Core Reference

Document authored by: Native Instruments
Product Version: 5.5 (06/2010)
Document version: 1.1 (06/2010)

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

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

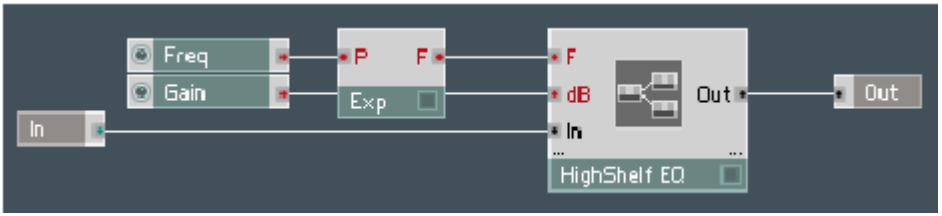
Table of Contents

1 First Steps in Reaktor Core

1.1 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:



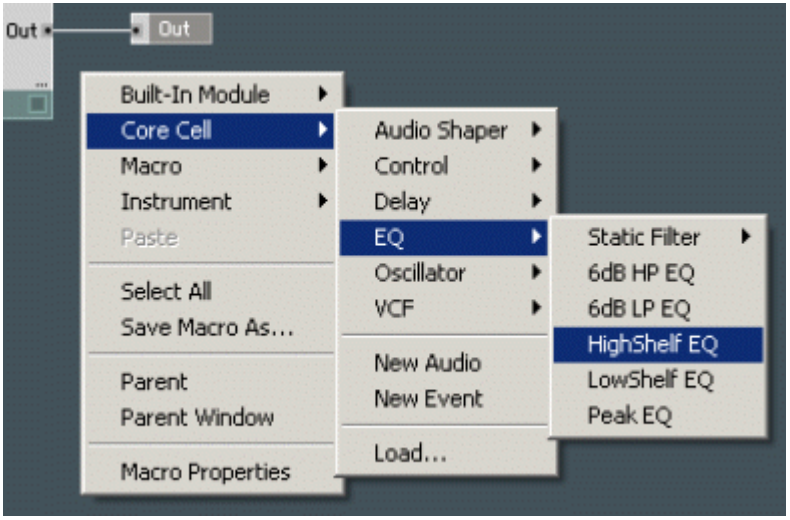
Inside of core cells are Reaktor Core structures. Those provide an efficient way to implement custom low-level DSP functionality as well as to build larger-scale signal-processing structures using such functionality. We will take a detailed look at these structures later.

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.

1.2 Using Core Cells

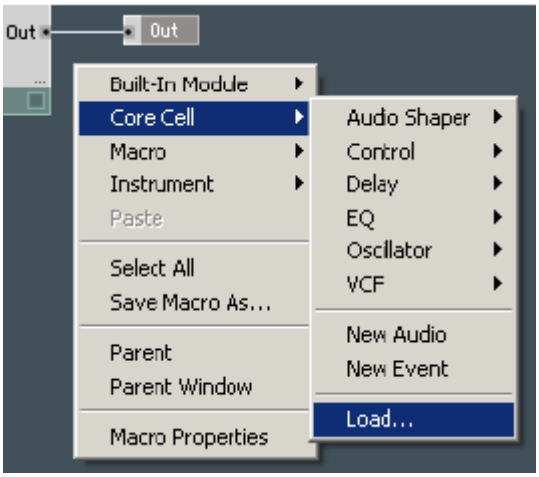
The core cell library can be accessed from primary-level structures by right-clicking on the background and using the *Core Cell* submenu:



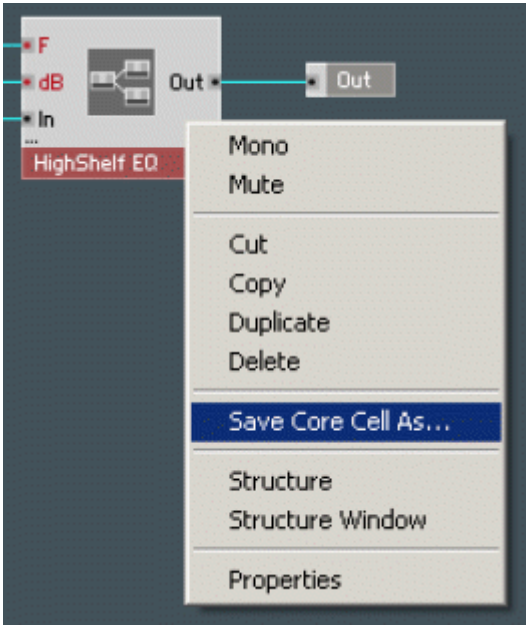
As you can see, there are all different kinds of core cells; they can be used in the same way as primary-level built-in modules.

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.

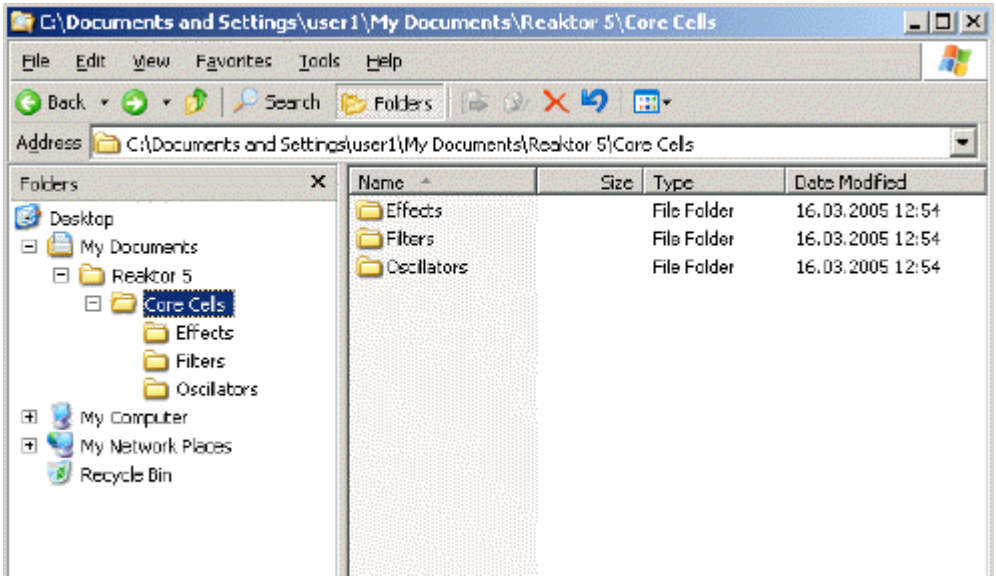
You can also insert core cells that are not in the library. To do that, use the Load... command from the Core Cell menu:



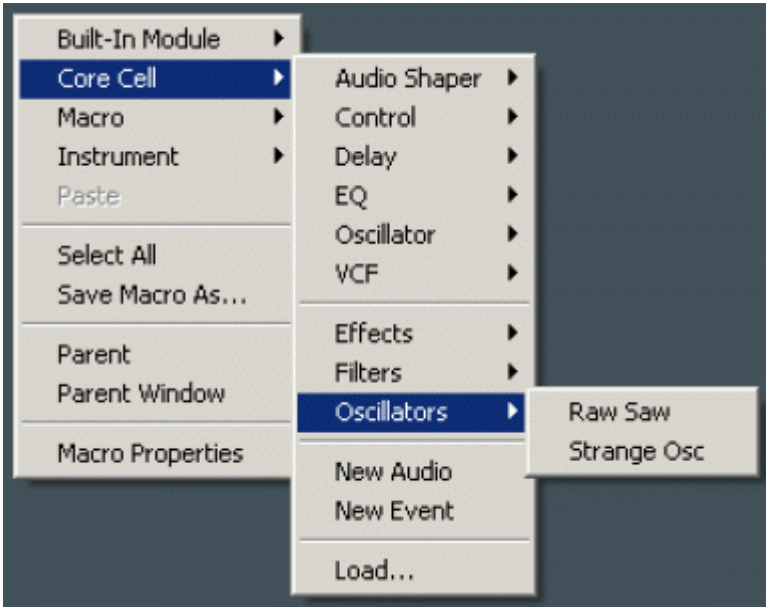
You may also want to save core cells you've created or modified, so that you can load them into other structures. To save a core cell, right-click on it and select Save Core Cell As:



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:



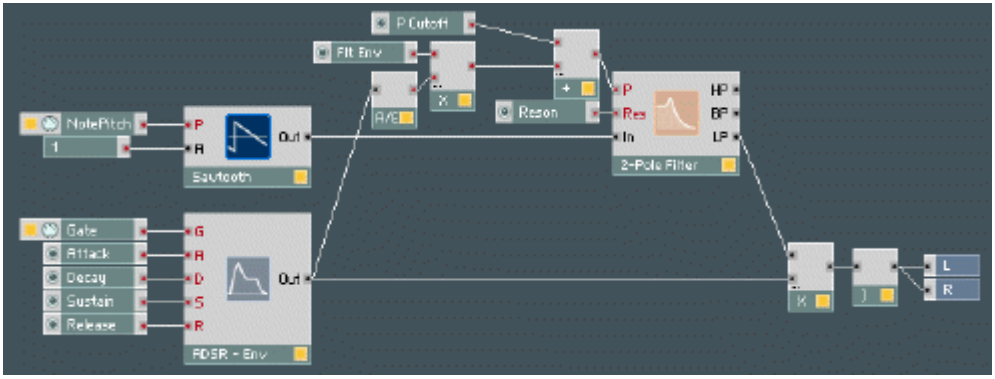
"My Documents\Reaktor 5" is the user library folder in this example. On your computer there may be a different path, depending on the choice you've made during installation and any changes you've made in Reaktor's preferences. Inside the user library folder there's a folder named "Core Cells". (Create 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:



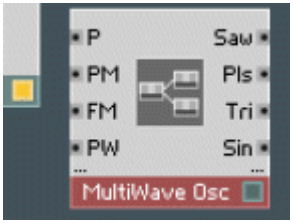
The menu contents are scanned once during Reaktor startup, so after putting new files into these folders, you should restart Reaktor.
Empty folders are not displayed in the menu; a folder must contain some files to be displayed.
Under no circumstances should you put your own files into the system library. 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.

1.3 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:

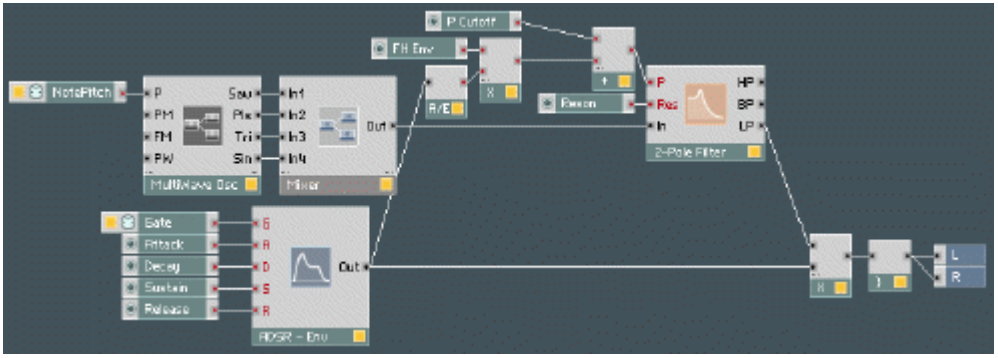


As you can see this is a very simple subtractive synthesizer consisting of one oscillator, one filter and one envelope. We are going to replace the oscillator with a different, more powerful one. Right-click on the background and select Core Cell > Oscillator > MultiWave Osc:



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 > 02-Mixer Amp > Mixer– Simple–Mono:

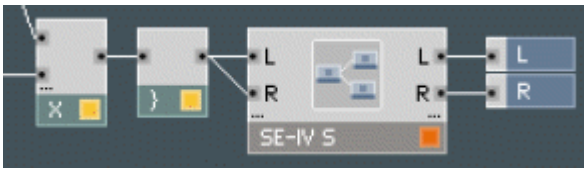
Connect the mixer and the oscillator together and use their combination to replace the sawtooth oscillator:



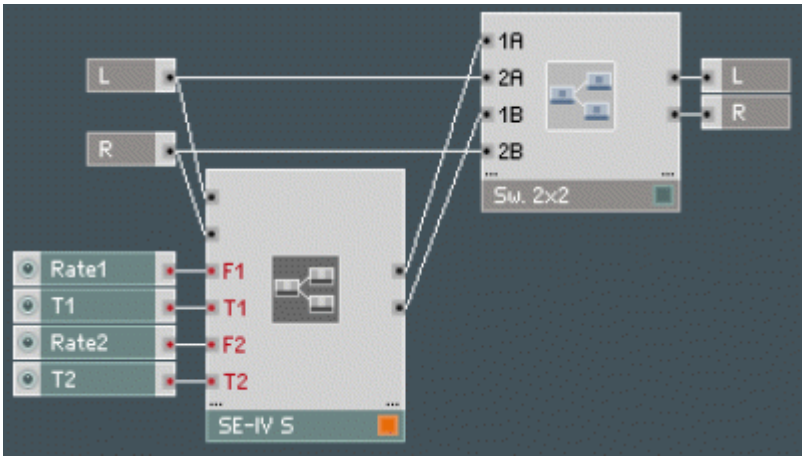
Switch to the panel view. Now you can use the four faders of the mixer to vary the waveform mix.

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:

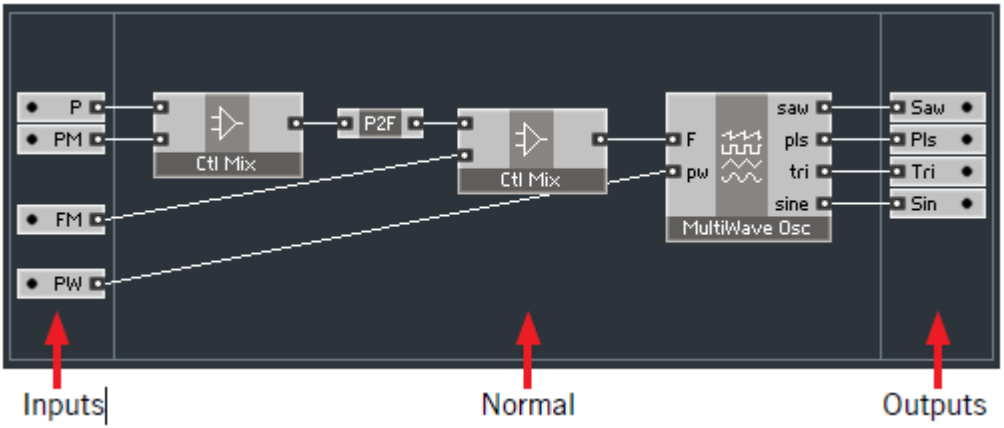


If you look inside the chorus you can see the chorus core cell and the panel controls:

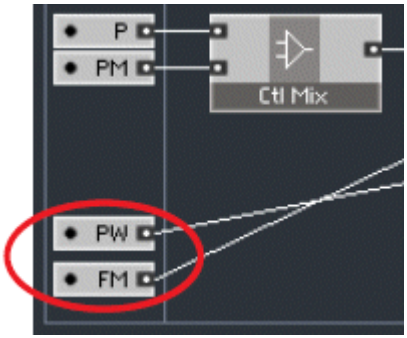


1.4 Basic Editing of Core Cells

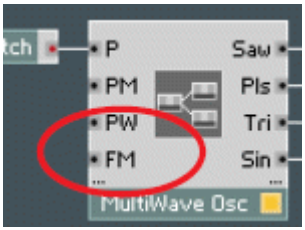
Now we are about to learn a few things about editing core cells. We are going to start with something simple: modifying an existing core cell to your particular needs.
First, double-click the MultiWave Osc to go inside:



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:

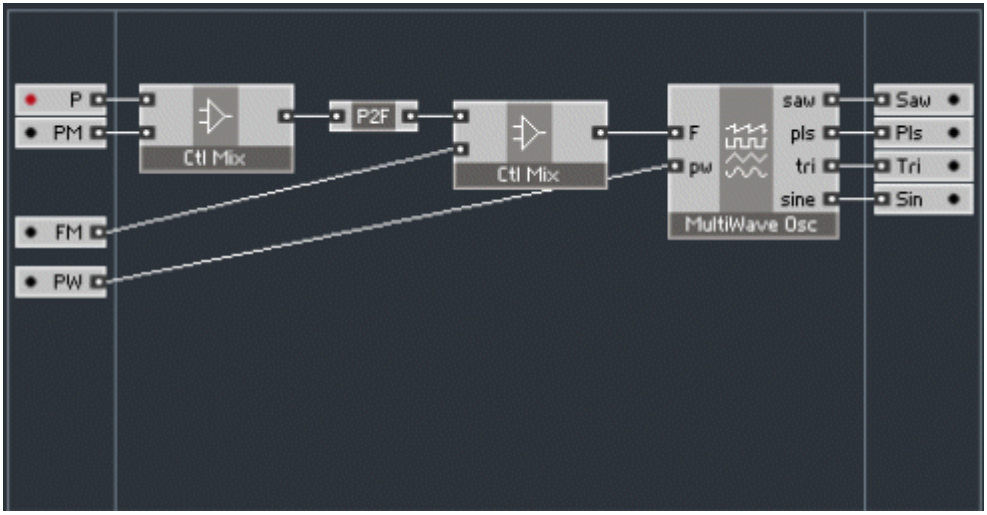


You can double-click the background now to ascend to the outside, primary-level structure and see the changed port order:

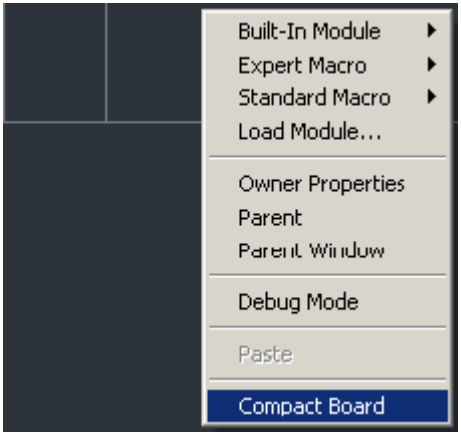


Now go back to the core level and restore the original port order:

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:

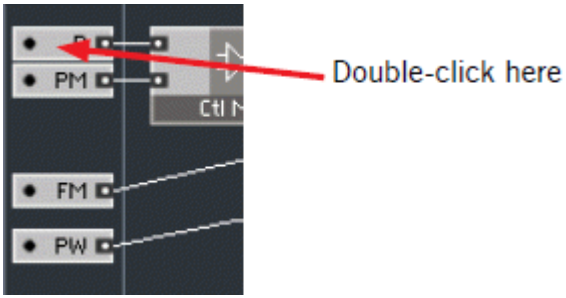


Now that we have learned to move the things around and rearrange the port order of a core cell, let's try a few more options.

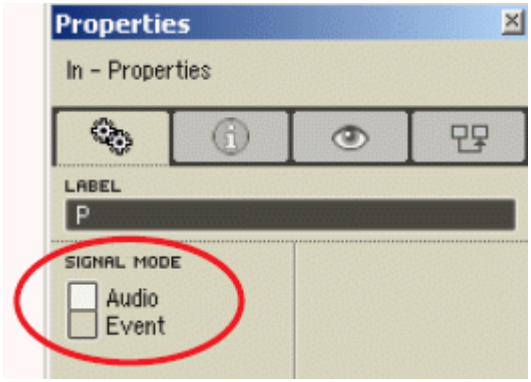
For a core cell that has audio outputs it's possible to switch the type of its inputs between audio and event (a more detailed explanation can be found later in this manual). In the above example, we used a MultiWave Osc module, all of whose inputs and outputs are au-

dio. 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:

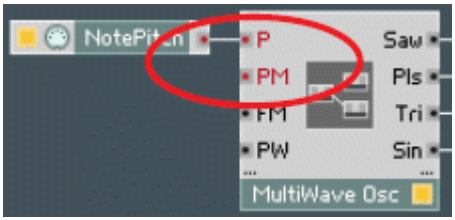


Switch the properties window to the function page, if necessary, by clicking on the cog wheel tab. You should now see the Signal Mode property:



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):

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.

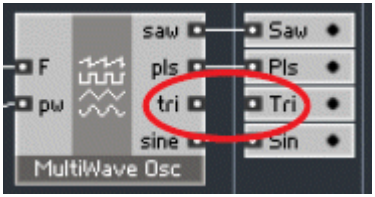


Sometimes it doesn't make sense to switch a port from one type to another. For example, it doesn't make sense to switch an input that receives a real audio signal (meaning real audio, not just an audio-rate control signal like an envelope) to an event rate. In some cases such switching could even ruin 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:

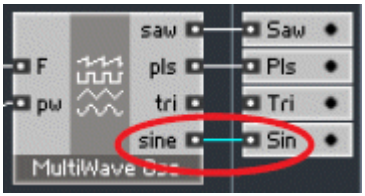
> **!** In a well designed core cell, an audio-rate control input can typically be switched to event mode without any problem. An event input can be switched to audio only if it doesn't have a trigger (or other event-sensitive) function.

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 con-

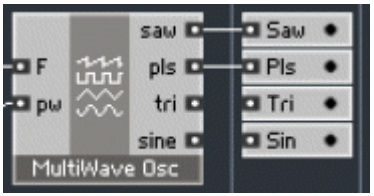
nection, drag the mouse to the any empty part of the background and release it. For example, click on the input port of the Tri output and drag the mouse into empty space on the background.



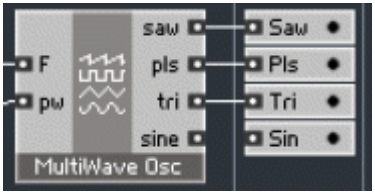
There's another way to delete a connection. Click on the wire between the sine output of the MultiWave Osc and Sin output of the core cell, so that it gets selected (you can tell that it's selected by its blue color):



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.

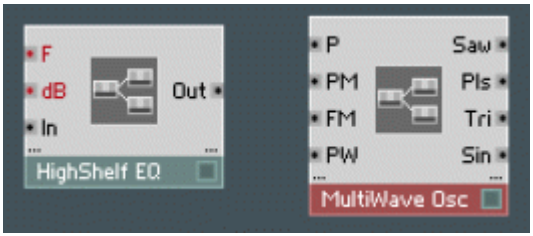
2 Getting Into Reaktor Core

2.1 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	Înputs	Outputs	Clock Src
Event	Event	Event	Disabled
Audio	Event/Audio	Audio	Enabled

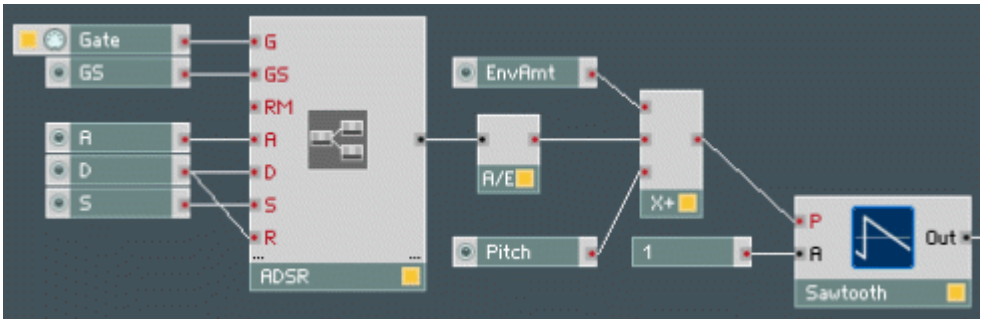
Therefore audio cells can implement oscillators, filters, envelopes, effects and other stuff, while event cells are suitable only for event processing tasks.
The HighShelf EQ and MultiWave Osc modules that you are already familiar with are examples of audio core cells (you can tell that by the fact that they have audio outputs):



And here is an example of an event core cell:

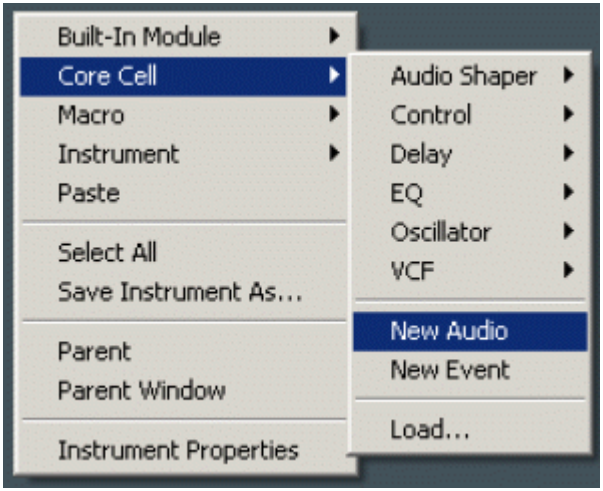
This module is a parabolic shaper for control signals, which can be used to implement velocity curves or LFO signal shaping, for example.

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:

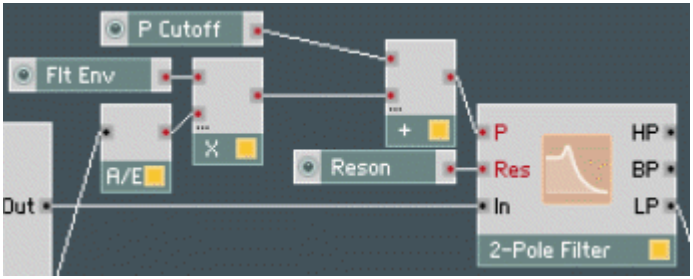


2.2 Creating Your First Core Cell

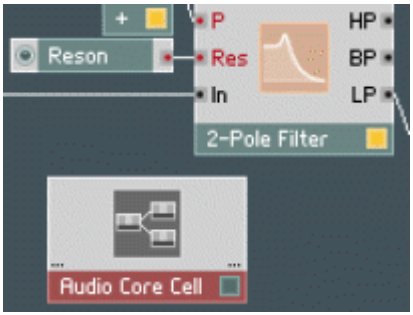
You create new core cells by right-clicking on the background in a primary-level structure and selecting Core Cell > New Audio or, for event cells, Core Cell > New Event:



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".):

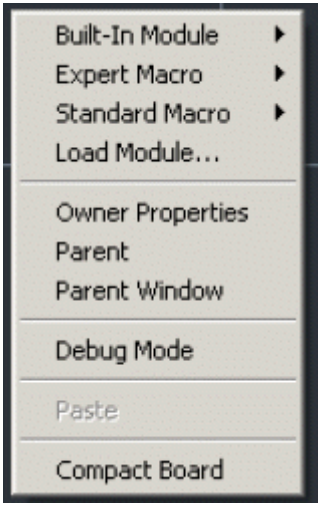


Because we need to apply the modulation at an event input, we also need to convert the envelope's output to an event signal, which we do with an A/E converter. As a result, our control rate is pretty low. Of course we could have used a converter running at a significantly higher rate (and eating up significantly 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:

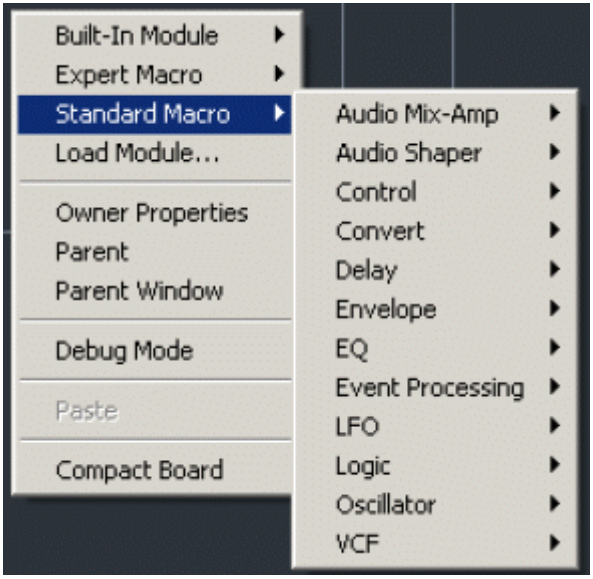


Double-click it to see its structure, which is obviously empty. As you surely remember, the three areas are meant for input, output, and normal modules:

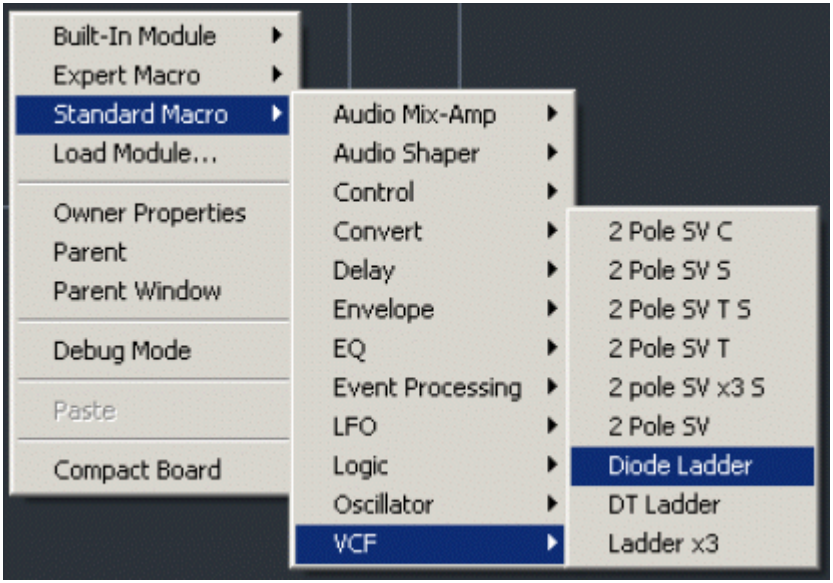
Attention: we are going to insert our first module into a core structure right now! Right-click in the normal area to bring up the module creation menu:



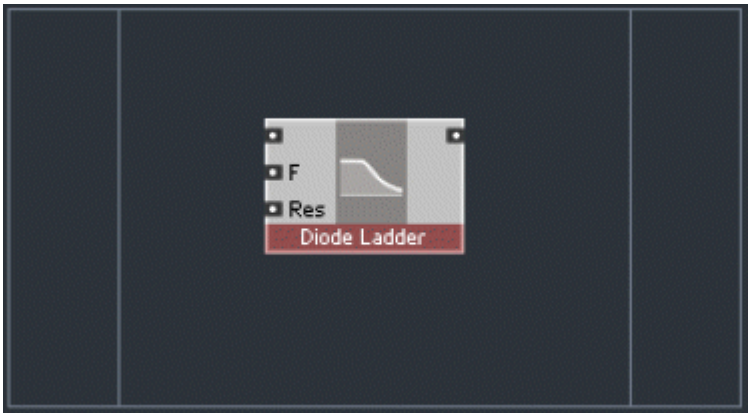
The first submenu is called Built-In Module and provides access to the built-in modules of Reaktor Core, which are generally meant to do really low-level stuff and will be discussed later.
The second submenu is called Expert Macro and contains macros meant to be used alongside built-in modules for low-level stuff.
The third submenu, called Standard Macro, is the one we want to use:



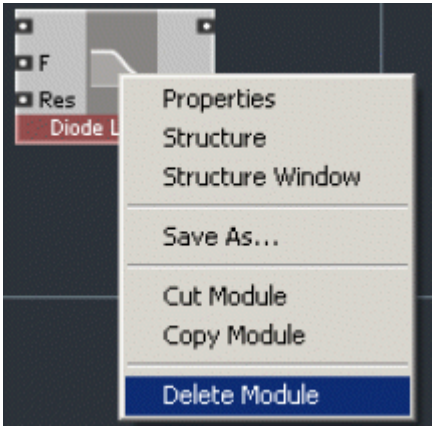
The VCF section could be promising, let's look inside:



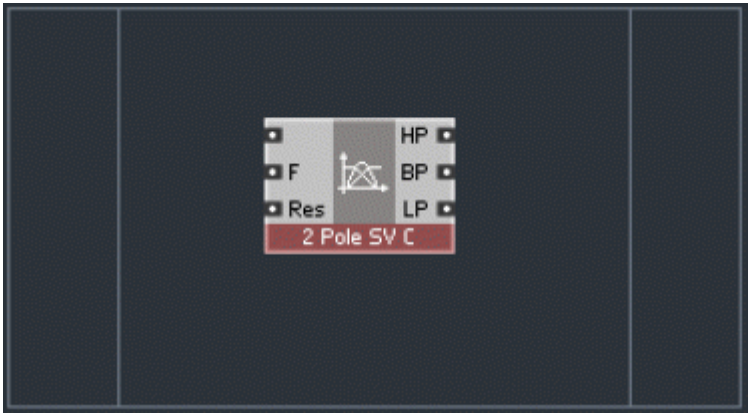
Let's try Diode Ladder:



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:

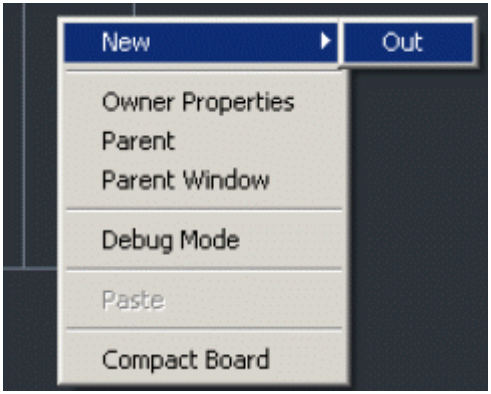


The other option is to select the module by clicking on it and pressing the Delete key.
After deleting the Diode Ladder, insert a 2 Pole SV C filter from the same VCF section of the Standard Macro submenu:



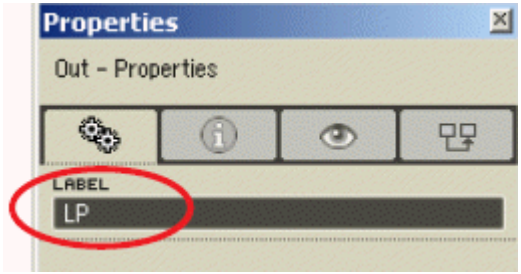
This is a 2-pole, state-variable filter and is similar to the one we are replacing (there are some differences, but they are quite subtle). What's important is that we can modulate this filter at audio rates.

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:

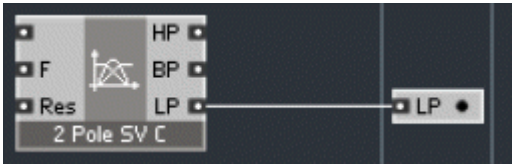


There's only one kind of module you can create there, so select it. This is what the structure is going to look like:

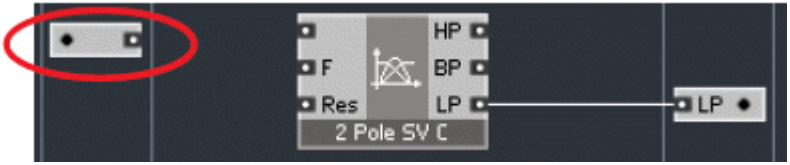
Double-click the output module to open the Properties window (if it's not already open). Type "LP" in the label field:



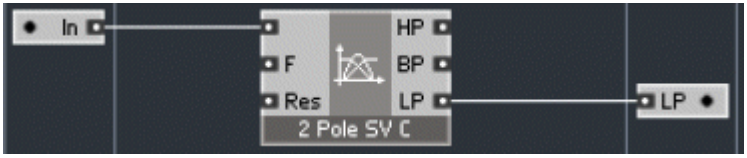
Now connect the LP output of the filter to the output module:



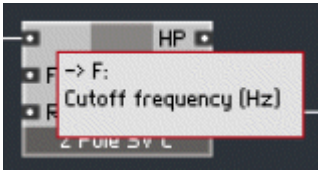
Now let's start with the inputs. The first input will be an audio-signal input. Right-click in the background of the inputs area and select New > In:



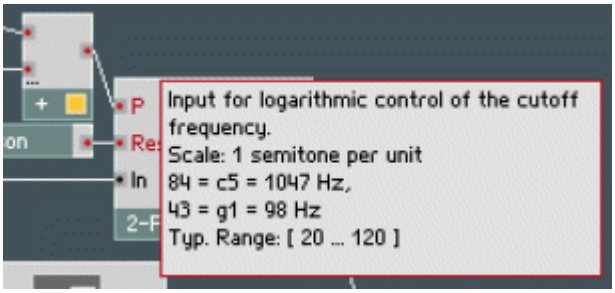
The input is automatically created with the right type – it's an audio input, as you can tell by the large black dot. Rename this input to "In" in the same way you renamed the output to "LP", then connect it to the first input of the filter module:



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 info button, the cursor with the “i,” is active), you’ll see the info text, which says “Cutoff frequency (Hz)”:

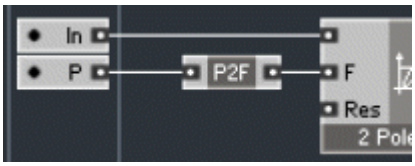


As we know, the cutoff of our primary-level filter module is controlled by an input labeled “P”, and as you can see from the info text, the signal uses a semitone scale.



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:

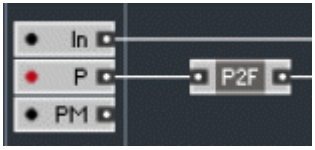
As the name implies (and the info text states), this module converts between P (pitch) and F (frequency) scales – exactly what we need. So let's create a second input labeled "P" and connect it using the P2F module:



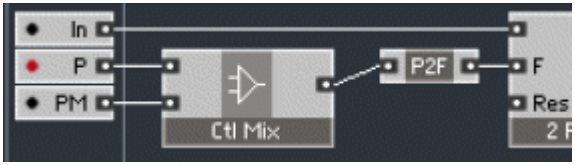
That should do it, but wait! In our instrument we have a "P Cutoff" knob defining the base cutoff of the filter, and to that is added the modulation signal from the envelope, which we have to convert to an event signal on the primary level in order to feed it into the P input of the filter. Now that 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.

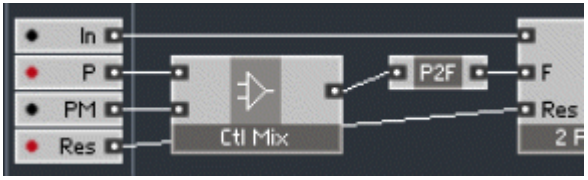
So first change the P input to the event mode (as described previously) and add another PM input, which should be in audio mode:



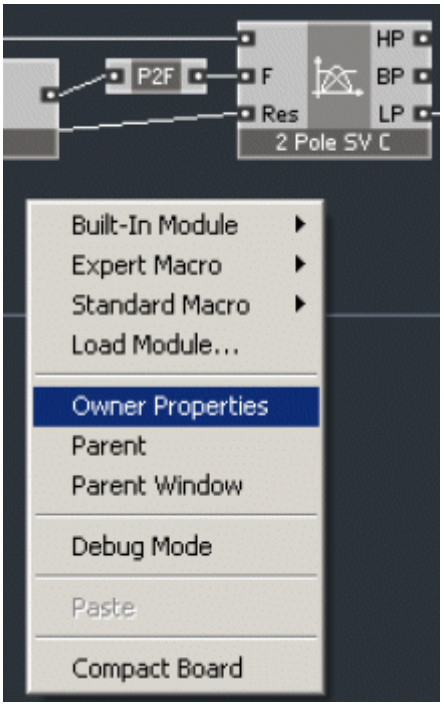
As a user of the Reaktor primary level, you probably expect us to add the two signals together now. In fact, we could do that, but in Reaktor Core the Add is considered a low-level module, and using it generally requires some knowledge of fundamental Reaktor Core low-level working principles. They are 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:



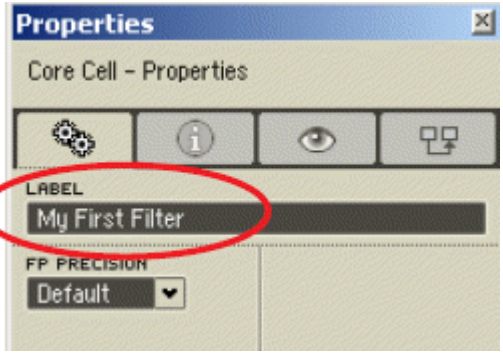
The last input that we need is a resonance input, it doesn't need to be at audio rate, so let's use an event input:



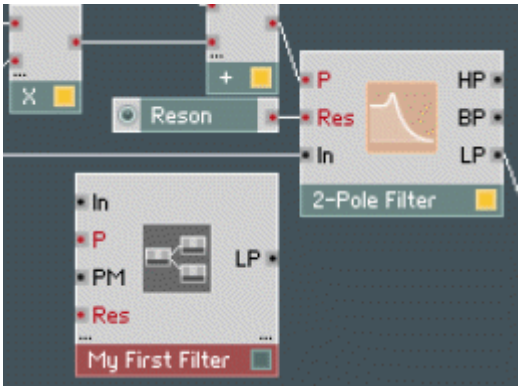
One other thing we need to do is to give our core cell a name. If the Properties window is already open, click on the background to display the core cell's properties. If it's not open, right-click on the background and select the Owner Properties command:



Now you can type some text into the label field:

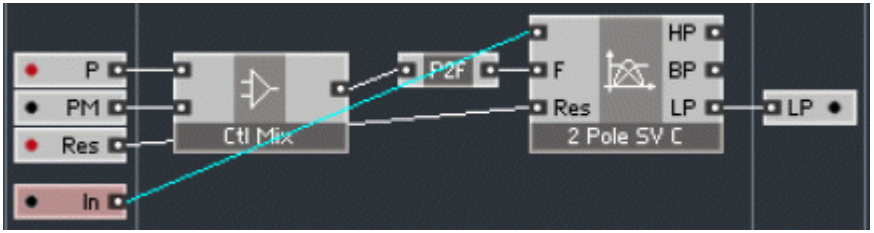


Double-click the background to see your result:



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.

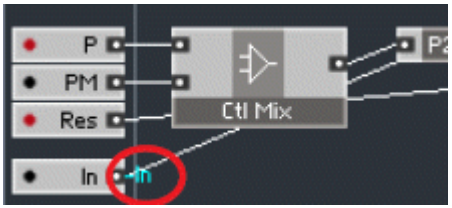
The first thing to do is go back inside and drag the audio-signal input all the way to the bottom:



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:

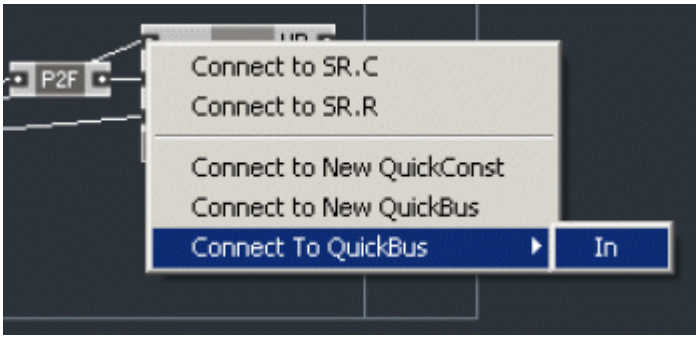
This is what you should see now:



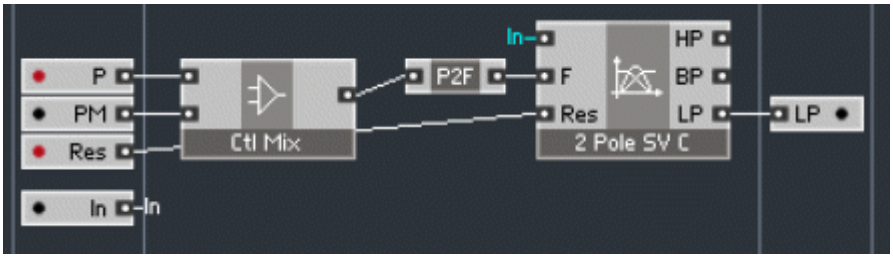
Also, the Properties window should open to display the properties of the QuickBus you've just created. The most useful QuickBus property is the ability to change its name (other properties are quite advanced, so don't touch them for now). You can open the Properties window later by double-clicking on 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.)

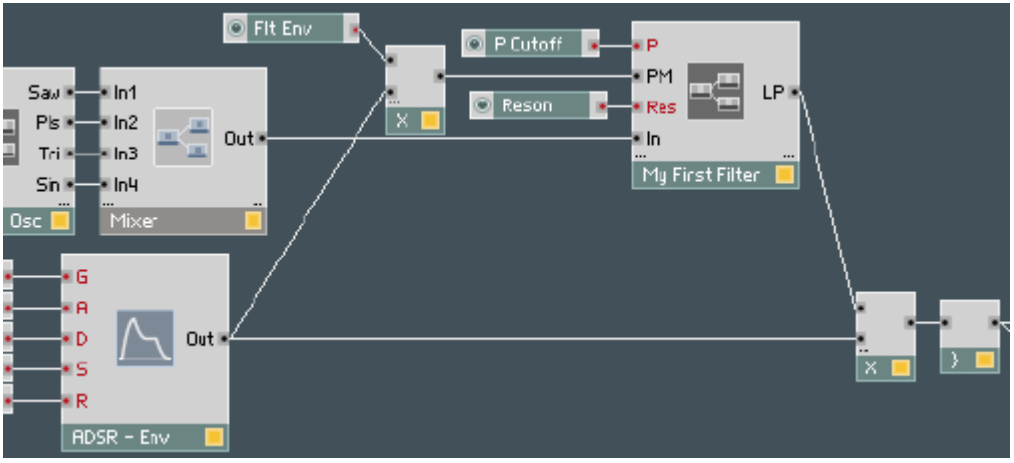
The next thing you should do is right-click on the top input of the 2 Pole SV C filter module and select Connect to QuickBus > In:



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:



Instead of a nasty looking diagonal wire, we get two nice references, stating that the input and output are connected by a QuickBus whose name is “In”.
Now we can go back out to the primary level and modify our structure to use the new filter we’ve just built. The Add and A/E modules can be thrown away. This is our final result:



Takes quite a bit more CPU, doesn’t it? Well, don’t forget that this filter is modulated at audio rate in pitch scale. If you don’t like it, you can still revert to the old structure or use the Multi 2-pole FM filter module from the primary level (slow envelopes, remember?), but we hope that you do like it. Even if you don’t, there are quite a few other filters with new features that you might like better. And, if you don’t like the new Reaktor Core filters, there are a whole bunch of other Reaktor Core modules you can try.

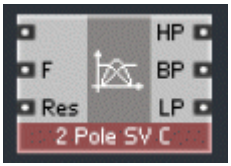
2.3 Audio and Control Signals

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:

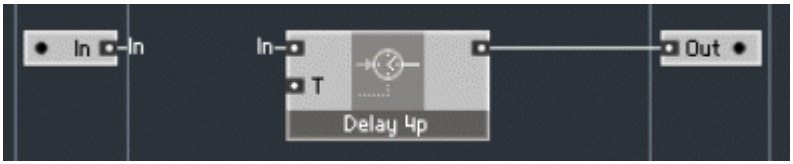
- A primary-level audio signal normally corresponds to either a Reaktor Core audio signal (for example, an output of an oscillator or an audio filter) or a Reaktor Core control signal (for example, an output of an envelope).
- A primary-level event signal is typically a control signal in terms of Reaktor 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.

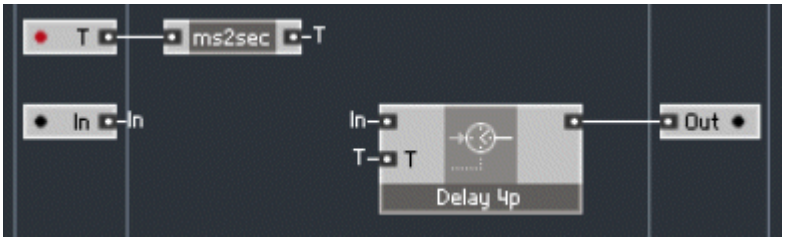
> **!** It's important to understand that when you select the type for a core-cell input port, you are choosing between the primary-level event and audio signals, not between Reaktor Core event and audio signals. The core-cell ports are the place where both worlds meet, and therefore, they use a bit 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:

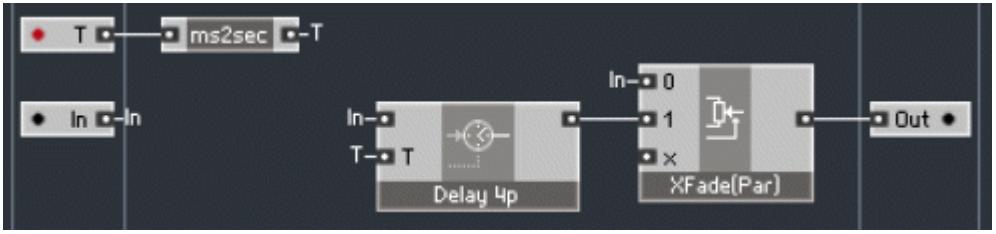
We obviously need an audio input and an audio output for our delay core cell. We will use a QuickBus connection for the input and a normal connection for the output:



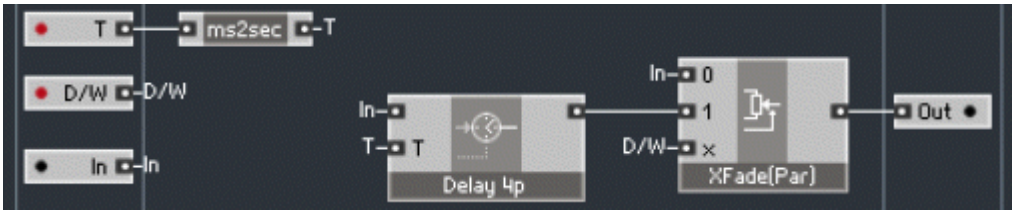
We also need an event input for controlling the delay time. One thing to be aware of here is that, on the primary level, the delay time is usually expressed in milliseconds, while Reaktor Core library delay macros expect it to be in seconds. No problem, there is a conversion module available for that. 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):

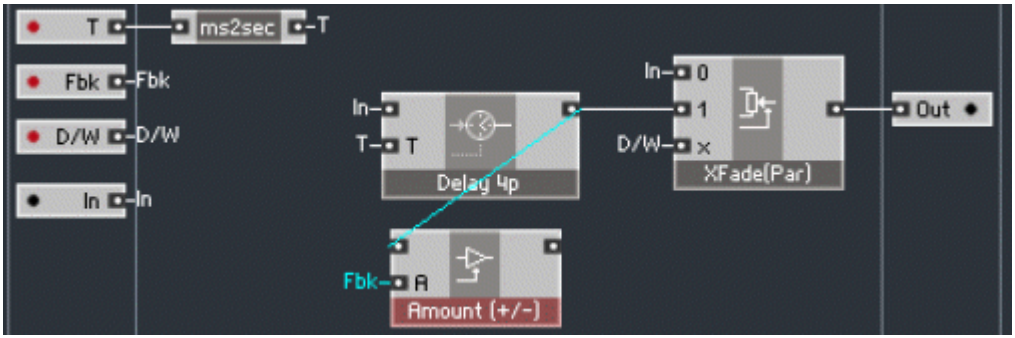


Here "(par)" stands for parabolic, which produces a more natural sounding crossfade than a linear crossfade. We will connect the control (x) input of the crossfade to a new event input to control the mix between the dry (unprocessed) and wet (delayed) signals. When the control signal is 0 we will hear only the original signal, and when it's 1, we will hear only the delayed signal:



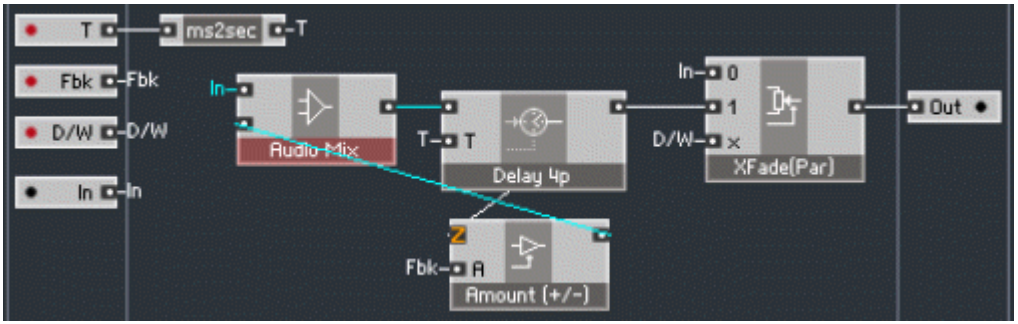
Now we can hear the original signal and the echo, but there's still only one echo. To have multiple echoes we need to feed a fraction of the delayed signal back to the delay input. First we need to attenuate the delayed signal. Following the same guidelines, use an audio amplifier to attenuate an audio 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:

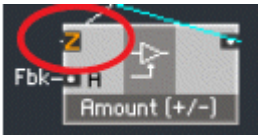


! The reasonable feedback amount range is something like [-0.9..0.9] here. When you try out this delay, be careful with the feedback amount, because you can easily reach excessive signal levels (there is no saturation in our circuitry yet). We could have embedded a safety feedback amount clipper into 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.

We need to mix the feedback signal with the input signal. An audio mixer (Standard Macro > Audio Mix-Amp > Audio Mix) is a natural choice



You may wonder what happened to the upper input of the Amount module above, which now shows a large orange "Z":

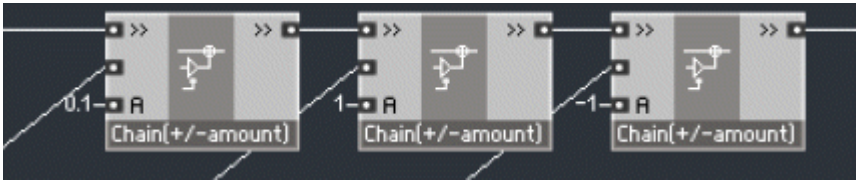


Actually, depending on the version of the software and other conditions, the Z sign could appear at some other input in the structure, but don't worry you too much about it. The Z sign indicates that a digital feedback has occurred in the structure, and it is meant for advanced structure design, where such information can be an important hint for the structure designer.

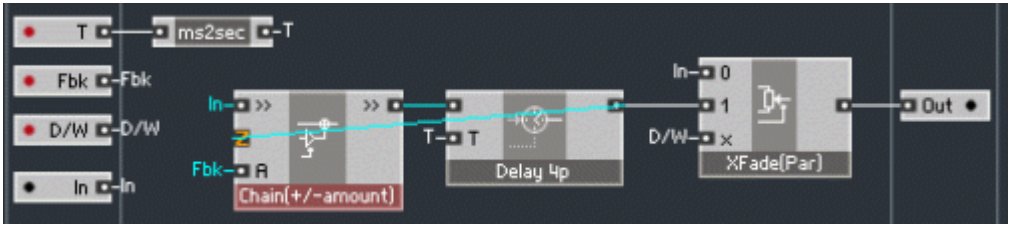
For simple structures like the one above, one normally needn't worry about the Z sign; its presence just shows that there will be a 1-sample delay (about 0.02ms at 44.1kHz, even less at higher sampling rates) at that point in the structure. We assume you won't notice if your delay time is 0.02ms off 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:



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.

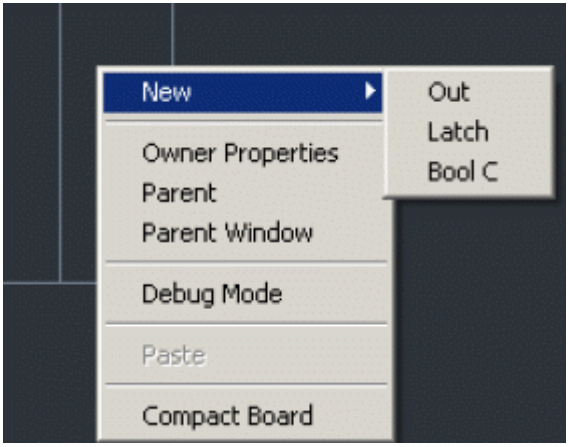
2.4 Building Your First Reaktor Core Macros

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.

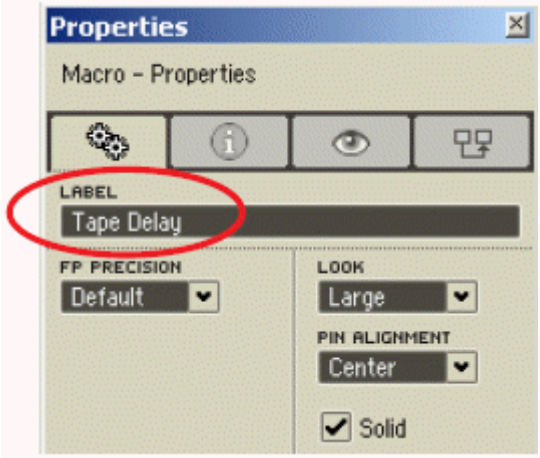
Let's start by deleting the delay macro from the structure and creating an empty macro instead. Right-click on the background as select Built-In Module > Macro:

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:

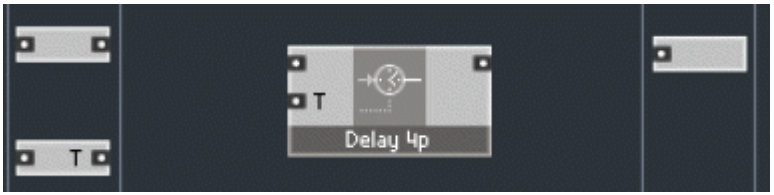


The remaining properties of the macro control various aspects of its appearance and its signal processing.

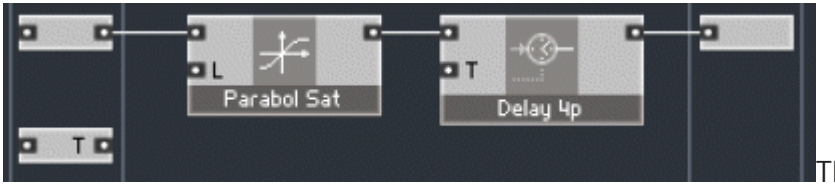
! While you are free to experiment with remaining properties as you see fit, we strongly advise against turning the Solid parameter off. We also advise changing the FP Precision sparingly. The meaning of these parameters will be described in the advanced topics of this manual.

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:



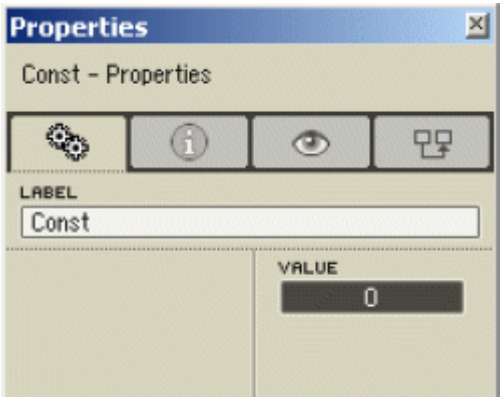
A simple emulation of the saturation effect can be done easily by connecting a saturator module before the delay. Saturator is a kind of signal shaper, so we will look for it among the audio shapers (because it is an audio saturator). Standard Macro > Audio Shaper > Parabol Sat:



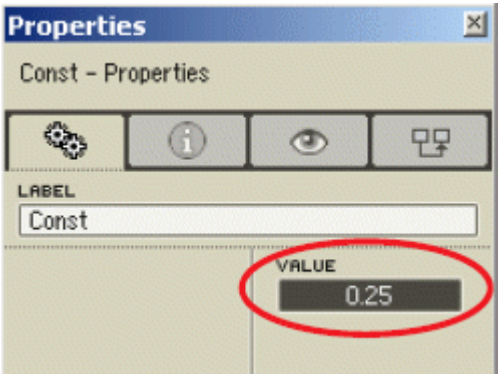
The input signal will now be saturated within the range of –1..+1. Actually, the range is controlled by the L input of the saturator module, if it is disconnected it defaults to 1. That might be surprising to you because you are probably used to disconnected inputs being treated as if they receive no 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 cog wheel tab):



In the value field type a new value of 0.25:

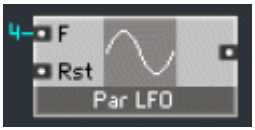


This is how the QuickConst should look now in the structure:

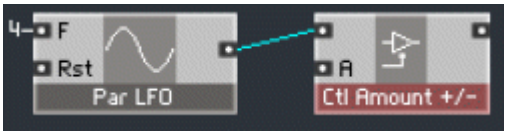
Let's explain what we have just done. The port on the left side of the input module specifies a so-called default signal. That means that if the input is not connected (on the outside of the macro), the default signal will be taken as the input source. In our case, if the T input of the Tape Delay macro 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 Quick-Const again here. A rate of 4 Hz seems reasonable so we can try that:

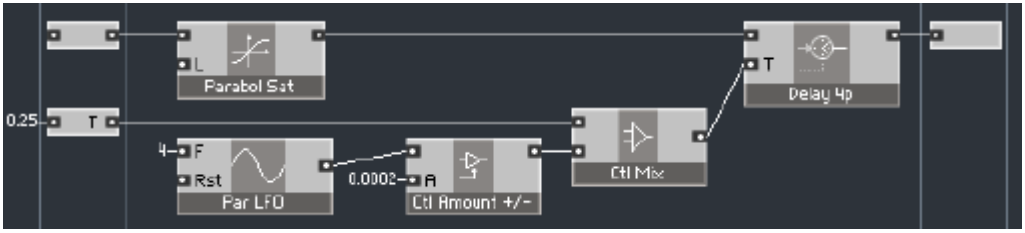


The Rst input is used for restarting the LFO; we won't need it for now.
Now we need to specify a modulation amount by scaling the output of the LFO. Currently the LFO output signal varies in the range –1 .. 1 and that is way too much. Because we are dealing with control signals here, we are going to use a control amount module, which is similar to the Amount amplifier we used for audio. Standard Macro > Control > Ctl Amount:

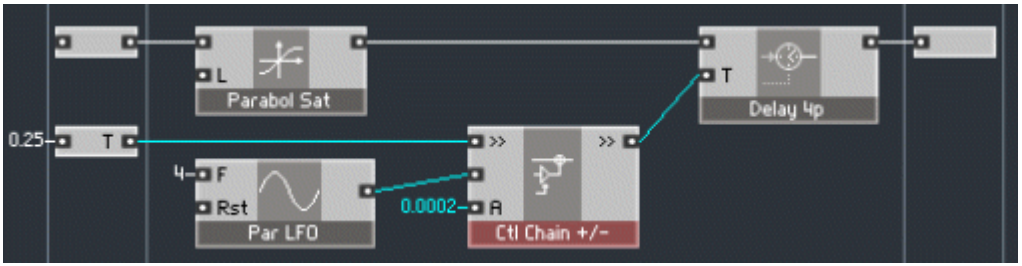


A modulation amplitude of 0.0002 should do fine, so we scale the signal to that amount:

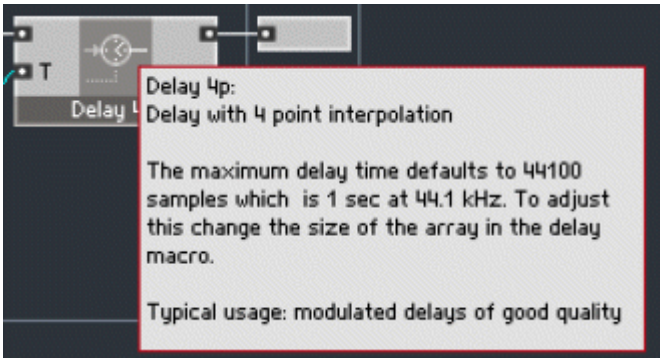
Ultimately, we can mix the two control signals (one from the T input and one from the Ctl Amount module) and feed them into the T input of the delay module. The already familiar Ctl Mix module can be used for that:



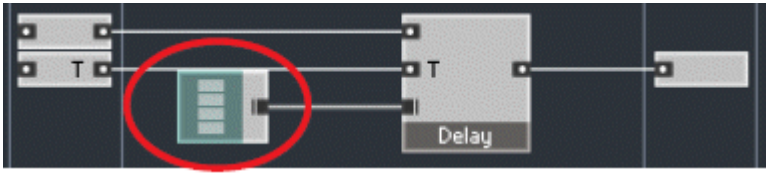
Actually, we have a Chain type of control mixer that is similar to the mixer we have for audio signals. We could use it to replace the Ctl Amount and the Ctl Mix modules in the same way we did earlier in the audio path. Standard Macro > Control > Ctl Chain:



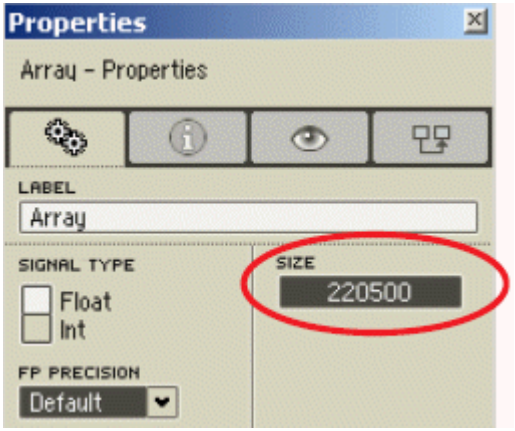
As one last touch for our macro, we are going to change the buffer size for our delay, which defines the maximum possible delay time. If you hold your mouse cursor over the Delay 4p macro (provided the cursor info button for info popus on rollover is active), you can read in the hint text that the default buffer size corresponds to 1 sec of delay at 44.1kHz:



Let's increase the amount to 5 seconds (44,100*5 = 220,500 samples). Because each sample requires 4 bytes, we need 220,500*4 = 882,000 bytes, which is almost 1MB). Double-click on the Delay 4p macro:

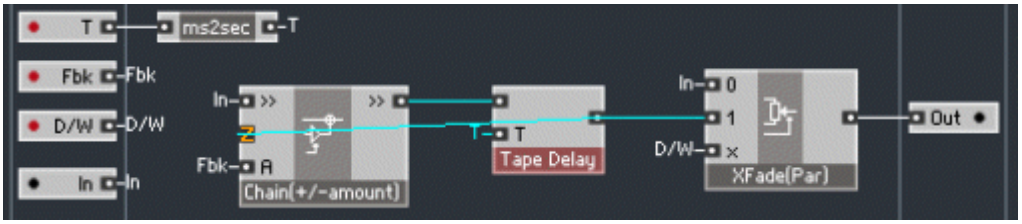


The module on the left is the delay buffer module. Double-click it (or right-click and select Show Properties) to edit its properties. Select the cog wheel tab and you should see the Size property. Change it to 220,500 samples:

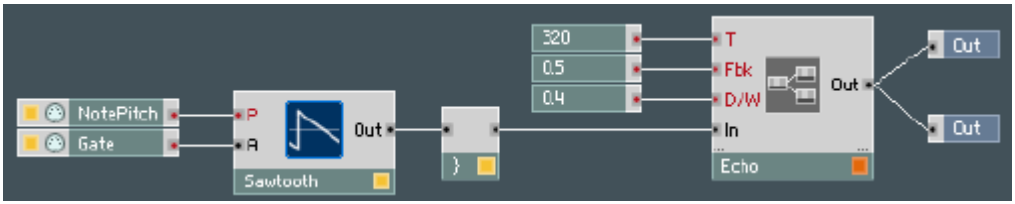


! As we have seen, a delay buffer for 5 seconds of audio takes almost 1MB of memory, so be careful when changing delay buffers. That's most important when the delays are used in polyphonic areas of the structure, because the size of the buffer will be multiplied by the number of voices.

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):

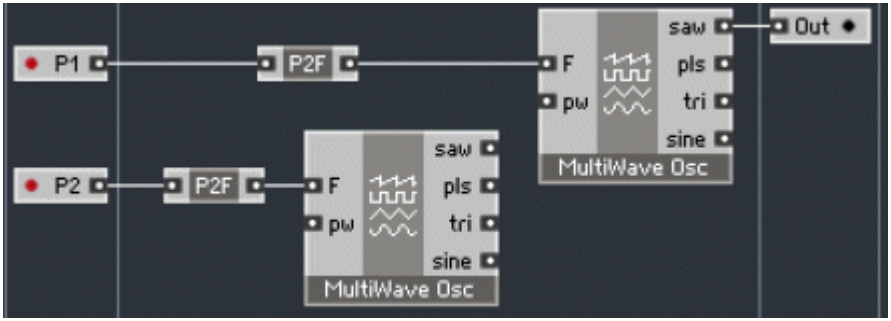


You might want to enhance it in various ways, for example, by providing knobs controlling the echo parameters, by using a real synthesizer as a signal source, and so on.

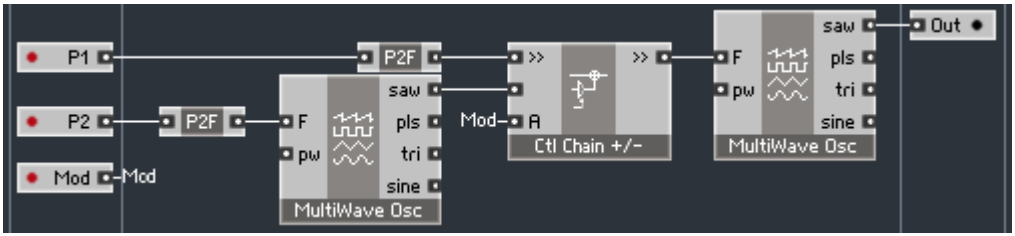
2.5 Using Audio as Control Signal

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:

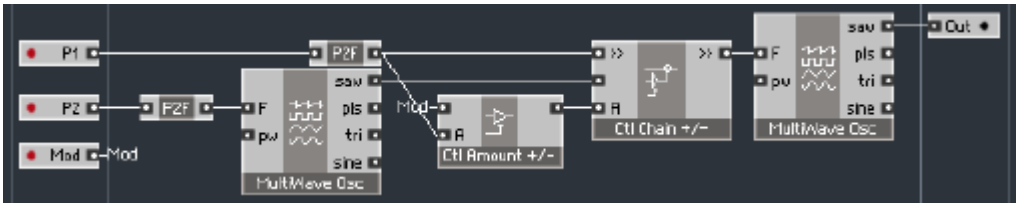
We need pitch control for both of the oscillators, and we are going to listen to the output of the second one, so let's create the necessary inputs and outputs:



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:



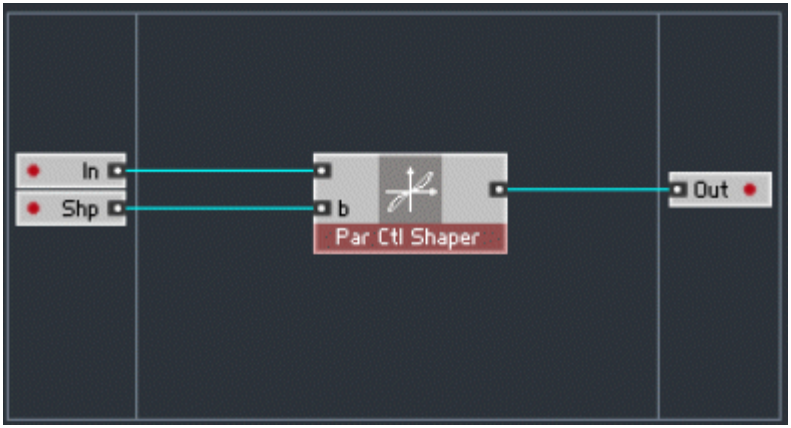
If you analyze the above structure from the point of view of control and audio signals you will notice that all of the signals in the structure except the outputs of the oscillators are control signals. The outputs of both oscillators are obviously audio signals. Notice, however, that we are misusing the output of the left oscillator as control signal at the point at which we feed it into the Ctl Chain mixer.

2.6 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:

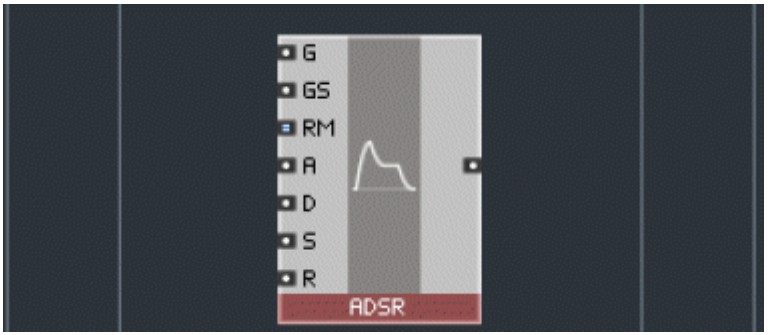


The control shaper receives an event rate control signal from the primary level (for example, a MIDI velocity signal, or a primary-level LFO signal), bends it according to the Shp parameter, and forwards the result to the output.

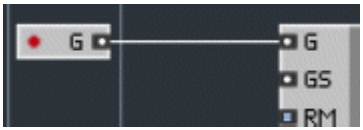
> ! An important restriction of event core cells, which we mentioned earlier, is that all clock sources are disabled inside them. That means that not only oscillators and filters, but also envelopes and LFOs do not work inside event core cells. Those modules are restricted to receiving events from the 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.

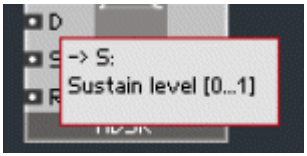
The first case is using an envelope in a core structure. As you can guess from the disabled-clock restriction on event core cells, this has to be an audio core cell. So, create a new audio core cell and choose Standard Macro > Envelope > ADSR:



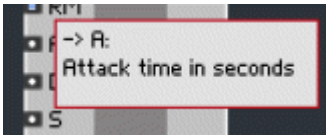
The top input of the envelope is a gate input, which works similarly to the gate inputs of primary-level envelopes—that is, it opens or closes the envelope in response to incoming events. For that we create an event input for our core cell:



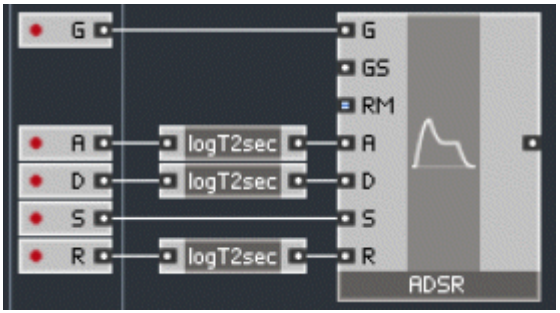
This input will translate the incoming primary-level gate events into the core events.
Now let's take a look at the A, D, S, R inputs. The S (sustain level) input works similarly to the primary level; it expects the incoming signal to be in the 0 to 1 range:



The A, D, R inputs are different, however. Unlike primary-level envelopes they expect time to be specified in seconds:

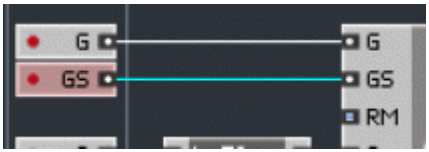


That can be solved by using a Standard Macro > Convert > logT2sec, which converts the primary-level envelope times to seconds:

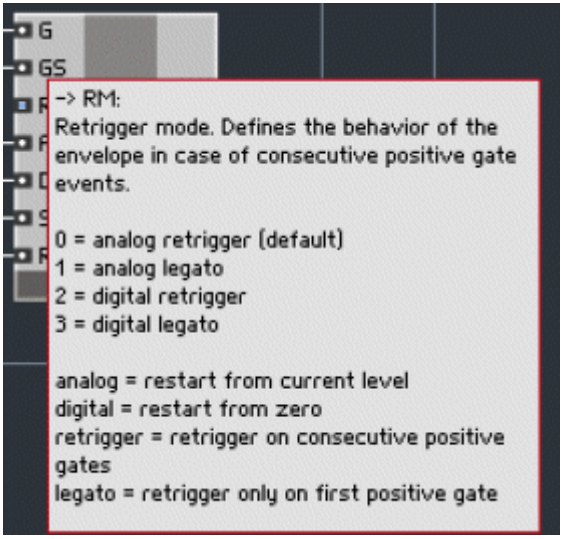


Although all inputs in the above structure are in event mode, the first input produces an event signal, whereas the others produce control signals.

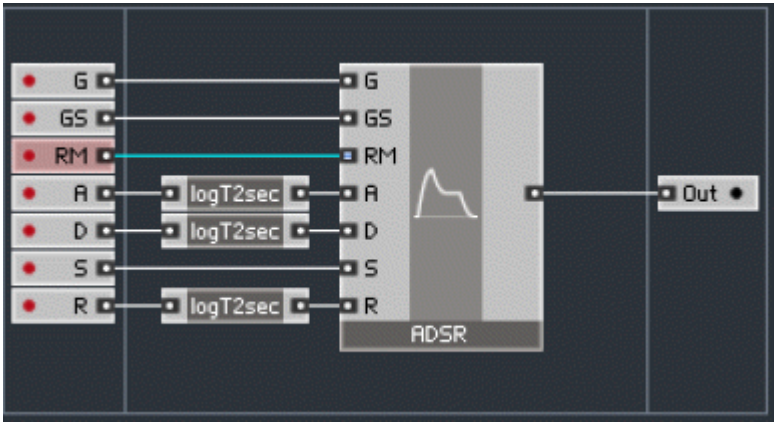
Our envelope still has two unconnected ports. The GS port sets the gate sensitivity amount. At 0 the envelope completely ignores the gate level and is always at full amplitude. At 1 the gate level has maximum effect, as on the Reaktor primary level. We can control this amount from the outside by adding another input:



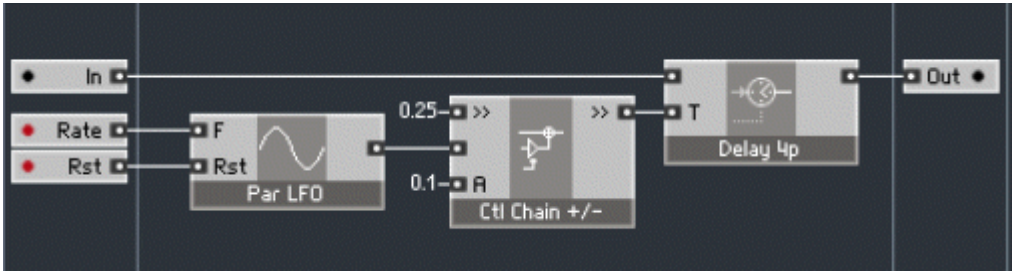
The RM port specifies the retrigger mode for the envelope:



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±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.

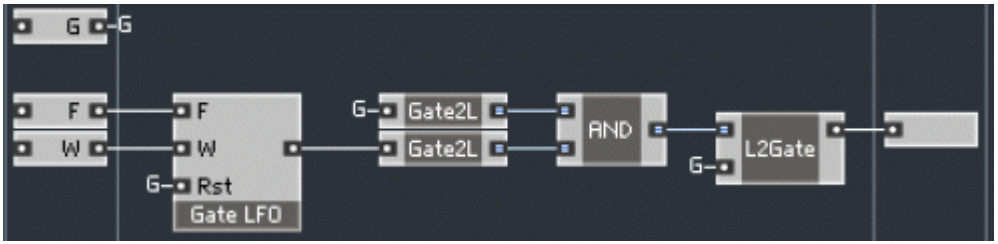
2.7 Logic Signals

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:

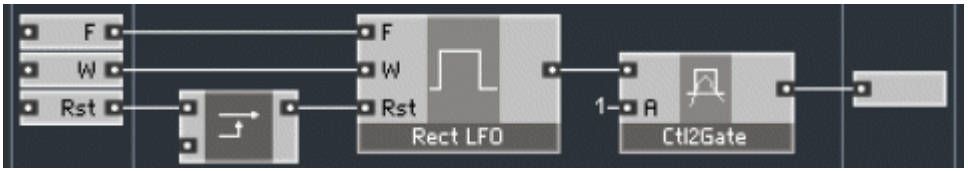
Notice that the ports of this module are integer type, just as was the RM input of the envelope. That is because, generally, logic signals carry only integer values; more precisely, they carry only values of 0 and 1.

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).

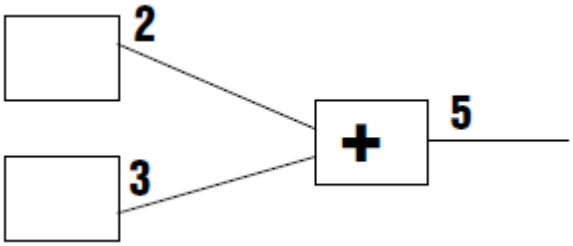
The module connected to the Rst input of the Rect LFO is called Value and can be found in Standard Macro > Event Processing. It ensures the LFO is restarted at zero phase by replacing the values of all incoming events by the value at its lower input, which is zero. The LFO output is converted into a 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.

3 Reaktor Core Fundamentals: The Core Signal Model

3.1 Values

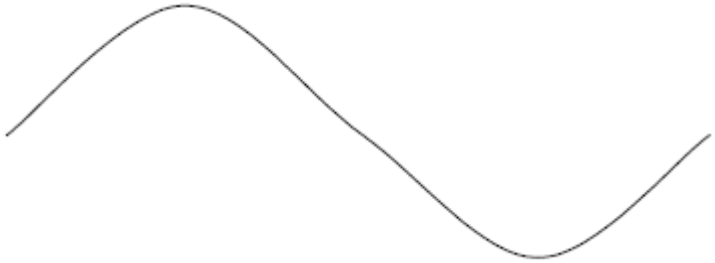
Most of the outputs of Reaktor Core modules produce values. (Producing a value means that at any moment in time there is a value associated with the output.) The values are available to all modules whose inputs are connected to those outputs.
In the following example an adder module gets values 2 and 3 from the two modules whose outputs are connected to its inputs, and it produces a value of 5 at its output.



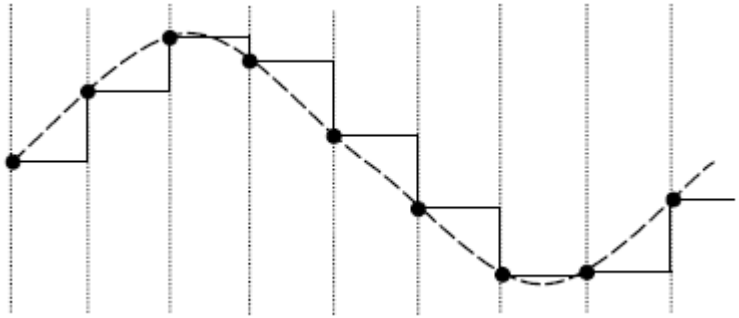
If you want to draw an analogy to the hardware world you can think of values as signal levels (voltages), especially with relatively large-scale modules such as oscillators, filters, envelopes, and so on. However, values are not limited to those kinds of processing—they are just values and can be used to implement any processing algorithm, not just voltage-modeling algorithms.

3.2 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:

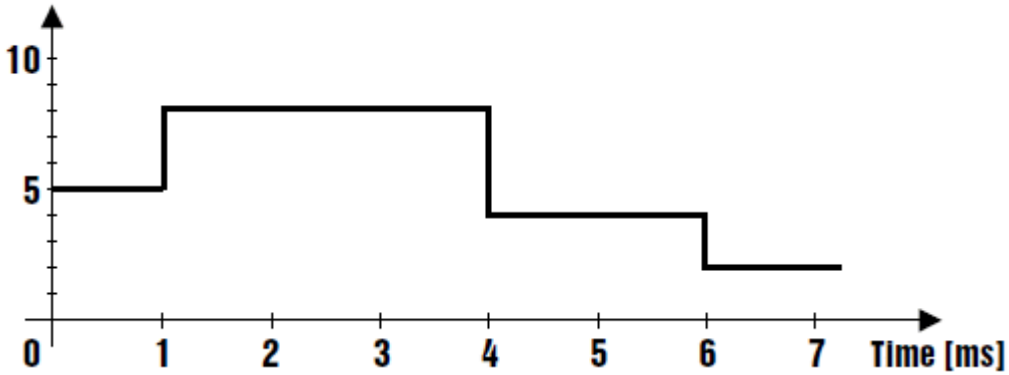


and its digital representation:

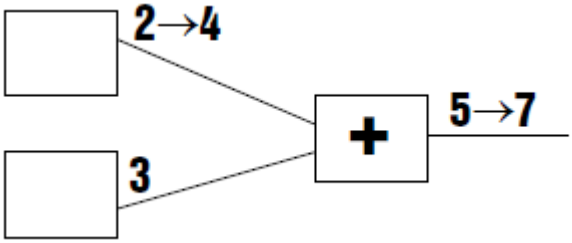


Because we are in the digital world, the outputs of our modules cannot change values continuously. On the other hand, we don't have to limit ourselves to changing values at regularly spaced points in time. For one thing, we do not have to maintain a particular sampling rate all over our structures. For 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.

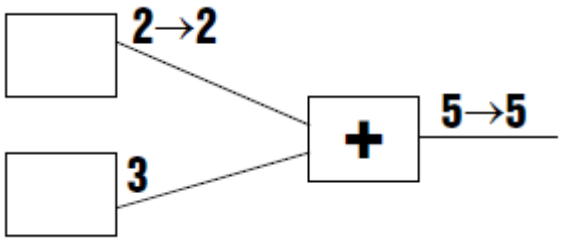
For example, at time zero the output of our adder could have a value of 5. The first change could occur at time 1 ms (one millisecond). The second change could occur at 4 ms. The third at 6 ms:



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.



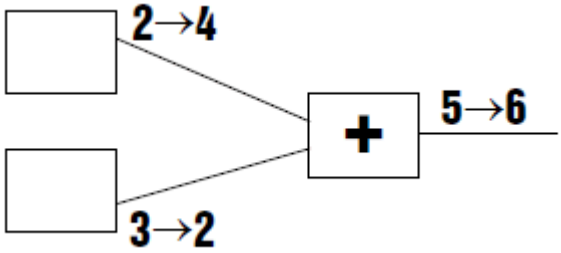
! The new value appearing at the output is not required to be different from the old one. However, the only way an output can change its value is by generating an event.

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.

3.3 Simultaneous Events

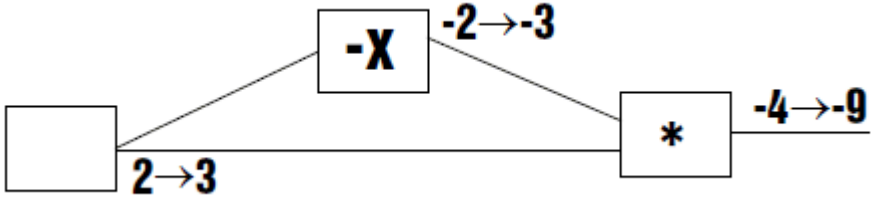
Consider the situation in which the two modules on the left side in the previous examples simultaneously produce an event.



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.

3.4 Processing Order

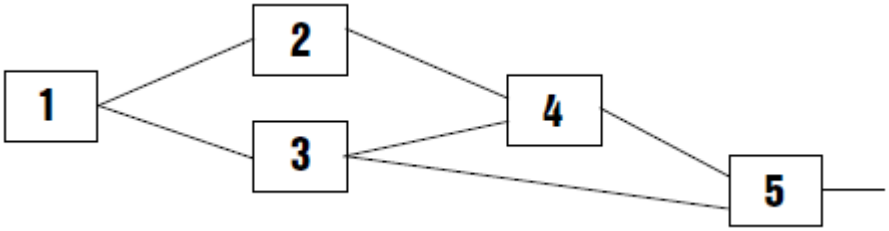
As you have seen from the previous examples, when a module sends an event, the downstream modules respond to that event. From that, one might conclude that, despite producing logically simultaneous events, the modules are definitely not processed simultaneously. One might further conclude that, for a 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.

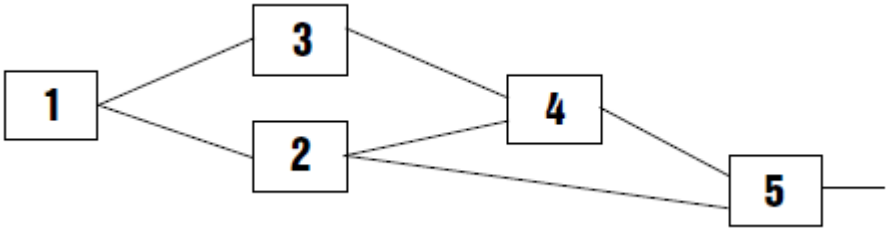
From the above rule it follows that as long as there is a one-direction connection path (always upstream or always downstream) between two modules, then there is a defined processing order for these two modules: the upstream module is processed first.

! 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:

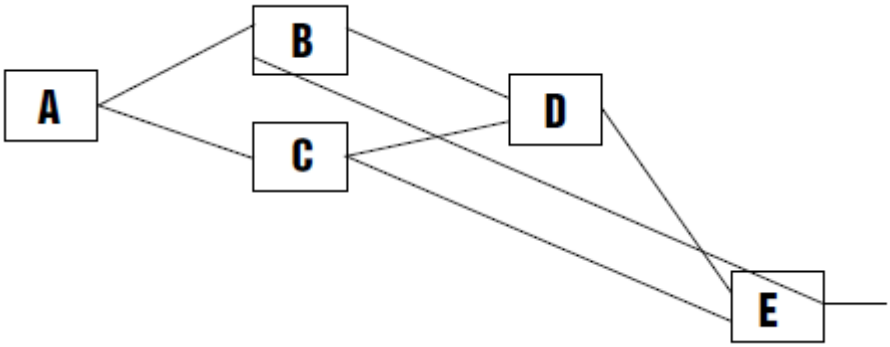


For the above structure, there is an alternative valid processing order:



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).

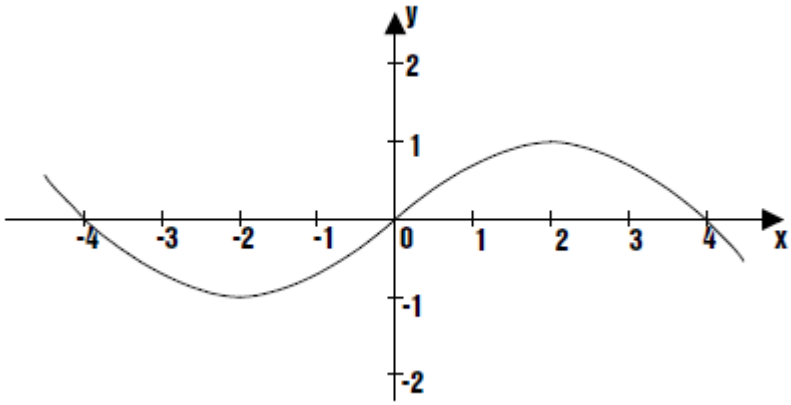
3.5 Event Core Cells Reviewed

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:

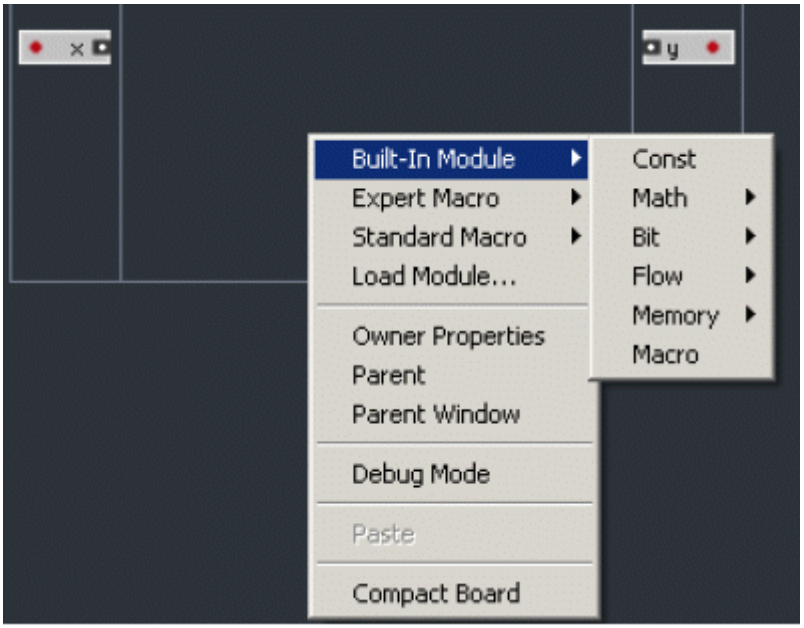
- **Event Inputs send** core events to the inside of the structure in response to primary-level events coming from outside. Because the outside primary-level events cannot arrive simultaneously at the inputs, the internally produced events also do not occur simultaneously.
- **Event Outputs** send primary-level 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:

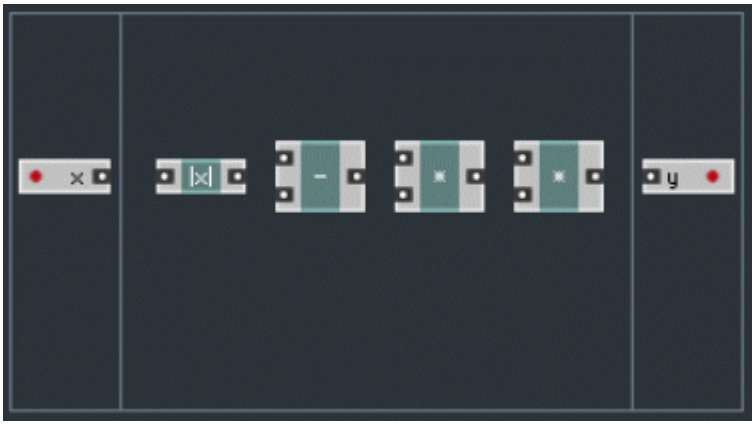


Let's start by creating a new event core cell with one input and one output, labeled "x" and "y", respectively.

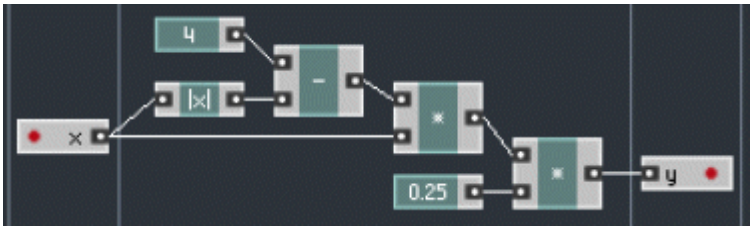
Now let's create the structure which computes the formula. We need to create **|x|** (absolute value), - (subtract), and two * (multiply) modules in the normal area. These are not core macros, but rather true Reaktor Core built-in modules. To insert built-in modules into core structures, right-click 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 submenu:



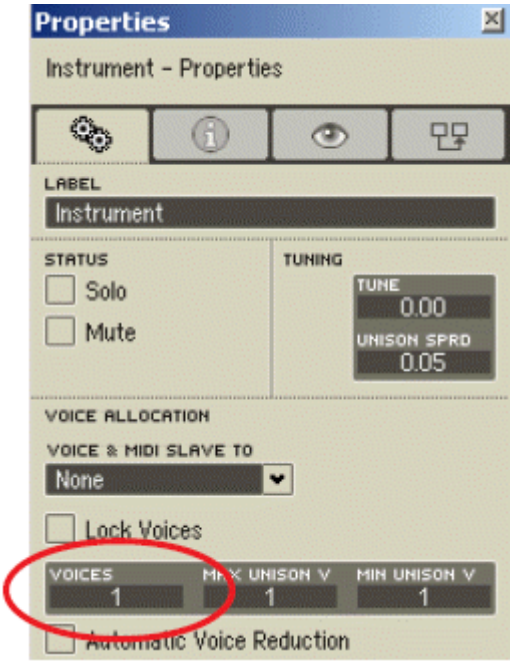
We'll need two constant values: 0.25 and 4. We could use QuickConsts exactly like we did earlier, but we can also insert real constant modules: Built-In Module > Const (as with the QuickConst, their values can be specified in the Properties window):



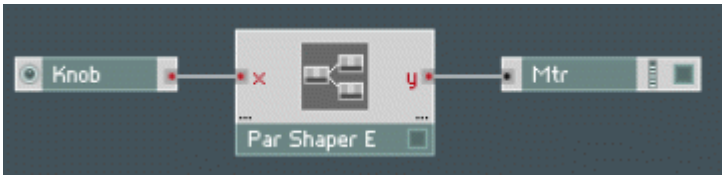
Of course, in this particular case there is no benefit in using Const modules instead of QuickConsts, but sometimes you might want to. For example, if the same constant has to be connected to multiple inputs, it may be better to use a Const module, because then you need only one of them and you also 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:

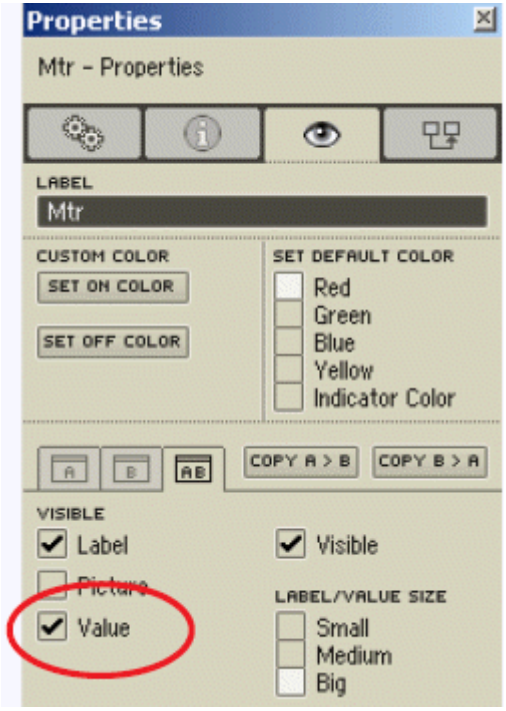
Now let's test it. Set the number of voices for the Reaktor instrument to 1, so that it will be easier to use a Meter module:



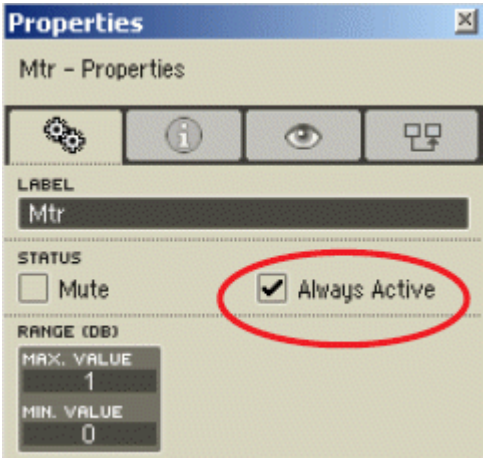
Create a Knob and a Meter and connect them to the input and output of your module:



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.

4 Structures with Internal State

4.1 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.

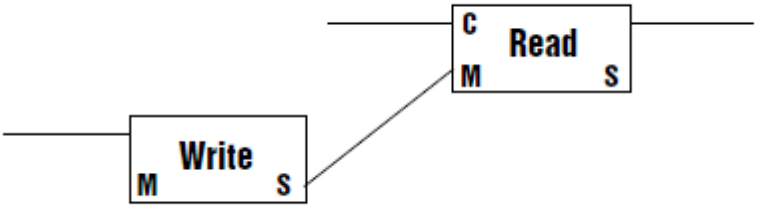
The value input (the upper one) will store the incoming value to the internal memory of the latch in response to an incoming event; nothing will be sent to the output. The clock input will send the last stored value to the output in response to an incoming event.

! Clock inputs (unless otherwise specified) completely ignore the value of the incoming event and respond only to the fact that the event is coming.

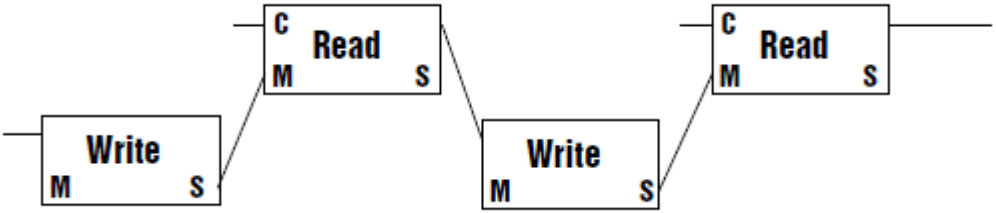
(Because now we are discussing clock signals, not latches, examples of using the Latch module will come later.)
Because there are modules with clock inputs, it should be clear that some of the signals in the structure do not carry any used (or, for that matter, useful) values. Some signals can even be 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.

4.2 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.



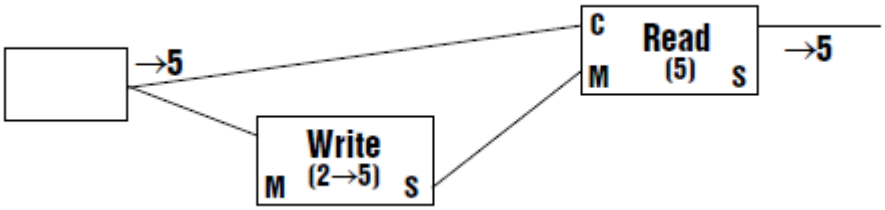
The above structure implements the functionality of the Latch macro (in fact, it is the internal structure of the Latch macro). The M and S pins of Read and Write modules are pins of Latch OBC type. The M pin is the master connection input, the S pin is the slave connection output. The master input of 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.

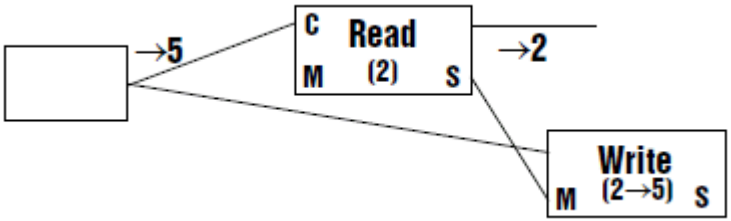
! The relative order of processing of OBC connected modules is defined using the same rules as for other modules: upstream modules are processed first.

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.

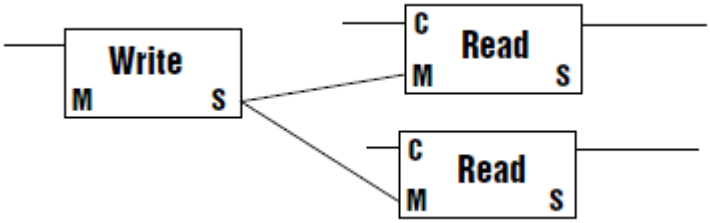
Now consider the second structure:



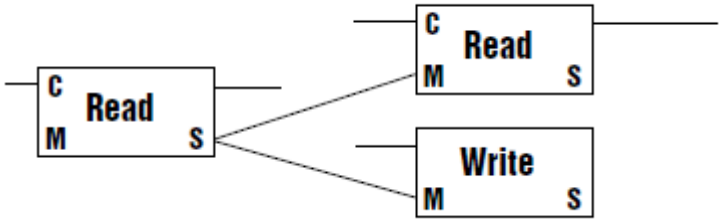
Here we have the opposite situation. First, the clock event arrives at the Read module, sending the stored value of 2 to the output. Only after that does the event arrive at the input of the Write module, changing the stored value to 5. This structure implements the functionality of a Z^{-1} block (one 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.

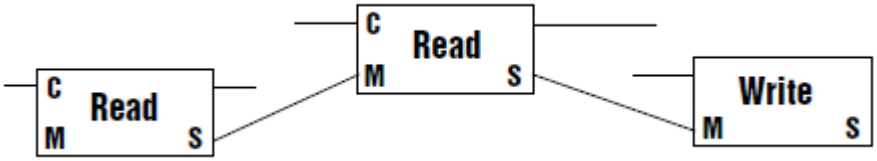
When there are more than two modules connected by OBC wires, they all share the same object. Then it becomes very important to know whether the order of specific read and write operations is important, and if so, what that order should be.
For example, in the following structure the relative order of the two read operations is undefined, but they both happen after the write operation, so it should be completely OK:



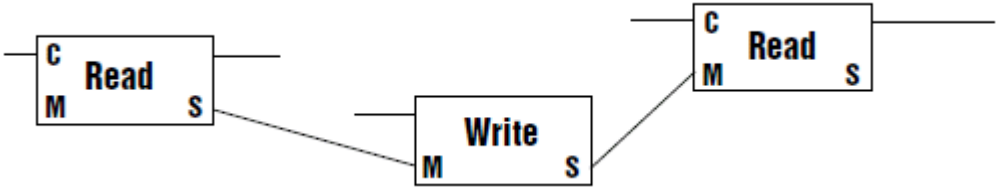
In the next structure, the relative order of the write operation and the second read operation is undefined. That can be a potentially dangerous structure and generally has to be avoided:



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.

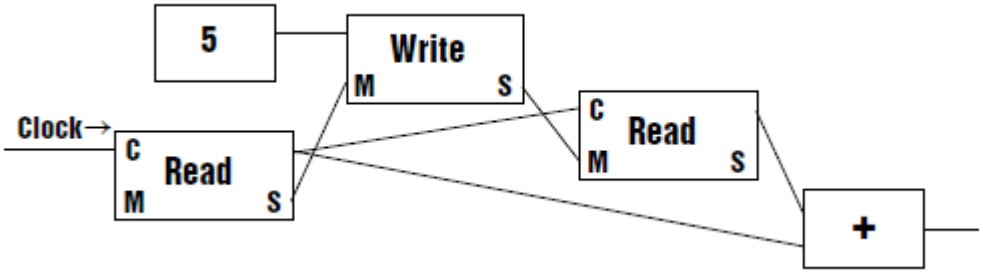
4.3 Initialization

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:

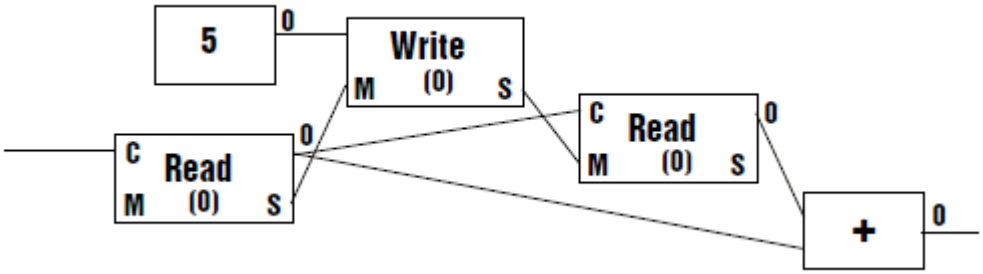
1. First, all state elements are initialized to some default values, usually zeroes. Particularly all shared memory and all output values of the modules will be set to zeroes, unless explicitly specified otherwise in the documentation
2. Second, an initialization event is sent simultaneously from all 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.

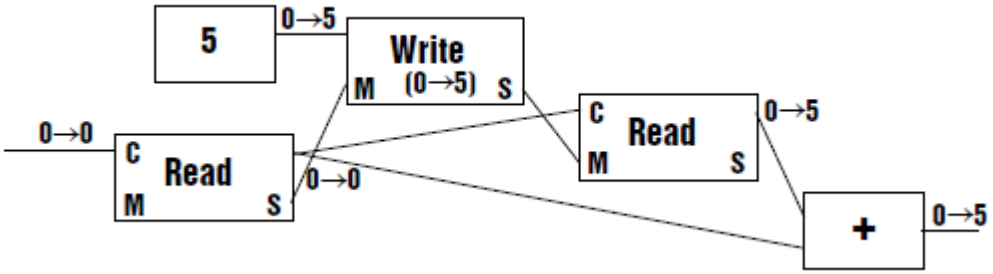
Here's a look at how initialization works:



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.

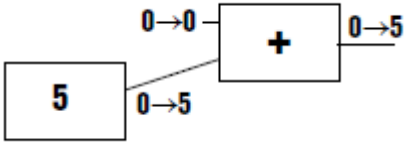


Then an initialization event is sent simultaneously from the clock source and from the constant 5.



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.

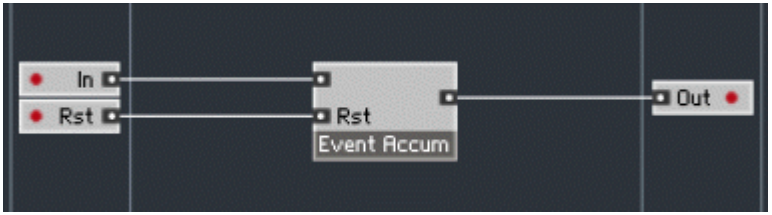


Above, an adder with one input disconnected and one connected to a constant module receives two simultaneous initialization events, one from the default zero constant connection and one from a real connection to a constant.

! 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.

4.4 Building an Event Accumulator

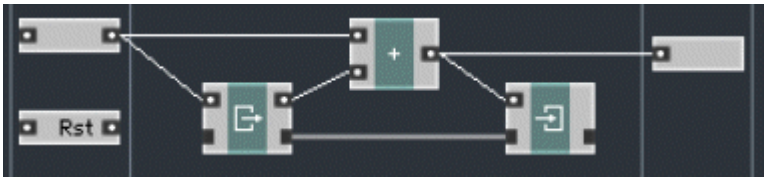
The event accumulator module that we want to build now is going to have two inputs: one for the event values to be accumulated, and one for resetting the accumulator to zero. There is also going to be one output, which outputs the sum of the accumulated events.
We are going to build this module in the 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) in 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.

4.5 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.

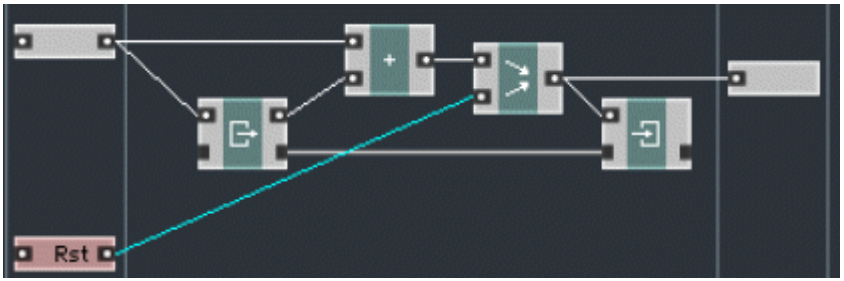
Here we have a special rule for the Merge module:

> **!** 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.

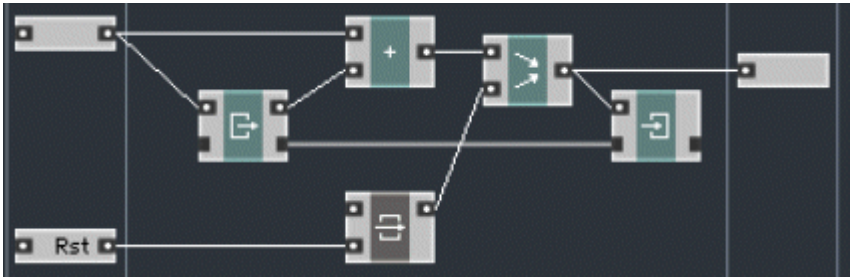
4.6 Event Accumulator with Reset and Initialization

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:



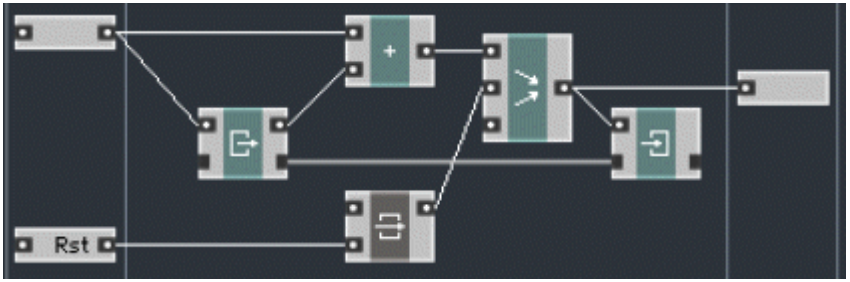
Now the reset works as specified.

The last thing we have to do is ensure the correct initialization, which of course requires defining what is the correct initialization. Let's take a look at how the above structure is going to be initialized.

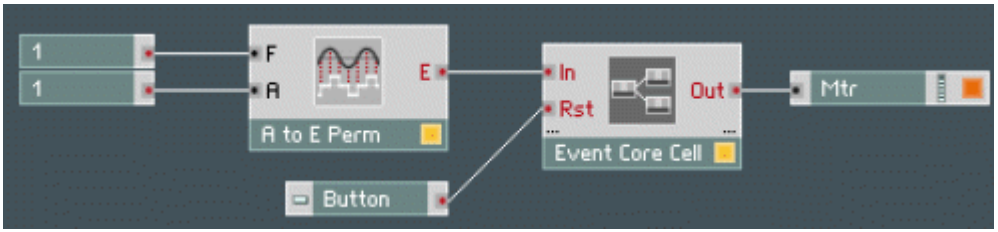
If the initialization event is sent simultaneously from the In and Rst inputs of the core cell top-level structure, and also from the implicit zero constant at the value input of the Latch, then the Latch triggered by the Rst input will send a value of zero to the second input of the Merge, overriding whatever value arrives at the first input of the Merge. Therefore, zero will be written into the internal state and sent to the output – perfect!

There's one little problem with that, however. It could be that the initialization event doesn't arrive at one or both of the ports. That could be because the initialization event didn't arrive at the corresponding input of the event core cell or because this macro is used in a more complicated Reaktor Core structure that also doesn't get the initialization event on all its wires (we will learn how that can be arranged later). So we need to do a last final modification to the structure to make it more universal.

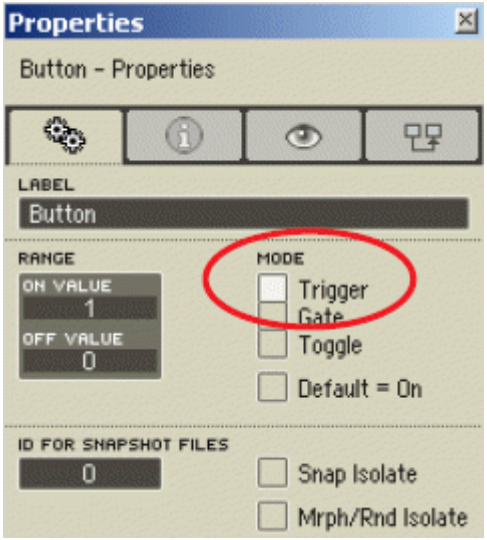
Go to the properties of the Merge module and change the number of inputs to 3.



Now, even if there was no event arriving at the Rst input, the implicit zero constant at the third input of the Merge would still send an initialization event, producing the correct output and initial state values.
Let's try out our new event accumulator by building the following primary level structure using the newly created Event Accum module.



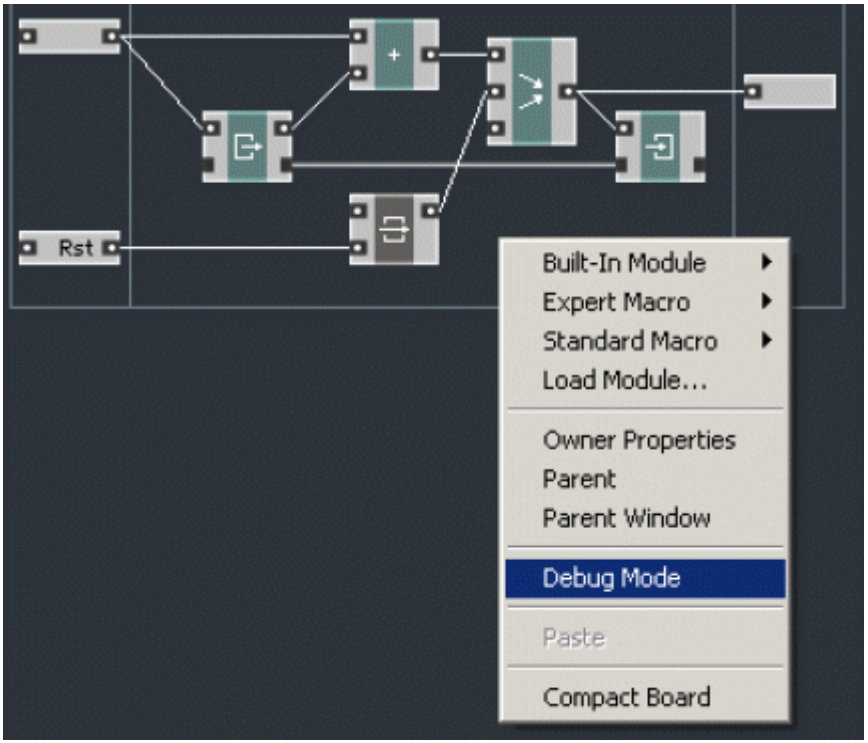
The instrument number of voices should be set to 1, and the meter should be set to display a value and to be always active, as in the previous example. The button should be set to the trigger mode.



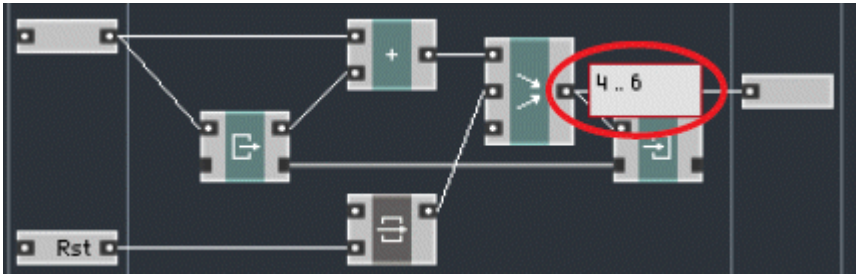
Now switch to the Panel and see the values incrementing in steps of 1 each second and resetting in response to pressing the button.

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.

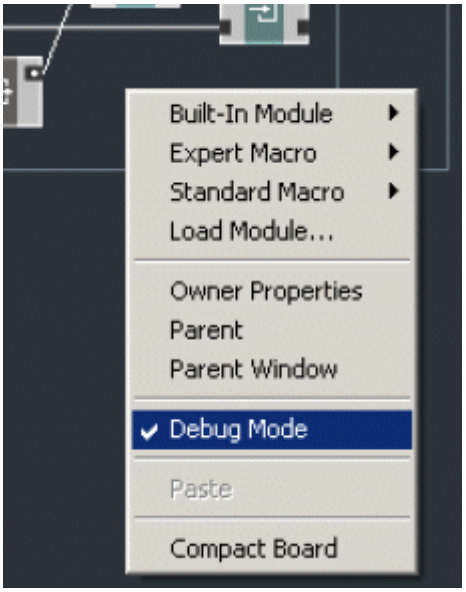
Ok, we already hear you complaining, and we've provided a compromise. You can disable the optimization for a particular core structure in order to see the output values. Let's try that with the structure we've just built. Right-click on the background and select Debug Mode:



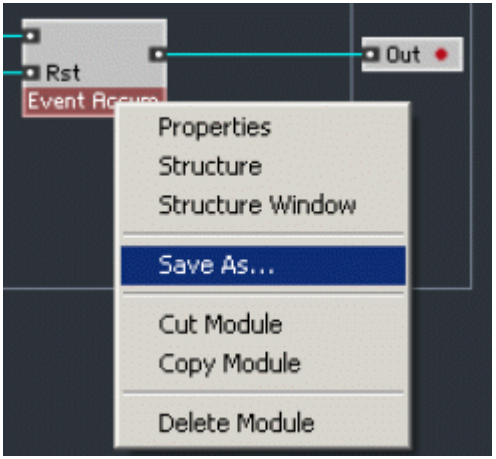
(You can do the same thing using the button with the bug icon on the toolbar). Now if you keep your cursor over a particular output, you will see its value (or range of values):



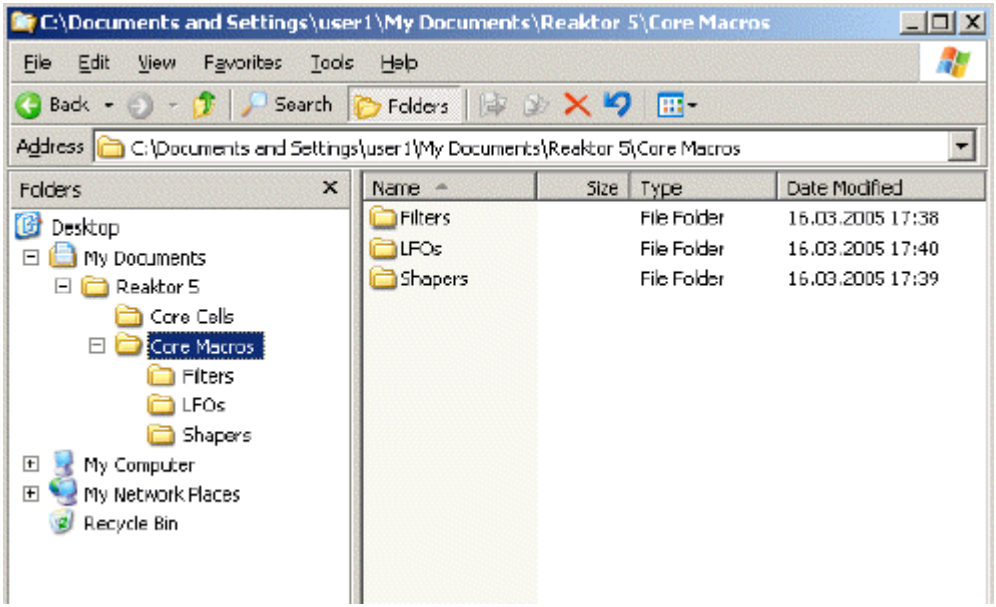
You can disable debug mode by selecting the same command (or pressing the button with the bug icon) again:



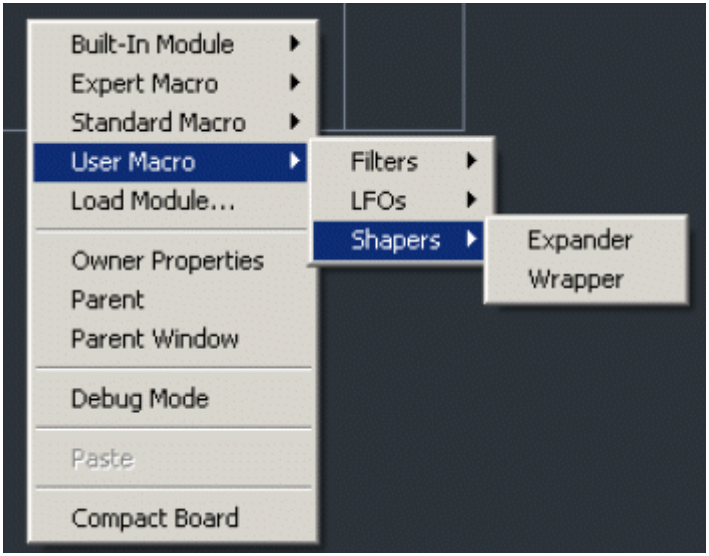
Also, it will be automatically turned off when you leave the structure, so you may need to enable it again for another structure. After debugging our core macro we might consider saving it as a separate file for future use. That can be done by right-clicking on the macro and selecting Save As...:



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:

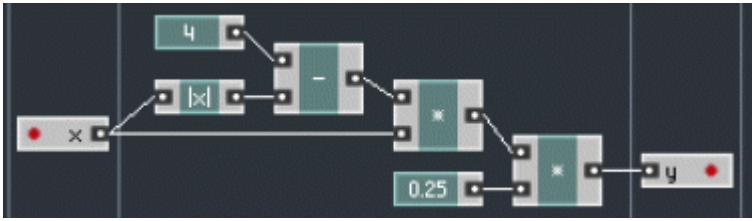


Similar restrictions apply to the Core Macros folder as apply to the Core Cells folder:

- empty folders are not displayed in the menu
- never put your own files into the system library, put them into your user library folder

4.7 Fixing the Event Shaper

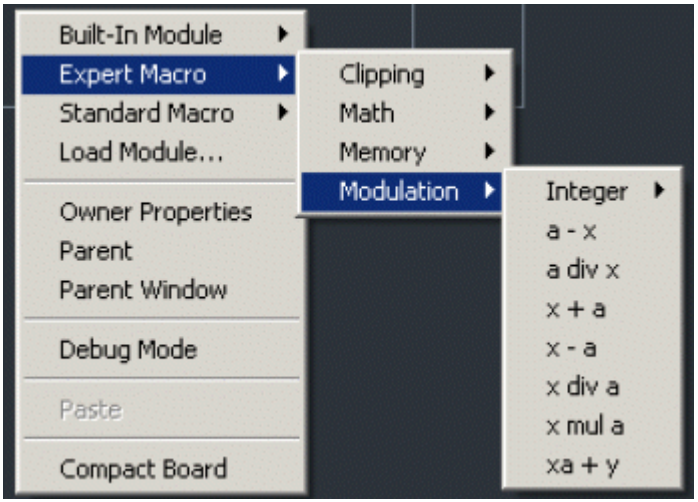
We can now discuss in more detail exactly what's wrong with the event shaper module structure we built earlier:



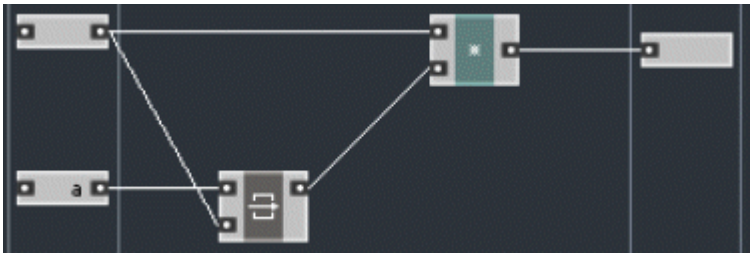
The problem is the initialization event. If you consider how the initialization of the above structure will happen you'll notice the following:

- 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

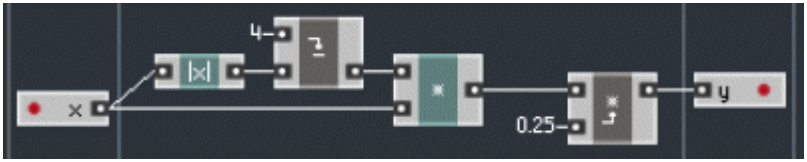
Thus, in case for whatever reason the initialization event does not occur at the input of the shaper, the output of the shaper will still receive the event from the last multiplier and will forward that event to the outside primary level structure.
Although for control signal processing purposes, that 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 modulatior 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 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.

5 Audio Processing at Its Core

5.1 Audio signals

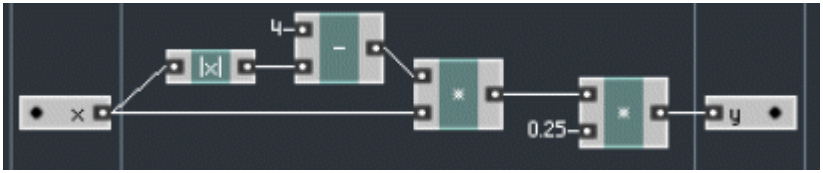
There is no special type for audio signals in Reaktor Core; audio signals are represented by events that, from the structure point of view, do not differ from any other event. What is different is that along the audio signal path the events are normally produced at regularly spaced time intervals, where 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.

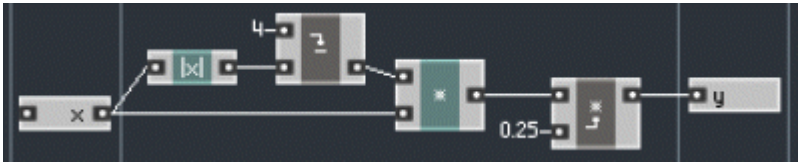
There is also a new output type which has to be used instead of event outputs.

- 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.

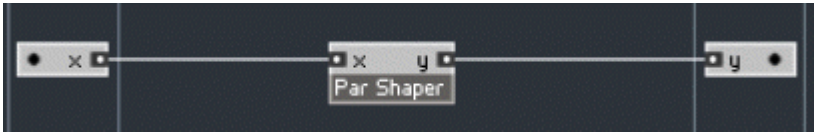
Now we are going to rebuild the same shaper we built for events in audio mode. Therefore we create a new audio core cell. Generally we can use exactly the same structure, except instead of event inputs and outputs we will have audio inputs and outputs:



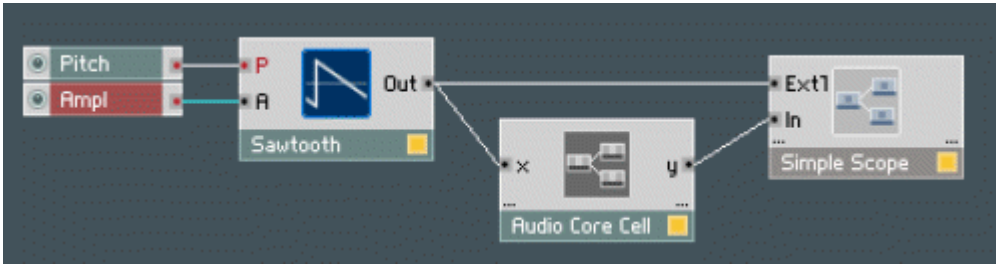
You may wonder why didn't we use the modulation macros in this case? The reason is that we are processing audio signals here, and audio signals always send the initialization event, so it's safe to do it this way. (You could use modulation macros if you prefer; it doesn't really matter.)
We could also 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 > 00 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.

5.2 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.

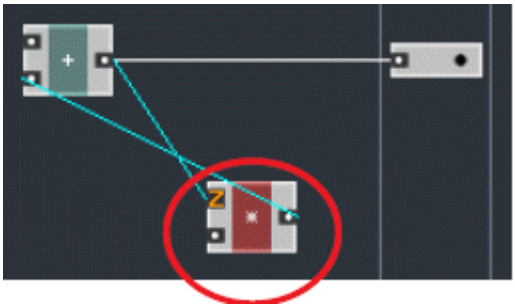
You can access the sampling-rate bus by right-clicking on any signal input and selecting “Connect to SR.C” for clock signal or “Connect to SR.R” for rate signal.

The connection will be displayed next to the input:

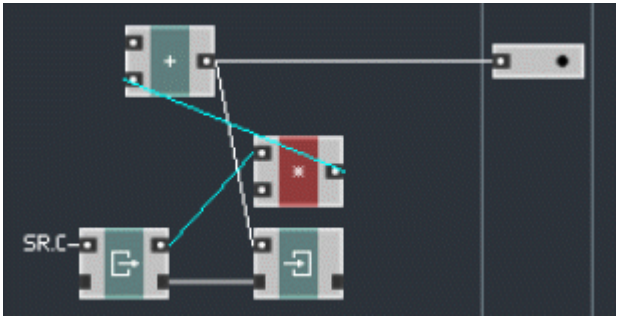
! The sample rate clock bus doesn't work inside event core cells.

5.3 Connection Feedback

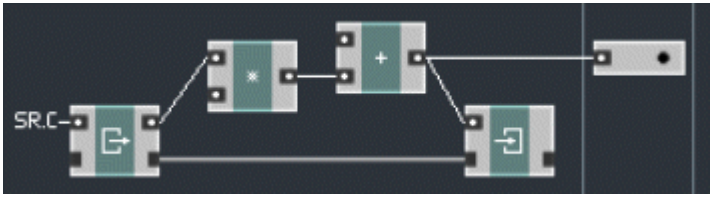
As we have already seen, the processing order rules cannot be applied if there are feedback connections within a structure. Therefore we need to provide additional rules, defining how feedback is handled.
The main rule is: Reaktor Core structures cannot handle feedback.
Well, not exactly so. You can 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.

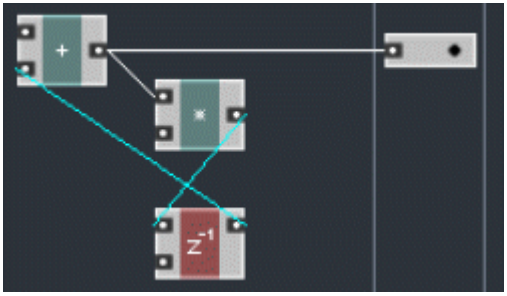
So, inserting an explicit Z^-1 module formally removes the feedback from the structure, while keeping it there logically (with a one audio sample delay).

> ! 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.

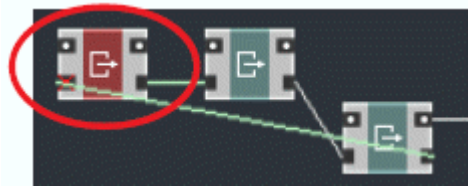
Here is a version of the above structure with a Z^-1 macro inserted (it can be found in Expert Macro > Memory submenu):



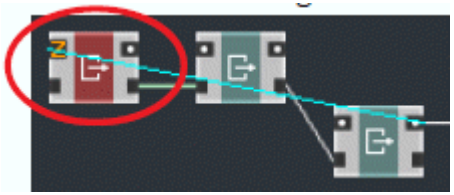
As you can see, the big orange Z mark is gone now. Also note that the 1-sample delay point is different from the one which was automatically inserted (the automatic one was on the wire going from the Adder output to the Multiplier input and now it's on the wire going from the Multiplier output to the 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:



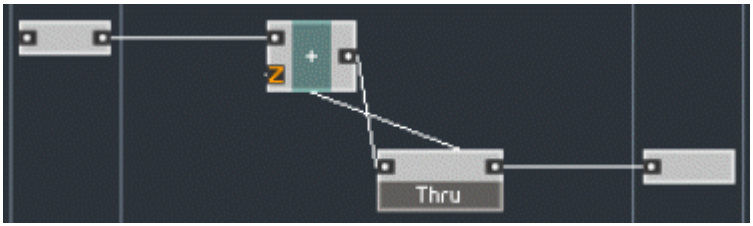
! In essence, this means that non-signal connections are never affected by feedback resolution, unless you make a completely non-signal feedback, which doesn't make any sense.

5.4 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:

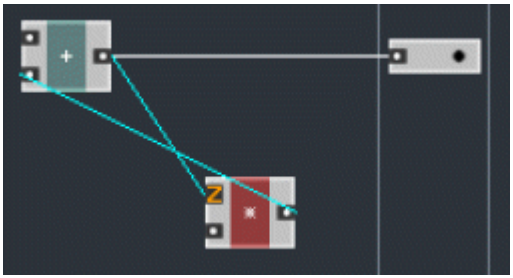
Now assume we build a feedback structure using this macro:



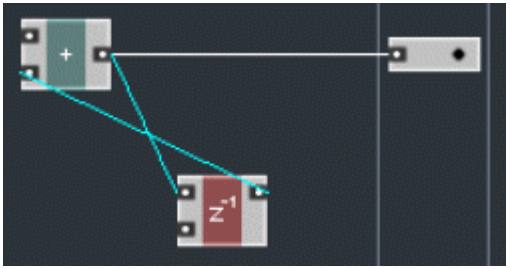
The feedback loop goes through two wires in the above structure and through another wire inside of the macro. Now where is the resolution going to occur? (OK, you can see in the above picture that is occurring at the adder input in this particular case, but we know it might as well have occurred at another 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.

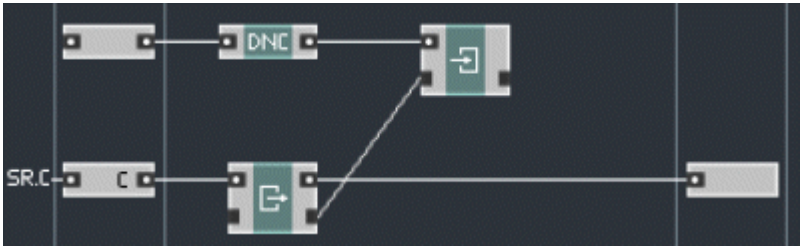
However, you can change that behavior and allow feedback resolution to happen inside the macros. In fact, you should have wondered, if macros are treated the same as built-in modules, how can a Z^-1 macro resolve the feedback. Consider the following structure:



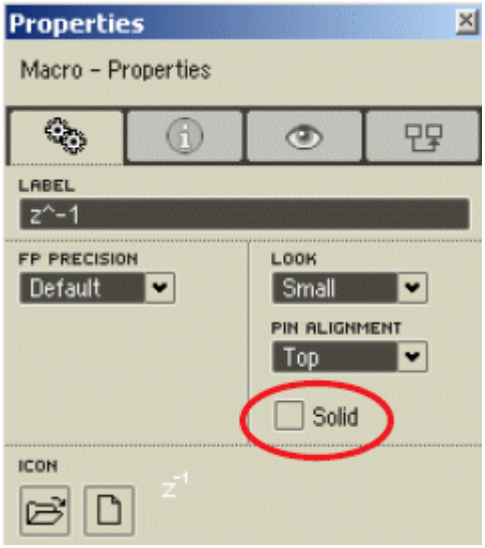
If macros and built-in modules are the same then nothing should change when we replace the multiplier by a Z^-1 macro:



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:

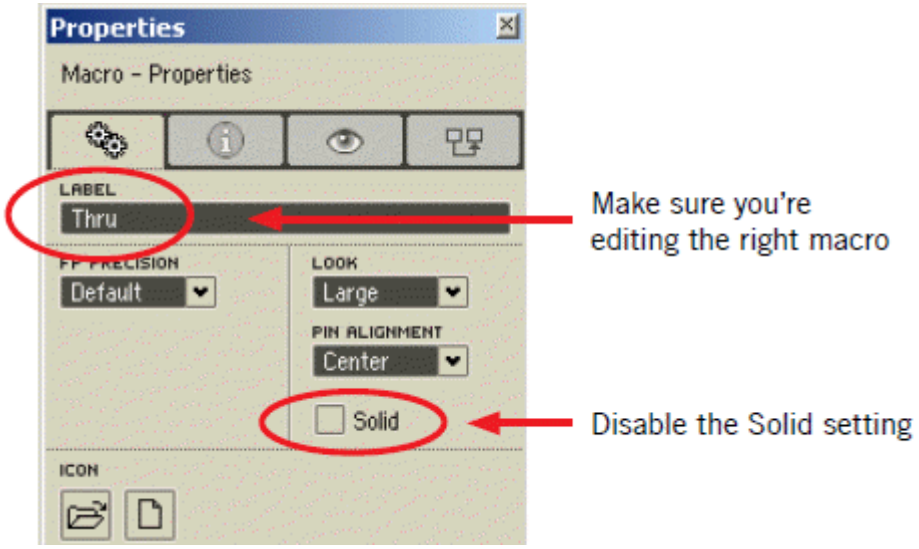


As you can see, the clock input of the macro is connected to the internal Read module. The default connection for this input is not to a zero constant, but the audio clock, and that's what you would want in most cases. The module connected between the upper input and the write module will be explained later, for now just ignore it.
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:

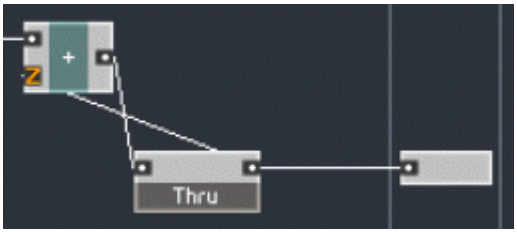


The Solid property tells the Reaktor Core engine whether the macro is to be considered as a solid built-in module for the purposes of feedback resolution or whether it is to be considered transparent. In 99% of the cases, you would want to keep this property on. That's because you typically don't want implicit feedback resolution to happen inside your macros.

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.

5.5 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:

$V_{out=} V_0 e^{-2\pi fct}$

where f_c is the filter cutoff in Hz, t is time in seconds and V_o=1V (initial voltage).

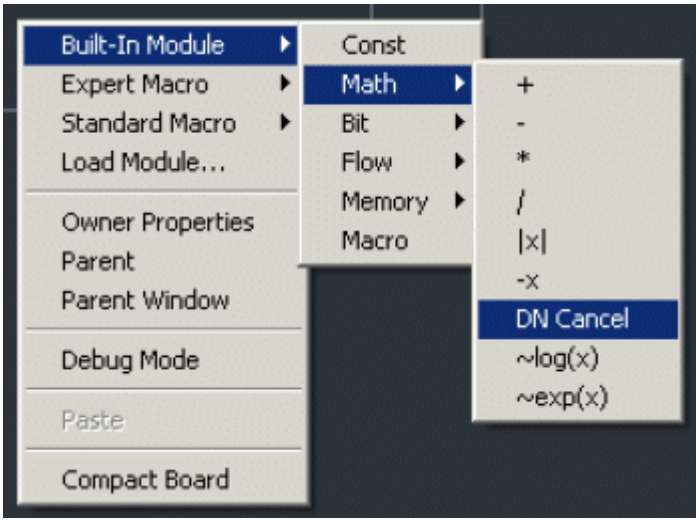
Then the output voltage will change as follows:

- 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:



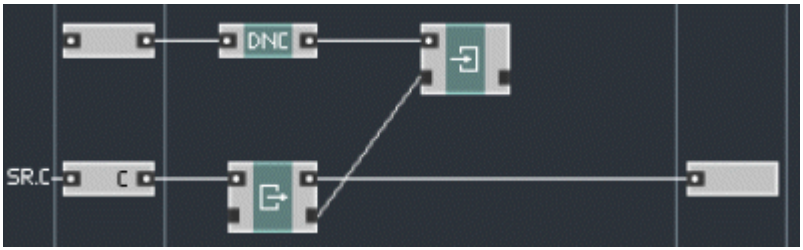
The Denormal Cancel module has one input and one output, and it tries to slightly modify the incoming value in a way that prevents denormals from occurring at the output:

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:

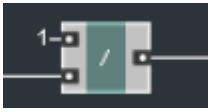


Now you probably can tell what the Denormal Cancel module is doing in there. Because you would often use the Z^-1 macro inside feedback structures, there's a good possibility of denormals occurring. We therefore decided to put the DNC module into the Z^-1 macro structure.
There's another version of this macro called Z^-1 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):

5.6 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.

5.7 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:

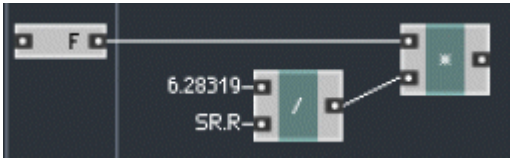
- $F_c = 2*\pi*f_c / f_{SR}$ where
- f_c is the desired cutoff frequency in Hz
- f_{SR} is the sampling rate in Hz
- π is 3.14159...
- F_c is normalized circular cutoff (in radians)

> **!** 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 a 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:

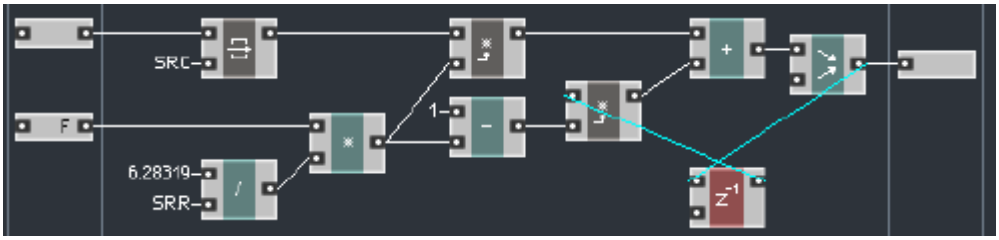
Now let's build the circuitry for converting the cutoff frequency into the normalized circular cutoff:



6.28319 is 2*π, 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.

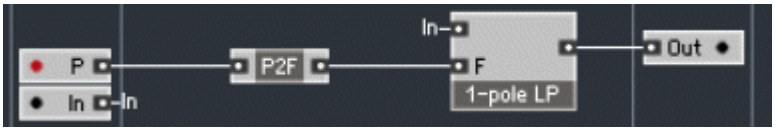
Let's build the circuitry implementing the filter's equation:



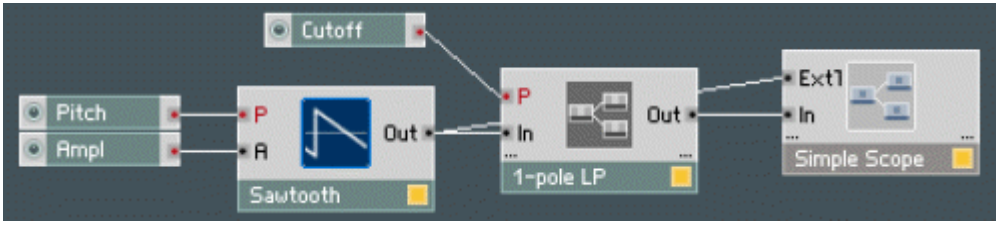
audio inputThe is latched just in case events at this input arrive asynchronously to the standard audio clock. That wouldn't be necessary in the core-cell structure, where an audio input is known to send events at correct times, but in a general core macro it is a very good practice.
Two modulation multipliers 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 cut-off-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:



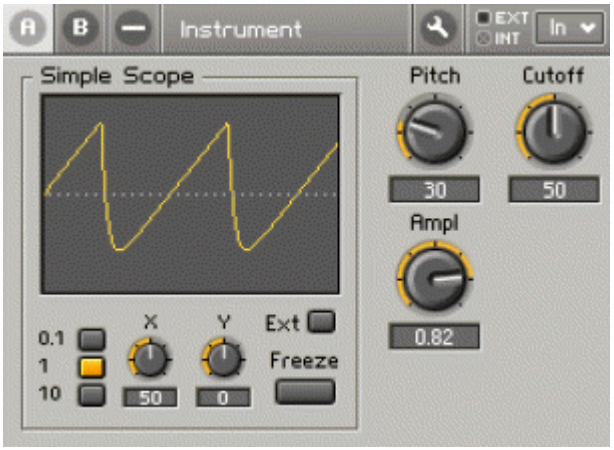
For testing we suggest using the following structure (don't forget about the 1-voice setting for the instrument):



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:

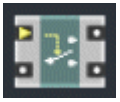


Move the cutoff knob and watch the signal shape changing.

6 Conditional Processing

6.1 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):

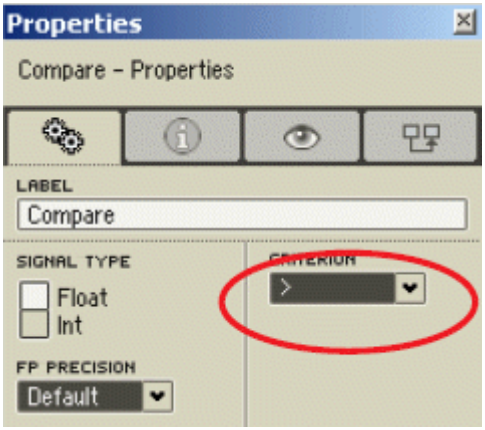


The Router module accepts events at its signal input (bottom) and routes them to either its output 1 (top) or its output 0 (bottom). The routing, i.e. whether the event goes to output 1 or output 2, depends on the current state of the Router, which is controlled from the Ctl input (top)
The Ctl input 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:

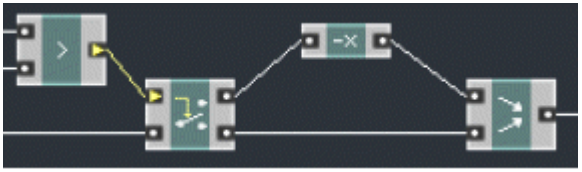


The available criteria are:

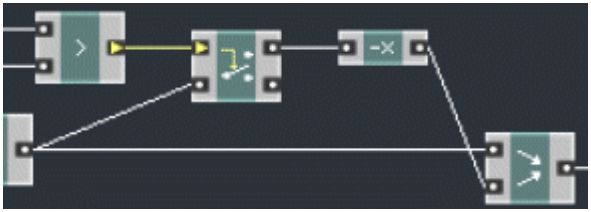
- = equal
- != not equal (≠)
- <= less or equal (≤)
- < less
- >= greater or equal (≥)
- > greater

> ! 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:



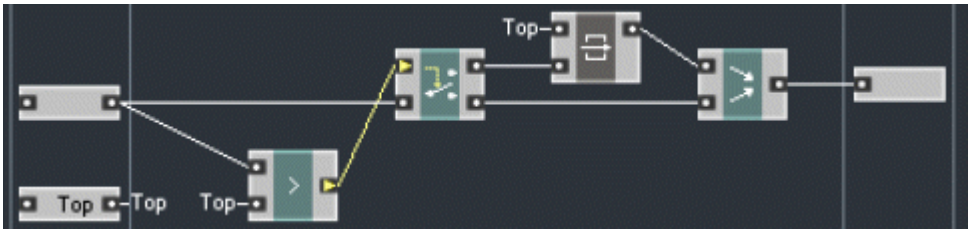
In this version, the 0 output of the Router is disconnected; therefore, the Router works as a gate, letting the events through only if it's in the 'true' state. The inverted value then arrives at the second input of the Merge, thus overriding the non-inverted value, which is always arriving at the first 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.

6.2 Building a Signal Clipper

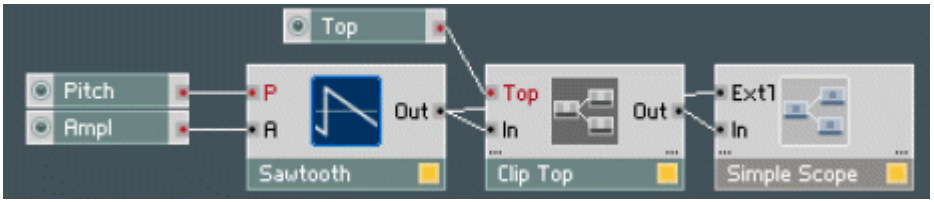
Let's build a Reaktor Core macro structure that would clip the incoming audio signal from the top at a specified level:



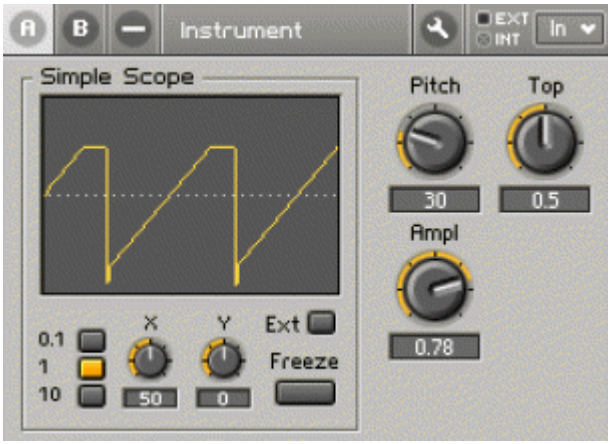
If the input signal is not greater than the threshold it will be routed to output 0 of the Router and, through the Merge, to the output of the structure. Otherwise, the signal will be routed to output 1, where it triggers the latch, sending the threshold value to the Merge instead. The same thing happens 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.

6.3 Building a Simple Sawtooth Oscillator

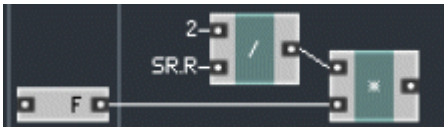
Let's build a simple sawtooth oscillator, generating a sawtooth waveform with amplitude 1 and a specified frequency. We will use the following algorithm: increment the output signal level at constant speed and at the moment the level becomes greater than 1 drop it by 2.

> ! 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.

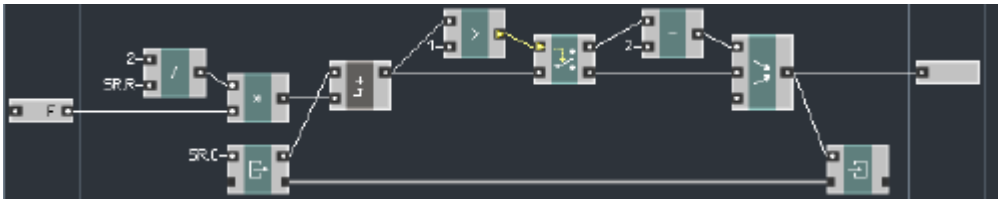
The incrementing speed defines the oscillator frequency by the following equation:

- $d = 2f / f_{SR}$

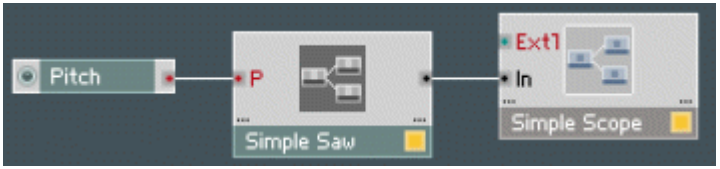
where d is the level increment per one audio sample, f is the oscillator frequency and fSR is the sampling rate. First we are going to build the circuitry for computing the incrementing speed:



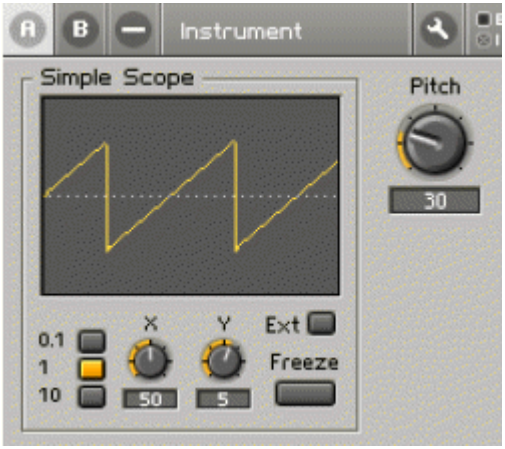
Now we need the increment loop. It's time to use a pair of Read and Write modules exactly as we did in the accumulator:



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:



7 More Signal Types

7.1 Float Signals

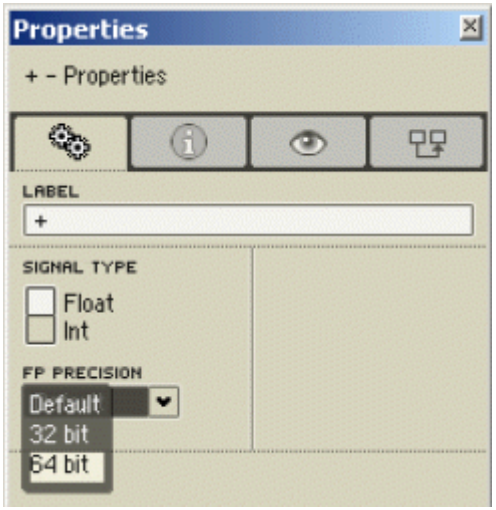
The most common signal type used for DSP (digital signal processing) on modern personal computers is floating point (float for short). Floats can represent a wide range of values, as large as 10^{38} (in 32 bit mode) or even 10^{308} (in 64 bit mode). As useful as they are, floats have a drawback – limited 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 π, 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).

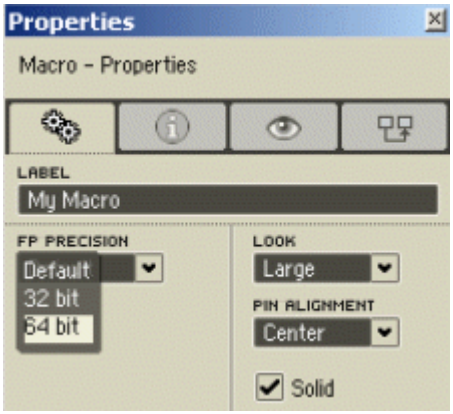
You can change the floating point precision for individual modules as well as for whole macros. For individual modules you do so in the module's FP Precision (floating point precision) property:



- default means use whatever precision is the default for the current structure
- 32 bit use minimum 32 bits of precision
- 64 bit 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.

7.2 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.

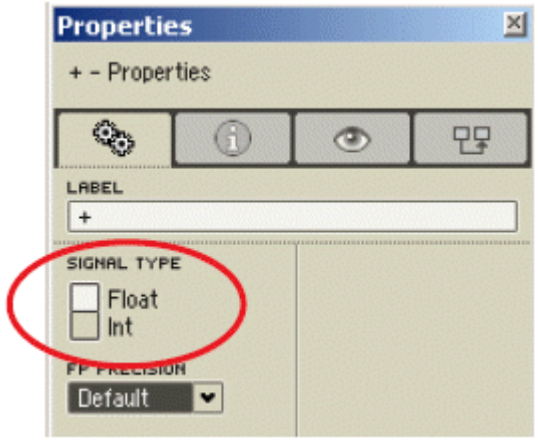
! Infinite precision for storage and processing of integer values is possible because they don't have any decimal digits after the period, so you can write them using a finite number of digits. Let's write down the number of seconds in an hour: 3, 6, 0, 0, done. It's that easy. If you try to write down the value of π 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:



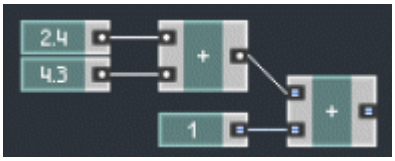
A module set to integer type will process the input values as integers and produce integer output values. You can tell that a module is in integer state by the fact that its signal inputs and outputs look different:

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).

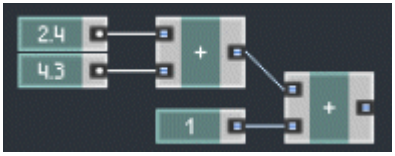
Integer signals can be freely interconnected with floats, but the wires created between different type signals will perform signal conversion, which can use a certain amount of CPU. At the time of this writing, the extra CPU usage is somewhat noticeable on PCs and quite significant on Macs. The OBC connections 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 in 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.

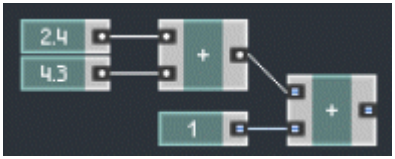
It is important to understand that turning the processing mode of an operation to integer and converting of a floating point result of the same operation to an integer is not the same. Let's consider an example. Here we are adding two numbers 2.4 and 4.3 as floats. The result is clearly 6.7, which when 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:

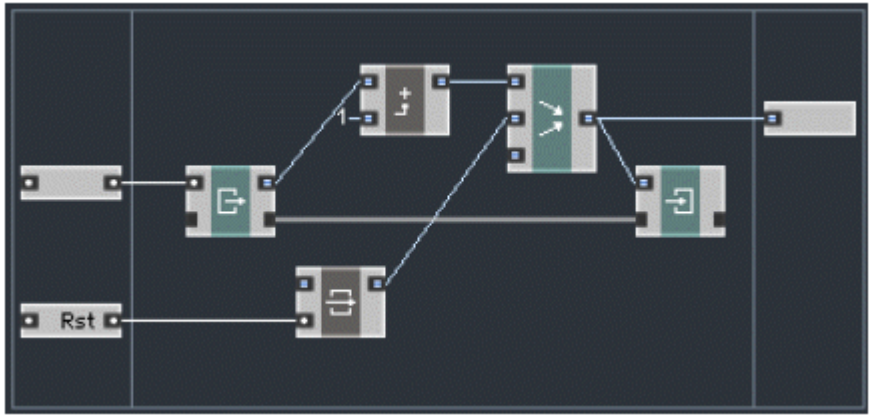


Here the clock input of the Read module is still float although the module has been set to integer mode (the OBC ports look the same regardless whether they are float or integer).

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).

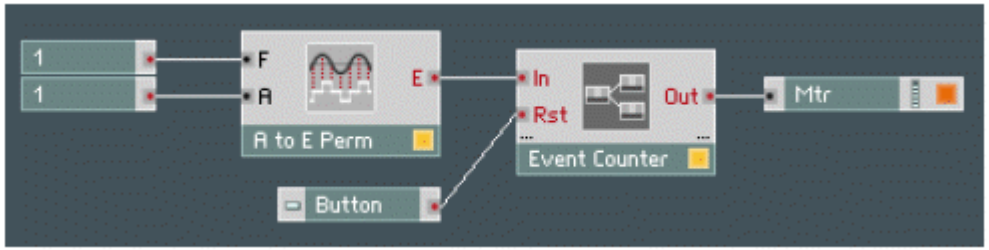
7.3 Building an Event Counter

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), buts work 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:

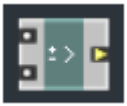
7.4 Building a Rising Edge Counter Macro

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:

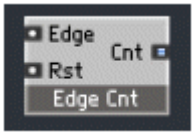
- 3.1 is sign-greater than –1.4
- 2.1 is sign-equal to 5.0
- 4.5 is sign-equal to –2.9

! 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):

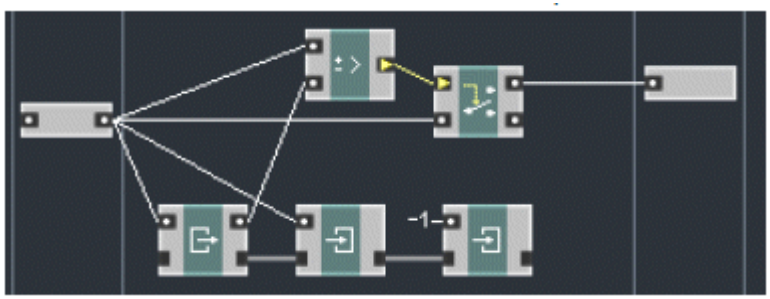


This module produces a BoolCtl signal at the output, so that you can connect it to a Router.
One of the possible uses of such a module is detecting the rising edges of an incoming signal. Below we are going to build a rising edge counter Reaktor Core macro:



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:

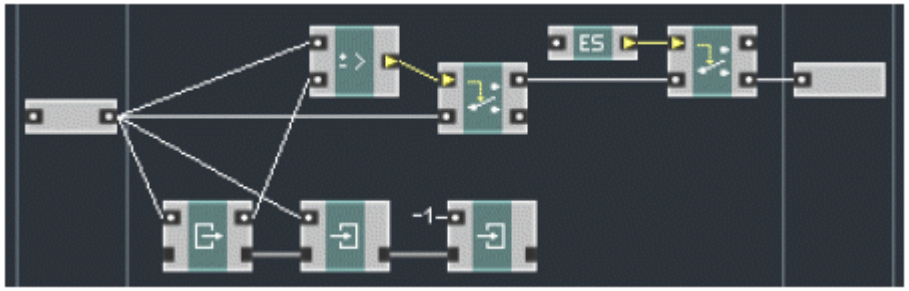
This is how the detector macro can be implemented:



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.

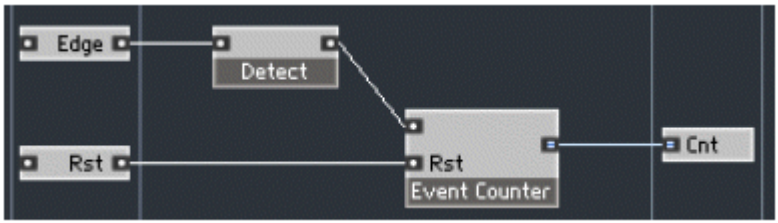
The Router controlled by the Sign Comparison module will gate the events, letting through only those where a sign change from negative to positive occurs.
It's not clear whether such a module will send an event during initialization or not, particularly because the storage is still zero at the time of 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:



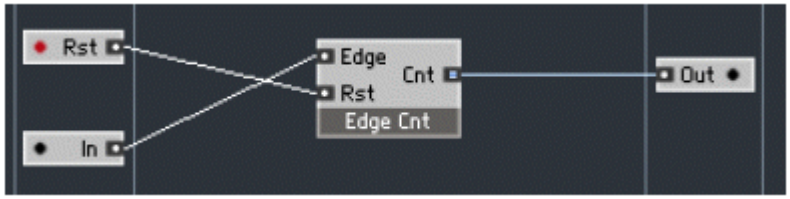
The ES Ctl module is an event sensitive control. The control signal produced by this module is true only if there is an incoming event at the input of this module. Because this input is disconnected in the above structure, which means it's connected to a zero constant, the only time the control signal 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.

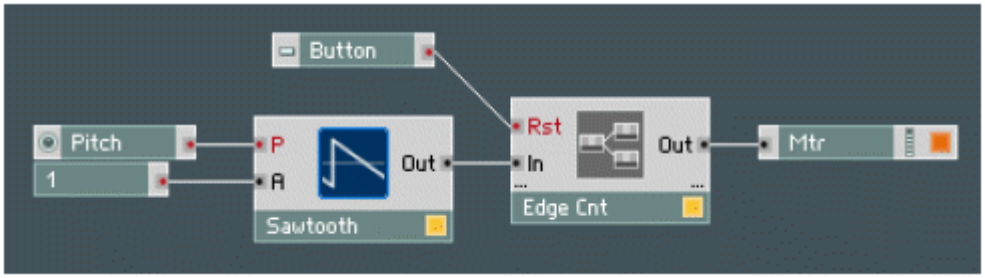
Now that we have a detector module we can connect it to the counting circuitry that we already have:



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:



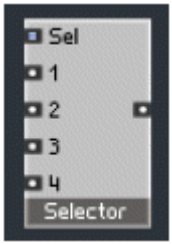
(Don't forget to set the Meter properties as in previous examples.)
Here is what you should see in the panel:

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.

8 Arrays

8.1 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.

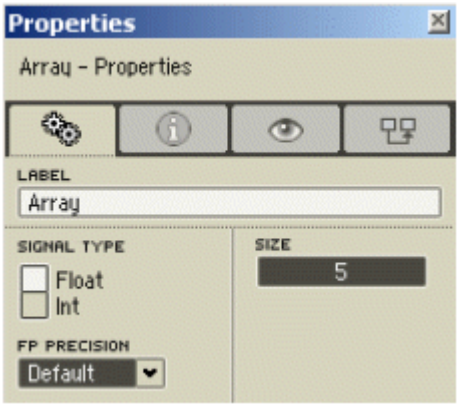
Here is a representation of this array using a table:

Index	Value
0	5.2
1	16.1
2	-24.0

Index	Value
3	11.9
4	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:



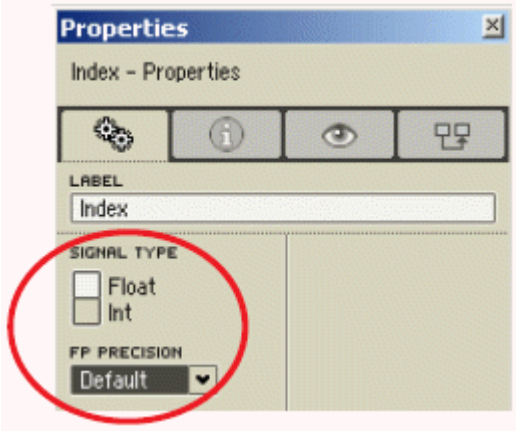
For example, for the above table of 5 elements we will need to specify the float data type and the size of 5.

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).

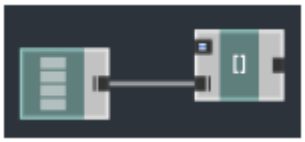
Array OBC signals corresponding to different item data types are, of course, not compatible.

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:



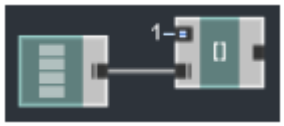
And now the connection:



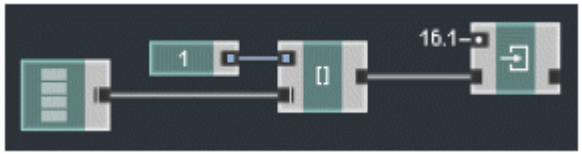
The upper input of the Index module is always integer type and accepts the index value. Here we are addressing the array element with the index of 1:

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:



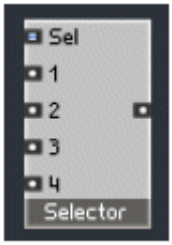
The output of the Index module is a Latch OBC type, which means that you can connect Read and Write modules (or even several of them) to that output. Of course, you need to take care that the Read and Write modules are set to the same data type as the data type of the Array and Index modules.
Here the 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.

8.2 Building an Audio Signal Selector

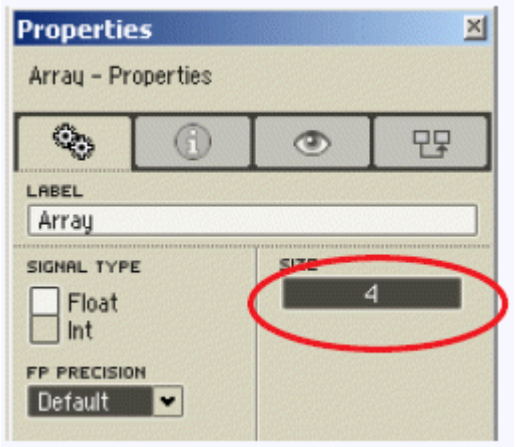
Now let's return to building the audio signal selector module:



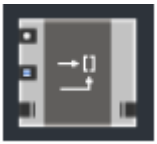
Here is an empty internal structure for this module:

We are going to use an array of 4 float elements for storing our audio signals:

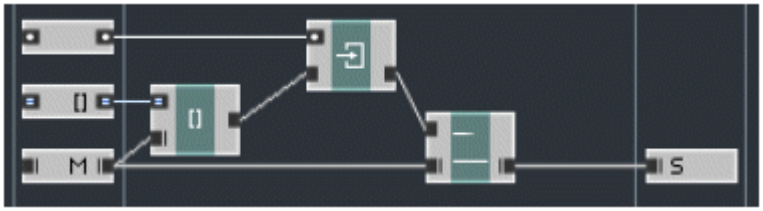
Here are the properties of the array module:



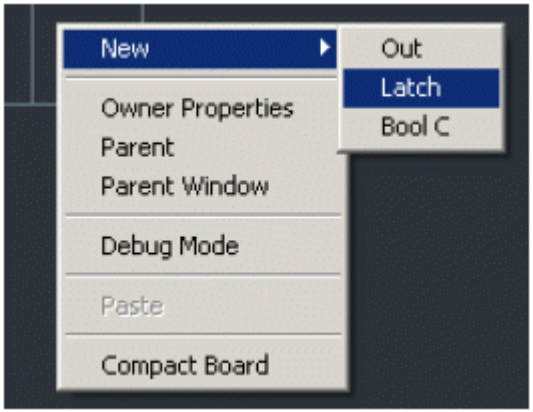
To write the input values into the array we will use the standard macro Write [] (Expert Macro > Memory > Write []):



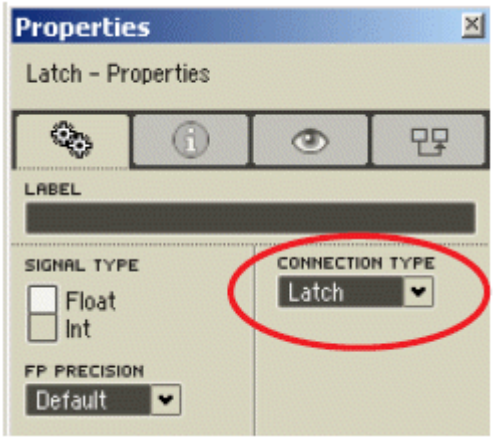
This macro internally has an Index module and a Write module, performing writing into the array element with a specified index:



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 BoolCtl 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:

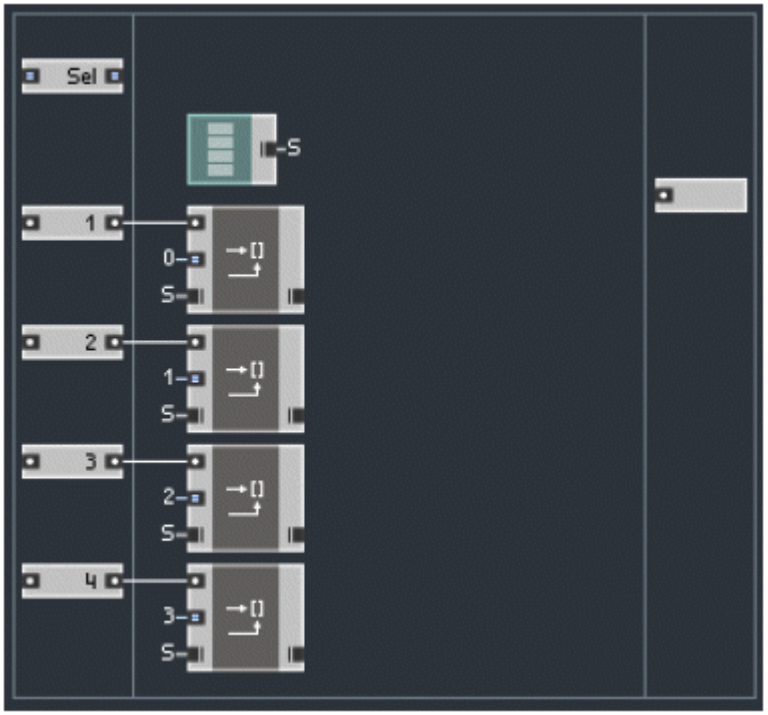


Setting the connection type to Latch or Array defines the OBC connection type between latch OBC and array OBC, respectively. For the Write [] macro ports this has obviously been set to Array type.
The module with two parallel horizontal lines is the R/W Order module (Built-In Module > Memory > R/W Order):

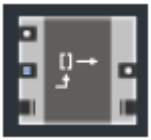
It does nothing except let the connection at its master (bottom) input through to its slave output. The upper input has absolutely no effect; however, because there is a connection at this input, it will affect the processing order of the modules. Therefore, everything connected to the S output of the 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.

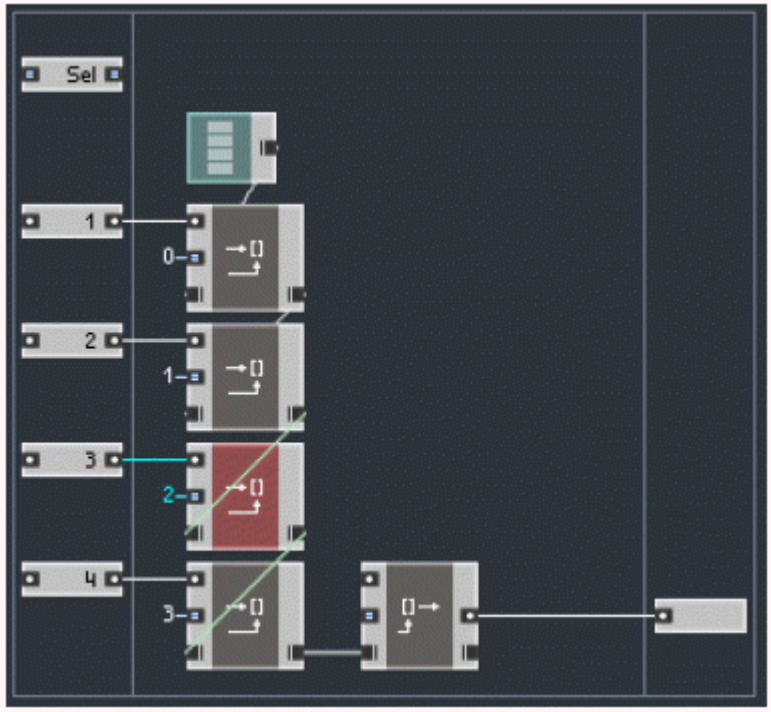
Like OBC ports, the R/W Order module has a Connection Type property. For this module the Connection Type property controls only the type of the M and S ports; the sidechain input is always in latch mode. See the description of R/W Order in the module reference section for details. Now let's build the circuitry for writing the input signals into the array:



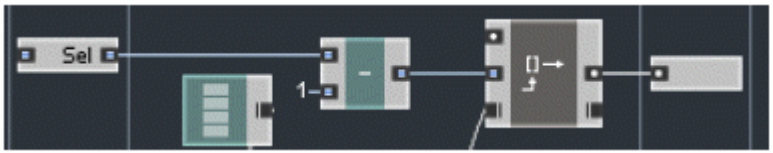
The four Write [] modules will take care of storing the incoming audio values into the array. We now need some circuitry to read one of the 4 values. We suggest using Read [] macro (Expert Macro > Memory > Read []):



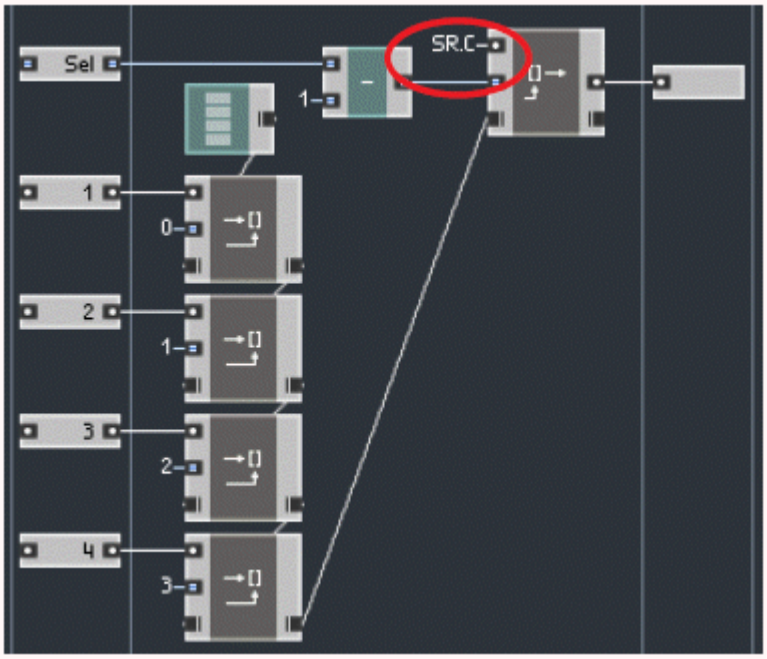
This macro reads from an array element whose index is specified at the integer input in the middle. The top input is the clock input for the read operation – it will send the read value to the upper output of the module in response to an incoming event. The ports at the bottom are, of course, the master 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.



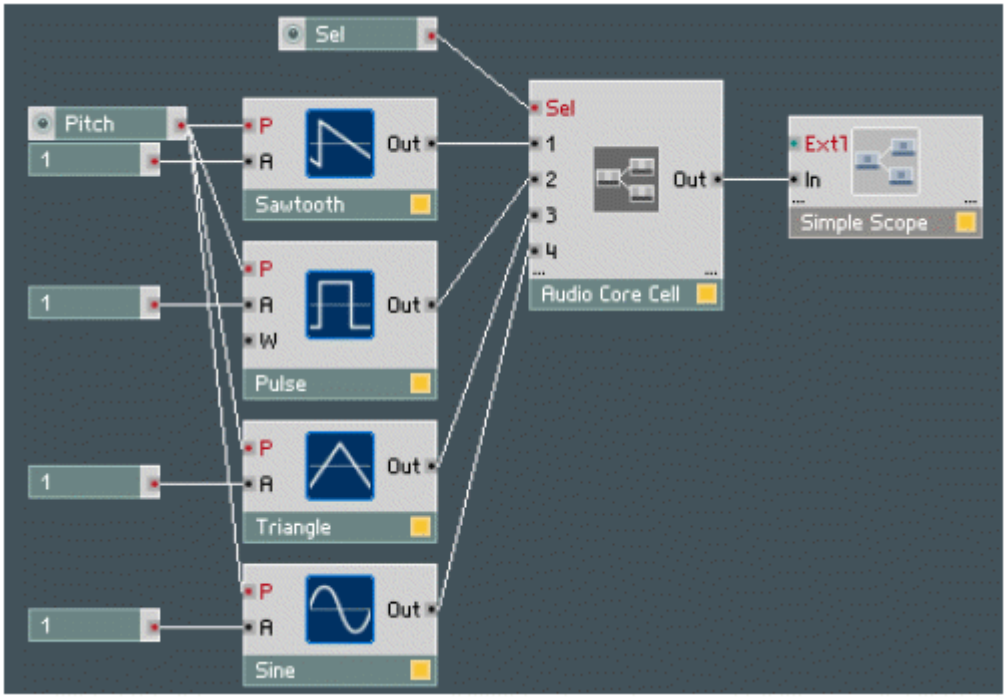
Now, what do we connect to the index input of the Read [] module? Because we want our selection value to be in the range 1 to 4, we need to subtract 1 from the Sel input value. Notice that we perform integer subtraction (and because Sel is just a control input we don't really need a modulation macro here):



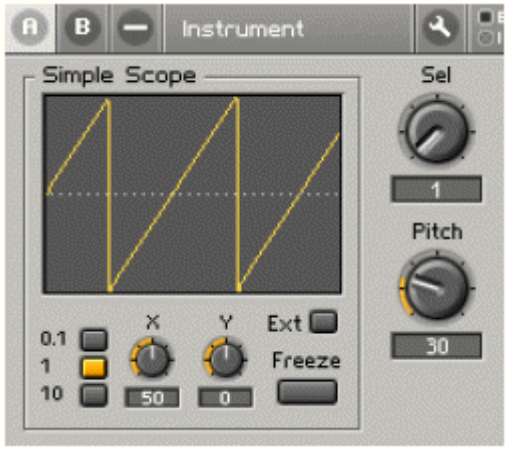
The last step is to clock the read module by the sampling-rate clock (because we are building an audio selector):



In general, you should also take care to clip the Sel input value to the correct range, but for simplicity, we did not do that here.
Here is the suggested test structure (the macro has been put into an audio core cell):



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:

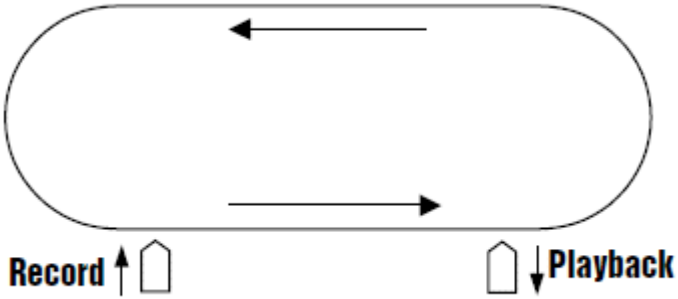


8.3 Building a Delay

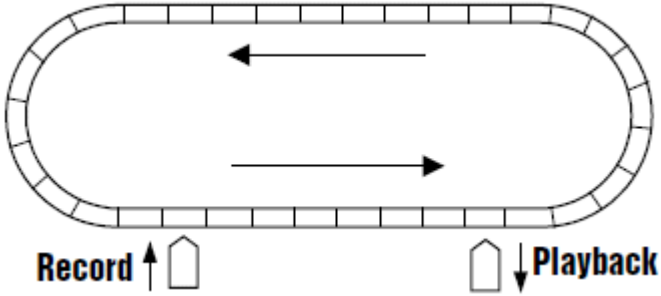
Now that we have some experience with arrays, let's build a simple audio-delay macro. The module will look like this:

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):

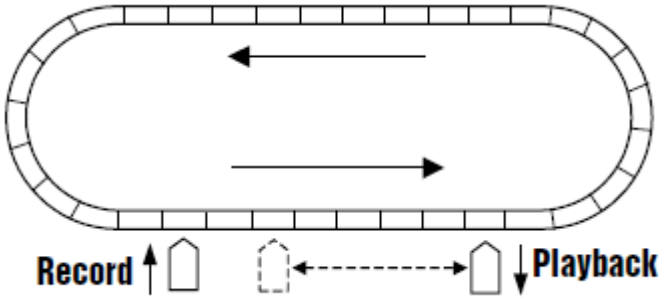
The T input expects the delay time in seconds.
If you take a look at an analog tape delay device, you'll see a tape loop combined with record and playback heads. Strictly speaking there's also an erase head, but for simplicity we can imagine that the record head does both the jobs of erasing and recording.



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:

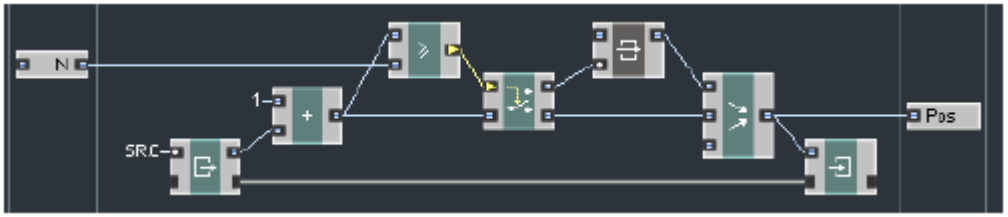
- array – to simulate our digital tape loop
- write index – this is our record head
- read index – this is our playback head

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.

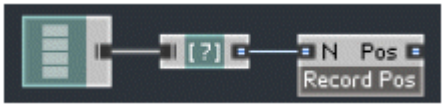
! This technique is quite common in programming and is called "circular buffer" or "ring buffer".

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:

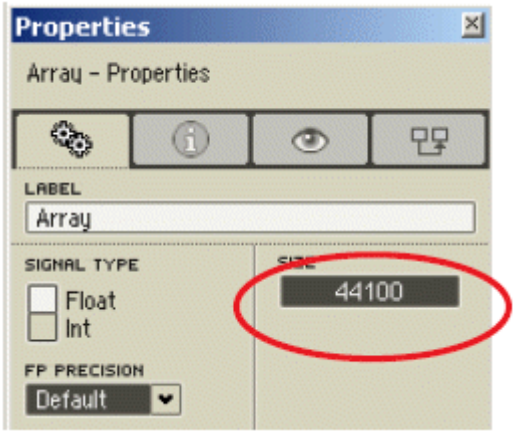
The N input should receive the number of elements in the array and the Pos output will carry the current writing position (index). Here is how this macro can be implemented (compare this to the sawtooth oscillator implementation):



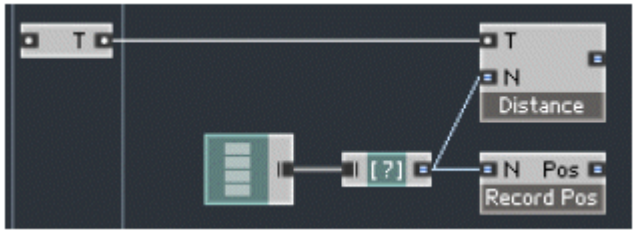
Note that the comparison module is set to >=. That was unimportant for the sawtooth oscillator, we could use >= or > there, but in integer computations the difference is in most cases critical. Using >= condition ensures that the write index will never reach a value of N (which would be out of range).
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:



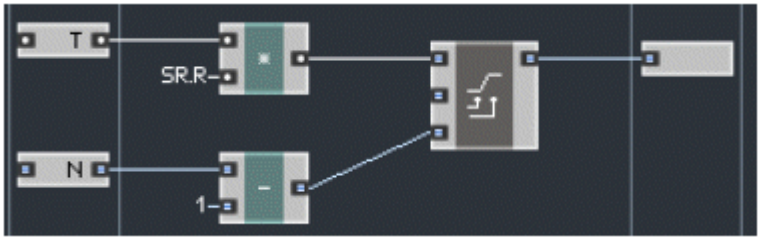
The size property of the array can be set to 44,100. This will allow us as much as 1 second of delay (actually one sample less) at 44.1 kHz sampling rate:



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:



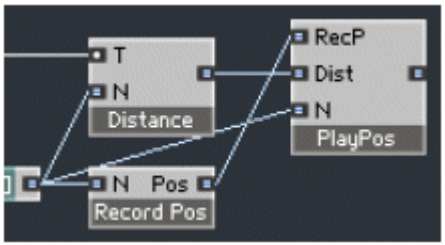
That can be done by multiplying the time in seconds by the sampling rate in Hz. We also should not forget to clip the result, a clipping macro Expert Macro > Clipping >IClipMin-Max should be good for that:



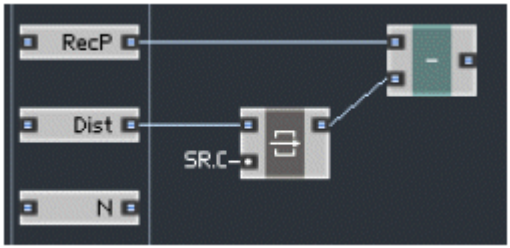
We clip to N-1 because that's the maximum distance between two different array elements. Note the conversion to integers, which is done after the multiplication.

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:

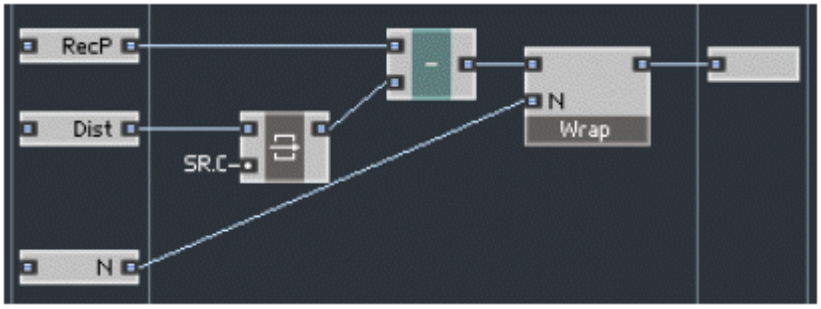


The distance value is latched because it is produced by a control signal input, which potentially can receive events at any time, and we do not want the subtraction happening at times other than at audio-clock events.

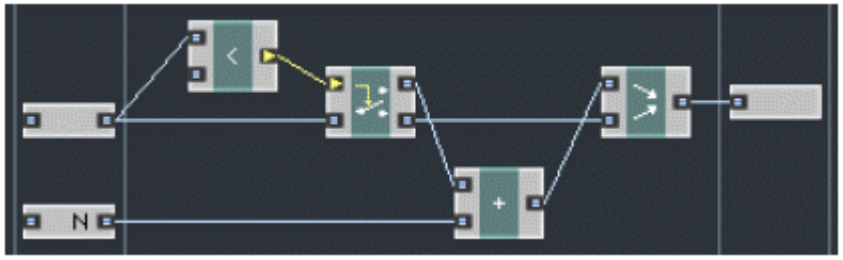
If we just subtract, the difference can turn out to be less than zero because our array is not a loop; its ends are not connected together. So we need to wrap the result:

- -1 must become N-1,
- -2 must become N-2,
- -3 must become N-3,
- and so on.

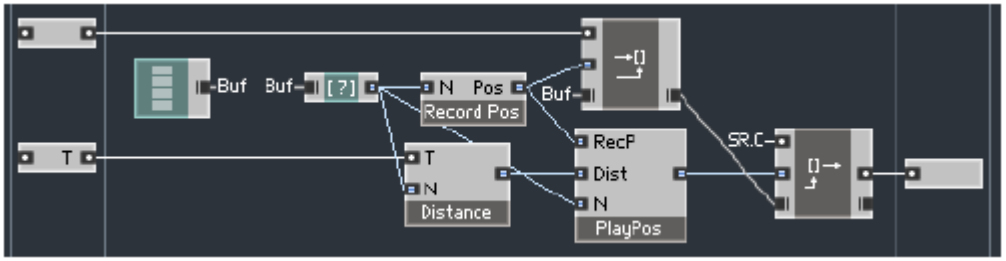
So we include another macro for wrapping:



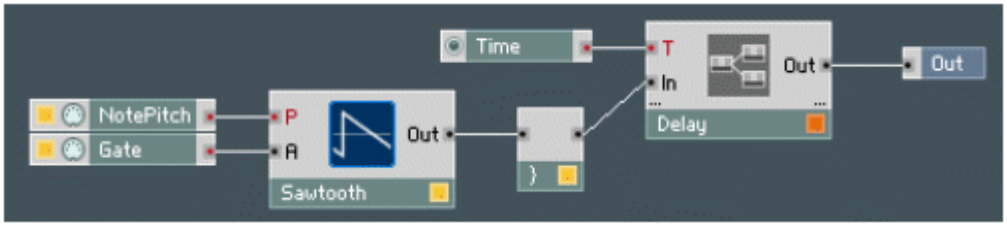
Because we know that the difference cannot be smaller than –N+1 (because RecordPos is between 0 and N-1 and Distance is between 0 and N-1), wrapping can be implemented as simple addition of N:



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:

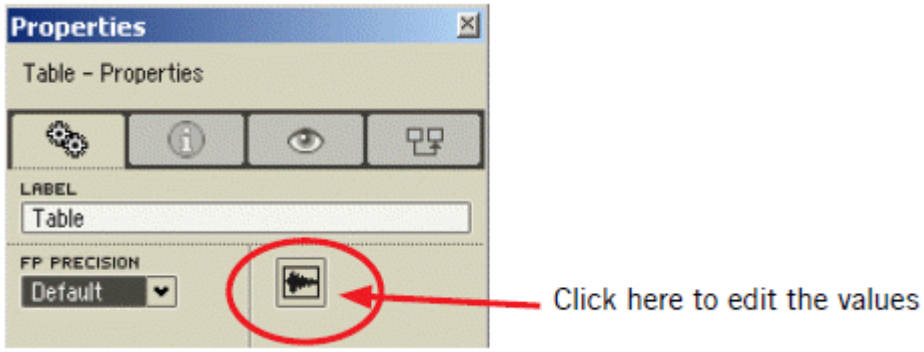
- 44,100*4 = 176,400 bytes, which is a little bit more than 172K (a kilobyte has 1024 bytes).

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.

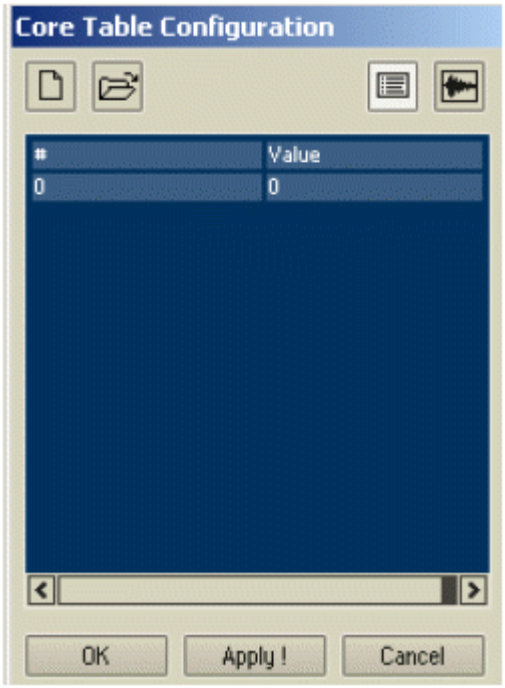
8.4 Tables

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:

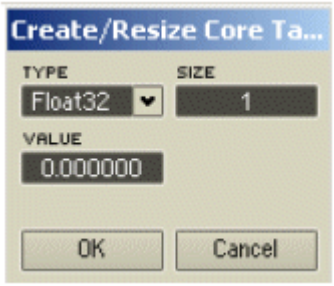


A new dialog will appear:



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 Open File 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:

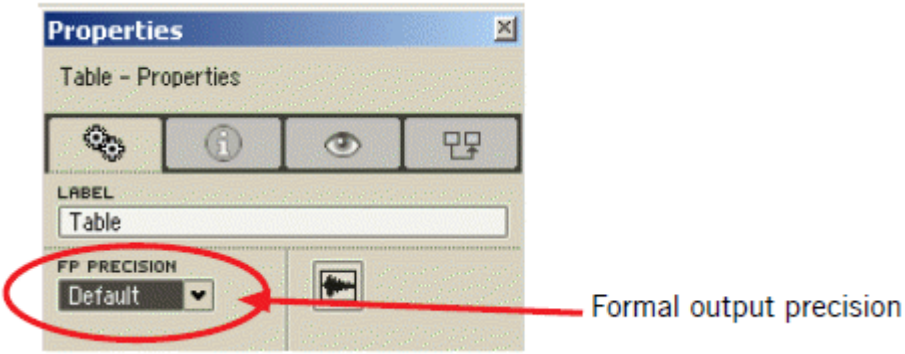
Inside this macro we are going to create a table module:

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:

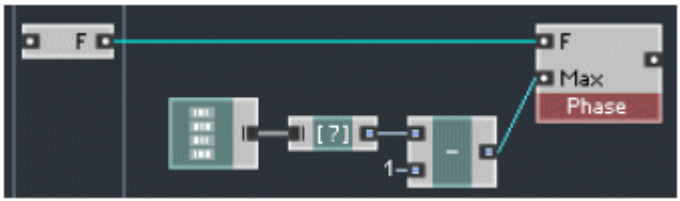
You can also view the loaded values as a waveform display. The List and Waveform buttons switch between the list and waveform view respectively.

Press OK to close the dialog and commit the loaded values to the table.

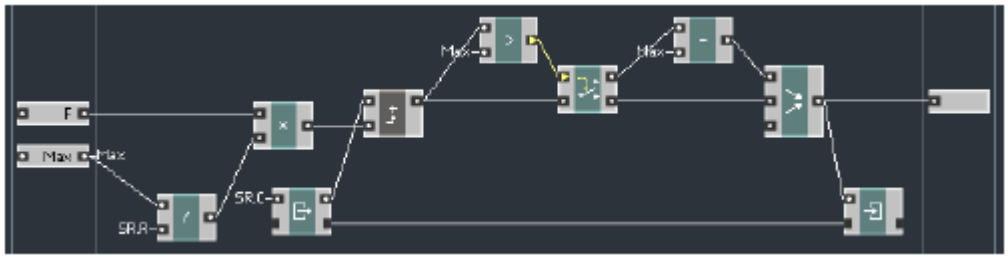
There's also an FP Precision property setting in the properties window for the table. It doesn't really control the precision of the values in the table (that you should have selected when importing the file or manually creating a list of values), but rather it sets the formal precision type of the table module's output. Generally, you would keep it set to default:



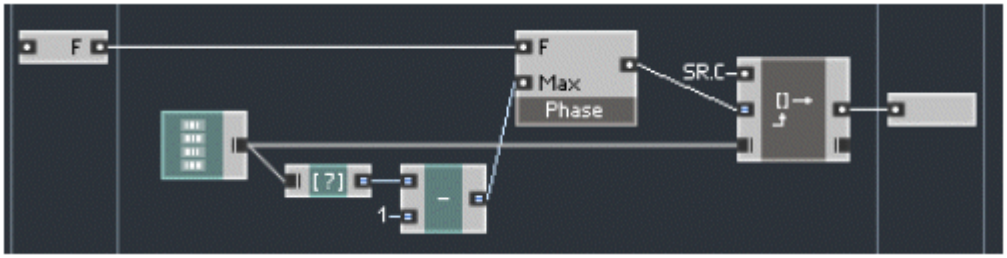
Now that we have the table, we can continue building the oscillator. At its core there will be a phase oscillator generating a rising sawtooth ramp signal from 0 to the size of the table minus one:



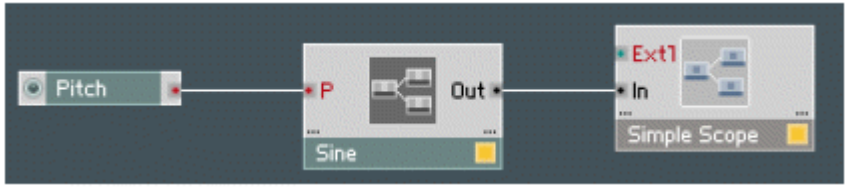
The phase oscillator implementation is similar to the sawtooth oscillator and to the recording position in the delay:



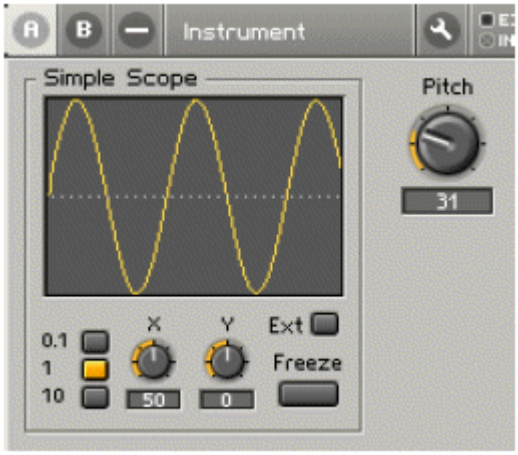
A Read [] module connected to the phase oscillator and clocked by the sampling-rate clock will access the corresponding table element and output its value.



Here's the suggested test structure (don't forget a P2F converter in the core cell):



An this is the panel view:



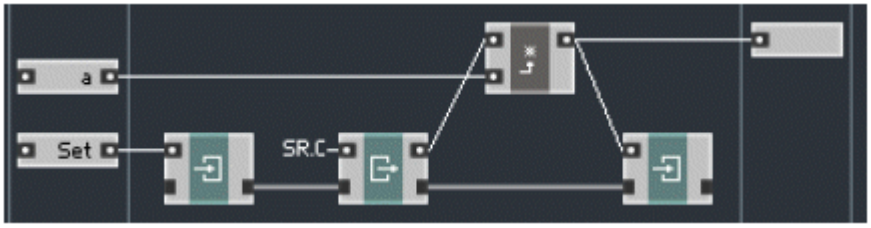
Of course, this is not a very clean sounding sine because we don't have any interpolation in there. We leave it up to you to build an interpolating version if you wish.

9 Building Optimal Structures

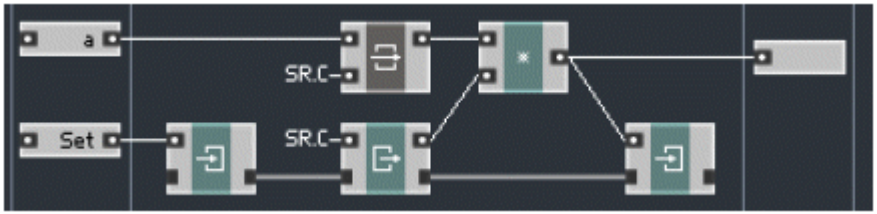
As a rule, no tool is ideal. The Reaktor Core technology is no exception. Although this technology is quite powerful on its own, you do need to know a few things to get the most out of it. So here are some essential tips and tricks to get you going.

9.1 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:

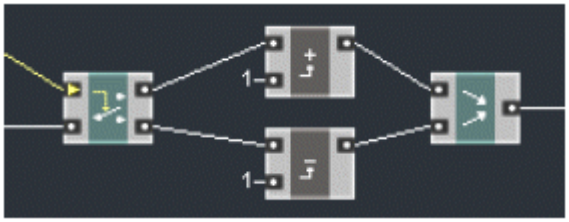


There were multiple examples of this technique throughout this tutorial.
Using latches has to do both with performance optimization and the correctness of your structures. Some typical mistakes in structure programming have to do with sending events to certain modules at improper times.
Don't be afraid 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.

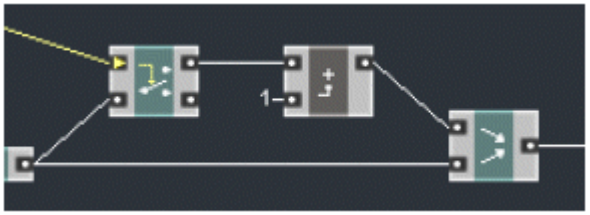
Latches are generally preferable to routing for event filtering because of their lower CPU cost. Try to use routers only where the processing logic dictates routing.

9.2 Routing and Merging

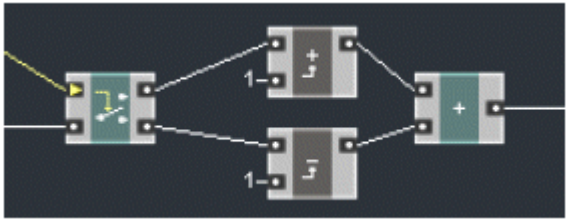
Routing can be more or less CPU intensive depending on the situation and the platform. If you can avoid routing without adding other CPU-intensive operations to your structure, do so.
Sometimes ES Ctl routing can be replaced by using Latches. If possible, do so.
If you split the event path into two branches using a Router, it's a good idea to merge the branches generated by the outputs of the Router:



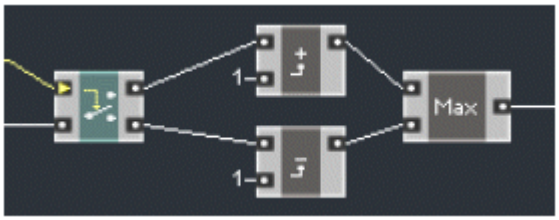
It's also a good idea to merge to the incoming (unsplit) event to the Router:



Merging is not necessarily done by using a Merge module. Any arithmetic or a similar module will do the job:



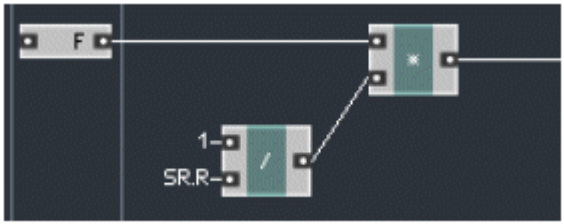
Merging can also happen inside a macro (depending on its internal structure):



It may be reasonable or necessary to merge the branches generated by different Routers, but beware of higher CPU loads in that case.

9.3 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:

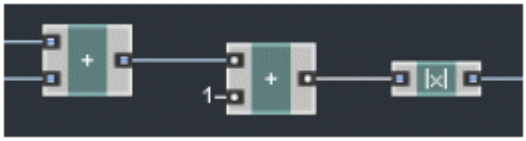


In the above structure, the division will be performed only when the sample rate changes, which should be pretty rare. Changes to the frequency will trigger only multiplication.
Compare that to the more straightforward implementation of the same formula:

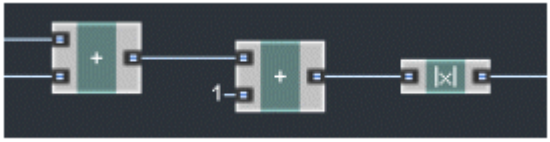
where the division would be executed in response to every change of frequency.

9.4 Conversions Between Floats and Integers

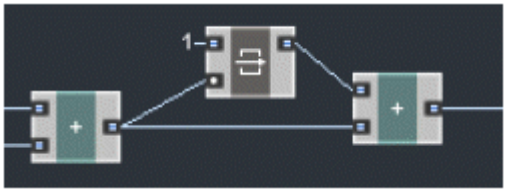
Generally, avoid all unnecessary conversions between float and integer numbers. Depending on the platform such conversions could use significant amounts of CPU. The conversions that are necessary to do are OK, of course.
Although the following structure might work as expected, in fact, there are two 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.

10 Appendix A. Reaktor Core User Interface

10.1 A.1. Core Cells

A core cell is 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.

10.2 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.

10.3 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.

10.4 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.

11 Appendix B. Reaktor Core Concept

11.1 B.1. Signals and Events

There are signals of float and integer types.
Float ports look like this:

An 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.

11.2 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.

11.3 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.

11.4 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).

11.5 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.

11.6 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.

12 Appendix C. Core Macro Ports

12.1 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.

12.2 C.2. Out

Accepts an incoming event at the inside input and forwards it unmodified to the outside.

12.3 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.

12.4 C.4. Latch (output)

Forwards an OBC connection from the inside of the macro to the outside of the macro.

12.5 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.

12.6 C.6. Bool C (output)

Forwards a BoolCtl connection from the inside of the macro to the outside of the macro.

13 Appendix D. Core Cell Ports

13.1 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

13.2 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.

13.3 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

13.4 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.

14 Appendix E. Built-in Busses

14.1 E.1. SR.C

Sends regular clock events at the sampling rate
INITIALIZATION EVENT: always sends an initialization event

14.2 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

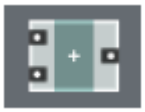
15 Appendix F. Built-in Modules

15.1 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.

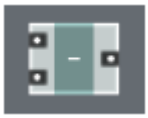
PROPERTIES
Value: the value to be sent to the output

15.2 F.2. Math > +



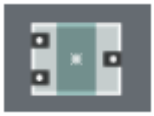
Produces the sum of the incoming signals at the output. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

15.3 F.3. Math > -



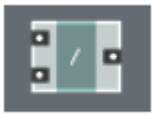
Produces the difference of the incoming signals at the output (the signal at the lower input is subtracted from the signal at the upper input). The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

15.4 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.

15.5 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.

15.6 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.

15.7 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.

15.8 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.

15.9 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

- **Base:** the logarithm base
- **Precision:** the approximation precision (better precisions require more CPU)

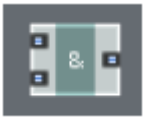
15.10 F.10. Math > ~exp

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

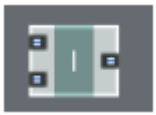
- **Base:** the exponent base
- **Precision:** the approximation precision (better precision requires more CPU)

15.11 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.

15.12 F.12. Bit > Bit OR

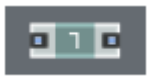


Performs the bitwise disjunction of the incoming signals. Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

15.13 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.

15.14 F.14. Bit > Bit NOT

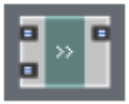


Performs the bitwise inversion of the incoming signal. Works only on integers. The output event is sent each time there is an event at the input.

15.15 F.15. Bit > Bit <<

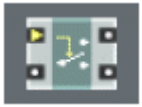
Bit-shifts the value at the upper input to the left (towards more significant bits). The amount of bits to shift by is specified by the lower input. The result for $N < 0$ and $N > 31$ is undefined (use it only for $0 \leq N \leq 31$). Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

15.16 F.16. Bit > Bit >>



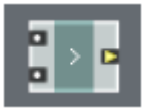
Bit-shifts the value at the upper input to the right (towards less significant bits). No sign extension is performed. The amount of bits to shift by is specified by the lower input. The result for $N < 0$ and $N > 31$ is undefined (use it only for $0 \leq N \leq 31$). Works only on integers. The output event is sent each time there is an event at either of the inputs or at both of them simultaneously.

15.17 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.

15.18 F.18. Flow > Compare

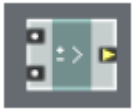


Produces a BoolCtl signal at the output indicating the result of comparison of the input values. The value at the upper input is placed to the left of the comparison sign and the value at the lower input to the right (so that the module on the picture above checks if upper value is greater than the lower one).

PROPERTIES

- **Criterion** the comparison operation to be performed

15.19 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

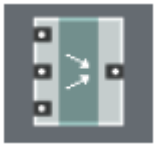
15.20 F.20. Flow > ES Ctl

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).

15.21 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).

15.22 F.22. Flow > Merge

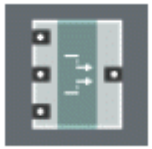


An output event is sent each time there is an event at any of the inputs or at several of them simultaneously. If only one input receives the event at a given time, the value of the output event will be equal to the value of the input event. If several inputs receive an event simultaneously, the value 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

15.23 F.23. Flow > EvtMerge

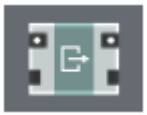


The functionality is similar to that of the Merge module, except that all input values are ignored. The value of the output event is undefined. This module is intended to be used for generating signals to be used as clocks. Works only in floating point mode, since the value is not meant to be used anyway.

PROPERTIES

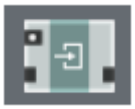
- **Input Count** number of inputs of the module

15.24 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.

15.25 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.

15.26 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)

15.27 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

15.28 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.

15.29 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.

15.30 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

 edit the values in the table

- **FP Precision** controls the formal precision of the output connection

15.31 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** The Open File button loads a new icon for the macro, the File button clears the icon (no icon assigned)

16 Appendix G. Expert Macros

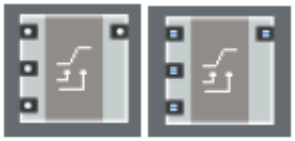
16.1 G.1. Clipping > Clip Max / IClip Max

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.

16.2 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.

16.3 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.

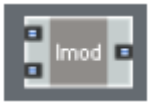
16.4 G.4. Math > 1 div x

Computes the reciprocal of the input value

16.5 G.5. Math > 1 wrap

Wraps the incoming value into the range [-0.5..0.5] (the wrapping period is 1).

16.6 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.

16.7 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.

16.8 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.

16.9 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.

16.10 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).

16.11 G.11. Math > sqrt (>0)

Square root approximation. Works only for inputs greater than 0.

16.12 G.12. Math > sqrt

Square root approximation (zero input is allowed).

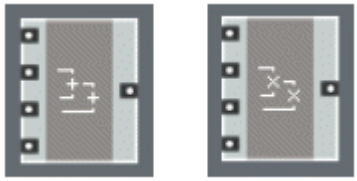
16.13 G.13. Math > x(>0)^y

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.

16.14 G.14. Math > x^2 / x^3 / x^4

Computes the 2nd/3rd/4th power of x.

16.15 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.

16.16 G.16. Math > Trig-Hyp > 2 pi wrap

Wraps the incoming value into the range [-π..π] (the wrapping period is 2π).

16.17 G.17. Math > Trig-Hyp > arcsin / arccos / arctan

Arcsine/arccosine/arctangent approximation.

16.18 G.18. Math > Trig-Hyp > sin / cos / tan

Sine/cosine/tangent approximation.

16.19 G.19. Math > Trig-Hyp > sin –pi..pi / cos –pi..pi / tan –pi..pi

Sine/cosine/tangent approximation (works only in the range [-π..π]).

16.20 G.20. Math > Trig-Hyp > tan –pi4..pi4

Tangent approximation (works only in the range [-π/4..π/4]).

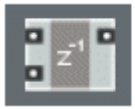
16.21 G.21. Math > Trig-Hyp > sinh / cosh / tanh

Hyperbolic sine/cosine/tangent approximation.

16.22 G.22. Memory > Latch / ILatch

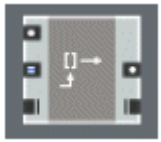
Latches (delays) the signal at the upper input until a clock event arrives at the lower input. If both events arrive simultaneously, the incoming signal will be let through immediately.

16.23 G.23. Memory > z^-1 / z^-1 ndc



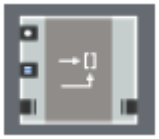
Sends out the last value that has been received at the upper input before a clock event arrives at the lower input in response to that clock event. If the clock input is disconnected the module will use the standard audio clock (SR.C) instead and effectively work as a one sample delay.
Both modules can automatically resolve feedback loops, however only z^-1 version provides denormal cancellation. The z^-1 ndc version is meant to be used only in the places where denormals are not expected.

16.24 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!

16.25 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!

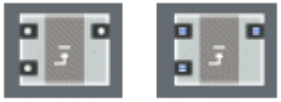
16.26 G.26. Modulation > x + a / Integer > Ix + a

Adds a parameter (lower input) to the signal (upper input) in response to an incoming signal event. Parameter changes do not generate events.

16.27 G.27. Modulation > x * a / Integer > Ix * a

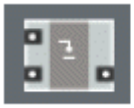
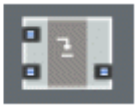
Multiplies the signal (upper input) by a parameter (lower input) in response to an incoming signal event. Parameter changes do not generate events.

16.28 G.28. Modulation > x – a / Integer > Ix – a



Subtracts a parameter (lower input) from the signal (upper input) in response to an incoming signal event. Parameter changes do not generate events.

16.29 G.29. Modulation > a – x / Integer > Ia – x

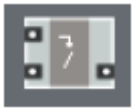



Subtracts the signal (lower input) from a parameter (upper input) in response to an incoming signal event. Parameter changes do not generate events.

16.30 G.30. Modulation > x / a

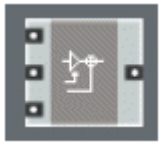
Divides the signal (upper input) by a parameter (lower input) in response to an incoming signal event. Parameter changes do not generate events.

16.31 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.

16.32 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.

17 Appendix H. Standard Macros

17.1 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

17.2 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.

17.3 H.3. Audio Mix-Amp > Audio Mix

Mixes two audio signals together.

17.4 H.4. Audio Mix-Amp > Audio Relay

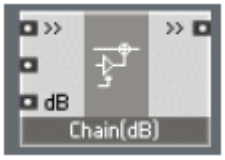
Switches between two input audio signals. If 'x' is greater than 0, picks up signal 1, otherwise signal 0.

17.5 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

17.6 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

17.7 H.7. Audio Mix-Amp > Gain (dB)

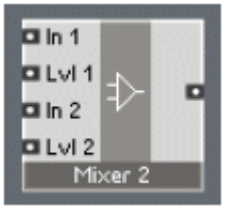
Changes the audio signal's amplitude by a given amount in dB.

- +6 dB doubles the amplitude
- 0 dB no change
- -6 dB halves the amplitude
- Typical usage: signal volume control in dB scale

17.8 H.8. Audio Mix-Amp > Invert

Inverts the polarity of the audio signal

17.9 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, ...).

17.10 H.10. Audio Mix-Amp > Pan

Pans the incoming audio signal using a parabolic curve.

- -1 hard left
- 0 center
- 1 hard right

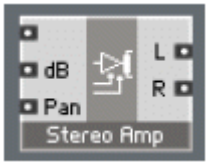
17.11 H.11. Audio Mix-Amp > Ring-Amp Mod

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)

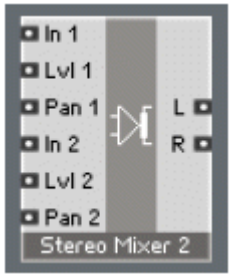
17.12 H.12. Audio Mix-Amp > Stereo Amp



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

17.13 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:

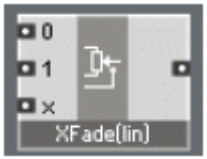
- -1 hard left
- 0 center
- 1 hard right

17.14 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.

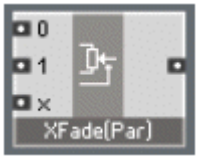
17.15 H.15. Audio Mix-Amp > XFade (lin)



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.

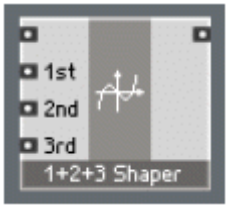
17.16 H.16. Audio Mix-Amp > XFade (par)



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

17.17 H.17. Audio Shaper > 1+2+3 Shaper



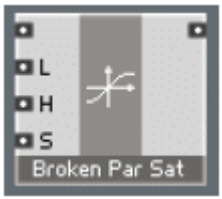
Provides audio-signal controllable shaping of 2nd and 3rd order. The 1st input specifies the amount of the original signal in the output (1=unchanged, 0=none). The 2nd and 3rd inputs specify the amounts of the 2nd and 3rd order distortion. respectively.

17.18 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

17.19 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.

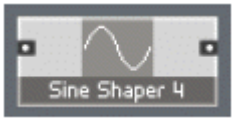
17.20 H.20. Audio Shaper > Hyperbol Sat

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.

17.21 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.

17.22 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.

17.23 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

17.24 H.24. Control > Ctl Amp Mod

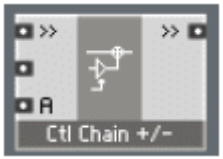
Modulates the control signal's amplitude by a given amount (AM) in linear scale.

- AM = 1 doubles the amplitude
- AM = 0 no change
- AM = -1 mutes the signal

17.25 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.

17.26 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

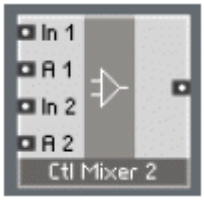
17.27 H.27. Control > Ctl Invert

Inverts the control signal's polarity

17.28 H.28. Control > Ctl Mix

Mixes two control signals.

17.29 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

17.30 H.30. Control > Ctl Pan

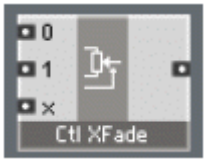
“Pans” a control signal using a parabolic curve.

- Pos = -1 hard left
- Pos = 0 center
- Pos = 1 hard right

17.31 H.31. Control > Ctl Relay

Switches between two control signals. If x > 0 picks up signal 1, else picks up signal 0.

17.32 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

17.33 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

17.34 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.

17.35 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.

17.36 H.36. Convert > logT2sec

Converts Reaktor primary-level logarithmic time (used for envelopes) to seconds.

- 0 translates to 0.001 sec
- 60 translates to 1 sec
- etc.

17.37 H.37. Convert > ms2Hz

Converts time period in milliseconds into corresponding frequency in Hz.
E.g. 100ms translates to 10 Hz.

17.38 H.38. Convert > ms2sec

Converts the time specified in milliseconds into time specified in seconds. E.g. 500ms translates to 0.5 sec.

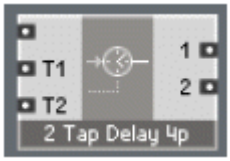
17.39 H.39. Convert > P2F

Converts a control signal from pitch scale to frequency scale.
E.g. pitch 69 translates to 440 Hz.

17.40 H.40. Convert > sec2Hz

Converts time period in seconds into corresponding frequency in Hz.
E.g. 0.1sec translates to 10 Hz.

17.41 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.

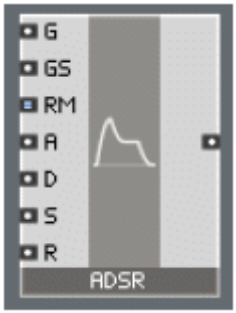
17.42 H.42. Delay > Delay 1p / 2p / 4p

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.

17.43 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.

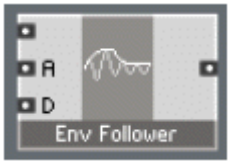
17.44 H.44. Envelope > ADSR



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

17.45 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.

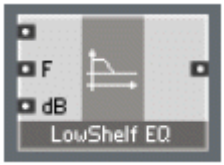
17.46 H.46. Envelope > Peak Detector

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.

17.47 H.47. EQ > 6dB LP/HP EQ

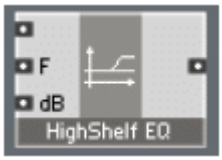
1-pole (6dB/octave) lowpass/highpass EQ. The F input specifies the cutoff frequency (in Hz) for both LP and HP outputs.

17.48 H.48. EQ > 6dB LowShelf EQ



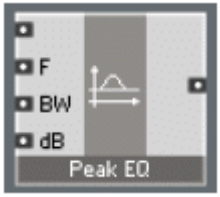
1-pole low-shelving EQ. The dB input specifies the low frequency boost in dB (negative values will cut the frequencies), the F input specifies the transition mid-frequency in Hz.

17.49 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.

17.50 H.50. EQ > Peak EQ

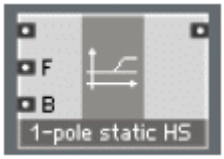


2-pole peak/notch EQ. The F input specifies the center frequency in Hz, the BW input specifies the EQ bandwidth in octaves and dB input specifies the peak height (negative values produce a notch).

17.51 H.51. EQ > Static Filter > 1-pole static HP

1-pole static highpass filter. The F input specifies the cutoff frequency in Hz.

17.52 H.52. EQ > Static Filter > 1-pole static HS

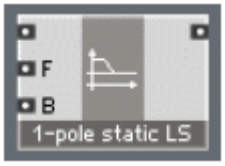


1-pole static high-shelving filter. The F input specifies the cutoff frequency in Hz and the B input specifies the high frequency boost in dB.

17.53 H.53. EQ > Static Filter > 1-pole static LP

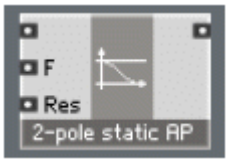
1-pole static lowpass filter. The F input specifies the cutoff frequency in Hz.

17.54 H.54. EQ > Static Filter > 1-pole static LS



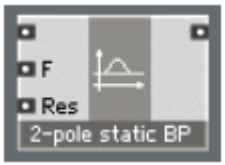
1-pole static low-shelving filter. The F input specifies the cutoff frequency in Hz and the B input specifies the low frequency boost in dB.

17.55 H.55. EQ > Static Filter > 2-pole static AP



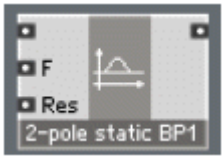
2-pole static allpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

17.56 H.56. EQ > Static Filter > 2-pole static BP



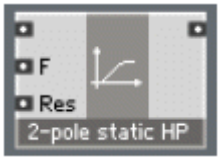
2-pole static bandpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

17.57 H.57. EQ > Static Filter > 2-pole static BP1



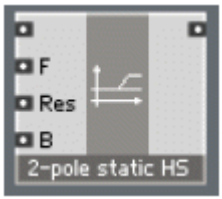
2-pole static bandpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1). The amplification at cutoff frequency is always 1 regardless of the resonance.

17.58 H.58. EQ > Static Filter > 2-pole static HP



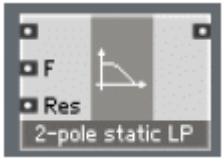
2-pole static highpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

17.59 H.59. EQ > Static Filter > 2-pole static HS



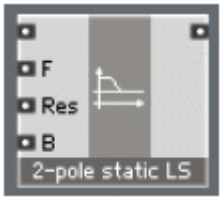
2-pole static high-shelving filter. The F input specifies the cutoff frequency in Hz, the Res input specifies the resonance (0..1), and the B input specifies the high frequency boost in dB.

17.60 H.60. EQ > Static Filter > 2-pole static LP



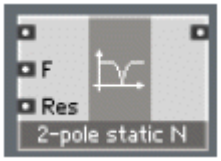
2-pole static lowpass filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

17.61 H.61. EQ > Static Filter > 2-pole static LS



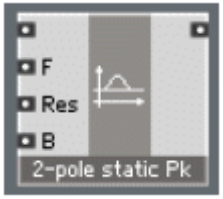
2-pole static low-shelving filter. The F input specifies the cutoff frequency in Hz, the Res input specifies the resonance (0..1), and the B input specifies the low frequency boost in dB.

17.62 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).

17.63 H.63. EQ > Static Filter > 2-pole static Pk

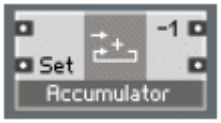


2-pole static peak filter. The F input specifies the cutoff frequency in Hz, the Res input specifies the resonance (0..1), and the B input specifies the center frequency boost in dB.

17.64 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.

17.65 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.

17.66 H.66. Event Processing > Clk Div

Clock frequency divider. The clock events arriving at the upper input will be filtered, allowing only 1st, N+1th, 2N+1th etc. events to come through (N is the value at the lower input and specifies the division ratio).

17.67 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.

17.68 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.

17.69 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.

17.70 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.

17.71 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).

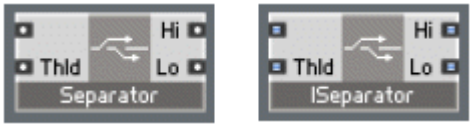
17.72 H.72. Event Processing > Impulse

Generates a one sample impulse of amplitude 1 in response to an incoming event. This module works only inside audio core cells.

17.73 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.

17.74 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.

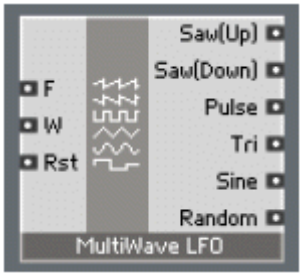
17.75 H.75. Event Processing > Thld Crossing

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.

17.76 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.

17.77 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).

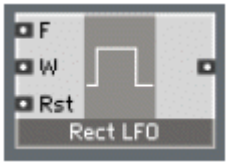
17.78 H.78. LFO > Par LFO

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).

17.79 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).

17.80 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).

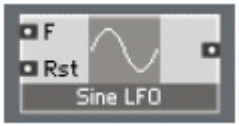
17.81 H.81. LFO > Saw(down) LFO

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).

17.82 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).

17.83 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).

17.84 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).

17.85 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.

17.86 H.86. Logic > Flip Flop

The output is flipped between 0 and 1 each time the clock input receives an event.

17.87 H.87. Logic > Gate2L

Converts gate signal to logic signal. Open gate produces output value 1, closed gate produces output value 0.

17.88 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.

17.89 H.89. Logic > EQ

Compares the two incoming integer values and outputs 1 if both values are equal, otherwise outputs 0.

17.90 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.

17.91 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.

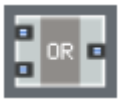
17.92 H.92. Logic > L2Gate

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.

17.93 H.93. Logic > NOT

Converts 1 to 0 and vice versa. For input values other than 0 or 1 the result is undefined.

17.94 H.94. Logic > OR

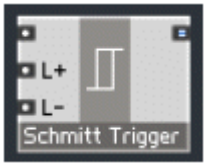


Performs a disjunction of two logical signals (the output is 1 if at least one of the inputs is 1). For input values other than 0 or 1 the result is undefined.

17.95 H.95. Logic > XOR

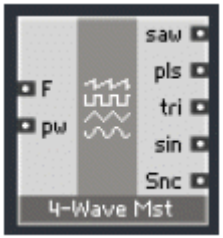
Performs an exclusive disjunction of two logical signals (the output is 1 if one of the inputs is equal to 1 and the other equal to 0). For input values other than 0 or 1 the result is undefined.

17.96 H.96. Logic > Schmitt Trigger



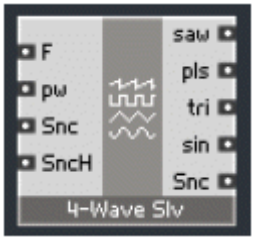
Switches the output to 1 if the input value becomes larger than L+ (default 0.67), switches the output to 0 if the input value becomes less than L- (default 0.33).

17.97 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 Slv oscillator.

17.98 H.98. Oscillators > 4-Wave Slv



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/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.

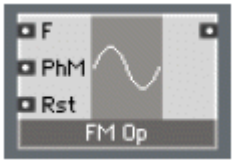
17.99 H.99. Oscillators > Binary Noise

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.

17.100 H.100. Oscillators > Digital Noise

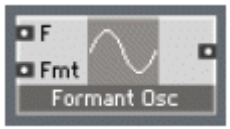
Digital white noise generator. Outputs random values in the range –1..1 An incoming event at the Seed input would (re-)initialize the internal random generator with a given seed value.

17.101 H.101. Oscillators > FM Op



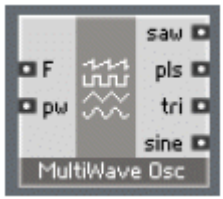
Classical FM operator. Outputs a sine wave whose frequency is defined by the F input (in Hz). The sine can be phase-modulated by the PhM input (in radians). An incoming event at the Rst input would restart the oscillator to the phase specified by the value of this event (range 0..1).

17.102 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).

17.103 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.

17.104 H.104. Oscillators > Par Osc

Generates a parabolic audio waveform. The F input specifies the frequency in Hz.

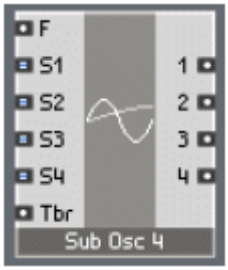
17.105 H.105. Oscillators > Quad Osc

Generates a pair of phase-locked sine waveforms with a phase shift of 90 degrees. The F input specifies the frequency in Hz.

17.106 H.106. Oscillators > Sin Osc

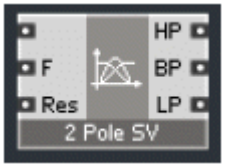
Generates a sine wave. The F input specifies the frequency in Hz.

17.107 H.107. Oscillators > Sub Osc 4



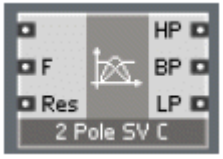
Generates 4 phase-locked subharmonics. The fundamental frequency is specified by the F input (in Hz). The subharmonic numbers are specified by S1..S4 inputs (range 1..120). The Tbr input controls the harmonic content of the output waveform (range 0..1).

17.108 H.108. VCF > 2 Pole SV



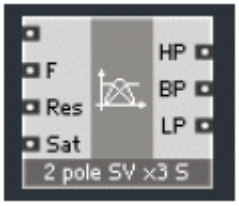
2-pole state-variable filter. The F input specifies the cutoff in Hz and the Res input specifies the resonance (range 0..0.98).
The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

17.109 H.109. VCF > 2 Pole SV C



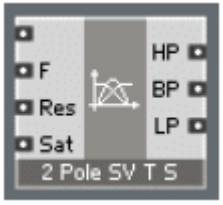
2-pole state-variable filter (compensated version). Offers an improved behavior at high cutoff settings. The F input specifies the cutoff in Hz and the Res input specifies the resonance (range 0..0.98). You also can use negative resonance values which will smear the slope further.
The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

17.110 H.110. VCF > 2 Pole SV (x3) S



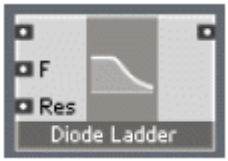
2-pole state-variable filter with optional oversampling (x3 version) and saturation. The F input specifies the cutoff in Hz, the Res input specifies the resonance (range 0..1), the Sat input specifies the saturation level (typical range 8..32).
The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

17.111 H.111. VCF > 2 Pole SV T (S)



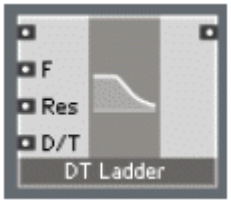
2-pole state-variable filter with table compensation and optional saturation (S version). Offers an improved behavior at high cutoff settings, but slightly different from the 2 Pole SV C version. The F input specifies the cutoff in Hz, the Res input specifies the resonance (range 0..1), the Sat input specifies the saturation level (typical range 8..32).
The HP/BP/LP outputs produce highpass, bandpass and lowpass signals respectively.

17.112 H.112. VCF > Diode Ladder



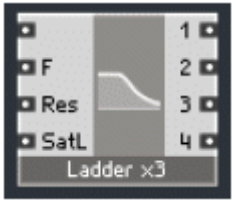
Diode-ladder filter linear emulation. The F input specifies the cutoff in Hz and the Res input specifies the resonance (range 0..0.98).

17.113 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).

17.114 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.

18 Appendix I. Core Cell Library

18.1 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

18.2 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.

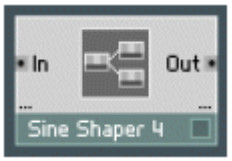
18.3 I.3. Audio Shaper > Hyperbol Sat

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.

18.4 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.

18.5 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.

18.6 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

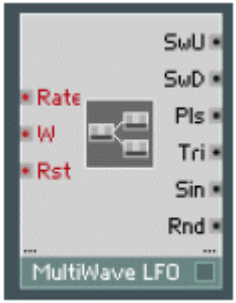
18.7 I.7. Control > Env Follower

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.

18.8 I.8. Control > Flip Flop

The output is flipped between 0 and 1 each time the trigger input receives an event.

18.9 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).

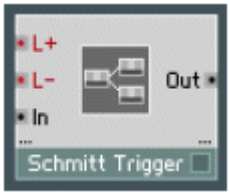
18.10 I.10. Control > Par Ctl Shaper

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

18.11 I.11. Control > Schmitt Trigger

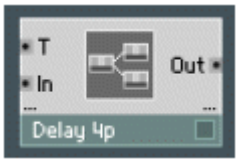


Switches the output to 1 if the input value becomes larger than L+ (default 0.67), switches the output to 0 if the input value becomes less than L- (default 0.33).

18.12 I.12. Control > Sine LFO

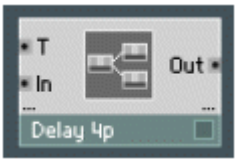
Generates a sine-shaped low-frequency control signal. The Rate input specifies the rate in Hz, the events at the Rst input restart the LFO at the phase specified by the event value (range 0..1).

18.13 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.

18.14 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.

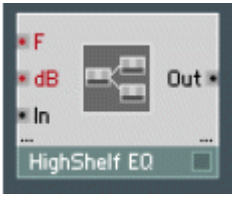
18.15 I.15. Delay > Diff Delay 4p

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.

18.16 I.16. EQ > 6dB LP/HP EQ

1-pole (6dB/octave) lowpass/highpass EQ. The F input specifies the cutoff frequency (in Hz).

18.17 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.

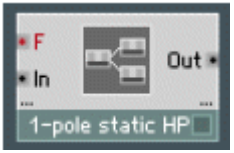
18.18 I.18. EQ > LowShelf EQ

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.

18.19 I.19. EQ > Peak EQ

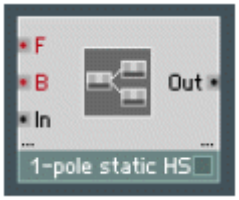
2-pole peak/notch EQ. The F input specifies the center frequency in Hz, the BW input specifies the EQ bandwidth in octaves and dB input specifies the peak height (negative values produce a notch).

18.20 I.20. EQ > Static Filter > 1-pole static HP



1-pole static highpass filter. The F input specifies the cutoff frequency in Hz.

18.21 I.21. EQ > Static Filter > 1-pole static HS

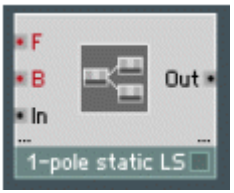


1-pole static high-shelving filter. The F input specifies the cutoff frequency in Hz and the B input specifies the high frequency boost in dB.

18.22 I.22. EQ > Static Filter > 1-pole static LP

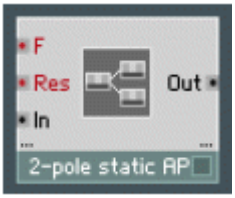
1-pole static lowpass filter. The F input specifies the cutoff frequency in Hz.

18.23 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.

18.24 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).

18.25 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).

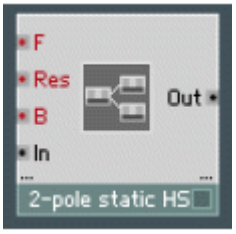
18.26 I.26. EQ > Static Filter > 2-pole static BP1

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.

18.27 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).

18.28 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.

18.29 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).

18.30 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.

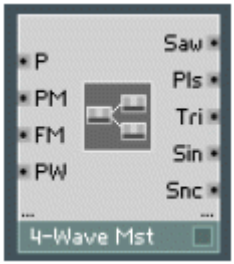
18.31 I.31. EQ > Static Filter > 2-pole static N

2-pole static notch filter. The F input specifies the cutoff frequency in Hz and the Res input specifies the resonance (0..1).

18.32 I.32. EQ > Static Filter > 2-pole static Pk

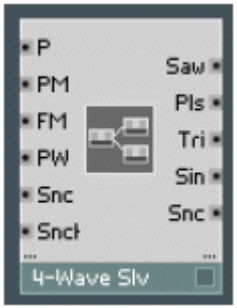
2-pole static peak filter. The F input specifies the cutoff frequency in Hz, the Res input specifies the resonance (0..1), and the B input specifies the center frequency boost in dB.

18.33 I.33. Oscillator > 4-Wave Mst



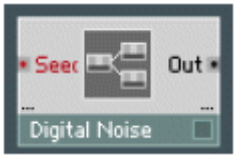
Generates 4 phase-locked audio waveforms. The oscillator pitch is specified by the P input (as a MIDI note number), can be modulated by the PM input (in semitones, exponential) and by the FM input (in Hz, linear). The pulse width is specified by the PW input (range –1..0..1, affects only the pulse waveform).
This oscillator can oscillate at negative frequencies and additionally offers a synchronization output for 4-Wave Slv oscillator.

18.34 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.

18.35 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.

18.36 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).

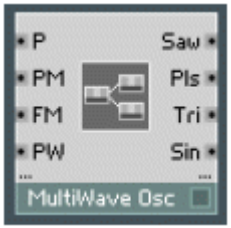
18.37 I.37. Oscillator > Formant Osc

Generates a waveform with a fundamental frequency specified by the the P input (as a MIDI note number) and the formant frequency specified by the Fmt input (in Hz).

18.38 I.38. Oscillator > Impulse

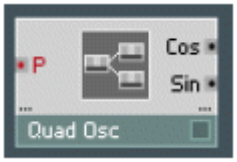
Generates a one-sample impulse of amplitude 1 in response to an incoming event.

18.39 I.39. Oscillator > MultiWave Osc



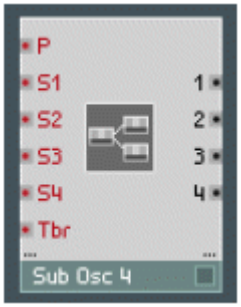
Generates 4 phase-locked audio waveforms. The oscillator pitch is specified by the P input (as a MIDI note number), can be modulated by the PM input (in semitones, exponential) and by the FM input (in Hz, linear). The pulse width is specified by the PW input (range –1..0..1, affects only the pulse waveform).
This oscillator cannot oscillate at negative frequencies.

18.40 I.40. Oscillator > Quad Osc



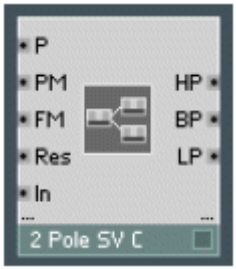
Generates a pair of phase-locked sine waveforms with a phase shift of 90 degrees. The P input specifies the pitch (as a MIDI note number).

18.41 I.41. Oscillator > Sub Osc



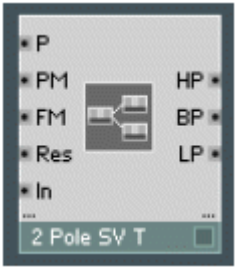
Generates 4 phase-locked subharmonics. The fundamental frequency is specified by the P input (as a MIDI note number). The subharmonic numbers are specified by S1..S4 inputs (range 1..120). The Tbr input controls the harmonic content of the output waveform (range 0..1).

18.42 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.

18.43 I.43. VCF > 2 Pole SV T

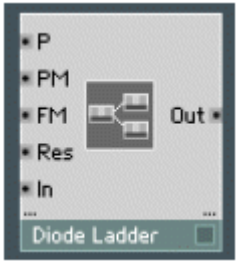


2-pole state-variable filter with table compensation. Offers improved behavior at high cutoff settings, but slightly different from the 2 Pole SV C version. The filter cutoff frequency is specified by the P input (as a MIDI note number), can be modulated by the PM input (in semitones, exponential) and 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.

18.44 I.44. VCF > 2 Pole SV x3 S

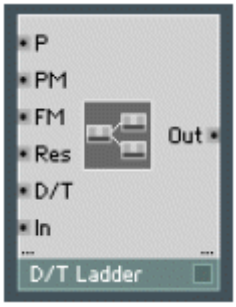
2-pole state-variable filter with optional oversampling (x3 version) and saturation. The filter cutoff frequency is specified by the P input (as a MIDI note number), can be modulated by the PM input (in semitones, exponential) and by the FM input (in Hz, linear). The Res input specifies the resonance (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.

18.45 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).

18.46 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).

18.47 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.