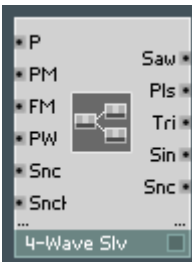
2-pole static peak filter. The *F* input specifies the cutoff frequency in Hz, the *Res* input specifies the resonance (0..1), and the *B* input specifies the center frequency boost in dB.

### I.33. Oscillator > 4-Wave Mst



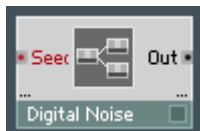
Generates 4 phase-locked audio waveforms. The oscillator pitch is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The pulse width is specified by the *PW* input (range  $-1..0..1$ , affects only the pulse waveform). This oscillator can oscillate at negative frequencies and additionally offers a synchronization output for *4-Wave Slv* oscillator.

### I.34. Oscillator > 4-Wave Slv



Generates 4 phase-locked audio waveforms. The oscillator pitch is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The pulse width is specified by the *PW* input (range  $-1..0..1$ , affects only the pulse waveform). This oscillator can oscillate at negative frequencies and can be synchronized to another *4-Wave Mst/Slv* oscillator. The *SncH* input controls the synchronization hardness (0 = no sync, 1 = hard sync,  $0..1$  = various degrees of soft sync). A synchronization output for another *4-Wave Slv* oscillator is also provided.

## I.35. Oscillator > Digital Noise



Digital white noise generator. Outputs random values in the range  $-1..1$ . An incoming event at the *Seed* input would (re-)initialize the internal random generator with a given seed value.

## I.36. Oscillator > FM Op



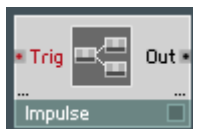
Classical FM operator. Outputs a sine wave whose pitch is specified by the *P* input (as a MIDI note number). The sine can be phase-modulated by the *PhM* input (in radians). An incoming event at the *Rst* input would restart the oscillator to the phase specified by the value of this event (range  $0..1$ ).

## I.37. Oscillator > Formant Osc



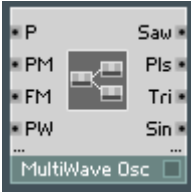
Generates a waveform with a fundamental frequency specified by the *P* input (as a MIDI note number) and the formant frequency specified by the *Fmt* input (in Hz).

## I.38. Oscillator > Impulse



Generates a one-sample impulse of amplitude 1 in response to an incoming event.

### I.39. Oscillator > MultiWave Osc



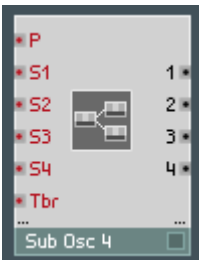
Generates 4 phase-locked audio waveforms. The oscillator pitch is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The pulse width is specified by the *PW* input (range -1..0..1, affects only the pulse waveform). This oscillator cannot oscillate at negative frequencies.

### I.40. Oscillator > Quad Osc



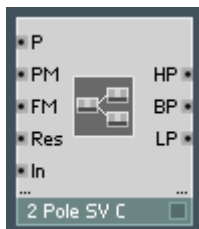
Generates a pair of phase-locked sine waveforms with a phase shift of 90 degrees. The *P* input specifies the pitch (as a MIDI note number).

### I.41. Oscillator > Sub Osc



Generates 4 phase-locked subharmonics. The fundamental frequency is specified by the *P* input (as a MIDI note number). The subharmonic numbers are specified by *S1*..*S4* inputs (range 1..120). The *Tbr* input controls the harmonic content of the output waveform (range 0..1).

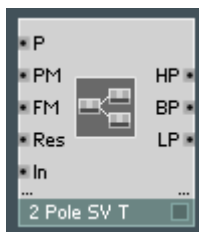
## I.42. VCF > 2 Pole SV C



2-pole state-variable filter (compensated version). Offers improved behavior at high cutoff settings. The filter cutoff frequency is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The *Res* input specifies the resonance (range 0..0.98). You also can use negative resonance values which will smear the slope further.

The *HP/BP/LP* outputs produce highpass, bandpass and lowpass signals respectively.

## I.43. VCF > 2 Pole SV T



2-pole state-variable filter with table compensation. Offers improved behavior at high cutoff settings, but slightly different from the 2 Pole SV C version. The filter cutoff frequency is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The *Res* input specifies the resonance (range 0..1).

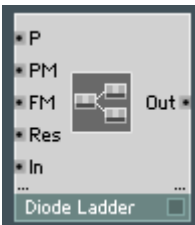
The *HP/BP/LP* outputs produce highpass, bandpass and lowpass signals respectively.

## I.44. VCF > 2 Pole SV x3 S



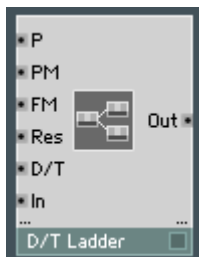
2-pole state-variable filter with optional oversampling (x3 version) and saturation. The filter cutoff frequency is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The *Res* input specifies the resonance (range 0..1), the *Sat* input specifies the saturation level (typical range 8..32). The *HP/BP/LP* outputs produce highpass, bandpass and lowpass signals respectively.

## I.45. VCF > Diode Ladder



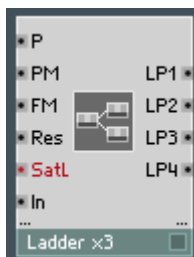
Diode-ladder filter linear emulation. The filter cutoff frequency is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The *Res* input specifies the resonance (range 0..0.98).

## I.46. VCF > D/T Ladder



Ladder filter linear emulation, can be morphed between diode and transistor ladder behavior. The filter cutoff frequency is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The *Res* input specifies the resonance (range 0..0.98), the *D/T* input morphs between diode and transistor (0=diode, 1=transistor).

## I.47. VCF > Ladder x3



An emulation of saturating transistor ladder filter (x3 times oversampled). The filter cutoff frequency is specified by the *P* input (as a MIDI note number), can be modulated by the *PM* input (in semitones, exponential) and by the *FM* input (in Hz, linear). The *Res* input specifies the resonance (range 0..1), the *SatL* input specifies the saturation level (typical range 1..32).

The outputs 1-4 are taken from the corresponding taps of the emulated ladder. Take the 4th tap for the 'classic' ladder filter sound.



# Index

## Symbols

1/96 Clock .....	50
1 / Square Root .....	60
12-Step .....	107
16-Step .....	107
4-Ramp .....	84, 121
4-Step .....	83
5-Ramp .....	122
6-Ramp .....	124
6-Step .....	106
8-Step .....	107

## A

$a * b + c$ .....	55
Accumulator .....	160
AD - Env .....	115
ADBDR - Env .....	117
ADBDSR-Env .....	118
Add .....	53
ADSR - Env .....	116, 117, 119, 120
Allpass 1-Pole .....	127
Amp .....	65
Appendix .....	194
AR - Env .....	115
Arccos .....	61
Arcsin .....	61
Arctan .....	62
A to E .....	177
A to E (Perm) .....	178
A to E (Trig) .....	177
A to Gate .....	178
Audio Modifier .....	149
Audio Outputs .....	271
Audio Smoother .....	181
Audio Table .....	158

Audio Voice Combiner .....	175
Auxiliary .....	172

## B

Bad values .....	285
Beat Loop .....	103
Bi-Pulse .....	81
Bi-Saw .....	69
Boolean control (BoolCtl) connections .....	289
Button .....	24

## C

Cells .....	<i>See</i> Core cells
Ch. Aftertouch .....	44, 49
Chopper .....	152
Clipper .....	150
Clock Oscillator .....	86
Compare .....	57
Compare/Equal .....	57
Constant .....	53
Controller .....	43, 48
Core cells .....	195
(re-)naming .....	216
audio .....	271
basic editing of .....	201
creating .....	208
event .....	246
event and audio .....	206
ports of .....	212
user library of .....	197
using .....	196, 199
Core events .....	<i>See</i> EventsCore

macros		Max.....	176
(re-)naming.....	227	Min .....	177
ports of.....	226	Expon. (A).....	58
Solid parameter of.....	227, 279	Expon. (F).....	58
user library of .....	266		
Core modules		<b>F</b>	
inserting.....	210	Fader.....	21
integer mode of.....	297, 299	Feedback .....	274
processing order of.....	254	indication of .....	224, 274
Core structure.....	201	resolution of .....	275, 299
Counter.....	160	Filter .....	126
Crossfade.....	64	FP Precision .....	227, 295, 321
Ctrl. Shaper.....	162, 163	Frequency Divider .....	157, 161
		From Voice .....	179
<b>D</b>			
D - Env .....	112	<b>G</b>	
DBDR - Env.....	113	Gate .....	24, 42
Debug mode.....	264	Geiger.....	87
Default signals of inputs		Grain Cloud .....	101
..... <i>See</i> Inputs		Grain Cloud Delay .....	146
Delay.....	142	Grain Delay .....	145
Denormal Cancel module .....	283		
Differentiator .....	140	<b>H</b>	
Diffuser Delay.....	144	H - Env .....	110
Distributor .....	64	High Shelf EQ.....	138
Divide x/y.....	55	High Shelf EQ FM .....	138
DR - Env .....	112	Hold .....	169
DSR - Env .....	113	HP/LP 1-Pole.....	126
Dynamic ports .....	20	HP/LP 1-Pole FM .....	127
		HR - Env.....	111
<b>E</b>		Hybrid modules .....	19
Envelope.....	109		
ES Ctl module .....	302, 325	<b>I</b>	
Event Processing.....	160	IC Receive.....	40, 192
Event Smoother .....	181	IC Send .....	40, 191
Event Table .....	170	Impulse .....	81
Event V.C.			
All .....	176		

Impulse FM.....	82
Impulse Sync.....	82
INFs.....	285
Initialization .....	301
Initialization event ..	256, 263, 268, ..... 271, 285, 302
In Port.....	190
Inputs.....	<i>See</i> Ports
default signals of.....	228
Integrator.....	141
Invert .....	54
Iteration.....	165

## K

Knob .....	23
------------	----

## L

Ladder Filter.....	135
Ladder Filter FM .....	136
Lamp.....	27
Latch module .....	
	252,254, 262, 287, 292, 299, 324
Level Lamp .....	28
Level Meter .....	30
LFO.....	109
Log (A) .....	59
Log (F) .....	59
Logic AND .....	163
Logic EXOR.....	164
Logic NOT.....	164
Logic OR.....	164
loop length.....	88
Low Shelf EQ.....	139
Low Shelf EQ FM .....	140

## M

Macros .....	<i>See</i> Core macros
--------------	------------------------

Master Tune/Level .....	182
Math .....	53
Merge.....	167
Merge module ..	260, 262, 291, 325
Meter .....	29
MIDI Channel Info.....	184
MIDI In.....	41
MIDI Out.....	48
Mirror 1 Level.....	151
Mirror 2 Levels .....	151
Mixer.....	65
Mod. Clipper .....	151
Modulation macros.....	268, .....271, 299, 324
Modules.....	<i>See</i> Core modules
Modulo .....	56
Mouse Area .....	37
Multi-Ramp .....	84
Multi-Sine .....	76
Multi-Step.....	83
Multi-Tap Delay.....	143
Multi/HP 4-Pole.....	133
Multi/HP 4-Pole FM .....	134
Multi/LP 4-Pole .....	131
Multi/LP 4-Pole FM.....	132
Multi/Notch 2-Pole.....	129
Multi/Notch 2-Pole FM .....	130
Multi 2-Pole .....	128
Multi 2-Pole FM.....	128
Multi Display .....	35
Multi Picture .....	31
Multiplex 16.....	107
multiplier .....	54
Multiply .....	54
Multi Text.....	32

## N

NaNs.....	285
Noise.....	86

Note Pitch.....	41
Note Pitch/Gate .....	48
Note Range Info.....	184

## O

Object Bus Connections (OBC) .....	296, 304
Off Velocity .....	43
on/off switch .....	168
On Velocity.....	43
Order.....	165
Oscillator .....	67
oscillator mode .....	88
OSC Receive .....	193
OSC Send .....	193
Out Port.....	190
Outputs .....	<i>See</i> Ports

## P

Panner.....	64
Parabol.....	72
Parametric equalizer.....	137, 138
Par FM .....	72
Par PWM .....	74
Par Sync .....	73
Peak Detector.....	156
Peak EQ.....	137
Peak EQ FM.....	137
Picture.....	30
Pitchbend .....	41, 48
Poly Aftertouch.....	44
Poly Display.....	35
Ports	
audio .....	271
audio/event.....	203
creating.....	212
event .....	246

event sending order of .....	247
relative order of.....	201
Power x y .....	59
Precision	
floating point .....	<i>See</i> FP Precision
Pro-52 Filter .....	135
Program Change .....	45, 50
Pulse.....	77
Pulse 1-ramp.....	79
Pulse 2-ramp.....	80
Pulse FM .....	78
Pulse Sync .....	78

## Q

QNaNs.....	285
Quantize .....	58
QuickBusses.....	217
QuickConsts .....	228, 248
QWERTY.....	194

## R

R/W Order module .....	309
Ramp .....	121
Random .....	87
Randomizer .....	161
Read module .....	253, 259, 306
initialization of .....	257
Read [] macro .....	310
Receive.....	190
Reciprocal.....	55
Rect./Sign .....	56
Rectifier.....	56
relay .....	168
Relay 1,2 .....	63
Router 1,2 .....	168
Router 1->M.....	168
Router M->1 .....	167
Router module.....	289, 325

## S

Sample & Hold .....	157
Sample Lookup.....	105
Sampler .....	89
Sampler FM .....	90
Sampler Loop .....	91
Sampling-rate clock.....	
..... 252, 275, 279, 287, 293, 311	
Saturator.....	149
Saturator 2.....	149
Saw FM .....	67
Saw Pulse .....	69
Saw Sync .....	68
Sawtooth .....	67
Scanner .....	63
Scope.....	33
Sel. Note Gate .....	42
Sel. Poly AT.....	44, 49
Selector .....	63
Send .....	190
Separator .....	166
Sequencer.....	106
Set Random .....	187
Shaper 1 BP .....	153
Shaper 2 BP .....	153
Shaper 3 BP .....	154
Shaper Cubic.....	155
Shaper Parabolic.....	155
Signal Path .....	63
Signals	
audio .....	220, 271
clock .....	<i>See</i> Clock signals
control .....	220
event .....	234
float.....	294, 326
integer .....	296, 326
Sign comparison .....	300
Sine .....	60, 74
Sine/Cos .....	61
Sine FM.....	75

Sine Sync.....	75
Single Delay .....	142
Single Trig. Gate .....	42
Slew Limiter .....	156
Slow Random .....	110
Snapshot .....	185
Snap Value.....	188
Song Pos .....	46, 51
Square Root .....	60
Stacked Macro.....	39
Start/Stop .....	45, 50
Step Filter .....	167
Stereo Amp .....	66
Stereo Pan .....	65
Subtract.....	54
Switch .....	26
Sync Clock .....	46
System Info.....	183

## T

Tapedeck 1-Ch .....	172
Tapedeck 2-Ch .....	175
Tempo Info.....	182
Terminal.....	190
Text.....	31
Timer .....	169
Toggle .....	24
To Voice.....	179
Transposing.....	194
Tri/Par Symm .....	71
Triangle.....	70
Tri FM .....	70
Trigger .....	24
Tri Sync .....	71
Tuning Info.....	183

## **U**

Unison Spread.....	187
Unit Delay.....	148

## **V**

Value.....	166
Voice Info.....	182
Voice Shift .....	180

## **W**

Write module...253, 259, 301, 306	
initialization of .....	257
Write [] macro .....	307

## **X**

XY.....	32
---------	----

## **Z**

Z <sup>-1</sup> module.....	254, 274,
.....	276, 288, 299