
BUILDING IN CORE

Document authored by: Vadim Zavalishin

Software version: 6.0.1 (11/2015)

**NATIVE INSTRUMENTS GmbH**
Schlesische Str. 29-30
D-10997 Berlin
Germany
www.native-instruments.de

**NATIVE INSTRUMENTS North America, Inc.**
6725 Sunset Boulevard
5th Floor
Los Angeles, CA 90028
USA
www.native-instruments.com

**NATIVE INSTRUMENTS K.K.**
YO Building 3F
Jingumae 6-7-15, Shibuya-ku,
Tokyo 150-0001
Japan
www.native-instruments.co.jp

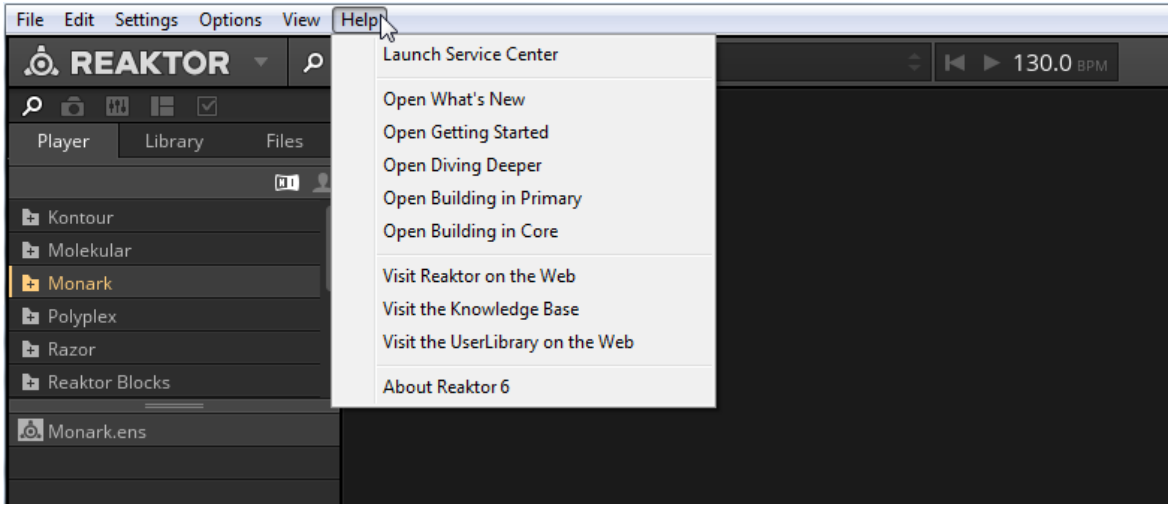
**NATIVE INSTRUMENTS UK Limited**
18 Phipp Street
London EC2A 4NU
UK
www.native-instruments.com

# Table of Contents

# 1 Welcome to Building in Core

Welcome to Building in Core, a manual describing the Core level in REAKTOR with its low-level building features. Using a run-time optimizing compiler, this graphical audio programming language can be used for implementing your own custom DSP algorithms. This document also includes reference information about the Core Macro Library, a comprehensive collection of DSP building blocks.

## 1.1 The REAKTOR 6 Documentation




The REAKTOR documentation is accessible from the *Help* menu

The documentation for REAKTOR 6 is divided into five separate documents, guiding you from loading and playing pre-built Ensembles to building your own Instruments.

- **REAKTOR 6 What Is New** is written for users who are already familiar with previous versions of REAKTOR and only describes the latest features in brief.

- **REAKTOR 6 Getting Started** is for new users. It is the only document needed for users who intend to use REAKTOR for loading and playing pre-built REAKTOR instruments and effects.
- **REAKTOR 6 Diving Deeper** expands on the concepts introduced in the Getting Started document. It provides more detail on subjects like Snapshots (REAKTOR's preset system), and introduces advanced topics like OSC control and combining multiple Instruments in one Ensemble.
- **REAKTOR 6 Building in Primary** shows you how to build your own Instruments in REAKTOR's Primary level. It focuses on a series of tutorials that guide you through building your first synthesizers, effects, and sequencers.
- **REAKTOR 6 Building in Core** describes the Core level of REAKTOR with its low-level building features, which can be used for implementing custom DSP algorithms. It includes reference information about the Core Macro Library, an comprehensive collection of DSP building blocks.

With the exception of the What Is New document, each of the documents listed above builds on the knowledge of the previous documents. While it is not necessary to read all of every document, some of the more advanced documents, like Building in Primary, assume knowledge from the previous documents.

## 1.2 Manual Conventions

This section introduces you to the signage and text highlighting used in this manual.

- Text appearing in (drop-down) menus (such as *Open...*, *Save as...* etc.) and paths to locations on your hard disk or other storage devices is printed in *italics*.
- Text appearing elsewhere (labels of buttons, controls, text next to checkboxes etc.) is printed in blue. Whenever you see this formatting applied, you will find the same text appearing somewhere on the screen.
- Important names and concepts are introduced in **bold**. Furthermore, **bold** text is used to stress important statements in the text.
- References to keys on your computer's keyboard you'll find put in square brackets (e.g., "Press [Shift] + [Enter]").

▶ Single instructions are introduced by this play button type arrow.

→ Results of actions are introduced by this smaller arrow.

An indented, gray paragraph contains additional, contextual information.

Furthermore, this manual uses particular formatting to point out special facts and to warn you of potential issues. The icons introducing these notes let you see what kind of information is to be expected:

The speech bubble icon indicates a useful tip that may often help you to solve a task more efficiently.

The exclamation mark icon highlights important information that is essential for the given context.

The red cross icon warns you of serious issues and potential risks that require your full attention.

## 1.3 Changes from REAKTOR 5

Here is a brief list of what has been changed in Core compared to REAKTOR 5.

- Two new major features, *Scoped Buses* (see section ↑3.3, Scoped Buses) and *Bundles* (see section ↑3.4, Bundles) have been introduced. Particularly the former SR.C and SR.R connections are now a part of the Scoped Buses/Bundles framework (see section ↑4.9, SR and CR Buses).
- Unification of audio and event Core Cells. Now there is only one Core Cell type. The input and output ports of the Core Cells can be switched between audio and event types (see sections ↑2.5.1, Inputs and ↑2.5.2, Outputs).
- SR.C does not send an initialization event anymore (see section ↑4.9.6, Internal Structure of Clock Buses).

  Upon importing the older Core Cells special adapter Macros are automatically inserted at the top level of the Structures to simulate the REAKTOR 5 behavior of SR.C.

- Nonsolid Macros and Structures now have a dedicated look (see section ↑2.6.3, Properties).
- The feedback loop highlighting now highlights the entire loop instead of just the resolution point (see section ↑4.10.1, Automatic Resolution).
- The Macro library has been completely restructured (see section ↑6, Macro Reference).
  - No more *Expert* / *Standard* parts.
  - New oscillators, filters, FX, LFOs, envelopes and smoothers.
  - Dedicated set of Macros for audio and control rate handling (see section ↑4.9, SR and CR Buses).
  - The former *Expert > Modulation* part is now found under *Math Mod*.
  - The logic signals have been slightly reworked (see section ↑6.1.9, Logic).
  - The *Flow* submenu contains a number of value-controlled routers.

- The *z^-1* Macros have been reworked. Now they are intended to be used exclusively for audio (see section ↑4.10.2, Manual Resolution). For other one-clock delay purposes use *Memory > Latch[-1]* or *Event Proc > Event[-1]* Macros.
- The *~exp* and *~log* Modules used inside the Macros (such as e.g. *Convert > P2F*) are typically set to the highest precision. Set them to a lower precision if the highest precision is not needed.

# 2 REAKTOR Core Fundamentals

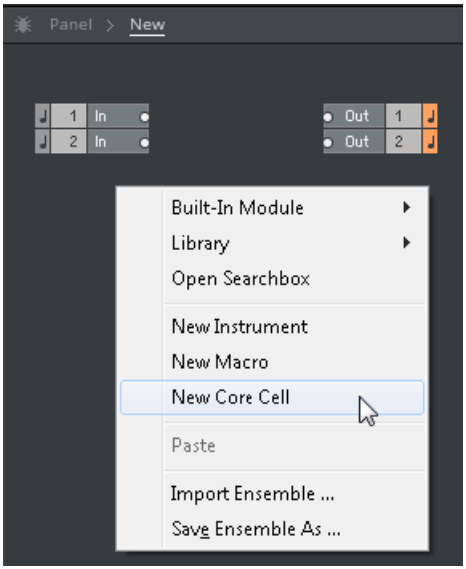
## 2.1 An Introduction to Core Cells

REAKTOR Core is a special level of functionality within REAKTOR with a distinct set of features. Similarly to the main part of REAKTOR (which we will refer to as the **Primary level** of REAKTOR), the **Core level** is also using a graphical signal flow paradigm. However, there are a number of important differences in the details between the Primary and the Core levels:

- REAKTOR Core is using an integrated run-time compiler, allowing efficient processing of highly detailed low-level Structures.
- The low-level Structure building features of REAKTOR Core make it specifically suitable for implementing custom DSP algorithms.
- The details of the rules of signal processing in REAKTOR Core are different from Primary level and are tailored to the needs of the DSP algorithm design.
- REAKTOR Core is using a different set of basic Modules and a different Macro library.

The REAKTOR Core Structures exist inside special wrapper objects called **Core Cells**. Core Cells can be considered as a special kind of Primary level Macros, where the difference is that inside Core Cells there is a different world of REAKTOR Core.

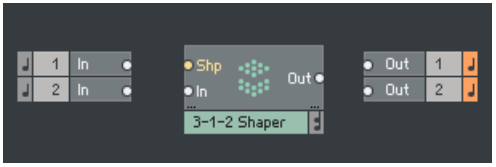
In order to create a new blank Core Cell right-click somewhere in the background of the REAKTOR Structure and select *New Core Cell* from the context menu:




Creating a new Core Cell from the context menu.

The prebuilt Core Cells from the factory and user libraries can be found in the *Library* menu among Primary Macros and Instruments.

Core Cells look like a special type of the Primary level Modules:

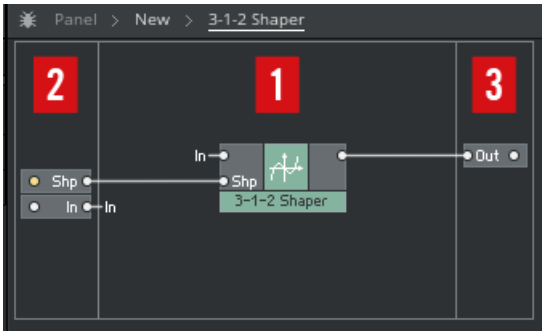



A Core Cell.

Similarly to Primary level Macros, Core Cells have internal Structures, which can be navigated to by double-clicking the Core Cell.

## 2.2 Core Structures

**Core level Structures** (or simply 'Core Structures') are the Structures contained inside Core Cells and Core Macros. These Structures look like follows:




The internal Structure of a Core Cell.

The most obvious difference from the Primary Structure is the frame with three different areas (1), (2) and (3).

(**1**) **Normal Modules area**: The bigger area in the middle (referred to as the **normal Modules area** or simply **normal area**) is where the majority of the Structure's Modules (more specifically, the so-called **normal Modules**) are typically located. The Core level Macros are also inserted here among the other normal Modules.

(**2**) **Input area**: The smaller area on the left is where the input port Modules are located.

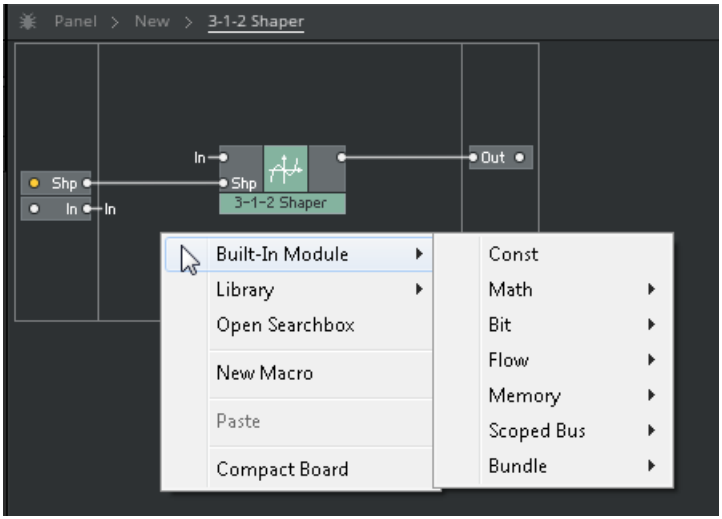
(**3**) **Output area**: The smaller area on the right is where the output port Modules are located.

The three areas can contain only the Modules of the respective **flavor**, that is the normal area can contain only the normal Modules and the input and output areas can contain only the input and output port Modules respectively.

The normal Modules can be moved freely in both horizontal and vertical directions. The input and output port Modules can be moved only vertically. Their vertical positions specify the order in which they appear on the outside of the Core Cell or a Core level Macro. Changing their relative vertical positions changes their outside order.

In order to create a new Module in a Core Structure, right-click on the background inside of one of these three areas. **Depending on which of the three areas is clicked, a context menu with a different list of available Modules appears.**

► For example, right-click on the normal area to access the normal Modules:

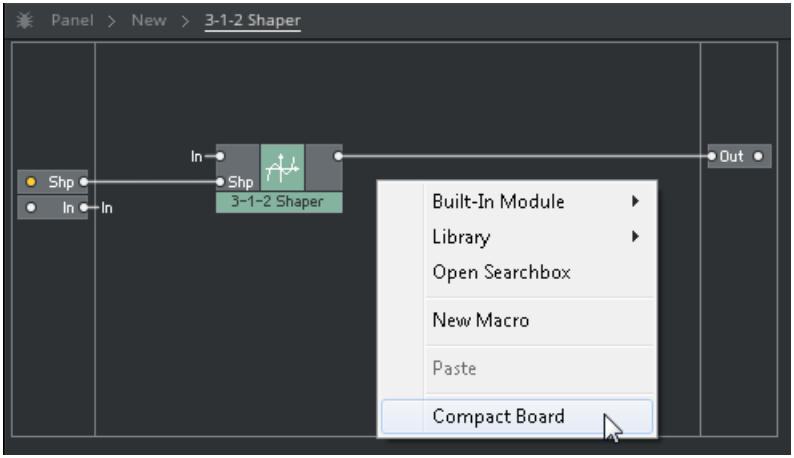



Pretty much any editing done to a Core Cell Structure will cause it to recompile. For small and simple Core Cells the compilation typically happens very quickly, but for large or complicated Core Cells it might take noticeable amounts of time. During the compilation, the progress is displayed in REAKTOR's toolbar by an orange gradient moving over the CPU load display:


## Structure Shrinking

As more Modules are inserted into the Structure and/or moved around within the Structure, the enclosing frame will grow to accommodate the Modules. The frame never shrinks back though.

► In order to make the frame shrink back, right-click on the background somewhere within the Structure and select *Compact Board* from the context menu.



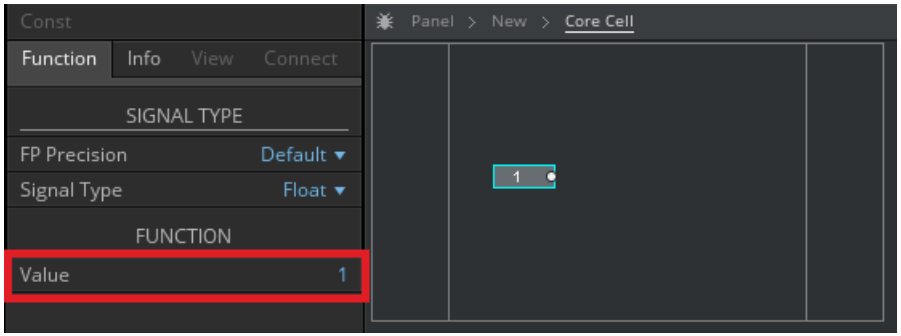

The shrinking will not change the distances between the Modules (except possibly the distances from the inputs and the outputs to the normal Modules). There is also some hard-coded minimum size which the shrinking will not go beyond.

## 2.3 Built-in Modules

The ideas and conventions behind the groups of Modules available in the *Built-In Module* submenu of the normal area context menu are discussed in different places of the manual according to their function. The information about specific Modules can be found directly within the software in the Module's info.

- *Const*: The Core constants are very similar to the Primary level constants. See section ↑4.4, Initialization for additional details.

The value of the Core constants cannot be changed in the name field of the Properties, one has to use the dedicated value property field:




- *Math*: See sections ↑4.1, Events and ↑4.2, Processing Order.
- *Bit*: This group is simply an extension of the *Math* group dedicated to the integer bit arithmetic.
- *Flow*: See section ↑4.8, Routing and Merging.
- *Memory*: See sections ↑4.3, Object Bus Connections (OBC) and ↑4.7, Arrays.
- *Scoped Bus*: See section ↑3.3, Scoped Buses.
- *Bundle*: See section ↑3.4, Bundles.
- *Macro*: See section ↑2.6, Core Macros.

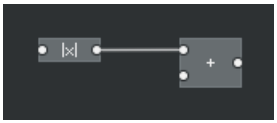
## 2.4 Core Connection Types

REAKTOR Core Structures are using several different types of connections. Respectively there are different types of the input and output ports of the Modules used inside Core. These types fall into several different classes.

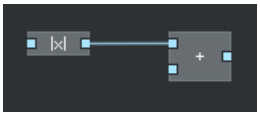
The connections between ports of different classes are not possible. Whether the connections between different port types within the same class are possible varies from one class to another.

### 2.4.1 Scalar

Scalar is the most commonly used class. Scalar connections can be either of **float** (floating-point) or **int** (integer) type, of which float is more commonly used.

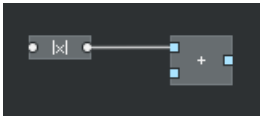


A connection between float ports.



A connection between int ports.

All scalar types are connection-compatible to each other (meaning it is possible to connect a float output to an int input or vice versa). The values of one type will be automatically converted into another wherever necessary:



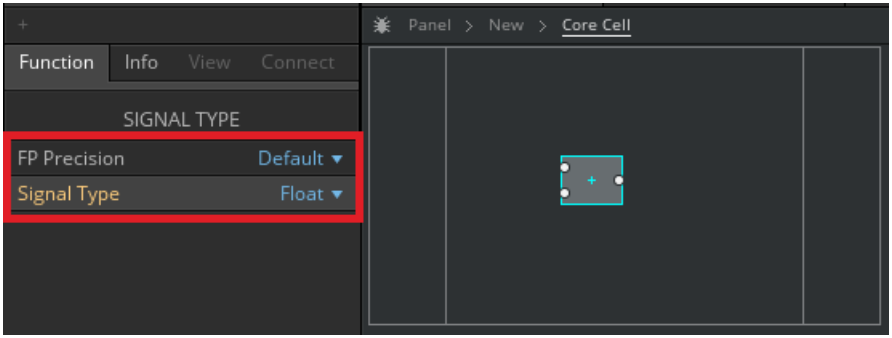
A float output connected to an int input.

Upon conversion from float to an int the value is rounded to the nearest integer.

For the values which are exactly in between, the rounding direction is conceptually undefined. For example, 1.5 can be rounded to 1 or to 2. It is not advisable to rely on a specific direction of that rounding for any of such 'exactly-in-between' values, even if it seems fully consistent and reliable in the current REAKTOR version. The rounding could change e.g. on a different hardware architecture or for whatever other reason. As a general rule one should not rely on the implementation details which are undefined by the specification, especially if they are explicitly mentioned as undefined.

### Type Configuration

For many of Core's built-in Modules, particularly for those which perform mathematical operations, the scalar type can be reconfigured in the Properties:



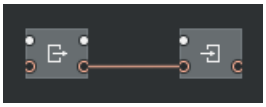

Scalar configuration properties of an adder.

The Signal Type property can be switched between Float and Int. The FP Precision setting (applicable only to the float mode) can be Default, 32 bit or 64 bit. Normally it should be left at Default, which means that the Module will use the default floating point precision of the given Structure, where the latter (unless reconfigured by the builder) is 32 bit. The 64 bit precision is needed in exceptional cases. The precision is not reflected visually in the look of Modules, ports or connections. The precision does not affect the compatibility of connections either, automatic precision conversion will occur wherever necessary.

> Conceptually, a 32 bit float precision setting does not guarantee that it will have exactly 32 bit precision. It means that the precision will be at least 32 bits. The same holds for 64 bit.
>
> The 32 bit float is the type used by Primary audio and event connections.

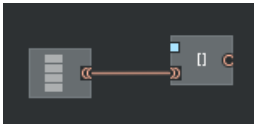
## 2.4.2 OBC

The OBC class is used for identification of the memory storage items (see the discussion in section ↑4.3, Object Bus Connections (OBC)), where the items can be of **latch** (for single variables) or **array** type. Each of these types can be float or integer.

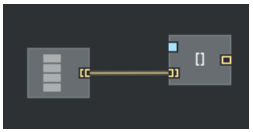


A connection between float OBC ports of type latch.

A connection between int OBC ports of type latch.



A connection between float OBC ports of type array.

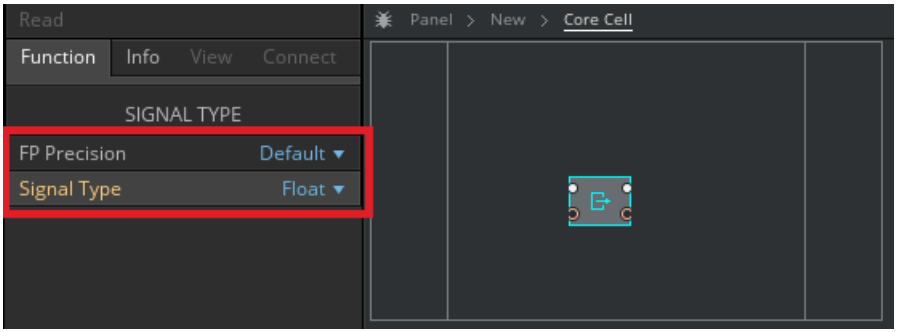


A connection between int OBC ports of type array.

Differently from the scalar connections, the OBC connections of different types are not compatible to each other.

## Type Configuration

Similarly to the scalar processing Modules, most of the OBC Modules feature the possibility of type configuration in the Properties:



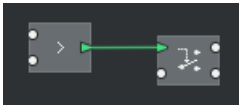

OBC configuration properties of a Read Module.

Besides affecting the OBC port types, these properties also affect the type of the memory storage associated with the OBC Module.

Differently from scalar processing Modules, the float OBC connections of different precision settings are not compatible to each other. Particularly, the Default precision setting is not compatible to the others (e.g. it is not compatible to 32 bit setting even if the actual precision corresponding to the Default setting is 32 bit). Normally the Default precision setting should be used. See sections ↑2.5.3, Properties and ↑2.6.3, Properties for further details of precision control.

### 2.4.3 BoolCtl

The BoolCtl class contains a single type, which is also called BoolCtl (Boolean control). These connections are used for router control signals (see section ↑4.8, Routing and Merging).

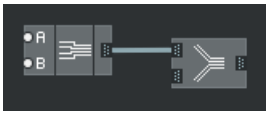


A connection between BoolCtl ports.

The BoolCtl type does not have any further configuration parameters.

### 2.4.4 Bundle

Bundles are 'multiwire cable' connections. They are simply containers for the nested connections.




A connection between Bundle ports.

The Bundle type itself does not have any configuration parameters. However the contents of the cable vary depending on its source. In principle any Bundle output can be connected to any Bundle inputs, but the connection may still be considered erroneous if the contents of the incoming Bundle do not match the expectations of the connection destination's owner. See section ↑3.4, Bundles for further details about using the Bundle connections.

### 2.4.5 Connection Defaults

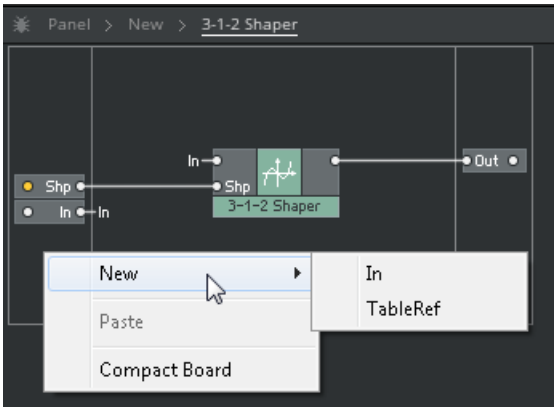
Inputs which are left unconnected still normally have a defined functionality. This functionality can be specifically defined by each Module for each of its inputs separately. Particularly, a Macro builder can specify the meaning of the disconnected Macro inputs for each of its inputs (see section ↑2.6.1, Inputs). In practice, however, there are most commonly used defaults. These defaults are used for the Macro inputs, unless specifically overridden by the Macro builder, and are also used for the built-in Modules, unless the Module's documentation says otherwise.

- **Scalar** inputs default to a zero constant. This means that the input behaves exactly as if there was a zero constant connected to it, including the initialization behavior (see section ↑4.4, Initialization).
- **Non-array OBC** inputs default to a unique storage location. This rule is abundantly used in storage manipulation (see section ↑4.3, Object Bus Connections (OBC)).
- **Array OBC** inputs, unless explicitly specified otherwise, should not be left disconnected. A disconnected array OBC input has undefined semantics, except that it is guaranteed not to affect the correctly connected OBC memory.
- **BoolCtl** inputs default to a false signal.
- **Bundle** inputs default to an empty Bundle.

# 2.5 Core Cells in Detail

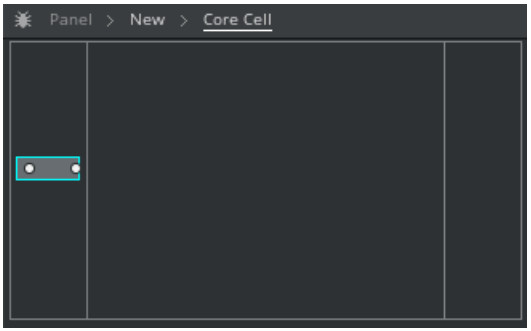
## 2.5.1 Inputs

There are only two different input port Module types available in the Core Cell's input port area context menu: *In* and *TableRef*.




The Core Cell's input port area context menu.

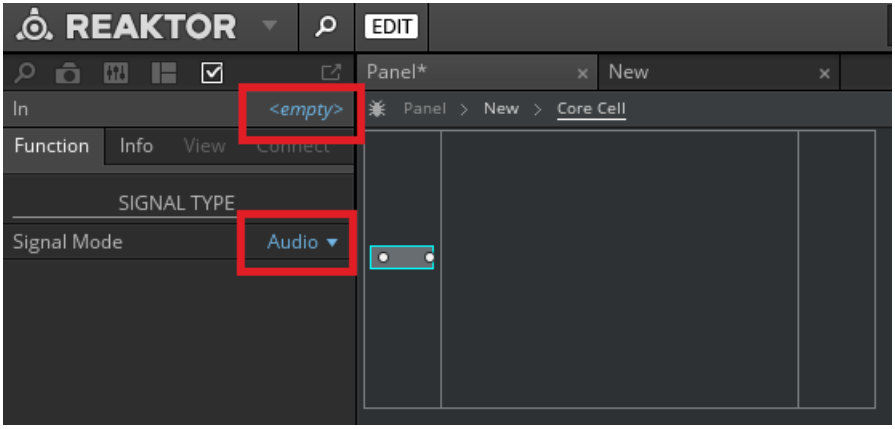
The *TableRef* input port type is described separately in the Building in Primary document. The Building in Core document deals exclusively with the Core Cell input ports of type *In*:




A Core Cell input port Module.

### Input Properties

The Properties of a Core Cell input port allow you to specify the port's name and select between the **Audio** and **Event-mode** of the port:



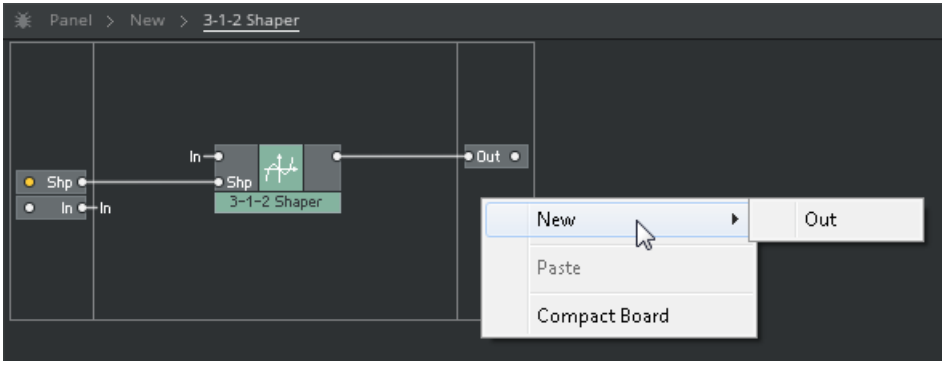

Properties of a Core Cell input port Module.

The Audio/Event-mode specifies whether the input will appear as a Primary level audio or event port on the outside of the Core Cell.

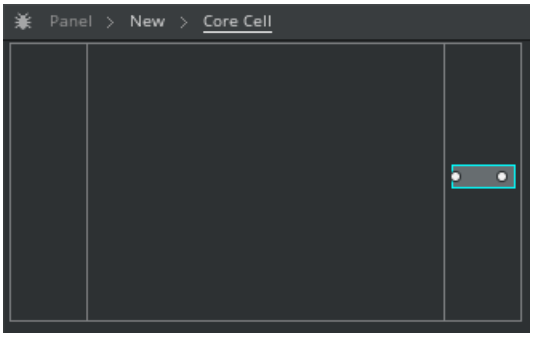
In the Info Properties tab one can specify the hint text for the port.

## 2.5.2 Outputs

There is only one output port Module type available in the Core Cell's output port area context menu: *Out*.



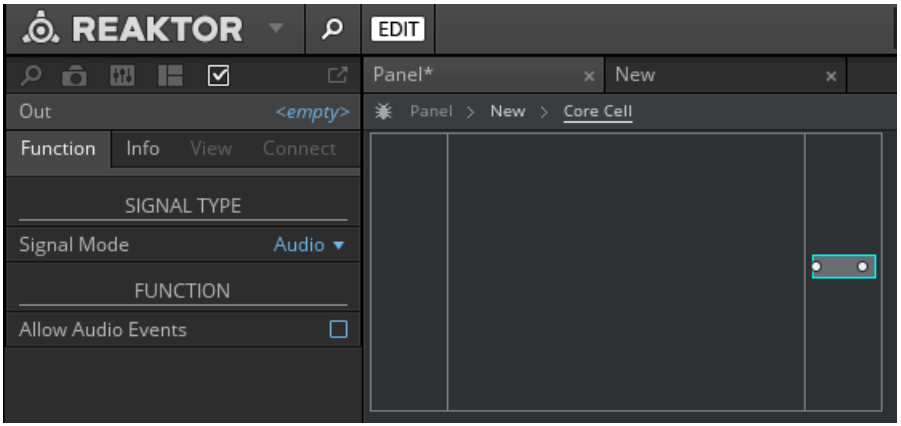

The Core Cell's output port area context menu.




A Core Cell output port Module.

## Output Properties

The Properties of a Core Cell's output port are similar to those of the Core Cell input port:




Properties of the Core Cell output port Module.

The Audio/Event-mode specifies whether the output will appear as a Primary level audio or event port on the outside of the Core Cell.

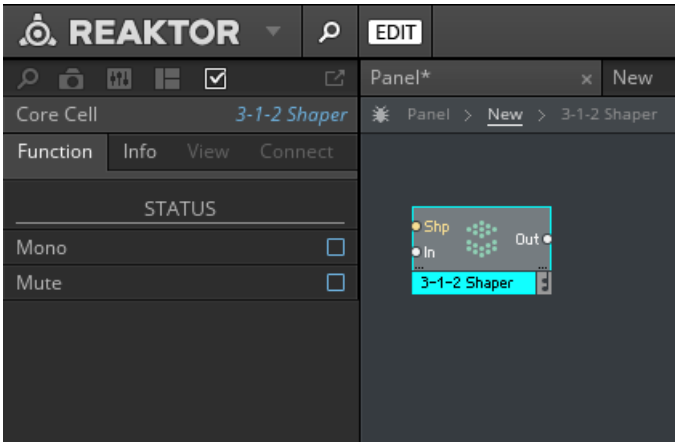
> The additional Allow Audio Events checkbox is an advanced option for the event-mode outputs. By default, the event-mode outputs block internal Core events that are originating from an audio source (those are the events obtained directly or indirectly from audio-mode inputs of the Core Cell and from the default internal clock sources of the Core Cell). This is important in order to avoid events being sent inadvertently from the Core Cell's outputs to the Primary level at audio rate, which is quite CPU-intensive. If you enable this checkbox, the event-mode output will not block any events. This option should be used with care. For the audio-mode outputs this check-box does not have any effect.

In the Info Properties tab one can specify the hint text for the port.

### 2.5.3 Properties

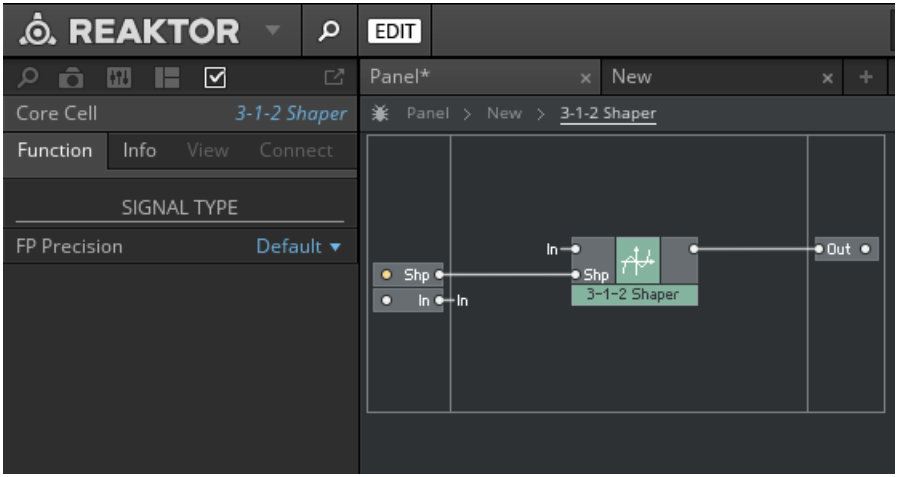
The Core Cells feature two distinct sets of Properties: the 'inside' (Core level) and the 'outside' (Primary level) ones.

The outside Properties of a Core Cell are accessible when the Core Cell is selected within the Primary Structure: They contain the Primary level configuration settings of the Core Cell.




The outside Properties of a Core Cell.

The inside Properties of a Core Cell are accessible by clicking the background of the Core Cell's internal Structure:




The inside Properties of a Core Cell.

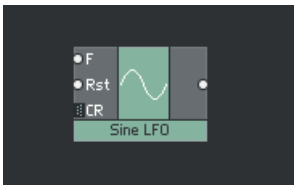
The Core Cell name and (editable) Info properties are shared between the inside and the outside Properties and can be equally accessed from either side.

The FP Precision property selects the meaning of the 'default' setting of the float type objects (Modules and ports) within the Core Cell's Structure. Simply put, the floating-point computations inside the Core Cell will be performed at this precision, unless explicitly overridden for certain elements within the Structure.

- Default: use the system-default setting, whatever that is. Technically this is 32 bit, the same as within Primary level. This setting should be used whenever possible (that is, unless there is a special reason to use a different setting).
- 32 bit: use 32 bit floats by default
- 64 bit: use 64 bit floats by default

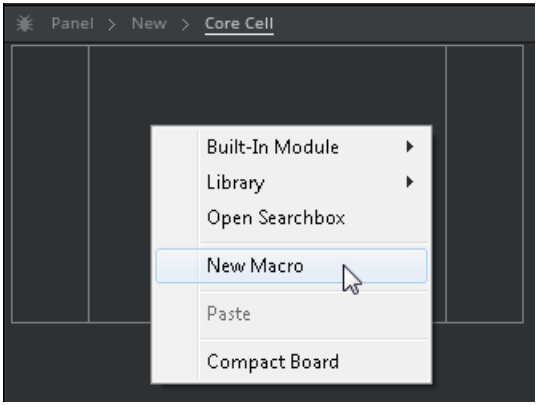
## 2.6 Core Macros

Conceptually seen, a **Core Macro** is not different from a Primary level Macro. This is simply a container Module which has another Core Structure inside:

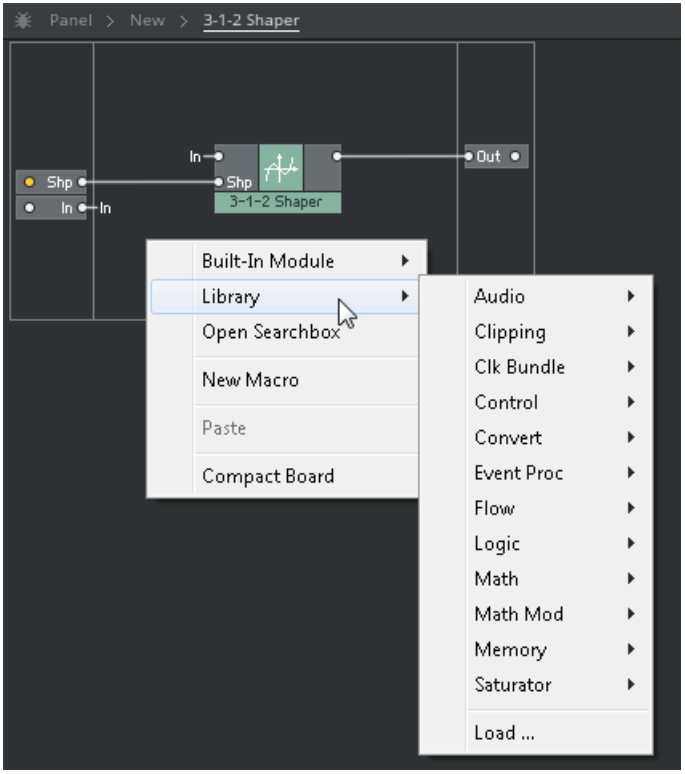



A Core Macro.

► To create a new blank Macro, right-click in the normal area of a Core Structure and select *New Macro* from the context menu.




In the normal area context menu next to the *Built-In Module* submenu there is a *Library* submenu containing the factory library and the user library Macros:

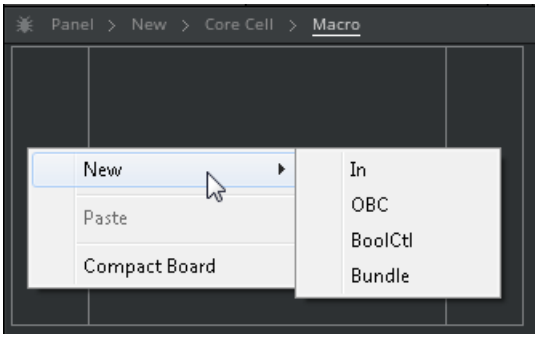



The Library submenu in the Core Macro's normal area context menu.

The internal Structures of Core Macros are very much like the ones of Core Cells, except that the available input and output port types are different.

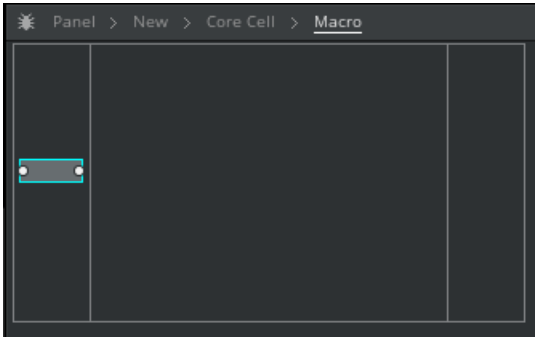
## 2.6.1 Inputs

The input types available in the input port area context menu of a Macro Structure correspond to different connection classes available in Core (see section ↑2.4, Core Connection Types for the connection classes and type discussion).




The Core Macro's input port area context menu.

- *In*: Creates a scalar type port. This is the most commonly used port type, which can be configured to float or int, of which float is most commonly used.
- *OBC*: Creates an OBC type port.
- *BoolCtl*: Creates a BoolCtl type port.
- *Bundle*: Creates a Bundle type port.

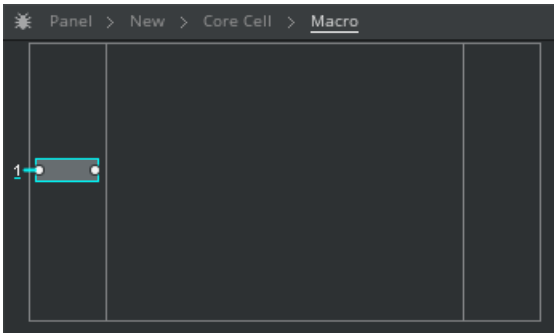



A Core Macro input port Module of float type.

The set of available properties of an input port Module varies depending on the Module type. Here one can switch the signal type between float and integer, control the floating point precision etc.

### Default Connection

Differently from Core Cell port Modules, which have only an output, the Macro input port Modules also feature an input. This input is typically left disconnected and is only used if one wants to override the default behavior of the corresponding port on the outside of the Macro. Technically, if a Macro input (on the outside of the Macro) is disconnected, the input connection of the corresponding input port Module inside the Macro will be used instead. So, if there is e.g. a constant of 1 connected to the 'default' input of the input port Module, then a disconnected Macro port will default to 1 rather than 0:




A Core Macro input port Module with a non-zero default signal.

## 2.6.2 Outputs

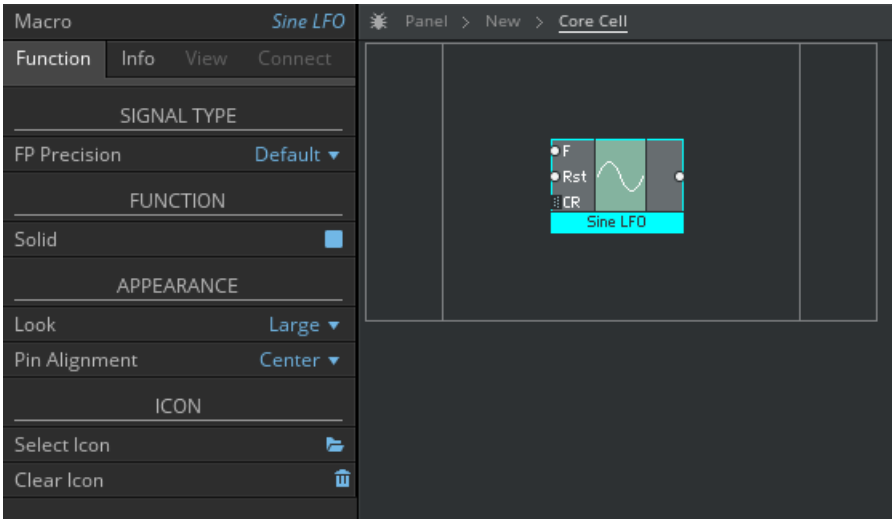
The output types available in the output port area context menu of a Macro Structure are exactly the same as the input types. Except for the absence of the 'default connection' functionality, the output port configuration is exactly the same as that of the inputs.

## 2.6.3 Properties

The Properties of a Macro can be accessed in two different ways:

- ► Select the Macro from the outside, from the Structure it is contained in.
- ► Left-click on the Structure background inside the Macro.

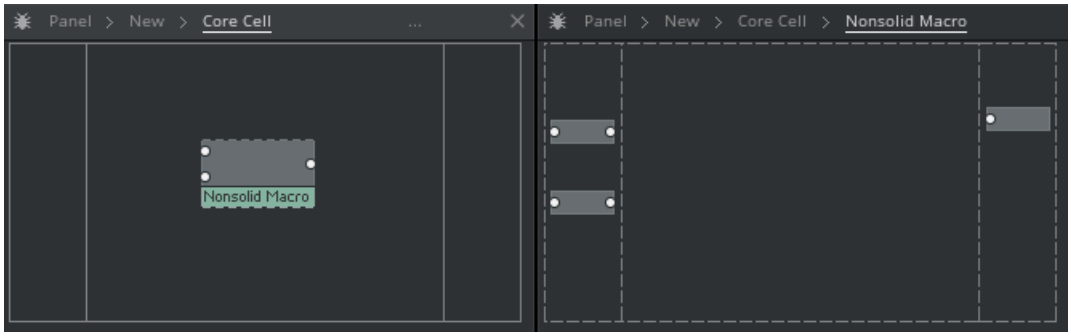
In other words, the inside and the outside Properties of a Macro are identical.



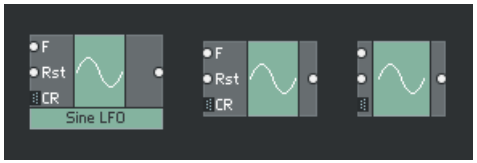

Properties of a Core Macro.

- FP Precision: controls the default precision of the float-type objects (Modules and ports) inside the Macro. It allows to specify the precision of the floating point computations inside the entire Macro (including the nested Macros), except for the areas which are explicitly configured to different precision settings. Usually this property should be left in the Default setting.
  - Default: use the same precision as the Structure that contains the Macro.
  - 32 bit: objects configured to default precision will use 32 bit precision floats (notice, that formally these objects are still considered to be configured to the default precision!) The respective float ports (scalar and OBC) on the outside of the Macro will appear as explicitly configured to 32 bit. In particular, a float OBC port which is configured to the default precision setting will appear as a default-precision port on the inside of the Macro and as a 32 bit precision port on the outside of the Macro.
  - 64 bit: the same as 32 bit, except the precision is 64 bit.
- Solid: controls the solidity of the Macro. Affects the resolution of the feedback loops (see section ↑4.10.3, Nonsolid Macros and Feedback Loops) and the visibility of the Scoped Buses (see section ↑3.3.6, Solid Areas). This property must be left in the 'On' state unless the Macro explicitly needs to be non-solid, otherwise it may make the Structure more

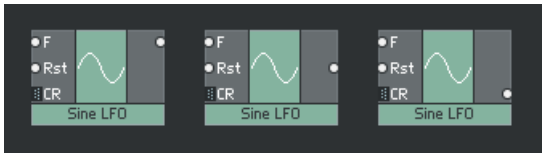
prone to difficult to find building errors and sometimes unnecessarily increase compilation times. The nonsolid Macros can be told visually by ragged edges, while their internal Structures can be told by dashed frames:




- Look: controls the outside look of the Macro. There are large, medium and small settings:




- Pin Alignment: controls whether the ports of the Macro should be aligned to the top, center or bottom in the outside look:

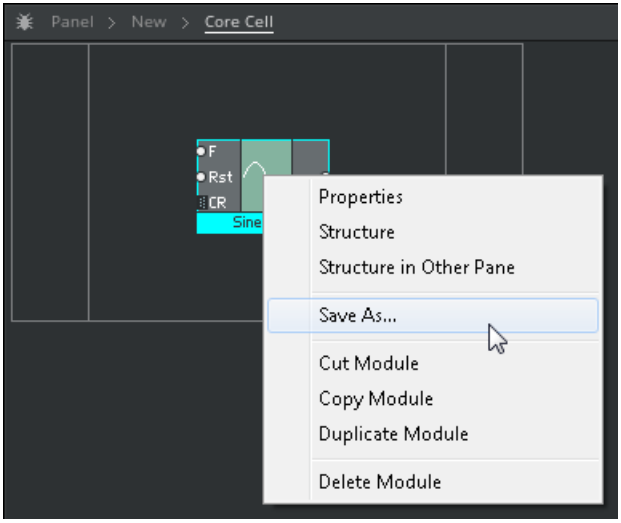



- Select Icon / Clear Icon: allows to load or unload a picture to be displayed in the outside look of the Macro. It is recommended to use a picture with an alpha channel (in PNG or TGA format).

## 2.7 Saving Core Cells and Core Macros

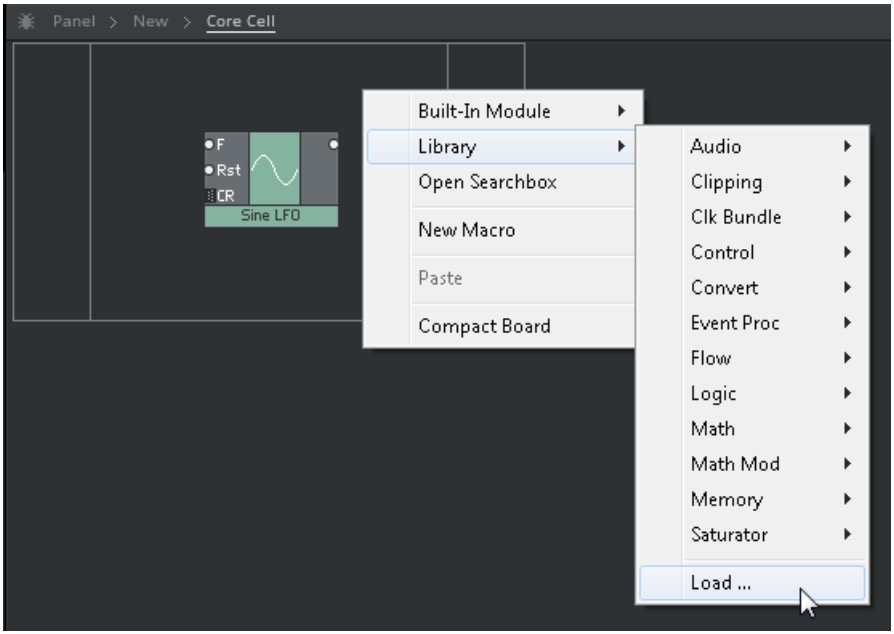
Similarly to the Primary Macros it is possible to save Core Cells and Core Macros as individual files as well as load them back into the Structure.

► To save a Core Cell or a Core Macro, right-click it and select *Save Core Cell As...* or *Save As...* respectively.




The *Save As...* function can also be used for built-in Core Modules. They are saved to the same file format as Core Macros. One can use this to save built-in Core Modules with preconfigured Properties, if desired.

► To load Core Cell and Core Macro files back, right-click on the background of the Primary or Core Structure (for Core Macros/Modules the right-click has to be within the normal area) and select *Library > Load...*.




Similarly to the user library folder "Primary" for Primary level objects, there is a folder called "Core" for Core level objects. The folders and the Core objects located in this folder will show up in the *Library* submenu after a separator.

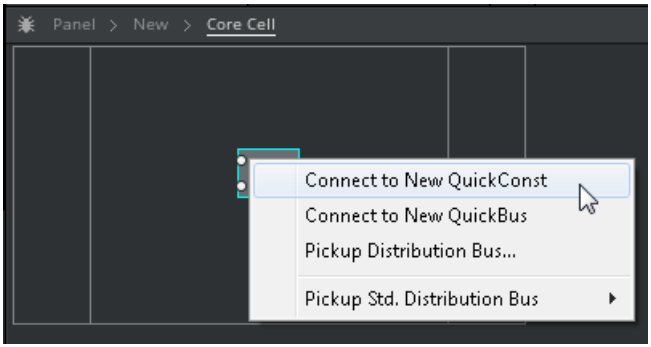
Notice that while Core Macros and Modules belong to the "Core" folder, the Core Cells belong to the "Primary" folder, since they are inserted from the Primary level and respectively need to be available in the Primary level context menu.

# 3 Additional Connectivity Features

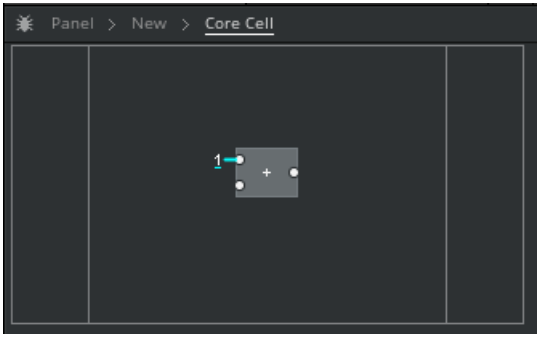
## 3.1 QuickConsts

QuickConsts are a lightweight alternative to *Const* Modules. To create a QuickConst do the following:

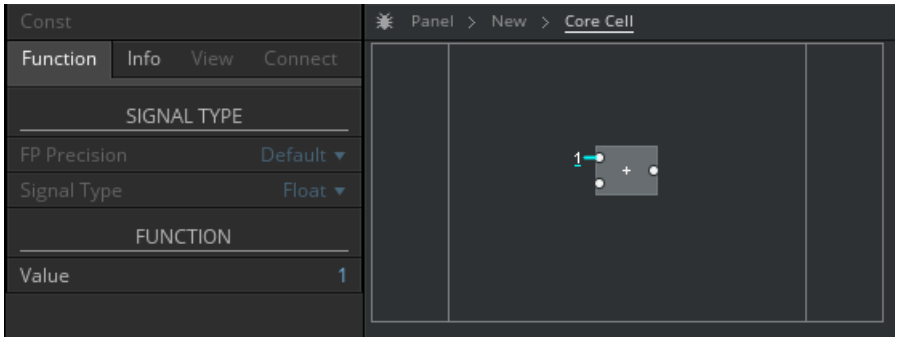
1. Right-click on a scalar input and select the *Connect to New QuickConst*.




2. A new QuickConst is created (if the input is already connected, the old connection will be deleted).



3. The value and the type of the QuickConst can be adjusted in the QuickConst's Properties.




The Properties show up whenever the QuickConst is selected.

- To select a QuickConst, simply click on the QuickConst.
- To delete a QuickConst, you can either select it and press [Del] or [Backspace], right-click on it and select *Delete Connection*, or click-and-drag from the input port into the Structure background.

Functionally the QuickConsts are fully identical to the ordinary *Const* Modules. However, unlike the *Const* Modules, they cannot be copied and pasted. Additionally, multiple connections to a QuickConst are not possible.

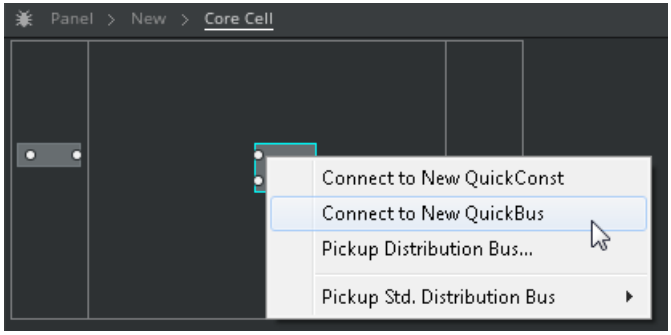
## 3.2 QuickBuses

QuickBuses are an 'invisible' alternative to the wires and can be used to reduce the connection clutter within a Structure.

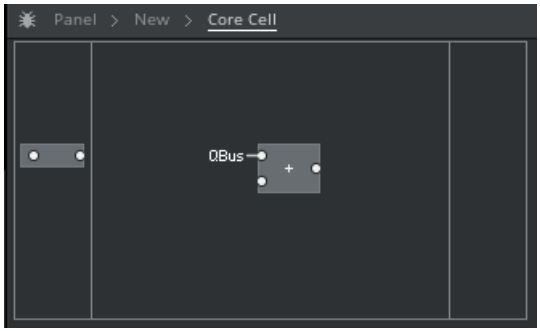
QuickBuses do not support cross-Structure connectivity. For cross-Structure connectivity use Scoped Buses (see section ↑3.3, Scoped Buses).

To create a QuickBus, do the following.

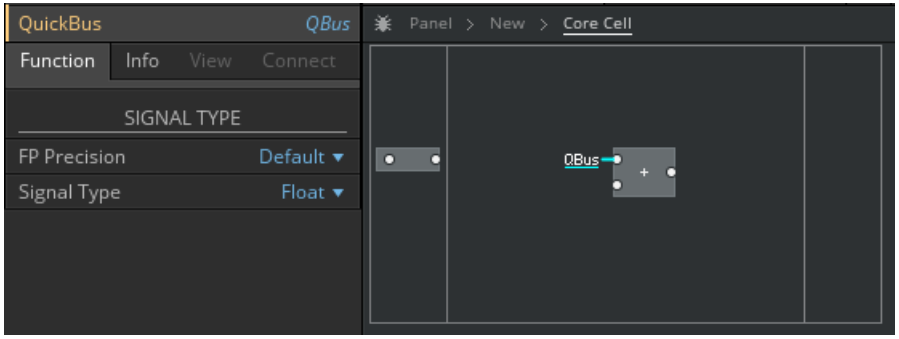
1. Right-click on an input or an output and select *Connect to New QuickBus*.




2. A new QuickBus with a default name (in this case "QBus") is created and the corresponding **QuickBus accessor** is connected to the port.




3. The name and the type of the QuickBus can be adjusted in the QuickBus's Properties.

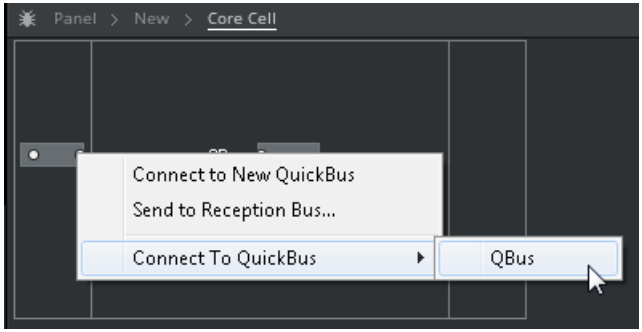


The QuickBus itself is not visible in the Structure. What is connected to the port is not the QuickBus itself, but the **QuickBus accessor**. Selecting an accessor (by clicking on it) switches the Properties to the associated QuickBus.

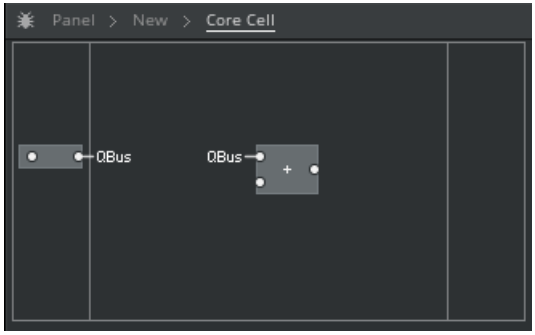
QuickBuses are supported for all port types, not just the scalar ports as QuickConsts.

In order for the QuickBus to establish a connection between an output port and an input port it has to be connected to both an input and an output. To connect a port to the existing QuickBus, do the following.

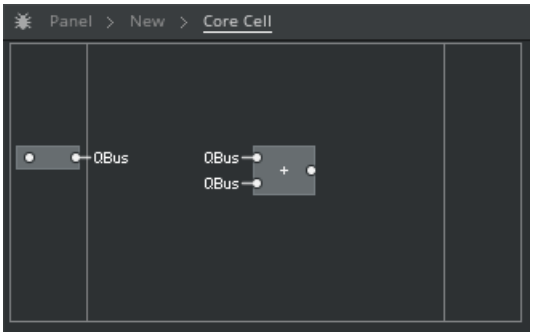
1. Right-click on the port and select the respective QuickBus's name in the *Connect to QuickBus* submenu.




2. Another accessor is created, thereby establishing a connection between the output and the input. Both accessors are referring to the same QuickBus object and thus are connected to each other.



The same QuickBus can be used to connect several inputs to one output. Simply right-click on an input and select the QuickBus from the *Connect to QuickBus* submenu to establish an additional connection to the same QuickBus:




Two connections established via a single QuickBus.

While it is possible to connect several inputs to a single QuickBus, at most one output can be connected to a given QuickBus at any given time. An attempt to connect a second output to the same QuickBus will disconnect the previously connected output from the QuickBus.

A QuickBus that is not connected to an output will be 'default-connected' according to the conventions explained in section ↑2.4.5, Connection Defaults. An input connected to a QuickBus accessor is formally a connected input (regardless of whether there is an output connected to the QuickBus or not), thus the default connection mechanism for this input is not active in this case. The explicit usage of QuickBuses which are not connected to an output is not advisable in finished Structures.

QuickBuses can have the same name within the Structure, even though giving each QuickBus a unique name is recommended. Duplicate names are automatically extended with extra indices "(1)", "(2)", "(3)" etc. These extra indices are not a part of the name and are merely a visual display feature, allowing you to distinguish between identically named QuickBuses. In principle these indices are intended to be used exclusively in temporary situations, such as after a QuickBus name collision upon copying and pasting. QuickBus name collisions in finished Structures are not recommended.

QuickBus connections can only be used between ports within the same Structure. Therefore, identically named QuickBuses from different Structures (if existing) represent completely different QuickBus objects and do not generate a collision either. In the terminology of programming languages, the QuickBus namespaces are per-Structure.

A QuickBus exists as long as there is at least one accessor associated with the QuickBus. The accessors are deleted in the same way as other connections and QuickConsts.

Disconnecting accessors by clicking and dragging to the background works only for the accessors at the input ports.

## 3.3 Scoped Buses

Scoped Buses allow invisible connections across several Structure layers. Unlike QuickBuses, Scoped Bus connections are asymmetric in the following sense:

- A QuickBus connection is established between two (lightweight) accessors.
- A Scoped Bus connection is established between a single (heavy-weight) **definition point** and one or multiple (lightweight) accessors.

The accessors connected to a definition point must be located within the **definition scope**. The definition scope includes the same Structure as the definition point and the Macros contained within that Structure (including the Macros contained within those Macros, etc.).

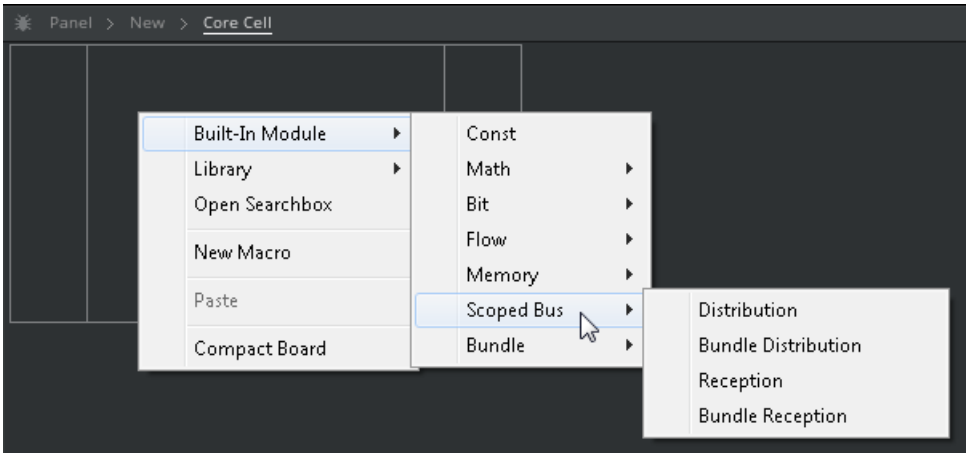
More strictly speaking the definition scope is the same **solid area** and all nested ones. See section ↑3.3.6, Solid Areas for more details.

### 3.3.1 Definitions

The definition points are provided in the form of built-in Modules and come in two different flavors:

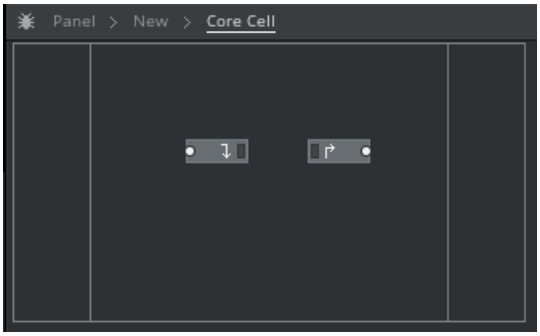
- **Distribution Scoped Buses**: The definition point acts as the output side of a connection (the signal flows from the definition point to the accessors). The input side of a connection is represented by a **scoped pickup accessor**. There can be multiple scoped pickup accessors connected to a single distribution bus. This is the most useful flavor of the two.
- **Reception Scoped Buses**: The definition point acts as the input side of a connection (the signal flows from the accessor to the definition point). The output side of a connection is represented by a **scoped send accessor**. The current implementation of REAKTOR Core allows only **single-reception Scoped Buses**, which do not support more than one connected scoped send accessor.

The connection classes supported by Scoped Buses in the current implementation of REAKTOR Core are scalar and Bundle. The *Built-In Module > Scoped Bus* menu allows you to create the definition points of distribution and reception flavors within these connection classes.




Scoped Bus submenu for creating distribution and reception definition Modules

A distribution definition Module has an input which is supposed to be connected to a signal source. A reception definition Module has an output which is supposed to be connected to a signal's destination:




Newly created scalar scoped distribution (left) and reception (right) Modules.

A newly created definition does not have a name and thus does not define a Scoped Bus. In order to define a Scoped Bus, the definition must be given a nonempty name in its Properties.

Both the distribution definition and reception definition Modules have a connection status indicator that turns green when one or more accessors are connected to this definition. It turns red in case of a definition conflict (see section ↑3.4.3, Definition Conflicts and Pickup Errors.).

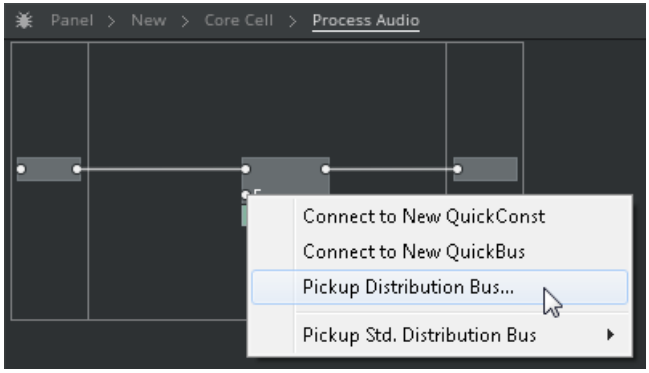

A distribution definition of a Scoped Bus named "Bus" with a green ('connected') connection status indicator.

### 3.3.2 Simple Scoped Access

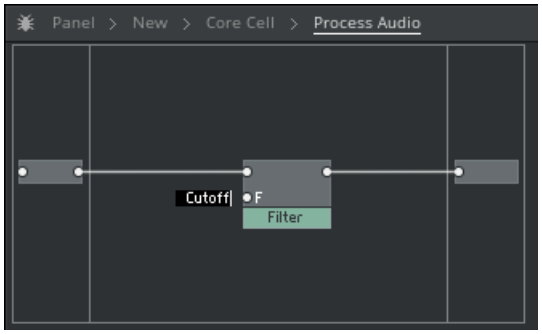
In order to make a scoped connection to a definition point an accessor needs to be created within the definition scope. For distribution buses a scoped pickup accessor (or simply 'pick-up') needs to be created, for reception buses a scoped send accessor (or simply 'send') needs to be created.

To create a scoped pickup accessor do the following:

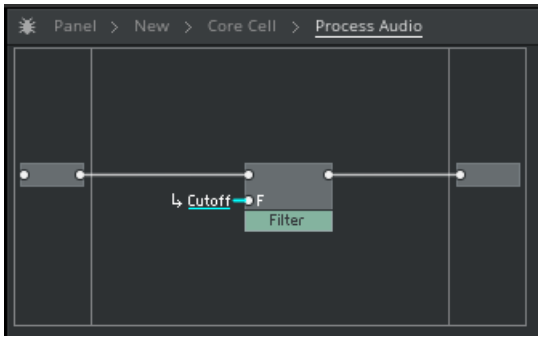
1. Right-click on an input of the scalar or Bundle type and select *Pickup Distribution Bus...*.



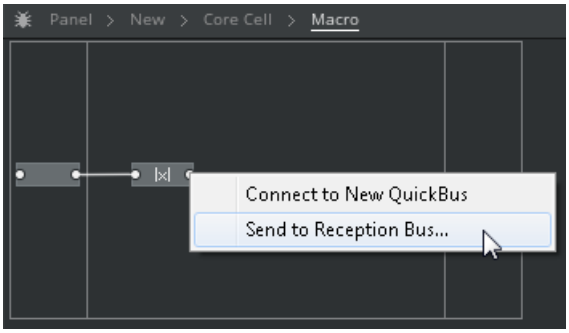

2. A text entry field appears next to the input. Enter a nonempty name for the scoped pickup accessor.




3. Press [Enter] to create a pickup with the name you just typed in. If you want to abort the creation of the pickup, press [Esc].

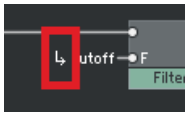


Scoped sends are created in the same way, except that you have to right-click on an output port:



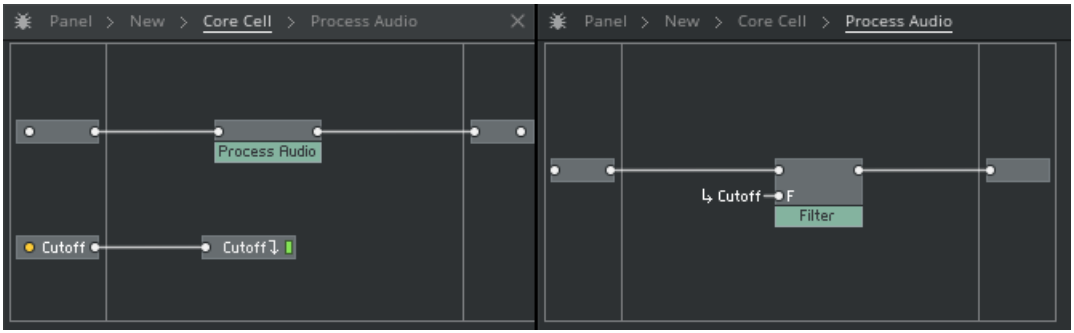

Creation of a scoped send.

The scoped accessors can be distinguished from QuickBus accessors by the 'broken' arrow next to them:



A scoped accessor is looking for an identically named definition, within the scope of which the accessor is located. The search is started in the same Structure (more correctly, in the same **solid area**, see section ↑3.3.6, Solid Areas) where the accessor is located. If an identically named definition is found, the accessor connects to that definition. If such definition is not found, the accessor looks in the parent Structure (parent solid area), and so on. If the definition is not found at the topmost (Core Cell) level, the accessor fails to connect.

The following example demonstrates a pickup named "Cutoff" connecting to the identically-named definition one level higher:




A scoped connection.

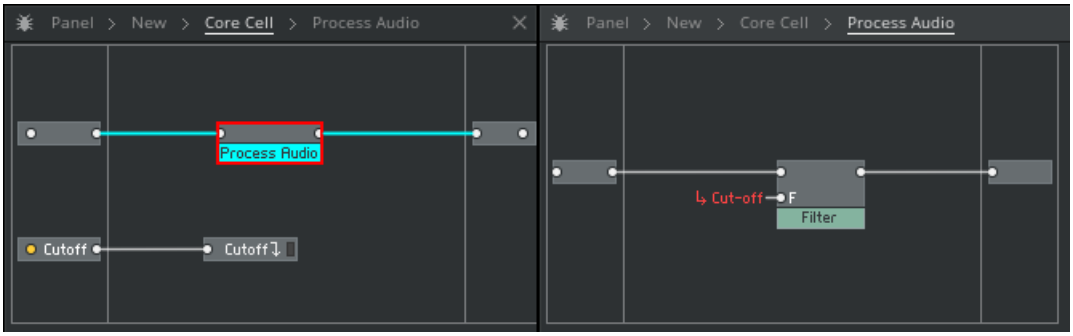
A scoped accessor can be selected by a single left-click. Differently from QuickBuses, accessor selection does not provide access to the bus's Properties.

A scoped accessor can be renamed by double-clicking it. Renaming is confirmed by pressing [Enter] or aborted by pressing [Esc], just like when giving an accessor a name upon creation. An accessor cannot have an empty name. If renamed to an empty name, the accessor is deleted. Otherwise, the deletion of scoped accessors can be done in the same way as for other connections and QuickBus accessors.

Differently from QuickBus accessors, which always stay connected to the same QuickBus and where the rename function renames the bus itself and not the accessors, the name of a scoped definition or of an accessor applies only to this specific definition or the accessor. If a scoped accessor or definition is renamed, it gets reconnected to a different definition or to a different set of accessors according to the new name.

### 3.3.3 Scoped Connection Errors

The scoped name lookup completely disregards the definition flavor and type (in programming language terminology, all Scoped Buses share the same namespace). If a scoped accessor finds a definition with a matching name that it cannot connect to (because the flavor and/or type are incompatible), it will fail to connect. Equally a scoped accessor fails to connect if it does not find a definition of a matching name at all. Such accessors are highlighted in red:



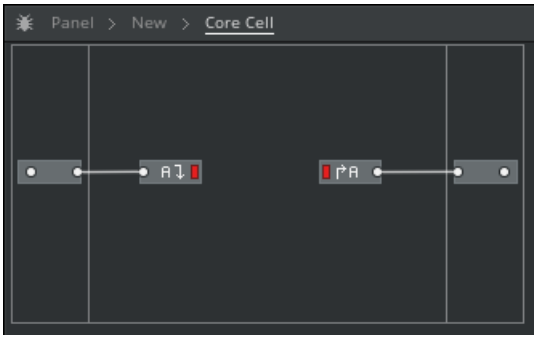

A scoped pickup accessor that fails to connect.

Aside from highlighting the accessor itself, the Core compiler highlights the parent Macros containing this accessor. The highlighting of Macros goes up in the Macro hierarchy as far as the 'error locality area' reaches.

The error locality area is (approximately) defined as the area that can affect the accessor error. E.g. if an accessor does not find a definition of a matching name, the parent Macros will be highlighted all the way up to the Core Cell's Structure. If, on the other hand, the error is due to a type or flavor incompatibility, the highlighting will go up only to the level of the respective bus definition.

### 3.3.4 Name Collisions and Definition Overriding

Two identically named definitions within the same Structure produce a name collision, regardless of their types and flavors. The connection status indicators of the conflicting definitions turn red:




Two conflicting definitions.

It is however possible to have identically named definitions if one Structure is a (direct or indirect) parent of the other. In this case the lower-level (the 'more nested') definition **overrides** the higher level one. Overriding means that from this Structure on, the new definition will be used by the respectively named accessors.

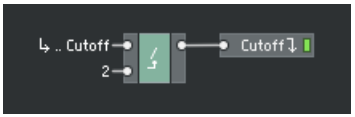
> The overriding effect is naturally following from the described earlier lookup process for a name-matched definition from the accessor. Clearly, the lookup process will find the 'most nested' definition above (or at) the accessor's level.

Identically named definitions in 'unrelated' Structures (that is, when neither of two different Structures is the parent of the other) do not produce a conflict, as their name visibility scopes do not overlap.

### 3.3.5 Parent Mode Accessors

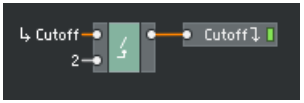
Sometimes, when overriding a definition, the new definition needs to be specified 'in terms' of the old definition. This makes it necessary (or at least highly desireable) to be able to access the old definition within the Structure of the new definition. This can be done by using a **parent-mode** accessor.

The accessor parent mode is specified by typing two dots in front of the accessor's name (when creating or renaming the accessor). The parent mode causes the accessor to ignore the definition within the same Structure and start the lookup for the matching definition one level higher, from the parent Structure. So, in case of a definition override, a parent-mode accessor will connect to the old rather than the new definition.




An example of parent-mode accessor usage

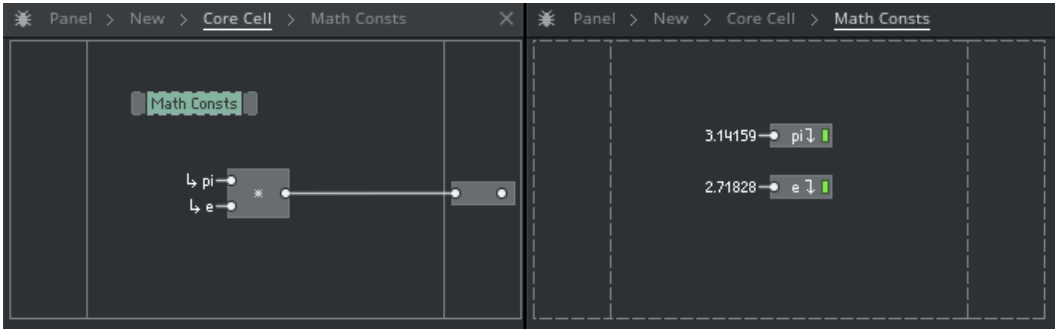
In the above example there is an (overriding) redefinition of the "Cutoff" Scoped Bus to a halved value. If the scoped pickup accessor on the left had not been set to parent mode, it would have connected to the new definition, creating a feedback loop (see section ↑4.10, Feedback Connections for a discussion of feedback loops):




An example of a logical error caused by the missing parent mode of the accessor.

### 3.3.6 Solid Areas

If a Scoped Bus is defined inside a Macro, the definition scope does not include the Macro's parent Structure. This makes it impossible to create a Macro the purpose of which is to define a Scoped Bus. This restriction can be lifted by turning off the Solid property of the Macro. In this case the Scoped Bus definitions contained directly within the Macro become visible on the parent level:




Using a non-solid Macro to define Scoped Buses for the parent Structure.

The Macro solidity also affects the resolution of the feedback loops (see section ↑4.10.3, Nonsolid Macros and Feedback Loops).

Whether using the non-solid Macros to define Scoped Buses within the parent Macro's context is good practice or not is questionable, since in this case the definitions are not visible at the top Structure of the definition scope. In any case this possibility should be used with care.

The description of the Scoped Bus functionality up to this point in the manual is approximate. It is only correct assuming that all Macros are solid. This assumption was done for simplifying the material presentation. A precise description of the Scoped Bus functionality follows below.

### A Precise Description of Scoped Bus Rules

The Scoped Bus mechanism of REAKTOR Core works in terms of the **solid areas** rather than in terms of Structures. A **solid area** is the area bounded by the boundaries of solid Macros. If a Macro is solid and so are all directly nested Macros, then the Macro's Structure constitutes a solid area. If all Macros in a Core Cell are solid, then there is no difference between Structures and solid areas. If in such a Core Cell one Macro is made non-solid, then the solid boundary between this Macro and its parent Structure disappears, meaning that the parent Structure and the Macro's Structure now constitute one solid area, etc.

Consequently, the following applies:

- The scope of a definition starts at the solid area of the definition and extends to all nested solid areas, unless overridden by another definition in a nested solid area.
- Two identically named definitions within one solid area produce a conflict.
- A parent-mode accessor looks up for the matching definition starting from the parent solid area, rather than the parent solid Structure.

## 3.3.7 Name Normalization and Escaping

### Normalization

Since scoped connections are established by name, it is very important that the names are visually distinguishable in the Structure. For that reason REAKTOR Core automatically collapses all whitespace within the names:

- Whitespaces in the middle of a name are collapsed to a single space.
- Whitespaces at the beginning and at the end of a name are eliminated completely.

This process is referred to as **normalization**.

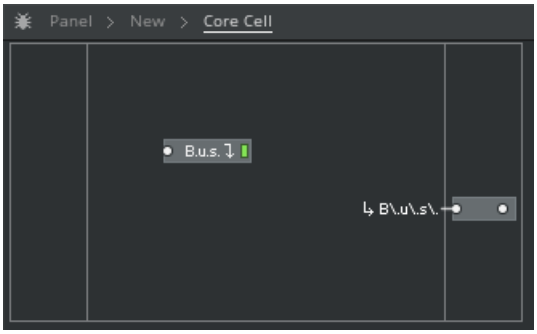
### Escaping

The dot symbol has a special meaning in the scoped accessors. It is used to specify the accessor parent mode. It is also used as a fiber separator in the Bundle subfiber pickup accessors (see section ↑3.4.4, Scoped Bus Subfiber Pickups). At the same time it is still possible to define a Scoped Bus with one or more dots within its name. In order to be able to access such definition the accessor must be able to specify that a certain dot should not be treated as a special syntactical element (parent mode specifier or fiber separator), but an ordinary dot. This possibility is implemented in the form of the so called **escaping**.

In order to specify that a certain dot in the accessor name is an ordinary dot, the dot must be prefixed by a backslash.

The backslash is referred to as the **escape character**.

The backslash thereby also obtains a special syntactical meaning and thus needs to be escaped itself if an ordinary backslash character is being meant.




Dot escaping in an accessor.

In the above case the Scoped Bus name ("B.u.s.") contains three dots. In the accessor each of these dots needs to be prefixed by a backslash.

Escaping is only applied to accessors, not to definitions.

If some special symbol within an accessor name is used at an inappropriate position (e.g. three dots are used in a row at the beginning of an accessor), then REAKTOR Core will not 'approve' it as a special symbol and will automatically escape it, explicitly highlighting its non-special meaning.

## 3.4 Bundles

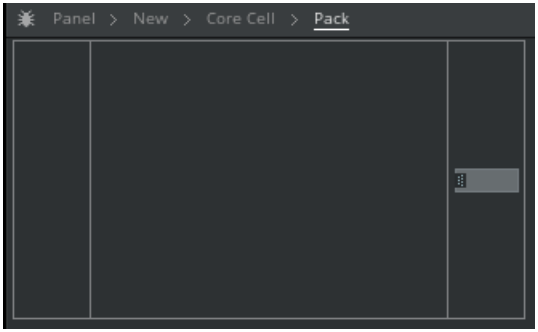
Bundles are a connection type, in which each 'cable' has multiple wires internally. These internal wires are referred to as the Bundle **fibers**.

### 3.4.1 Bundle Pack

In order to put several fibers into one Bundle use a *Built-In Module > Bundle > Pack*:

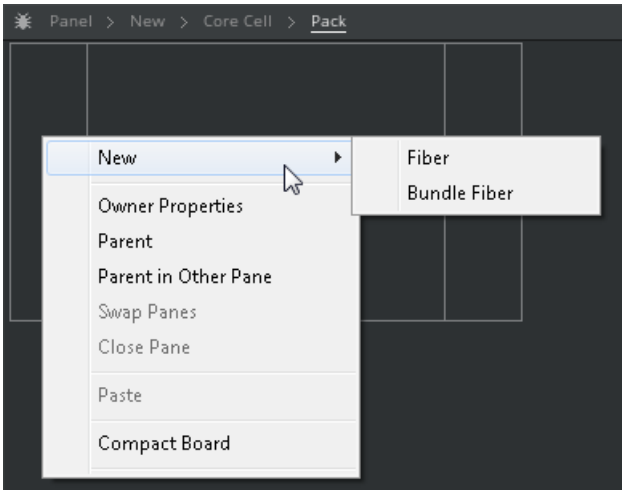
A newly created Bundle Pack Module.

A newly created Bundle *Pack* Module has no inputs and one Bundle output. Having no inputs means that the outgoing Bundle has no fibers. In order to add fibers to the Bundle, navigate into the internal Structure of the *Pack* Module by double-clicking on it, in the same way as it is done with Macros. The *Pack* Module's internal Structure will show up:




The empty internal Structure of a newly created Pack Module.

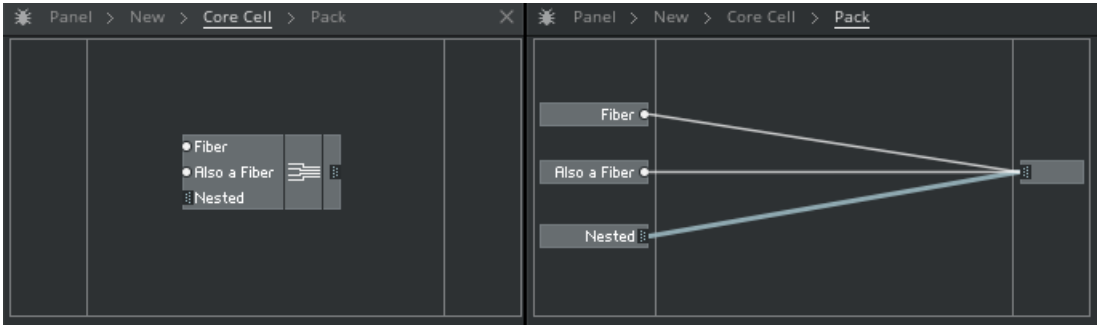
The only editable part of the Pack Module's internal Structure is the input area on the left. The inputs correspond to the Bundle fibers and are referred to as **fiber definitions**. Currently, only scalar and Bundle fiber definitions are supported:




The Pack Module's input area context menu.

The fiber definitions of Bundle type allow to nest Bundles.

The fiber definitions created in the *Pack* Module's internal Structure show up as inputs on the outside of the Module:




Fiber definitions and the corresponding inputs of the Pack Module.

Each fiber definition has to be given a name. Definitions with empty names cannot be picked up on the other side of the Bundle.

Similarly to the Scoped Bus names, the fiber names are automatically normalized.

Most commonly the output of the Pack Module is connected to:

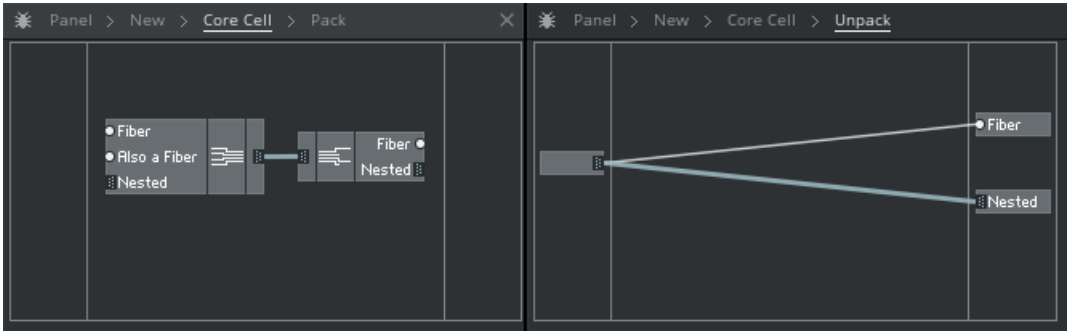
- a Macro output port from the inside of a Macro (if the *Pack* Module is contained within a Macro and the Macro output interface is employing a Bundle), or
- a Macro input from the outside of a Macro, or
- a scoped distribution definition.

### 3.4.2 Bundle Unpack

In order to access individual fibers within a Bundle use *Built-In Modules* > *Bundle* > *Unpack*:

A newly created Bundle Unpack Module

A newly created Bundle *Unpack* Module has a Bundle input and no outputs. The Bundle input is supposed to be connected to the Bundle, the fibers of which need to be accessed. The outputs of the *Unpack* Module correspond to the fibers which need to be accessed and are created as outputs in the *Unpack* Module's internal Structure. They are referred to as **fiber pickups**.




Fiber pickups and the corresponding outputs of the Unpack Module.

The fiber pickups need to be given names identical to the names of the fibers that are being picked up. They also need to have a compatible type.

It is not necessary to pick up all the fibers of the Bundle in the *Unpack* Module. Only a subset of the fibers may be picked up. E.g. in the above example the "Also a Fiber" fiber is not being picked up. The pickups do not have to be arranged in the same order as definitions either.

## 3.4.3 Definition Conflicts and Pickup Errors

### Definition Conflicts

The fibers within a Bundle are getting distinguished by their name. Thus, a Bundle should not have two identically named fibers. If it does, then these fibers generate a conflict error:


A definition conflict created by two identically named fibers.

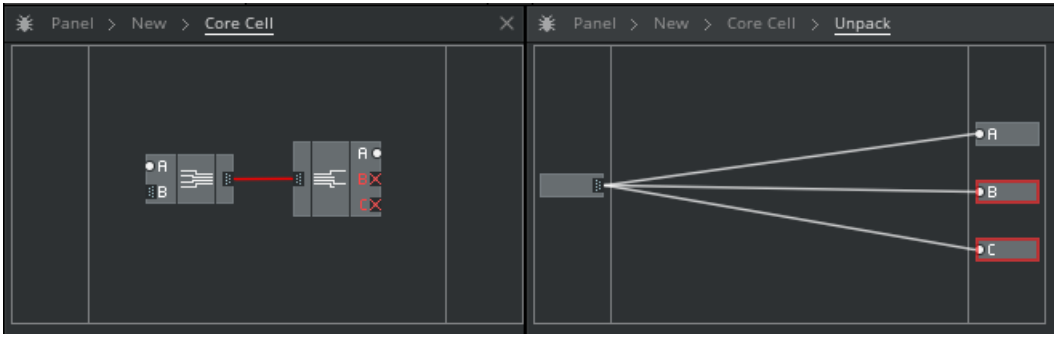
A conflicting definition does not generate a fiber in the Bundle.

More precisely, all conflicting definitions of the same name generate a **phantom** fiber in the Bundle. Phantom fibers cannot be picked up, but are participating in Bundle merging (see section ↑3.4.5, Bundle Merging) and splitting (see section ↑3.4.6, Bundle Splitting).

Definitions with empty names are considered void and do not conflict with each other.

## Pickup Errors

An attempt to pick up a missing fiber, a phantom fiber or an incompatible fiber will generate a pickup error:




Pickup errors generated by missing and incompatible fibers.

The above picture shows two errors:

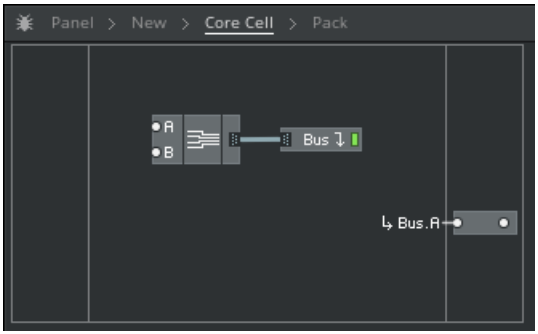
- The error in the pickup "B" is due to the type incompatibility (since the fiber "B" is of Bundle type, while the pickup is scalar).
- The error in the pickup "C" is due to the missing fiber (since the fiber "C" is not present in the incoming Bundle).

Pickup errors cause the Bundle wire to be highlighted in red all the way up to the definitions. This is done for use cases such as connecting together two library Macros by a Bundle wire. If the Bundle output of one Macro is incompatible to the Bundle input of the other Macro, the wire between the Macros will be highlighted in red. The wire highlighting also helps tracing the erroneous connection all the way between the definitions and the pickups in both directions.

> This highlighting approach follows the same idea of the error locality discussed in section ↑3.3.3, Scoped Connection Errors.

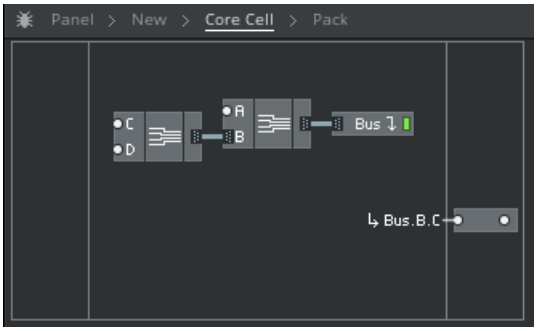
### 3.4.4 Scoped Bus Subfiber Pickups

As a convenience feature, when accessing a Bundle Scoped Bus, the scoped pickup accessor can access individual fibers of a Bundle. To do this, simply type the fiber name after the bus name, separated by a dot in between:




Scoped subfiber pickup.

In case of nested Bundles, subfiber pickups can be chained within one scoped pickup:

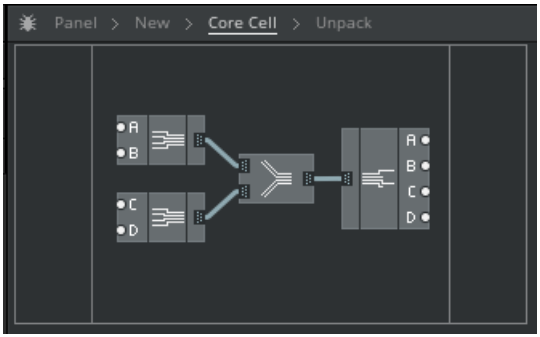



Scoped nested subfiber pickup.

There is no corresponding feature for scoped send accessors.

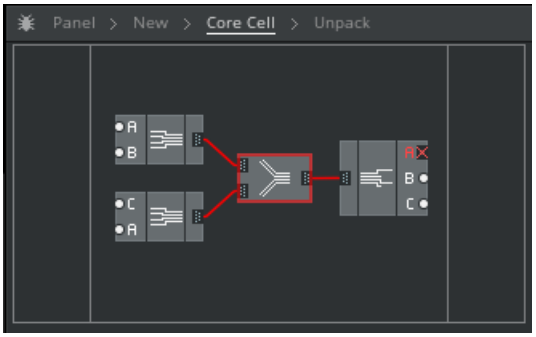
### 3.4.5 Bundle Merging

As a means of Bundle manipulation, two Bundles can be merged together into a single Bundle by the *Built-In Module > Bundle > Merge*:




Merging two Bundles into a single one.

The merged Bundles should not have identically named fibers, otherwise a merge conflict will occur. The conflicting name is turned into a phantom fiber in the output Bundle and cannot be picked up:

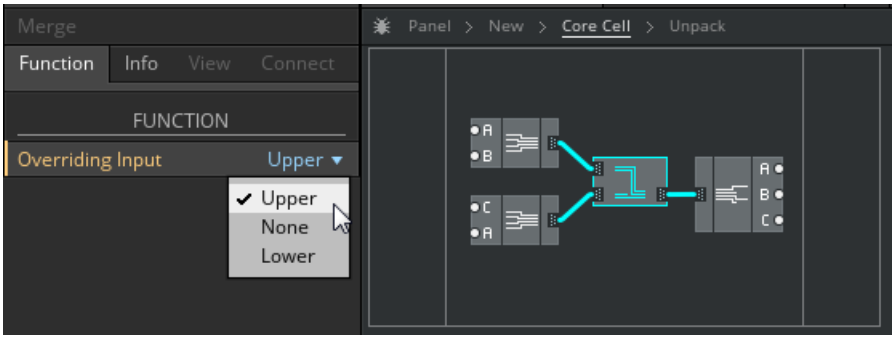



A Bundle merge conflict.

> A phantom fiber can generate further merge conflicts downstream if merged with identically named fibers.

## Override Merging

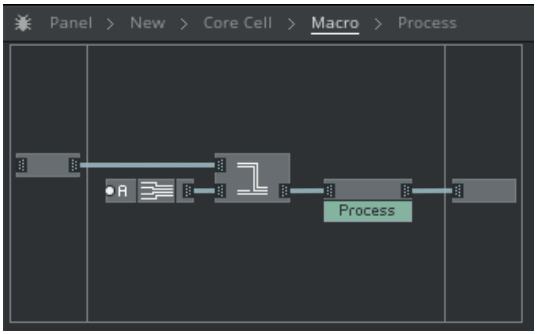
By selecting an overriding input in the Bundle *Merge*'s Properties the Bundle *Merge* Module can be switched into override mode:




An override mode Merge.

This causes the fibers at the **overriding input** to take precedence over the fibers of the other input in case of a name conflict.

The overriding mode is also useful as a means to provide default definitions for optional Bundle fibers in a library or framework Macro:




Using the override mode to provide defaults for optional fibers.

In the above example the incoming Bundle is expected to contain certain fibers, among which the "A" fiber is optional. If the "A" fiber is not present in the incoming Bundle, a default scalar fiber of zero value is inserted into the Bundle before it goes further into the "Process" Macro.

In the 'providing the defaults' use case the main Bundle has to go into the overriding input of the *Merge* Module, while the defaults should be connected to the other ('lower priority') input.

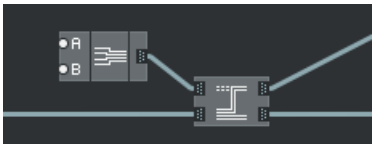
### 3.4.6 Bundle Splitting

The reverse of Bundle merging is Bundle splitting, done by *Built-In Module* > *Bundle* > *Split*:

A Bundle Split Module.

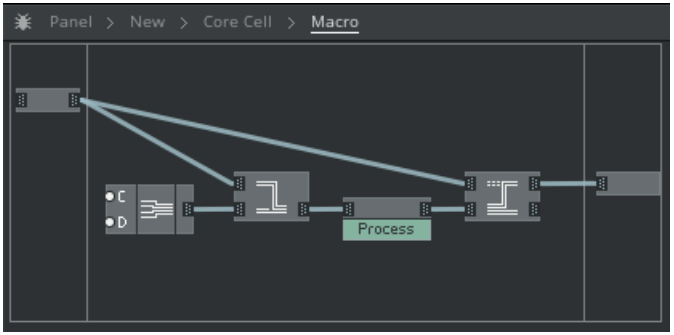
The Bundle that needs to be split should be connected to the lower input of the *Split* Module. The upper input of the *Split* Module is the **template** input. It controls the routing of the split fibers. The routing control is purely name-based. The fibers in the lower input for which there is an identically-named fiber in the template are routed to the upper output. The remaining fibers are routed to the lower output.

One can explicitly 'route off' certain fibers from a Bundle by connecting a 'dummy' *Pack* Module to the template input:



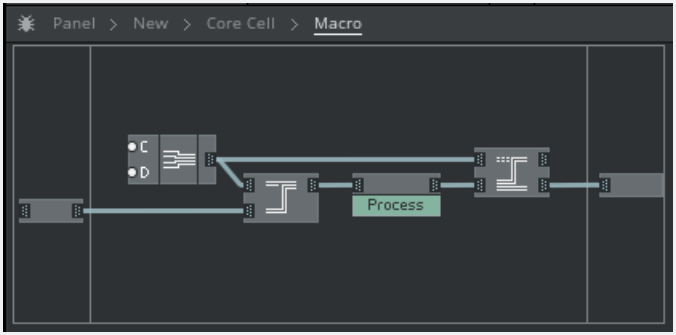

'Routing off' of fibers "A" and "B" from the Bundle.

Another use case is to remove the fibers that have been 'temporarily' merged into the Bundle:




Removing the temporarily merged 'default fibers'.

In the above example it is important that the *Split* Module is using the original Bundle as the template. A possible alternative implementation using the defaults as the template will not work correctly if the default fibers are already present in the input Bundle of the Macro, because these fibers are routed off by the *Split*:



# 4 Processing Model of Core

## 4.1 Events

A fundamental concept of signal processing in Core is the **event**. Similarly to the Primary level, an event within REAKTOR Core simply means that at some point in time a value has been generated at some output port. This value is immediately transmitted further downstream along the wires (or connections of other kinds), which is referred to as 'an event is being sent from the output to the connected inputs.' The downstream Modules then usually generate further events at their outputs in response to the incoming events.

The default event triggering convention in Core is the following: **a Module sends an output event in response to an incoming event at any of its inputs**.

This convention is most commonly used for the math processing Modules such as the ones found in *Built-In Module > Math* and *Built-In Module > Bit* menus. See section ↑5.1, Expression Computation for an example of efficient use of this convention. Nevertheless, there are a number of other different triggering conventions used for certain groups of Modules or individual Modules (constants, Core Cell inputs, Modulation Macros, audio processing Macros, audio generation Macros etc.). These conventions are described in the respective areas of the manual.

Differently from Primary, within REAKTOR Core Structures there is no distinction between audio- and event-mode connections. In Core everything is an event. An audio signal is simply consisting of events regularly sent at the sampling rate.

### Conversion between Primary and Core Signals

The conversion between Primary audio/event signals and Core event signals is done according to the following:

- The audio-mode input ports of Core Cells produce events at the sampling rate of the Primary Ensemble.
- The event-mode input ports of Core Cells forward Primary level events arriving on the outside of the Core Cell to the Core Cell's internal Structure.

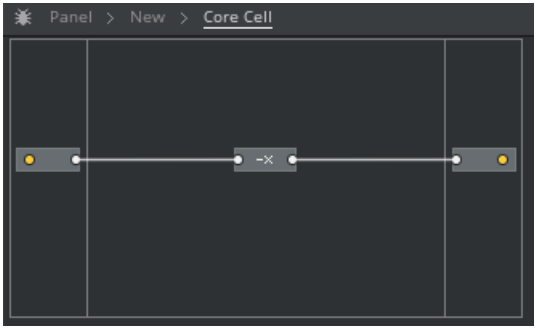
Respectively:

- The audio-mode output ports of Core Cells simply make their internal (Core level) value available to the external Primary level. This internal value is simply the value of the most recently arrived event.
- The event-mode output ports of Core Cells simply forward the internally arriving events to the external Primary level.

The Primary level does not support simultaneous events (see section ↑4.2, Processing Order for the discussion of simultaneous events in Core). If several Core events arrive simultaneously at the event-mode outputs of a Core Cell, they will be sent to the outside Primary Structure consecutively, one after the other. The order of sending is 'top-to-bottom', according to the visual ordering of the Core Cell output ports. The events are sent immediately upon the completion of the respective handler (see section ↑5.6.1, Core Cell Handlers). Particularly, if the events are generated from the audio handler, they will be sent before any further audio processing is done in the Primary Structure.

### Event Processing and Triggering Example

The following example Core Cell Structure implements an event signal inverter Core Cell:




A simple audio inverter Core Cell.

The Structure consists of an event-mode input port Module, an event-mode output port Module and a Core level inverter Module connected in between, where the latter can be found under *Built-In Module* > *Math* > *-x*. Since in Core there is no difference between audio and events, the built-in Core level inverter Module does not need to be configured (neither automatically nor manually) to an audio or an event-mode.

The processing of an incoming Primary level event by the above Core Cell is happening like follows:

1. A Primary level event is arriving at the Core Cell's event input
2. The Core Cell's event input is converting the Primary event into the Core event, which is then sent from the output of the input port Module inside the Core Structure.
3. The event reaches the input of the *-x* Module.
4. The *-x* Module is inverting the value and sends the result as its output event.
5. The event reaches the input of the Core Cell's output port Module.
6. The Core Cell's output port is converting the Core event into the Primary event and sends it further within the Primary level.

The above Core Cell can be turned into an audio inverter Core Cell by simply reconfiguring the input and output port Modules of the Core Cell to the audio-mode. In this case the input port Module will send the events regularly at the audio rate. Configuring the Core Cell's ports to an event-mode input and an audio-mode output is also possible.

One also can configure the input to the audio-mode and the output to the event-mode. In this case the expected behavior of the Core Cell is to send the Primary level output events at an audio rate. However, since the Primary level handles the events differently from Core, such a dense event stream on the Primary level will typically cause a noticeable CPU overhead. In order to avoid accidentally doing this by mistake, the Core Cell's output port has the Allow Audio Events property, which is disabled by default. When disabled, it will prevent the events originating at the Core Cell's audio-mode inputs (or another audio rate source, such as SR.C and CR.C) from being sent to the Primary level, blocking the events completely. The property has no effect on audio-mode ports.

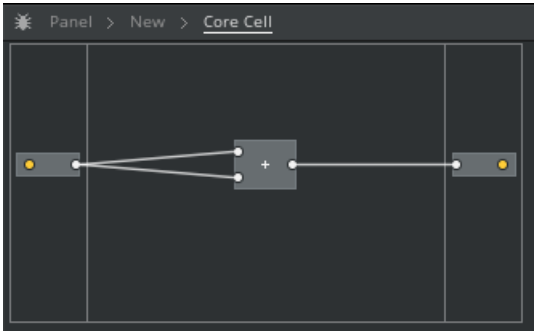
> Events derived from an audio rate source by routing (e.g. custom CR.C of a lower rate) will be equally blocked. On the other hand, the events derived from the event-mode Core Cell ports are not blocked. The initialization event (see the discussion of the initialization event in section ↑4.4, Initialization), including the initialization event obtained from the audio-mode Core Cell ports, is not blocked either.

## 4.2 Processing Order

As long as a Core event is processed exclusively in a serial fashion, there is no difference between Primary and Core event processing. The event is simply propagated downstream from one Module to the other. There is however a fundamental difference if the event propagation path splits into several branches.

On the Primary level an event that is sent to more than one destination is forwarded to each of these destinations in turn, one destination after the other. If the event paths later merge back again, then, instead of one event arriving at the mergepoint, there are two or more, depending on how many parallel paths the event was split into before merging back (for details about Primary level event processing, see the Building in Primary document).

In Core (logically seen) an event is sent to all such parallel destinations **simultaneously** on all paths. The following Core Cell Structure illustrates this:



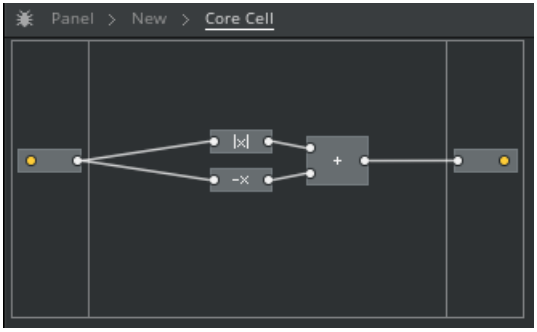

A Core Cell Structure illustrating the simultaneity of parallel events in Core.

The purpose of this Core Cell is simply to double the values of the incoming events. Instead of multiplying the values by 2, the signal's value is added to itself by using the adder Module (*Built-In Module > Math > +*).

Each incoming event is sent to the two inputs of the adder on two parallel paths. In Primary, the incoming event would first arrive at one input of the adder, producing an output event at the adder's output, and later it would arrive at the other input of the adder, producing another output event at the adder's output. Thus, for each incoming audio event the adder would have

generated two output events. This would lead not only to a doubling of the value, but also to a doubling of the count of the incoming event. In Core, the event arrives **simultaneously** at both inputs of the adder, so the adder produces a **single** output event.

The simultaneity of parallel events is of course extended to more complicated Structures, e.g. like the following one:




Another Core Cell Structure illustrating the event parallelism in Core.

In this Structure the event is going through the absolute value Module on the upper path and through the inverter on the lower path, and then both paths are merged by the adder. Of course, internally one of the two parallel Modules (the absolute value and the inverter) will be processed before the other. However, both the output event of the absolute value Module and the output event of the inverter Module will arrive at the adder's inputs simultaneously, thereby producing one event at the adder's output.

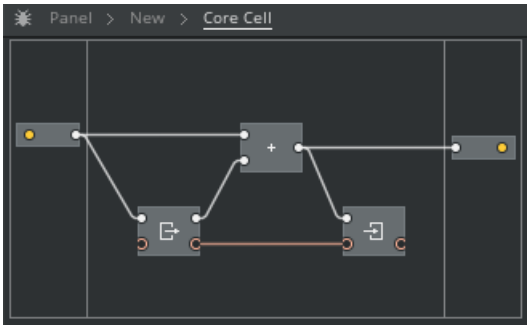
Thus, logically seen, it is impossible to tell which of the two parallel Modules (the absolute value and the inverter) has been processed first. From the formal point of view, both Modules can be considered as being processed simultaneously.

The processing order rule in Core is the following. If two Modules are connected by a unidirectional path (that is one can start at one of the Modules and reach the other one by always going only 'downstream' or only 'upstream') then the upstream Module is processed before the downstream one. If there is no unidirectional path between the two Modules, then their relative processing order is undefined. Such unordered Modules can be considered as 'simultaneously processed' ones.

The 'simultaneous processing' paradigm is not applicable to the explicit accesses to the memory by the OBC Modules. Memory accesses from the unordered Modules are serial and unordered.

## 4.3 Object Bus Connections (OBC)

Explicit memory storage is handled in Core by the *Read* and *Write* Modules performing the memory reading and the memory writing operations respectively. These Modules can be found under the *Built-In Module > Memory* menu. For the purpose of identifying the storage items referred to by these Modules, Core uses a dedicated connection class, called OBC (Object Bus Connection):




A simple event accumulator Core Cell.

In the above Core Structure there is a *Read* Module on the left, followed by an adder, followed by a *Write* Module. The direct connection between a *Read* Module and a *Write* Module is an OBC connection. It means that the *Read* and the *Write* Modules access the same storage (or the same variable, in terms of text-based programming languages). This storage is implicitly created by the mere fact that the *Read* and the *Write* Modules want to access some storage. Had there been no OBC connection between the *Read* and *Write* Modules, they would have accessed two independent storage items.

All Read and Write Modules connected together by OBC connections access one and the same memory storage (as long as there are no special OBC Modules, like e.g. the *Index* Module in between them).

The above Structure implements a simple event accumulator and works as follows:

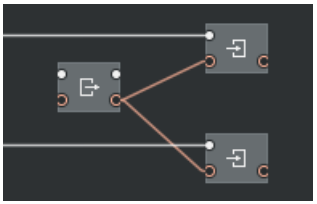
1. An incoming event from the Primary level is converted into a Core event by the input Module and is sent to the *Read* Module and to the adder.

2. The event arrives at the upper input of the *Read* Module. This is the clock or triggering input of the *Read* Module. In response to this clock event the *Read* Module reads the value from the memory and sends it from its upper output (the initial value of the memory is zero).
3. The event from the *input Module* and the event from the *Read* Module simultaneously arrive at the inputs of the adder. The adder computes the sum and sends it from its output.
4. The output event from the adder is sent to the Core Cell's output and also to the *Write* Module. The *Write* Module stores the new value of the accumulated sum in the memory, while the output forwards the same value to the Primary level.

The arrays and array connections which are a special kind of OBC are discussed separately in section ↑4.7, Arrays.

### Memory Access Ordering

In typical cases the memory accesses are naturally ordered by the OBC connection between the *Read* and the *Write* Modules. However, it is possible to construct Structures where these accesses are not ordered relative to each other:



Unordered memory accesses.

In the above picture the two *Write* Modules are ordered relative to the *Read* Module, but are not ordered relative to each other. If an event comes simultaneously on both wires, then it is not defined which value will be written first and which second. **Such Structures are generally not recommended, unless the access order is explicitly unimportant.**

As in the other cases of unspecified behavior, it is highly unrecommended to rely on the de-facto ordering of conceptually unordered OBC Modules, even if this ordering seems consistent.

As a general rule, it is recommended to ensure the correct ordering of the OBC Modules (accessing the same memory) by arranging them serially along one and the same OBC wire.

Macro boundaries by themselves do not ensure any ordering (please see section ↑4.7.4, Advanced OBC Ordering for a more detailed explanation).

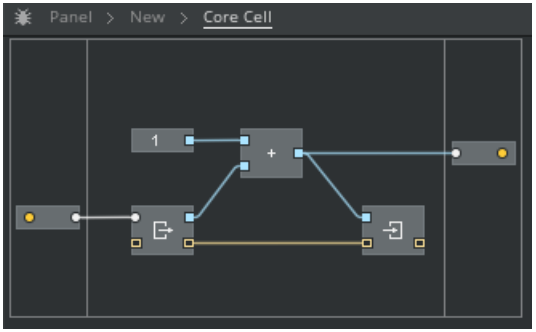
## 4.4 Initialization

Upon loading, before the regular processing begins, the Core Structures are getting initialized. The Core Cell initialization is performed as a part of the Primary level initialization process (please see the Building in Primary document for more information).The Core Cell initialization runs in the following steps.

- Firstly, all writable memory of the Core Cell is reset to zero. This includes the states of the output ports and the OBC storage.
- Then, an initialization event is sent simultaneously from all initialization event sources, which include the following.
  - Audio-mode Core Cell inputs. These always send the initialization event. The value of the event is equal to the value available on the outside of the Core Cell at the moment of the Core Cell's initialization.
  - Event-mode Core Cell inputs. These send the initialization event if and only if there is a Primary level initialization event coming from the outside at the moment of the Core Cell's initialization. The event's value is equal to the value of the incoming Primary level event.
  - Constants and QuickConsts (also including unconnected inputs without special default meaning as well as scoped and Bundle pickups which failed to connect). These always send the initialization event. The value of the event is equal to the constant's value. The constants never send any other events afterwards.
- After that, the Primary level initialization process might send additional events after the Core's initialization event is finished. From a Core perspective these events are regular events, not initialization events.

## Initialization Example

The following Core Cell Structure demonstrates the effects of the initialization process (**the Structure has a bug, which is embedded there for the demonstration purposes**):




A (faulty) event counter Core Cell.

At the first glance, this Structure should implement an event counter (notice the integer mode of the Modules used in the Structure). However, this Structure has a bug. If no Primary events have ever been sent to the Core Cell's input, the output value of the Core Cell (and the value stored internally in the OBC memory of the counter) is still one rather than zero. Furthermore, even though there is no incoming initialization event on the Primary side, the event counter Core Cell sends an initialization event (of value 1) from its Primary level output.

The reason for this are the details of the Core Structure initialization which runs as follows:

1. The shared OBC storage of the *Read*/*Write* Module pair and the output port states are zeroed.
2. The initialization event is sent from the constant "1" Module. Assuming there is no incoming initialization event from the Primary level at the Core Cell's input, there are no other initialization event sources in the Structure.
3. In response to the incoming event from the constant Module the adder computes the sum of the input values, where the lower input value is obtained from the just zeroed output of the *Read* Module. Thus, the adder outputs a value of 1.
4. The value of the sum is written into the OBC storage and is sent to the Core Cell's output.

Such problems can be avoided by using latches (discussed in section ↑4.5, Clocks and Latches) and/or Modulation Macros (discussed in section ↑4.6, Modulation Macros).

## 4.5 Clocks and Latches

A factory library *Latch* Macro is available under *Library > Memory > Latch*:

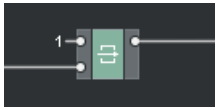
A Latch Macro.

> An integer version of the latch is available in the same menu under the name *ILatch*.

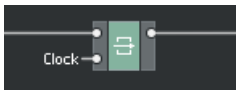
The upper input of the *Latch* Macro is the value input. The lower input of the *Latch* Macro is the event or the **clock** input. The output combines both into one event where the value is taken from the upper input and the clock or the triggering is taken from the lower input.

The *Latch* Macro can be used for two different purposes:

- Replacing a value of an event with another value:



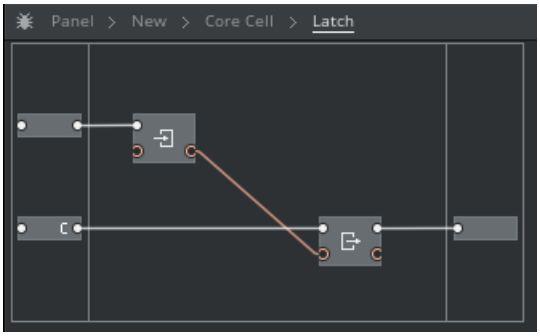

- Delaying a sent value until a certain event (clock) occurs:




Both purposes simply represent two different viewpoints on one and the same process.

> The second usage purpose (delaying a value) is the reason for the *Latch* Macro's icon and for the alignment of the *Latch* Macro's ports.

The internal Structure of the *Latch* Macro is very simple:




The internal Structure of the Latch Macro.

In this Structure it is imperative that the *Write* Module precedes the *Read* Module in the OBC chain. This ensures that if the value and the clock events arrive simultaneously at the *Latch* Macro's inputs, the value will be used to generate the output event.

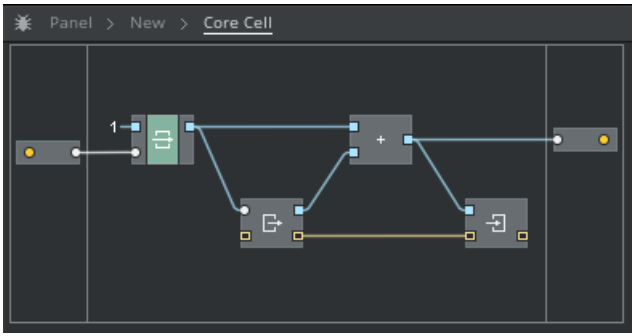
> The opposite ordering (the *Read* Module preceding the *Write* Module) is used to build the one-sample delay ($z^{-1}$) Macro discussed in section ↑4.10.4, Resolution Mechanism.

The compiler recognizes the '*Write* Module followed by a *Read*' Module OBC pattern and treats it in special, optimized way. Depending on the circumstances, using a *Latch* Macro at a given position in a Structure will produce more efficient code than without using it. As a general rule, the Core Structures should employ latches where logically appropriate without attempting to minimize their amount.

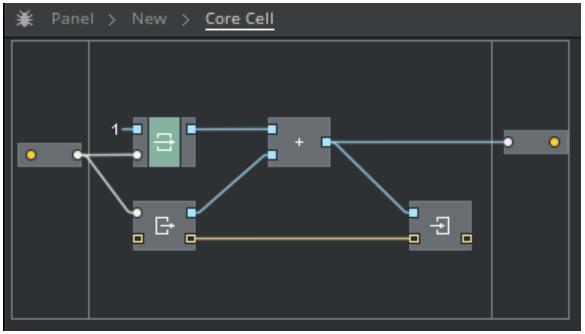
## Usage Examples

The faulty event counter discussed in section ↑4.4, Initialization could have been fixed in one of the two following ways using the latches.

- Use an integer event accumulator Structure (similar to the float event accumulator discussed in section ↑4.3, Object Bus Connections (OBC)) and use an integer latch (*ILatch*) Macro to replace the values of the incoming events with 1:




- Start with the faulty Structure discussed in section ↑4.4, Initialization and use a *Latch* Macro to make sure that the output event of the constant Module does not trigger the addition:




Notably, the two Structures are pretty much identical, differing only by the specific pickup points of the clock signals for the *Latch* and the *Read* Module.

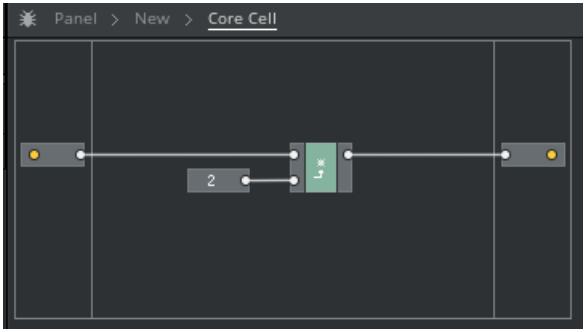
> The clock inputs completely ignore the incoming signal's value. The Core compiler is aware of that and drops out the respective computations in the incoming signal's path if these values are not needed otherwise. Particularly, the compiler will drop a value conversion from int to float if an int signal is used at a float clock input. For that reason it does not matter whether a clock input is configured to int or float type. The convention used in Core however is that all clocks are float.

The most appropriate way to fix the event counter is by using Modulation Macros, discussed in section ↑4.6, Modulation Macros.

## 4.6 Modulation Macros

The basic math operation Modules follow the standard Core triggering convention: any input triggers an output event. Sometimes a different convention is desired, where some inputs should not trigger the output. This convention is implemented in the Modulation Macros, found under *Library > Math Mod* (for Math Modulation).

The following alternative implementation of the event doubler Core Cell from section ↑4.2, Processing Order illustrates the concept of Modulation Macros:




An alternative implementation of the event doubler Core Cell using a modulation multiplier.

The Macro in the middle is the modulation multiplier, available under *Library > Math Mod > x mul a*. If one of the usual built-in multipliers had been used instead, the Structure would have had the same bug (for exactly the same reason) as the event counter in section ↑4.4, Initialization. With the modulation multiplier, the constant Module does not trigger the addition, thus the Structure does not have a bug.

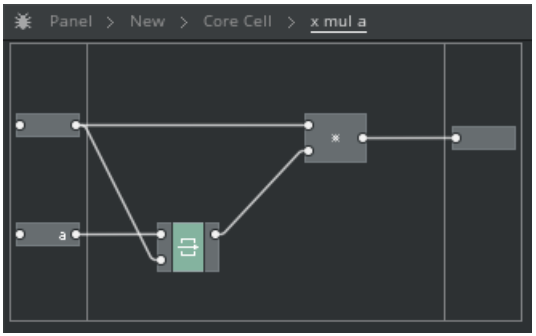
The name 'modulation multiplier' comes from the use case where the amplitude of an event stream is being modulated by some other 'gain' signal. Clearly, in this case it is not desired that the events of the 'gain' signal trigger the multiplication:


Modulation of the amplitude of an event stream

This use case is also the reason for the icon and the port alignment of the Macro. The arrow in the icon corresponds to the modulation signal input. Additionally, in the name of the Macro and in the labels of the ports the modulation inputs are denoted with the letters from the beginning of the Latin alphabet ("a", "b", ..).

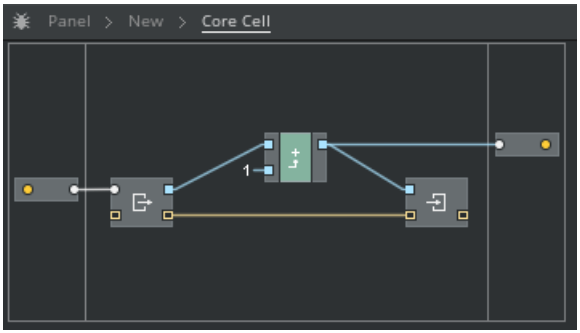
Occasionally the opposite to the modulation concept is used: the *tapping*. While the modulation inputs are inputs that are explicitly declared as non-triggering, a tapping input is explicitly declared as triggering (whereas the remaining inputs are assumed non-triggering by default).

The internal Structure of the modulation multiplier is using a *Latch* Macro to prevent the lower input from triggering the computation:




The internal Structure of the modulation multiplier.

Another good illustration of Modulation Macro usage is yet another way of fixing the event counter of section ↑4.4, Initialization, besides the two bug-fixing approaches discussed in section ↑4.5, Clocks and Latches. The fix is using an integer modulation adder, which is available under *Library > Math Mod > Integer > Ix + a*:




A bug-fixed event counter using a modulation adder.

## 4.7 Arrays

REAKTOR Core supports arrays as a special kind of OBC storage. While ordinary OBC storage is implicitly defined by the mere fact of using it (by reading or writing), the arrays need to be explicitly defined by placing an array Module into the structure. There are two array flavors:

- Writable array (available under *Built-In Module > Memory > Array*): The number of array elements is specified in the Properties. As with all writable memory, writable arrays are zero-initialized upon initialization.

- Read-only array or *Table* (available under *Built-In Module > Memory > Table*): The table is supposed to be prefilled by the Structure builder using the Table Editor available from the *Table* Module's Properties.

The array memory is allocated per voice. Thus an array of 1024 floats in a 16 voice setting needs a total memory for 16*1024=16384 floats, which, considering that each float needs 4 bytes of storage, is equal to 64 Kbytes.

The read-only tables are allocated just once and are not multiplied with the number of voices. Furthermore, multiple tables sharing exactly identical sets of data are internally combined into a single table. Thus having multiple copies of the same table Module containing a large table does not noticeably increase the memory usage.

The output port of the *Array* or *Table* Modules is of a special **array OBC** type (which is further subdivided into int and float types with variable float precision). Thus a *Read* or a *Write* Module cannot be directly connected to an array. In order to be able to read or write into an array, a single array element needs to be selected first for access.

A Macro port of the Latch class can be configured to the array connection mode in the Properties.

### 4.7.1 Array Indexing

The array element selection is performed by using the *Index* Module (*Built-In Module > Memory > Index*):

An Index Module.

- The lower input of the *Index* Module is of the array OBC type and is supposed to be connected (directly or via some Macro ports and other connectivity features) to the array to be accessed.
- The output of the *Index* Module is of the usual (latch) OBC type and is supposed to be connected, directly or indirectly, to one or more *Read* and/or *Write* Modules.
- The upper input of the *Index* Module is of scalar int type. An incoming event at this input causes a selection of an array element of the corresponding index for the OBC output of the Index Module.

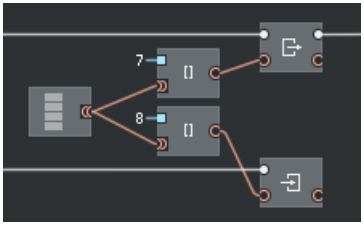
The indexing is zero-based. That is the valid index range is from 0 to N-1, where N is the array size.

The result of accessing the array elements out of the valid index range is unspecified, except that such accesses should not cause memory corruption or crash REAKTOR.

In the current implementation of the Core compiler the safety of the out of range index accesses is ensured by internally clipping the index value (in the unsigned integer mode). However, as with other unspecified behavior, the specific details of this safety mechanism should be never relied upon. Correctly built REAKTOR Core Structures should never use out of range index values.

Sometimes it is convenient to be able to automatically determine the size of an array. This function is provided by the *Built-In Module > Memory > Size []* Module. The Module outputs the total number of the array elements.

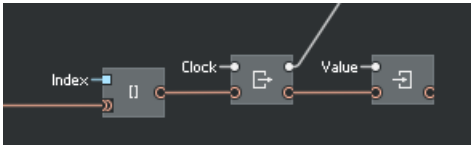

The *Index* Module rarely should be used directly. Typically *Read []* and *Write []* Macros should be used instead. For the sake of illustration of the Index Module's functionality the following two cases are presented:




Usage of raw Index Modules to access array elements (unrecommended).

In the above picture the two *Index* Modules are used to access the array elements with the indices 7 and 8 respectively, where the 7th element is accessed for reading and the 8th element for writing.

In principle, it is possible to attach a *Read* and a *Write* Module to the same *Index* Module, e.g. in series. In this case they are obviously accessing the same array element.



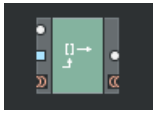

A Read and a Write Module sharing the same Index Module (untypical and unrecommended).

Neither of the two cases is a recommended way to access the array.

### 4.7.2 Read [] and Write [] Macros

The array accesses typically should be done by using the *Read []* and *Write []* Macros available under *Library > Memory*.

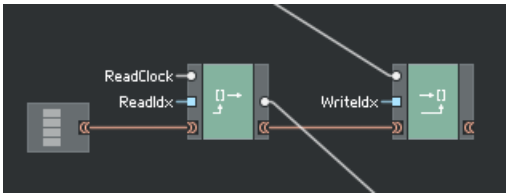
- The *Read []* Macro has a clock input, an index input and an array OBC input. The clock input triggers the array reading operation at the specified index, sending the read value at the upper output of the Macro.



- The *Write []* Macro has a value input, an index input and an array OBC input. An incoming value at the write input causes a write operation at the respective index.

It would have been possible to implement the *Read []* Module without the clock input, by using the event at the index input as the triggering signal for the reading operation. Besides being inconsistent with the *Write []* Macro this would have caused unnecessary CPU overhead if the reading is performed without an actual change of the index. With the index input being separate, the (rather small) CPU overhead associated with the indexing is occurring only in response to the events at the index input.

The array OBC inputs and outputs of the *Read []* and *Write []* Macros are supposed to be used to order the array accesses relatively to each other:




An example of the ordering of read and write accesses to an array.

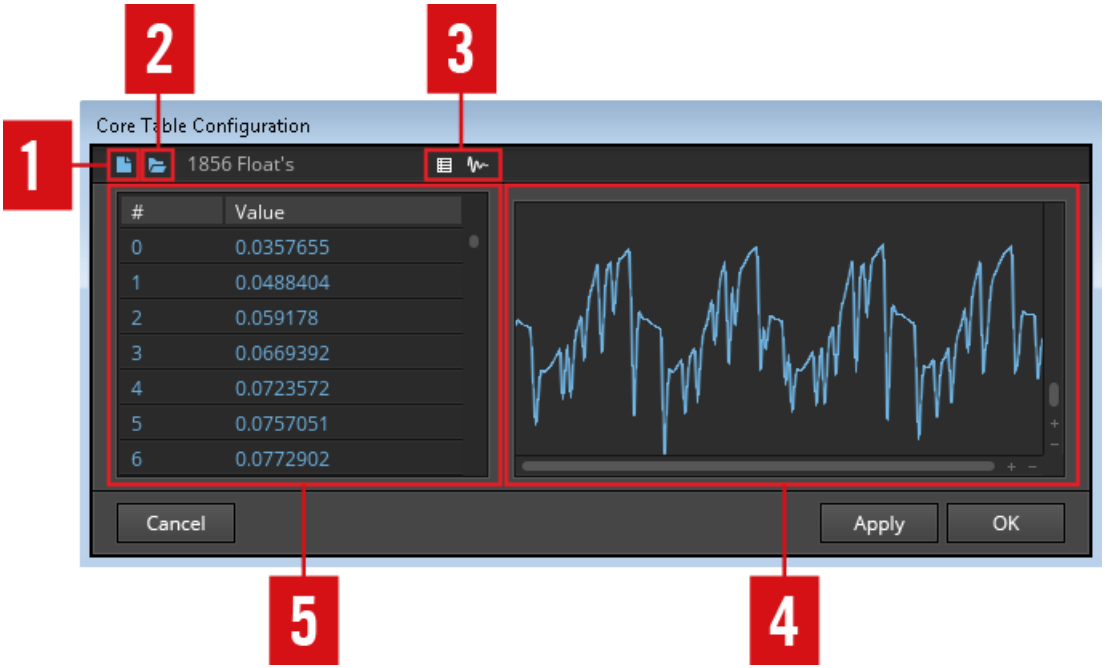
### 4.7.3 Table Specifics

The Table Module works pretty much as an usual array, except that writing into the table is not possible. The tables also have specific properties.

- FP Precision: In case the table contains float data, this configures the precision of the outgoing float OBC connection type (and thus the precision of the compatible *Read* accesses). The precision of values internally stored in the table is not affected by this setting.

  The *Read []* and *Write []* Macros can be connected to a table with a non-default FP precision by changing the FP precision setting of the *Read []* and *Write []* Macros.

- Table Editor: Calls up the table editor, which allows to fill the table with values.




The Table editor.

(**1**) **Create/Resize Table:** This button allows you to configure the table's data type and size and to pre-fill it with a specified initial value.

(**2**) **Import Table Data**: This button allows you to load the table from a file. A number of audio and text file formats are supported, where the latter can be used to import special numeric tables prepared in other software.

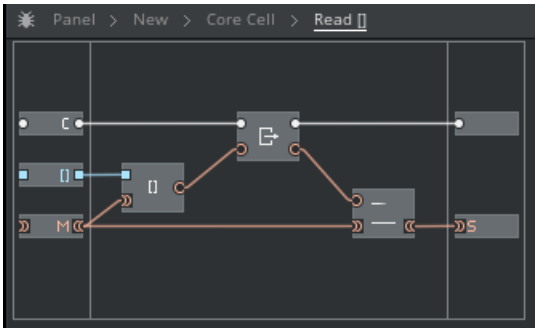
(**3**) **Layout**: These buttons turn the numeric display and the wave display on or off.

(**4**) **Wave display**: The wave display displays the contents of the table as a wave.

(**5**) **Numeric display**: The numeric display can also be used to edit the individual values in the table.

### 4.7.4 Advanced OBC Ordering

Some advanced OBC ordering topics can be nicely illustrated by the internal Structure of the *Read []* Macro:

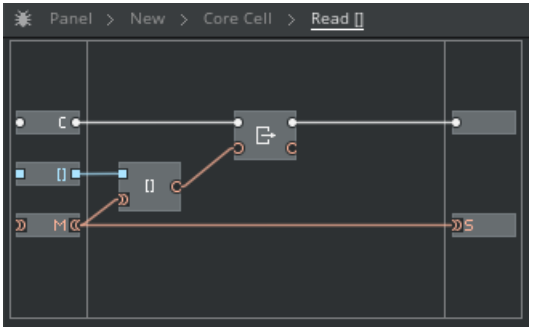



The internal Structure of the Read [] Macro.

The function of the *Index* and *Read* Modules is clear. The third Module (the one next to the OBC output of the Macro) is the *R/W Order* Module (available under *Built-In Module > Memory > R/W Order*).

> The *R/W Order* Module is also used inside the *Write []* Macro at the similar position.

Upon the first look this Module might seem to be completely redundant, because the *Read []* Macro could have been implemented without it:




A faulty implementation of the Read [] Macro without R/W Order.

However, in this case the OBC output of the Macro is not ordered relatively to the *Read* Module inside the Macro. This means that Modules and Macros (such as e.g. a *Write []* Macro) connected to this OBC output will not be ordered relative to the reading operation.

The *R/W Order* Module inside the *Read []* Macro does nothing more than providing a formal ordering between the *Read* Module and the OBC output of the Macro, due to the mere fact that the *Read* Module is connected to its upper input. The lower input of the *R/W Order* Module simply forwards the incoming OBC connection to the output.

The *R/W Order* Module can also be used for explicit ordering of non-array OBC connections, although there such need is much less likely to occur.

The need for the explicit ordering by the *R/W Order* Module is not affected by the setting of the Solid property of the containing Macro. This might be improved in future versions of REAKTOR Core (without loss of compatibility with older Structures).

## 4.8 Routing and Merging

The *Router* Module (*Built-In Module > Flow > Router*) is used to direct the event flow to one of two alternative paths, where the event flow direction can be switched at runtime:

A Router Module.

Depending on the router state (controlled by the upper input) the events arriving at the lower input are directed either to the upper output or to the lower output. The upper input (the control input of the router) is of BoolCtl type and can be in the **true** or **false** state.

- If the control input is in the **true** state then the events are routed to the upper output.
- If the control input is in the **false** state then the evens are routed to the lower output.

The BoolCtl connection does not transmit any events. It only transmits the state but no triggering information. Therefore it is not a signal in the same sense as the scalar type class.

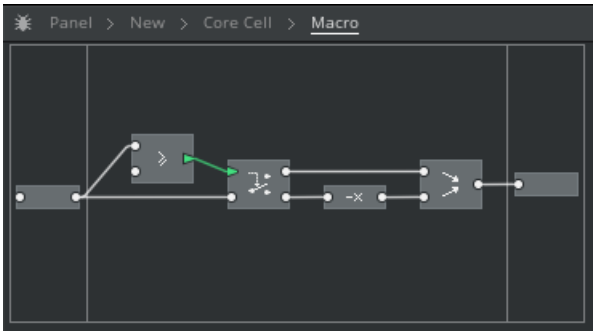
The BoolCtl sources are provided by *Compare*, *Compare Sign* or *ES Ctl* Modules in the *Built-In Module* > *Flow* menu.

The comparison criterion of the *Compare* and *Compare Sign* Modules can be adjusted in their Properties.

The *Compare Sign* Module is useful for detecting zero (and other thresholds) crossings.

There are no Boolean operation built-in Modules to combine different BoolCtl connections together. Use the Macros from *Library* > *Logic* (at the cost of a somewhat higher CPU load) if you need standard Boolean logical operations.

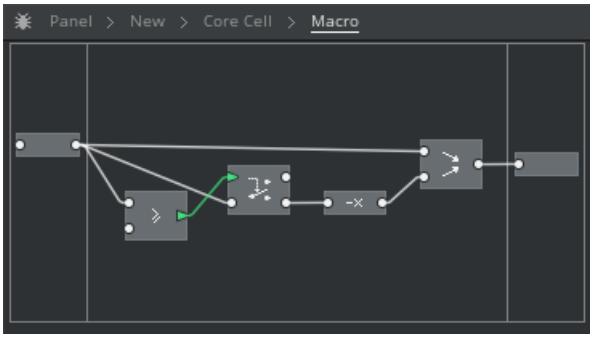
A signal split by a *Router* Module can be merged back by the *Built-In Module* > *Flow* > *Merge*:




Merging the two paths back. The Macro Structure implements the absolute value functionality.

The *Merge* Module simply lets all incoming events go through to the output. If the events arrive simultaneously at both inputs, the lower input has priority.

Instead of merging the two paths of the split signal, another common pattern is to merge back with the original signal:




Merging back to the original signal. The Macro Structure implements the absolute value functionality.

Since the lower input of the *Merge* Module has higher priority, in case the signal goes through the inversion path, the inverted signal at the lower input of the *Merge* Module has priority over the original (unrouted) non-inverted signal.

The *Merge* Module does not need to be used exclusively in combination with routers. One can merge arbitrary events.

In the current implementation of the REAKTOR Core technology this kind of 'chaotic' merging (as compared to merging the split paths back together or to their original source) might overload the compiler. Therefore 'chaotic merging' needs to be used with some care. In cases of large amounts of chaotic merging, the compilation time may grow excessively. Event reclocking by using latches and Modulation Macros often helps (see section ↑5.6.3, Routing and Merging).

The math processing Module such as the built-in adder, multiplier etc. technically perform event merging. The difference to the *Merge* Module is solely in how the output value is computed. Therefore the math processing Structures are subject to the same 'chaotic merging' considerations.

The *Merge* can be made to have more than 2 inputs by configuring the number of inputs in its Properties. The priority rule for simultaneous events is the same: the priority is highest at the lowest input. Intuitively this can be imagined as the events being locally ordered from the top input

to the bottom one, so that the lowest event is the one which arrives latest of all and thus overrides the value prepared for the output.

## 4.9 SR and CR Buses

A number of signal processing Modules need to be clocked by some regular trigger sources. The typical examples are oscillators, filters, envelopes and LFOs. REAKTOR Core provides default clock sources for such Modules. There are two sources provided in the form of Bundle Scoped Buses with two different names:

- **SR** (for Sampling Rate): The bus with this name is the source of the audio clock.
- **CR** (for Control Rate): The bus with this name is the source of the control clock.

The SR bus is intended to be used for audio processing Modules like oscillators and filters, while the CR bus is intended to be used for control processing Modules like envelopes and LFOs. However, this SR/CR distinction is purely conventional and is made with the sole purpose of being able to provide different audio and control clocks 'by default'. There is always a possibility of overriding the default connection and/or providing even more clocks by defining further buses with different names and the same Structure.

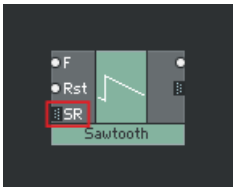
The default CR source in REAKTOR Core is identical to SR signal-wise, runs at the audio rate and has nothing to do with the Control Rate source in the REAKTOR Primary. If desired, the Primary Control Rate can be 'imported' into the CR bus by using the *Library > Clk Bundle > Control > CR From Prim* Macro.

Both SR and CR bus Bundles have identical internal Structure and theoretically can be used in place of each other. Practically this is usually not recommended. Particularly, the conversion between different clock rates needs to be different for audio and control signals.

The default built-in SR and CR buses are provided 'one level higher' than the top Structure of the Core Cell, thereby allowing to redefine them already within this top-level Structure without causing a definition conflict. If such redefinition is done, the 'previous' (default) bus has to be picked up using the parent-mode scoped accessors, as usually. See ↑4.9.4, CR rate change and ↑4.9.5, SR Rate Change for the details of redefining SR/CR.

### 4.9.1 Connections to the Clock Buses

A Core Macro Library Module that needs an SR or a CR connection has by convention a Bundle input, labelled SR or CR respectively:




The SR input of an oscillator.

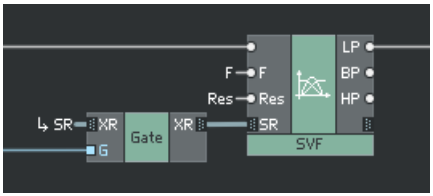
> The SR/CR input convention is not used for the low-level library Macros. Particularly, the *z^-1 ffd* and *z^-1 fbk* Macros connect themselves to the SR bus, even though they do not have an SR input.

If the SR or CR input is not connected, it automatically picks up the SR or CR bus. In the case of the above oscillator, in the default unconnected state of the SR input, this input will be automatically picking up the SR bus. Should this default connection be undesired, some other clock Bundle source needs to be manually connected to this input. This automatic connection convention is employed for effectively all signal generating and processing Modules in the Core Macro Library.

### 4.9.2 Clock Gating

Sometimes it is desired to turn off audio or control processing within a given part of the Structure. This can be achieved by **gating** the clock.

In the following Structure the audio clock is gated for a 2-pole SVF filter:




SR clock gating.

The *XR Gate* Macro (*Library > Clk Bundle > XR Gate)* on the left interrupts the SR clock as long as the gate signal G is zero. As soon as G is turned to 1 the clock begins to run.

> Upon turning from 0 to 1, the *XR Gate* sends a reset signal (see section ↑4.9.6, Internal Structure of Clock Buses) along the clock Bundle, causing the connected Modules to reinitialize their internal states if necessary. Should the reset signal be undesired in this case, use an *XR Freeze* Macro instead.

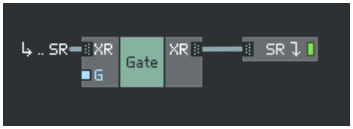
The "XR" stands for either "SR" or "CR". That is, the XR Macros are equally usable for both SR and CR Bundles. Some other clock Bundle Macros are usable only for SR or only for CR Bundles. Those Macros are located under *Library > Clk Bundle > Audio* and *Library > Clk Bundle > Control* respectively.

The *XR Gate* Macro only gates the clock, but does not cause the outputs of the connected audio- and control-processing Macros, such as the LP output of the filter in the above picture, to be zeroed (this is done for the CPU efficiency reasons, associated with the event merging issues discussed in section ↑5.6.3, Routing and Merging). It is the builder's responsibility to avoid picking up the 'frozen' output values by some other running parts of the Structure downstream.

The automatic default connection of SR/CR input is implemented only for the signal generating and processing Modules like oscillators, filters, envelopes, LFOs, etc. The Modules dedicated to the handling of SR/CR bus themselves (such as clock gates, dividers, etc. (located in *Library > Clk Bundle > Control*) always need to be connected explicitly.

### 4.9.3 SR and CR redefinition

If a whole area of the Structure is being controlled by a gated clock, then instead of wiring up the output of the *XR Gate* Macro to each of the clocked Modules, it is convenient to put all such gated Modules into one Macro Structure and locally redefine the clock bus within this Structure:

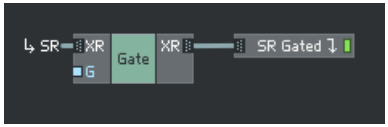



Redefining the SR bus.

Notice the parent-mode accessor usage at the input of the *XR Gate*, which ensures that the higher-level SR bus, rather than the one defined on this level, will be picked up.

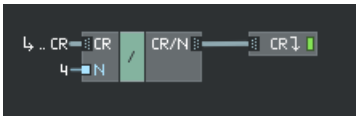
Another version of the same approach is discussed in section ↑5.2, Generation of Audio and Other Regularly Clocked Events.

Some kind of an 'in-between' approach is to define a Scoped Bus (or a QuickBus) with a different name and explicitly connect to this bus. Notice that there is no need for the parent-mode accessor at the *XR Gate*'s input in this case.




### 4.9.4 CR rate change

In certain advanced Structure building scenarios one control rate is insufficient. Or the default control rate, which is equal to the audio rate, may be considered too high for the builder's purposes. In this case the CR rate division may be employed to generate lower control rates. E.g. the following Structure fragment employs the *CR Div* Macro (*Library > Clk Bundle > Control > CR Div)* to locally redefine the CR bus to a 4 times lower rate:

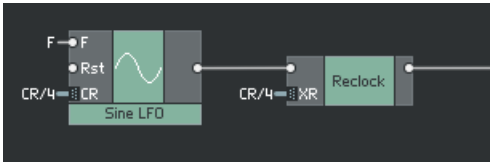



Redefining the CR bus to a 4 times lower rate.

Of course, the other discussed possibilities to connect to the new CR (including explicit wiring, QuickBuses and differently named Scoped Buses) can be used as well.

## Smoothing

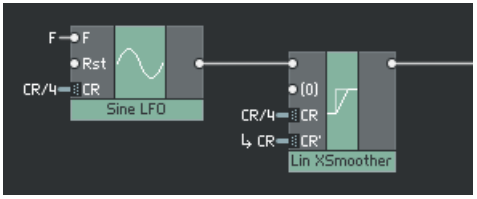
Converting a signal from a higher control rate to a lower control rate can be done by the *XR Reclock* Macro (*Library > Clk Bundle > XR Reclock)*:




Reclocking control signal to a lower rate.

> In principle the *XR Reclock* Macro can be also used for control signal upclocking or audio up- or downclocking. Usually, however, this will create undesired artifacts that can be avoided by using other means instead.

Conversion from a lower control rate to a higher control rate is achieved by **smoothing**. A linear cross-control rate smoother is provided by the *CR Lin XSmoother* Macro (*Library > Clk Bundle > Control > CR Lin XSmoother)*:



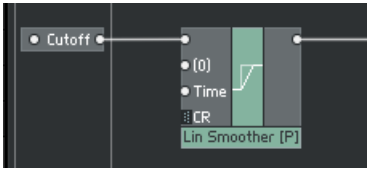

Smoothing from a lower control rate to a higher one.

The *CR Lin XSmoother* Macro is designed to be used by hard-locked control rates, the ratio of which is an integer number. E.g. in the above picture the CR/4 control rate is obtained by applying the *CR Div* Macro to the CR control rate. **If this condition is not met, the smoother will not work correctly.**

> The "(0)" input of this and other smoothers is intended to provide an initial value for the smoother. If the input signal of a smoother is clocked, then according to the clocking conventions (see sections ↑4.9.6, Internal Structure of Clock Buses, ↑5.2, Generation of Audio and Other Regularly Clocked Events and ↑5.3, Processing of Audio and Other Regularly Clocked Events) the input signal cannot arrive during the initialization or a Structure reset. Thus, it will be not possible to use the input signal to specify the initial state of the smoother. One way around it is to merge in a dedicated initial value signal using a *Merge* Module in front of the smoother's input. However this is a kind of hidden violation of the mentioned clocking convention. Instead, one should connect the initialization signal to the "(0)" input of the smoother. If the input signal is not following the clocking convention (like e.g. a signal which comes from a knob) and provides a reasonable initial value setting for the smoother, then the "(0)" input can be left disconnected. The smoother will use the input signal as the initial value source in this case.

## Irregular Permanent Smoothing

Smoothing can also be applied to convert an irregularly clocked signal (or a signal which comes completely sporadically, such as a stream of values from a panel knob) to remove the steps in the signal. Such smoothing will produce a stream of values clocked at some control rate. Depending on the specifics of the particular smoother, however, the signal events might not be sent all the time, but only until the destination value is reached by the smoother. E.g. in the following Structure the "Cutoff" knob's value at the Macro's input is being smoothed by a *Lin Smoother [P]* Macro (*Library > Control > Smoother > Lin Smoother [P]*):




Permanent smoothing of an irregular control event stream.

The *Library > Control > Smoother > Lin Smoother [P]* Macro is not considered a CR-handling Macro, since it does not modify the CR Bundle or explicitly convert from one rate to another. For that reason it is located under *Library > Control* rather than *Library > Clk Bundle > Control*. For the same reason it automatically connects to the CR scope bus.

The *Lin Smoother P* Macro generates a permanent stream of output events at a rate defined by CR bus ("P" for Permanent). Whenever there is a new event coming from the "Cutoff" input port Module, the smoother generates a ramp of duration specified by the Time input (in seconds) from the old value to the new value.

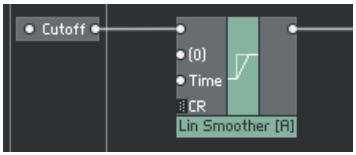
If the Time input is not connected, there is some preconfigured default. The "(0)" input has the same function as for *CR Lin XSmoother*.

## Irregular Automatic Smoothing

Imagine the control signal processing Structure downstream from a smoother. The processing in this Structure may follow different conventions, such as the following:

- The control signal processing Structure is clocked by the CR clock.
- The control signal processing Structure is unclocked; the processing is triggered by the incoming control signal events.

In the latter case it might be undesirable that the smoother is generating the output events all the time, thereby causing the control signal processing to run all the time as well. Rather, the smoother should only generate the output events until the target value is reached and then stop. This behavior is implemented by the *Lin Smoother [A]* Macro (*Library > Control > Smoother > Lin Smoother [A]*, "A" for Automatic), which differs from *Lin Smoother [P]* only in exactly this aspect:

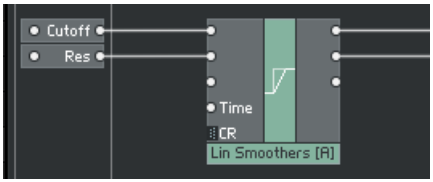



Automatic smoothing of an irregular control event stream.

## Irregular Batch Smoothing

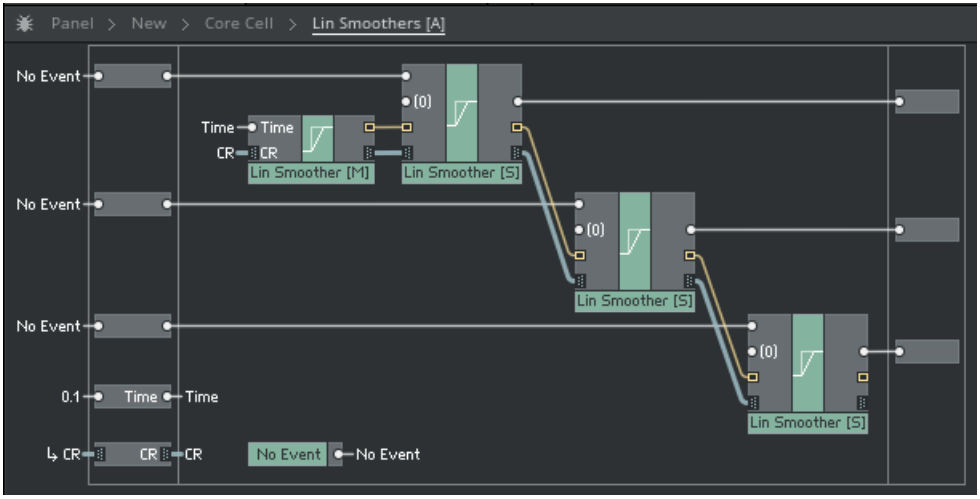
The automatic stop option however still requires a runtime check on every control clock in order to determine whether the smoother is active or not. Such runtime check also consumes CPU time. Therefore it is a good idea, if one has a group of several of such smoothers, to group them together, performing one single runtime check for the entire group.

This approach is implemented by the *Lin Smoothers [A]* Macro (*Library > Control > Smoother > Lin Smoothers [A]*):




Batch smoothing of two irregular control event streams.

Internally the *Lin Smoothers [A]* Macro contains a 'master' smoother (containing the master check) and a chain of 'slave' smoothers:




The internal Structure of Lin Smoothers [A] Macro.

The slave smoothers are the ones doing the smoothing itself and can be added or removed by the user of the Macro according to the number of signals which need to be smoothed. The "No Event" default for the Macro inputs prevents the unused inputs from initiating a smoothing ramp upon reset. The maximum number of slave smoothers supported by the current implementation is 32, which should be sufficient for most purposes.

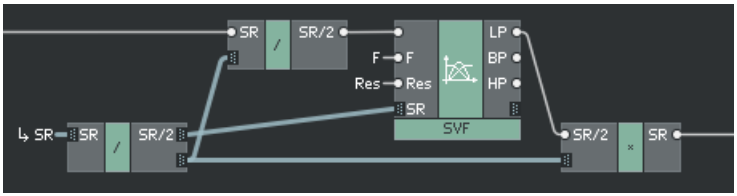
> Sticking too many smoothers together in this fashion will produce some small overhead, due to the fact that the per-smoother runtime checks will be performed for all smoothers as long as at least one of the smoothers is active.

### 4.9.5 SR Rate Change

A conversion between audio signals of different rates is done differently from control signals. While control signals can be naively downsampled and smoothed upon upsampling, the audio signal resampling requires special kinds filtering techniques.

> In the current version of REAKTOR Core the clock rates can be only lowered, not raised. Thus, oversampling is currently not possible. The rate conversion Macros are of course available in both directions: downsampling (used upon entering the downsampled area) and upsampling (used upon leaving the downsampled area).

Thus, a set of library Macros provided for the SR rate change is different from those for CR:



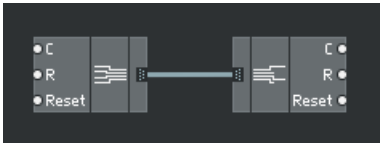

A locally downsampled SVF (State-Variable Filter)

The *SRdiv2 Master* Macro (*Library > Clk Bundle > Audio > SRdiv2 Master)* on the very left is the downsampling master, which is driving the downsampling converter *SRdiv2 In* on the left and the upsampling converter *SRdiv2 Out* on the right. Thus the filter in between is running at halved sampling rate. You can drive multiple *SRdiv2 In* and *SRdiv2 Out* converters with the same master.

Often it might be reasonable to put the entire downsampled area inside a dedicated Macro. In this case it is useful to locally redefine the SR bus to the downclocked one using the same trick as it was explained for the CR bus in ↑4.9.4, CR rate change. Note that you still need to explicitly resample all incoming and outgoing audio signals of the downsampled area by using SRdiv2 In and SRdiv2 Out Macros.

## 4.9.6 Internal Structure of Clock Buses

A clock bus is a Bundle containing three different fibers named "C", "R" and "Reset". For the convenience of the builders the Macro library contains preconfigured Bundle *Pack* and *Unpack* Modules for the SR and CR buses under *Library > Clk Bundle > Audio > XR Pack* and *XR Unpack*:




Preconfigured Pack and Unpack Modules for the SR/CR Bundles from the library

The details of each of the fibers are described below. The examples of the low-level usage of the clock buses can be found in sections ↑5.2, Generation of Audio and Other Regularly Clocked Events and ↑5.3, Processing of Audio and Other Regularly Clocked Events.

### "C" Fiber

This is the clock fiber which sends regular events at the clock rate. The audio processing Macros are supposed to send the output audio events only in response to the clock events. Same holds for regularly clocked control events.

The convention requires that no event is sent on the C fiber during the initialization. This particularly means that the audio signal outputs and clocked control outputs of Core Macros are not sending during the initialization.

The value of the C fiber is unspecified and should not be relied upon.

### "R" Fiber

This fiber transmits the clock rate in Hz.

The convention requires that the R event is sent during the initialization. Afterwards it is sent as needed.

The reason for this convention is to simplify the event processing in the use cases of the R fiber. Since the R fiber is guaranteed to send the initialization event, it can be safely used as a divisor in the computation of oscillator increment steps, normalized filter cutoffs, and so on.

The events on the R fiber may cause additional computation overhead in the consumer Structures, as they are reconfiguring themselves to a new rate, and should not be sent indiscriminately, but rather only when the rate changes or is likely to change (filtering out the duplicate values sent on this fiber is probably a little excessive, while sending events here at a regular rate is a highly questionable practice).

### "Reset" Fiber

This fiber transmits a zero-value reset event. The consumers of the SR Bundle are supposed to reset their state (if necessary) in response to this event. E.g. a filter should reset its state, while a free-running oscillator does not have to do anything in response to this event.

The convention requires that the Reset event is sent during the initialization. Afterwards it is sent as needed.

The convention requires that the Reset event always has a zero value.

The convention specifies the Reset event semantics as 'starting again without the past'. E.g. a Reset event can be sent upon resuming a paused clock to cause the Structures (such as e.g. filters) to reinitialize their states.

The reason for sending the Reset event during the initialization is to simplify the initialization of the Structures, which otherwise might need to take special measures to make sure that the same initialization steps are performed during the initialization and during further resets. The reason for the zero value of the Reset event is that a typical response to the Reset event is to zero the states, therefore the Reset event can be directly stored to the state memory of the respective Structures.

## 4.10 Feedback Connections

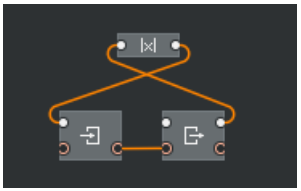
### 4.10.1 Automatic Resolution

REAKTOR Core allows the usage of the feedback loops in the Structures, provided each feedback loop contains at least one scalar (float or int) wire. The loop will be handled by automatically placing an implicit one-sample audio delay somewhere within the loop,

'Audio' means that this implicitly inserted delay will be clocked by the SR.C clock obtained from the current scope. The delay also responds to SR.Reset by zeroing its memory.

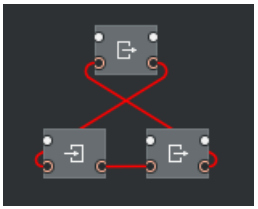
It is not guaranteed that REAKTOR Core will use at maximum one implicit one-sample delay per detected feedback loop. In some cases it is possible that more delays will be inserted. For exact control over the feedback loop resolution, one needs to use *z^-1 fbk* Macros explicitly (see section ↑4.10.2, Manual Resolution).

REAKTOR Core highlights the feedback loop visually by turning the connection color to orange:



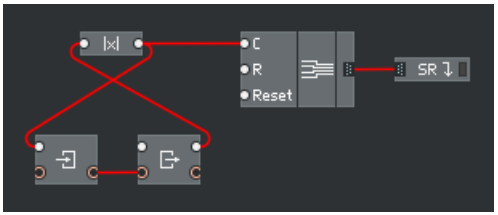
A highlighted feedback loop.

If the loop does not contain any scalar wires then it is an error, which is highlighted in red:



A purely non-scalar feedback loop.

Equally, if an SR bus definition is contained in or depends on a feedback loop, such loop is an error, even if it does contain scalar wires:

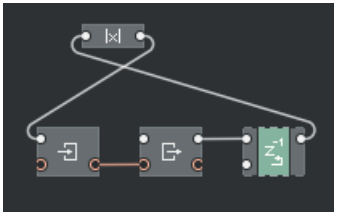



SR bus dependent on the feedback loop

> The reason for this is that the implicit audio delay used for the feedback loop resolution needs access to SR.C and SR.Reset, which creates another feedback loop after the resolution of the first one. See section ↑4.10.4, Resolution Mechanism for the details of the delay's Structure.

## 4.10.2 Manual Resolution

If exact control over the feedback loop resolution is needed, a *z^-1 fbk* Macro (*Library > Memory > z^-1 fbk)* has to be used. In this case the compiler does not highlight the loop anymore:



The use of the z^-1 fbk Macro to explicitly resolve the feedback loop.

The *z^-1 fbk* Macro implements the same one-sample delay Structure which is used by the automatic resolution. Differently from the automatic resolution, by using this Macro the builder can exactly specify the position of the one-sample delay.

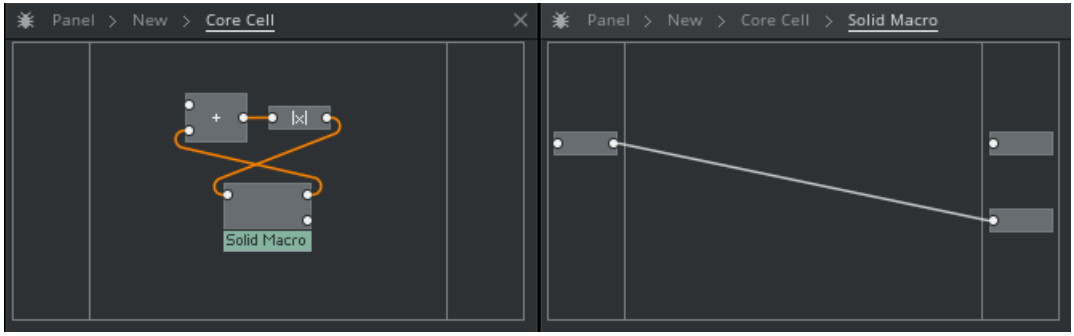
The *z^-1 ffd* Macro (*Library* > *Memory* > *z^-1 ffd*) is not intended and cannot be used for the feedback resolution. Its intention is implementation of feedforward 1-sample audio delays.

Neither of the above two Macros is supposed to be used for other purposes than audio rate 1-sample delays. For delaying other kinds of events by one tick use the *Latch[-1]* and *Event[-1]* Macros.

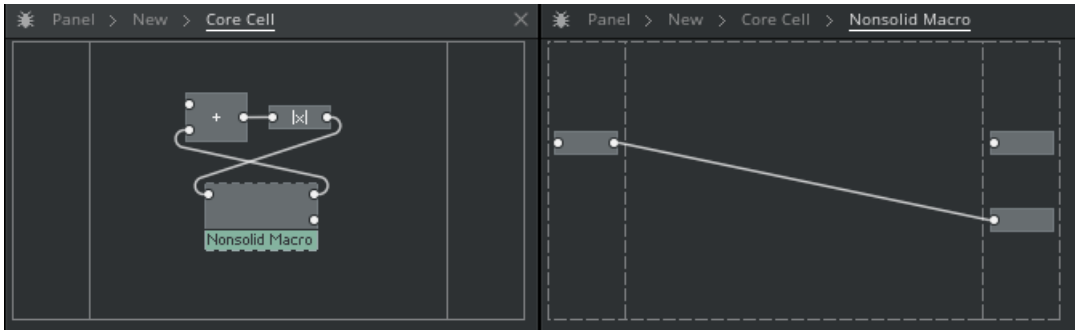
### 4.10.3 Nonsolid Macros and Feedback Loops

The Solid property setting of the Macro is affecting how the Macro is treated in respect to the feedback loops.

- **Solid Macros** 'cannot be entered' by feedback loops. This means that, regardless of the internal contents of the Macro, a feedback loop around this Macro is always a feedback loop. A Macro is treated as if it didn't have any internal Structure and was just a single solid block:



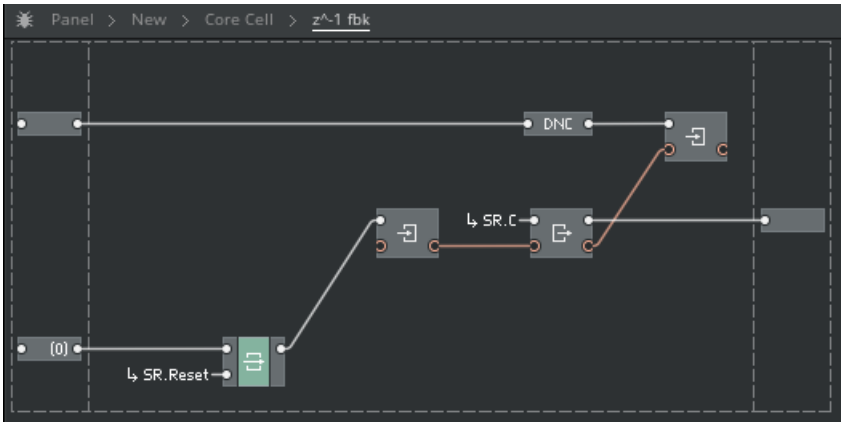
- The boundaries of **nonsolid Macros** are transparent to the feedback loops. If there is no *unidirectional* connection from an input to an output internally inside a nonsolid Macro, there will be no loop:




In particular, the *z^-1 fbk* Macro needs to be nonsolid in order to be able to resolve the feedback.

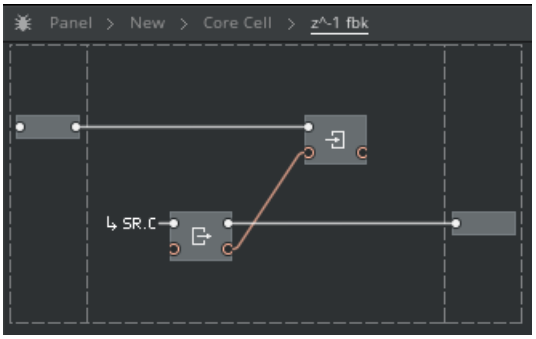
## 4.10.4 Resolution Mechanism

Internally the z^-1 fbk Macro loops like follows:




The internal Structure of the z^-1 fbk Macro.

The *Latch* and the *Write* Modules on the left are responsible for handling the SR.Reset, where the "(0)" input can be used to specify the value to reset to. The *DN Cancel* Module takes care of denormals (see section ↑5.5, Denormals and Other Bad Numbers). The essential part of the Structure is based on the remaining two Modules:




The essential part of the z^-1 fbk Macro's internal Structure.

Since the *z^-1 fbk* Macro is nonsolid, the following two Structures are equivalent:

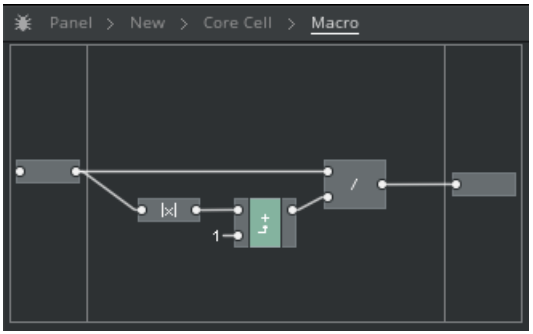

An equivalent expansion of the z^-1 fbk Macro (the nonessential parts of the z^-1 fbk's internal Structure are omitted).

So, formally from the compiler's perspective, there is no feedback loop at all (and thus no highlighting either).

# 5 Building Practices and Conventions

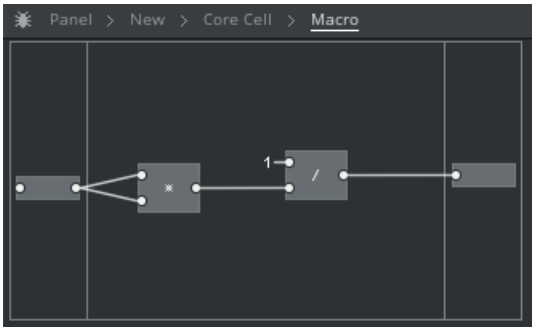
## 5.1 Expression Computation

The simultaneity of event processing in REAKTOR Core makes the computation of single-argument mathematical expressions quite straightforward. One thing to keep in mind is that, as a general rule, it is better to connect the constants to the Modulation Macros:




Computation of the x/(1+|x|) formula by a Core Macro.

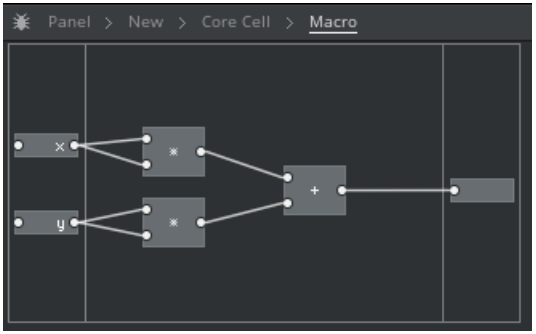
The reason for using Modulation Macros is clearly illustrated by the following Structure:




A questionable implementation of the 1/x^2 formula.

The problem is that if there is no input event during the initialization, the constant at the upper input of the divider will trigger the computation. Since no event has ever arrived at the lower input of the divider, though, the value at this input is zero. Therefore a division by zero will be performed, causing an *INF* value to be sent further downstream (see section ↑5.5, Denormals and Other Bad Numbers for further details on why this should rather be avoided) However, if it is certain that the initialization event will arrive at the lower input of the divider, the above Structure can be used.

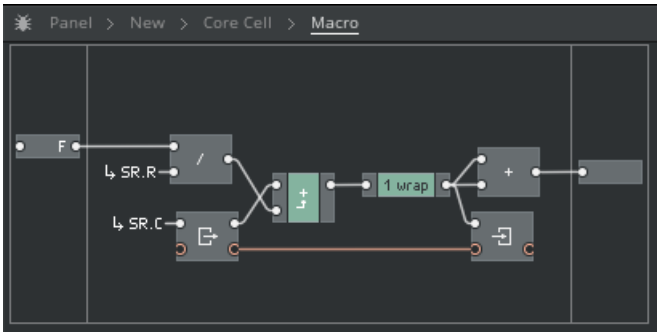
The computation of multiple argument expressions is fairly straightforward as well, except that one needs to watch for potential initialization issues.




Computing x^2+y^2.

## 5.2 Generation of Audio and Other Regularly Clocked Events

The audio-generating Structures typically should be internally clocked by the SR.C:



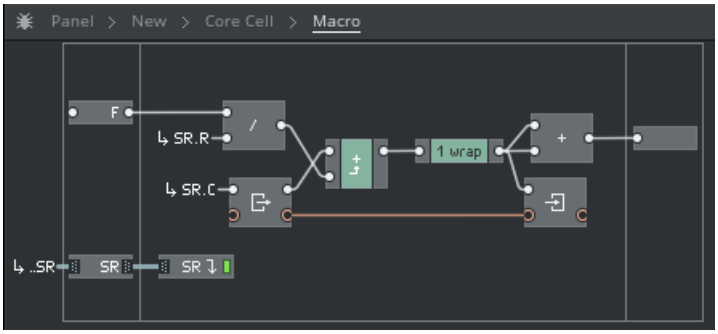

A naïve (non-antialiased) sawtooth generator Macro.

The *1 wrap* Macro used in the above Structure is found under *Library > Math*.

There is a general convention for the audio generating and processing Modules that they should perform their 'sample tick' computations and send output events **in and only in** response to the SR.C. The usage of the modulation adder in the above sawtooth oscillator ensures that no additional events (from the F input) will cause iteration triggering and/or sending of an output event.

The same convention applies for CR-clocked processing and is a good candidate for the application to other regularly clocked Structures.

Rather than binding the Macro to the usage of the SR bus available in the context, it might be useful to allow an SR Bundle to be connected explicitly to a dedicated input of the Macro. A useful pattern for this is illustrated in the following picture:




A Macro with a dedicated SR input port.

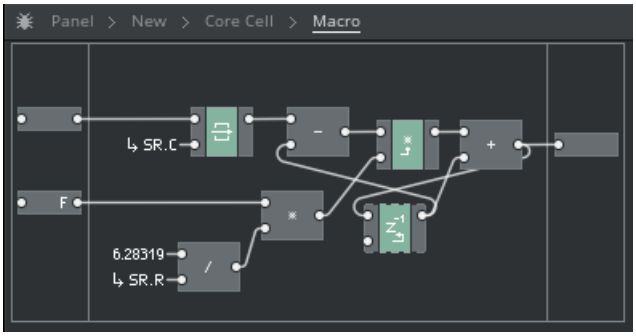
The Macro employs the Scoped Bus overriding technique to redefine the SR bus locally. If there is no connection at the SR input of the Macro, the redefined bus will simply be connected to the SR bus from the Macro's parent context, so the Macro will work exactly in the same way as if the SR bus was not redefined at all. However, if there is an explicit connection at the SR input of the Macro, the Macro will use the explicitly connected SR Bundle instead.

> An audio generating Structure might also need to perform some specific actions in response to the SR.Reset. For a free-running oscillator, however, the Reset event can be completely ignored.

The above trick involving the local redefinition of the SR Scoped Bus will not work correctly inside a non-solid Macro, since the bus redefinition will also affect the outside Structure. In this case the internal wiring of the SR.C, SR.R and SR.Reset needs to be done using normal wires and/or QuickBuses.

## 5.3 Processing of Audio and Other Regularly Clocked Events

The processing of audio and other regularly clocked events is similar to the generation. Besides the usage of Modulation Macros in places where the control signals are mixed with audio events, it is a good idea to reclock the audio input with SR.C just in case that a non-audio signal is connected to the input from the outside (one particularly common case of this is leaving an input disconnected, which implies a zero constant):




A naïve 1-pole lowpass filter.

> Discrete-time difference LTI systems can be directly implemented in REAKTOR Core by simply wiring up the system's block diagram using the *z^-1 fbk* and *z^-1 ffd* Macros for $z^{-1}$ blocks, modulation multipliers for gains, and adders for summation points. The above picture provides an example.

In the above Structure the *Latch* and the *z^-1 fbk* Macros serve as SR.C-clocked signal sources, which are then combined in various ways. The only other signal which is coming in (the normalized cutoff control signal derived from the input "F") is connected via a Modulation Macro. This ensures that no other clocks or trigger sources can accidentally generate an output event.

The same convention as for the audio generators is applied here: the output signal events must be produced **in and only in** response to SR.C. The same convention applies to CR.C and potentially to other regular clocks.

> The SR.Reset fiber is automatically handled by the internal Structure of the *z^-1 fbk* Macro, so that the filter's state is zeroed in response.

In the above filter Structure the processing of the cutoff control input "F" is not done by the Modulation Macros. If the F signal is clocked by CR.C, the filter Structure might seem to violate the above guidelines of processing the clock signals, however in fact this does not.

The clocking guidelines apply to entire Library or framework Macros, not to some parts of their internal Structure.

Imagine a modulation multiplier had been used to combine F with the 6.28319/SR.R value. Furthermore, imagine there was change of the sampling rate without a change of the F input's value at the same time. In this case, the multiplier output would not have been updated, which would have clearly been a bug.

As to whether one should have used a modulation divider to compute 6.28319/SR.R, this does not matter since SR.R should be guaranteed to arrive during the initialization.

Had this guarantee not been in place, the divider could have produced an INF value at the initialization, which would most likely result in a NaN value stuck in the filter feedback loop (see section ↑5.5, Denormals and Other Bad Numbers).

## 5.4 Rounding

There is only one type of rounding available in Core by default: rounding to the nearest integer. The other rounding modes (rounding down, rounding up and rounding towards zero) are not provided. There are a number of technical reasons behind this, related to the precision and efficiency of the generated code.

While in principle the other rounding modes could be implemented as Core Macros, often the same Structures can be implemented just using the round-to-nearest mode, thereby not losing the efficiency. For example, an interpolated read from a buffer in a classical implementation rounds the non-integer position *x* down to an integer *n* and then performs a linear interpolation between the values at *n* and *n*+1, where the interpolation weights are 1-(*x*-*n*) and *x*-*n*. However, instead one could round (*x*-0.5) to the nearest integer. Since for values with a fractional part of exactly 0.5 the round-to-nearest direction is not specified, sometimes, for an exact integer *x*, the value *x*-0.5 will be rounded to *n*=*x*-1 rather than *n*=*x*. However, for the interpolation it does not matter, since the interpolation weights in this case will be 0 and 1 instead of 1 and 0, producing the same result.

> One might need to ensure that the buffer has at least one 'extra sample' on each side of the region, which is read by the interpolator (this is usually a good idea regardless of the rounding mode used, since there can be precision losses in the computations). These extra samples can be set to an arbitrary value (within the same value range as the typical values in the buffer), since the interpolation weights for the extra samples will be very close to zero.

In situations where a division result needs to be rounded down, usually an integer division is meant semantically and this is what normally should be used as well. Note that the integer division rounding mode is 'towards zero'.

The reason that the integer division rounding mode is 'towards zero' rather than 'down' is that this is the mode supported by modern processors. Implementing a rounding down mode instead would produce a higher CPU load. However, there is no difference between the two modes for non-negative integers.

The rounding down mode of the integer division seems to be generally more useful in signal processing applications. Therefore, if future processors support this mode natively in an efficient way, it is possible that Core integer division will also employ this mode. While in this case Core will probably implement a legacy mode switch, it is recommended to stay away from relying on specific rounding direction when there are negative integers involved in the division.

## 5.5 Denormals and Other Bad Numbers

The IEEE floating point representation, which is typically used on modern computers, has several different classes of special values, which are often processed at a much higher CPU time cost. A single value of this sort occasionally popping up in the real-time data should not cause noticeable harm. The problem is that quite often these values tend to stick in the state memory for quite long periods of time, if not forever (that is until the memory reinitialization).

### Denormals

In 32 bit floating point, denormals are the values approximately in the absolute magnitude range between $10^{-37}$ and $10^{-45}$. The values below this range cannot be represented in this format and are replaced by zeros.

For 64bit floating point the denormal range is approximately between $10^{-307}$ and $10^{-324}$ in magnitude.

One most critical situation where denormals can appear is exponentially decaying feedback loops (such as feedback loops in the filters, feedback delays, reverbs, etc.). Denormals appear when the signal level decays to $10^{-38}$ and stays there for a while until it drops below $10^{-45}$, which, depending on the decay speed, can take quite a while.

Denormals are also generated in some other situations as well.

In order to mitigate the above problem, REAKTOR Core offers a *DN Cancel* Module whose purpose is to kill the denormals. In the current implementation it is done by adding a small value to the incoming signal.

- If the incoming value is within a typical signal value range (approximately above $10^{-10}$), then the addition of the denormal compensation value will not change the incoming value at all, due to the limited precision of the floating point.
- If the incoming value is lower (including the denormal range), it will be possibly modified, but, due to the limited precision of the floating point, will also never produce a denormal result.

Because the denormals are so likely to appear in feedback loops, the library *z^-1 fbk* Macro includes a *DN Cancel* internally.

An additional side effect of the *DN Cancel* embedded into *z^-1 fbk* is that the *z^-1 fbk* acts as an internal excitation source for the self-oscillating filters, like circuit noise in an analog filter. In absence of the *DN Cancel* such filters would not produce signals on their own without an external excitation signal.

Denormals are rarely a problem in feedback loops that are not exponentially decaying. For example, typical oscillator Structures include some kind of feedback loop, which is implicitly formed between the *Read* and the *Write* Modules in the oscillator's phase increment Structure. However, since this Structure is not exponentially decaying, but rather produces a repeating ramp signal, the denormals are not a problem here.

Modern CPUs can often be configured in such way that denormals do not cause extra CPU load (typically in this mode the CPUs simply flush them to zero). So, denormals have become less of a problem in recent years. Nevertheless, it is probably reasonable to take measures against them anyway because:

- we do not know how the situation with handling of denormals by CPUs will develop in the future.
- at some point REAKTOR Core might support a different processor architecture, which does have problems with denormals.
- *DN Cancel* can serve as an internal excitation source for self-oscillating filters.

### Infinities

Dividing a non-zero value by zero will produce an infinity (INF). INFs have a tendency to 'stick' in the processing path for a long time, and they also have an increased risk of being converted into a **NaN**.

### NaNs

A NaN (not-a-number) is typically produced when some mathematical operation cannot deliver a valid result. If combined with another value a NaN would normally produce another NaN, which results in the tendency of NaNs to 'stick' in the processing paths, similarly to the INFs.

> Initialization problems in filter Structures are one typical source of stuck NaNs, quite likely with some infinities generated along the way.

There is no embedded way in REAKTOR Core to kill NaNs (or infinities). It is the responsibility of the builder to prevent their appearance.

## 5.6 Optimization hints

### 5.6.1 Core Cell Handlers

When compiling a Core Cell, the compiler generates a number of different functions, or 'handlers', which correspond to the Primary level internal Module API:

- **Initialization handler:** This handler is invoked once at the very beginning of processing and also upon a 'hard reset' of a Core Cell.

- **Event handlers:** Each of the event-mode inputs of the Core Cell has a corresponding event handler. The event handler unconditionally sends the event from the respective input into the Core Cell's Structure.
- **Audio handler:** This handler is invoked on each audio tick. The audio clock event is unconditionally sent from all respective sources inside the Core Cell.

Each of the handlers is compiled and optimized separately.

- The compiler identifies the areas of the Structure which do not receive any events within the particular handler and does not generate the respective code.
- The compiler identifies the events which are being sent unconditionally within the particular handler (e.g. an audio clock within the audio handler) and does not generate any runtime checks for these events.
- For events that are sent conditionally the compiler generates the runtime checks, which determine whether a particular event is being sent or not.

### 5.6.2 Latches and Modulation Macros

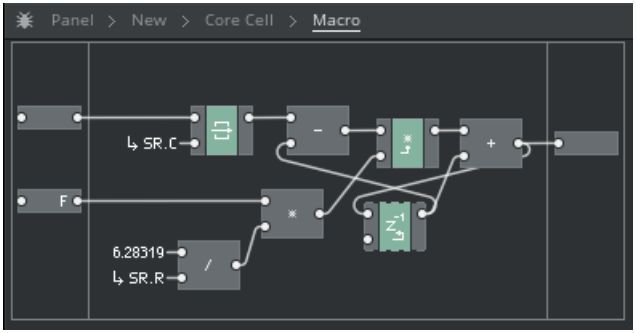
Often in Structure building there is a question of whether a Latch should be used at a particular position. In answering this question the logical function of the Latch and the associated CPU consumption can be considered. As it was already mentioned, the compiler is treating latches (and the 'read followed by a write' pattern in general) in a special optimized way. Thus, the relationship between the usage of a Latch at a particular place and the associated change in the CPU load is not straightforward.

- Generally, if it is not necessary to store the value into the memory (because the value is immediately read afterwards anyway), the compiler will not do so. In such situations a Latch by itself will not add to the CPU load.
- On the other hand, not placing a Latch on some signal path may result in a more complicated triggering logic of the downstream Structure and thus in a higher CPU load produced not by the Latch itself, but by this downstream Structure.

Thus, there is no general rule whether using a Latch will increase the CPU load or decrease it. It is best to simply use latches wherever logically appropriate.

> As Modulation Macros are simply shortcuts for the mathematical operations combined with Latches, the same applies to the Modulation Macros.

The already familiar naive 1-pole lowpass filter Structure illustrates:




A naïve 1-pole lowpass filter.

Depending on the clocking of the signal at the upper input there are the following two typical situations:

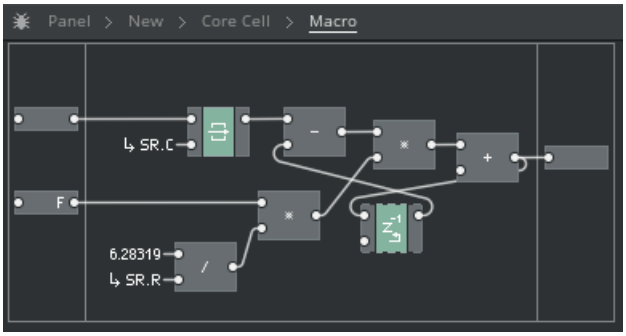
- If the input signal is already clocked at the audio rate, the latch at the input is fully transparent and is not going to produce any compiled code.
- If the input signal is not an audio rate signal, the latch will store its value upon the arrival of the input event, the value being read back upon every audio clock.

The compiler will also handle a number of more complicated situations in regards to the input signal clocking relative to the audio clock.

As for the cutoff control signal at the F input, the same situations are handled in the following ways:

- If the cutoff is an audio rate signal, the output of the first multiplier will be directly used by the internal multiplier of the Modulation Macro during the audio clock processing.
- If the cutoff is not an audio rate signal, the internal latch of the modulation multiplier will store the value, and read it back upon the following audio clock tick.

On the other hand, the following Structure illustrates the case where a usual multiplier had been used instead of the modulation multiplier:




A naïve 1-pole lowpass filter with a buggy use of a non-modulation multiplier.

In the above Structure the following problems would have arisen if a non-audio event arrives at the F input.

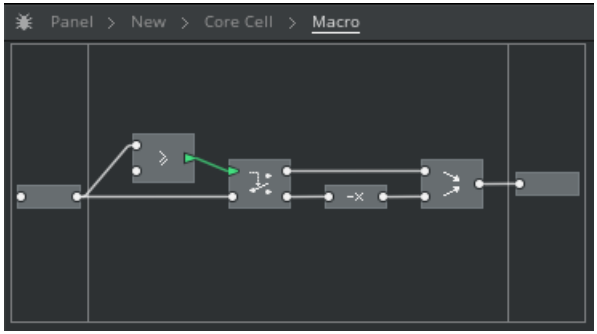
- The non-audio event at the F input will trigger the computation and the sending of the new output value at an inappropriate (non-audio-clock) time moment.
- The subtraction, which is normally done on the audio clock, will not be triggered during this non-audio computation, since neither the *z^-1 fbk* Macro nor the input latch send an event. However, the second multiplier and the adder will be triggered and the computation result will be stored in the *z^-1 fbk* Macro, overriding the previously stored state. Thus, the filter will have performed a 'halfway update cycle', basically destroying its own state.
- The triggering conditions of the second multiplier and of the adder will be a mixture of the audio clock and the cutoff event. If the audio clock is not the default clock (which is unconditionally sent during the audio handler), but something else (e.g. downsampled or gated audio clock), the mixture will require the compilation of a runtime check before the second multiplier, causing a higher CPU load.

Thus a removal of the *Latch* from the Modulation Macro in the above 1-pole filter Structure could have caused both incorrect functioning and a higher CPU load.

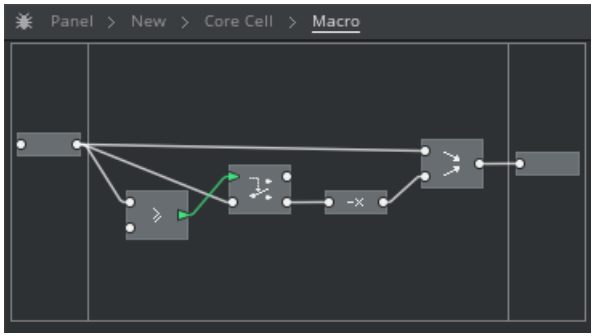
### 5.6.3 Routing and Merging

The compiler recognizes the following routing/merging patterns:

- Two signal branches from a router are merged back afterwards. Such merging is treated like an endpoint of the splitting:

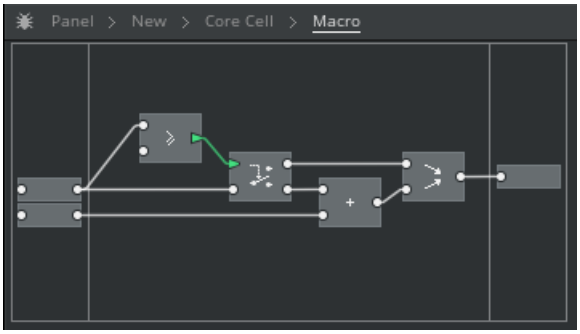



- A branch is being merged back to the pre-router signal. This is also a kind of endpoint of the splitting:




Triggering-wise, most of the arithmetic Modules are equivalent to the *Merge* Modules and are also subject to a similar recognition by the compiler.

The above patterns imply that no additional triggering events have been merged into the branch (e.g. the branch is consistently employing Modulation Macros to mix in further signals). The following Structure illustrates this:




An incorrect merging of two branches.

The Structure is supposed to add the lower input to the negative values of the upper input, while the non-negative values should be left untouched. The Structure however neither works as expected nor the mergepoint really merges the two signal branches back (triggering-wise).

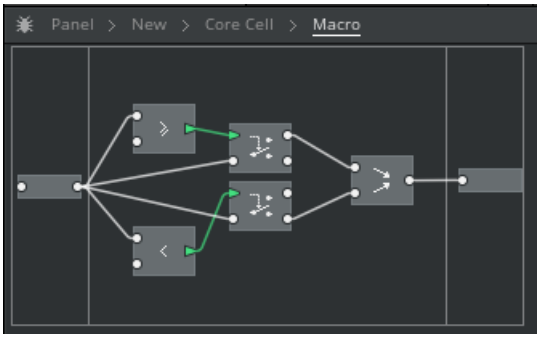
- If the events arrive simultaneously at both inputs of the Macro, the event at the lower input will trigger the adder. The adder will use the latest negative value of the upper input (which is available at the lower output of the router) as the other addend. The result of the addition will be sent through the *Merge* Module to the Macro's output regardless of the sign of the value at the upper input.
- If an event arrives at the lower input only, the situation is similar. The adder will use the latest negative value as the other addend and send it through the *Merge* to the output.
- If an event arrives at the upper input only, the Structure works as expected. The value is forwarded to the output untouched.

Regarding the triggering conditions of the output, rather than being identical to those of the input, they are a complicated mixture of the triggering conditions of the first and the second inputs and the sign of the value at the upper input. Thus, no splitting endpoint occurs triggering-wise.

Replacing the adder with a modulation adder will fix all of the above problems.

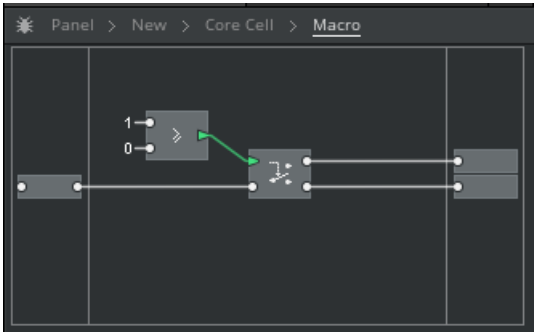
The above example and other kinds of merging do not correspond to the splitting endpoints, because the post-mergepoint triggering information is not identical to the pre-router triggering information. Therefore there is additional CPU overhead at the merging point due to the necessity of checking the runtime triggering condition for the subsequent signal path.

The compiler does not recognize identical or related conditions originating from different BoolCtl sources. A merging of such signals will produce additional runtime overhead as any merging of unrelated signals:




A merging of two formally unrelated signals.

Neither does the compiler identify 'always true' or 'always false' conditions from numerical comparisons:




The compiler will not recognize that the event is always routed to the upper output.

Disconnected BoolCtl ports **are** recognized as providing constant control conditions.

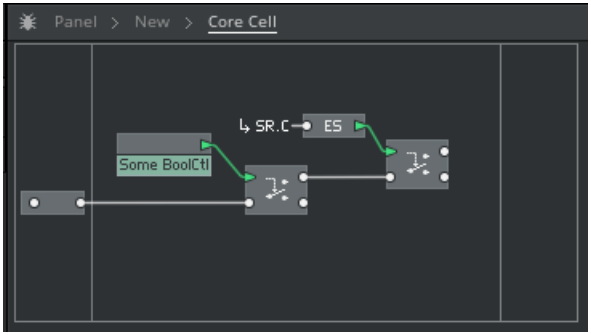
## ES Ctl

The BoolCtl signal produced by an *ES Ctl* Module is in the true state when there is an incoming event at the *ES Ctl* Module's input at the very same,moment, and in the false state otherwise. Thus, the *ES Ctl* Module allows you to check whether some event and the event at the Router's input are arriving simultaneously.

The routing controlled by *ES Ctl* Modules is subject to further special optimization, where a router can be found to be in an 'always true' or an 'always false' state. This is particularly detected in the following cases:

- The signal at the *ES Ctl* Module is the 'triggering parent' of the input signal of the Router.
- The event at the ES Ctl Module is mutually exclusive with the event at the Router's input, **because it lies on a different branch of some other Router or because the event simply never arrives within a particular handler.**

This is illustrated by the following picture:




ES Ctl controlled by a 'parent' trigger.

Within the audio handler both the SR.C and the audio-mode Core Cell input (in the inputs area on the left) are triggered unconditionally. Therefore the SR.C and the audio-mode input are considered as the same (unconditional) clock source. So, within the audio handler, the audio rate clock from the Core Cell's input is being sent through the first router. Thus, the input of the second Router is some subset of the audio rate events and there is always an SR.C coming to the ES Ctl simultaneously with the second Router's input events. Due to the 'parent-child' relationship between these two events (one being obtained by routing the other), the compiler is aware of this relationship and will deduce that the second Router can be always considered to be in the **true** state.

Within the initialization handler the compiler will detect that the SR.C will never come at all and will deduce that the second Router is always in the **false** state.

> Actually, compared to the event and especially the audio handlers, it is not crucial to ensure that the initialization handler is optimized, because the initialization handler is executed rarely. The REAKTOR Core compiler also performs less optimization for the initialization handler, often for the sake of being able to perform better optimization for the event and audio handlers.

## Reclocking

At each mergepoint the compiler attempts to detect whether a splitting endpoint occurs. In cases where 'chaotic routing' (when the routing branches are not merged back together, but rather some signals with unrelated triggering sources are mixed) has been used in excessive amounts, the analysis time can grow drastically. In the worst cases this can cause the compiler to appear to 'hang indefinitely' (the compiler's progress bar stops completely).

> NI is aware of this issue and is looking for a solution.

In a 'chaotic merging' situation practically each Module represents a new mergepoint with a new set of triggering conditions. While the analysis of these triggering conditions consumes the compilation time, the respective runtime check eventually generated by the compiler consumes the runtime.

'Chaotic merging' by itself is not necessarily bad. The builder simply needs to be aware of the associated CPU overhead. Such overhead can be completely acceptable e.g. in processing of a rarely sent control events. The mentioned growth of the compilation times is another problem requiring the builder's awareness.

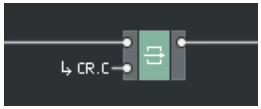
To kill the problem of 'chaotic merging' at its very root the builder can do the following:

- Use Modulation Macros at all logically appropriate places. The Modulation Macros copy the triggering information from their triggering input (as long as they have only one triggering input, which is true for the majority of such Macros), thus the signal stays within the same routing/triggering branch.

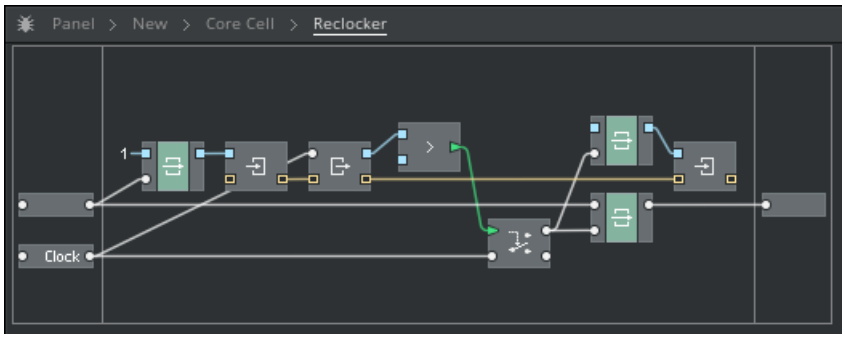
- Use control and audio clocks to make sure that the merged signals are either identically clocked or at least clocked with a small number of different clocks. It is even better if the clocks are derived from some common master clock in a hierarchical fashion, because merging a 'subclock' to its 'parent' will simply produce the 'parent' clock again.
- Use the technique of **reclocking**.

Reclocking means that the builder places Latches at a number of points within a 'chaotically merging' structure. The Latches should be clocked by events with a relatively simple triggering. Some candidates for such clocks are the following.

- A control or an audio clock signal:




- A control or an audio clock signal, put through a router, which is controlled by a Boolean variable, where the latter is being set upon any arriving merged event and reset upon the event passing through the Router:




Other strategies based on the specifics of given Structures can be used.

### 5.6.4 Numerical Operations

When computing mathematical expressions, various issues need to be taken into account.

## Relative Cost

The relative CPU cost of various operations can vary from one processor to the other. However, broadly speaking, the following hints can be kept in mind:

- The cheapest operations are float/int addition, float/int subtraction, float multiplication. Negation, absolute value and integer bitwise operations are equal or slightly more expensive. *DN Cancel* in the current implementation is using an addition internally.
- Float/int division and integer multiplication are noticeably more expensive.
- Float exp and log Modules are the most expensive.
- The cost of runtime condition checks (occurring in routing and non-split endpoint merging) can noticeably vary depending on the circumstances. At any rate they are probably less desirable than a couple of cheap mathematical operations such as addition, so whenever possible, the latter might be preferred.
- The *Read* and *Write* operations are comparable in cost to the cheapest mathematical operations. They might be marginally more expensive for the arrays (this applies to the array memory access, array indexing is more expensive).
- Array indexing internally contains a safety clipping mechanism, which is essentially a runtime check. So avoid sending unnecessary events to the array *Index* Module. Equally, consider sharing an array *Index* Module between a *Read* and a *Write* in such situations.

## Relative Order

Check the relative order of the expression computation and consider how often various input events arrive. For example, the following Structure is taking into consideration that the sampling rate changes much more rarely than the frequency control signal, therefore the division will be done only upon sampling rate changes, otherwise only the much cheaper multiplication will be performed:

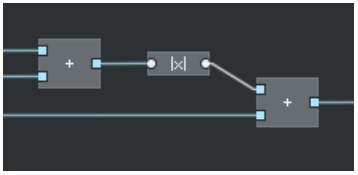

Computation of the normalized frequency for an oscillator.

Had the above Structure been implemented just by dividing F by SR.R, the division would have been performed on each F event, potentially causing a noticeable increase in the CPU load.

## Type Conversions

Conversions between floats and integer and **between floats of different precisions** are generally comparable in cost to the cheapest mathematical operations, but this could drastically vary depending on the platform (and used to in the past). Therefore, as a general rule, unnecessary conversions should be avoided.

The following Structure for example does two unnecessary conversions between float and int:



Unnecessary implicit type conversion.

Even though this Structure should typically perform correctly (unless the integer values exceed the range where the integers are exactly representable by floating point), the CPU load of the Structure is higher than necessary due to the type conversions implicitly occurring on both sides of the absolute value Module.

## Avoid Bad Numbers

Various special types of values causing higher CPU load have been mentioned in section ↑5.5, Denormals and Other Bad Numbers. These have to be avoided. Since these values can be particularly produced when mathematical operations are computed for the out-of-range inputs, care needs to be taken to keep the values in the supported range.

Even if a particular out-of-range value does not result in a 'bad number' in a given implementation of REAKTOR Core, this might change in the future versions or on different hardware. The result of processing an out-of-range value should not be relied upon.

# 6 Macro Reference

## 6.1 Low-level Macros

### 6.1.1 Math

The Macros for computing various mathematical operations, which are not implemented by the built-in Modules: maximum, minimum, reciprocation, square root, rounding, wrapping etc. As with math built-in Modules, each input is a triggering one.

#### Trig-Hyp

Trigonometric and hyperbolic functions and their inverses.

#### Shaper

Functions for control signal shaping.

#### BLT Prewarp

Frequency prewarping for the bilinear transform (used in digital filter construction). Various argument ranges and precisions.

### 6.1.2 Math Mod

Modulation Macros for computing various mathematical operations. See section ↑4.6, Modulation Macros for the explanation of the modulation concept.

The 'tap' Macros are a special kind of Modulation Macros, where the triggering is done by the 'control' input rather than by the 'signal' inputs.

#### Shaper

Modulation versions of control signal shaper functions.

### 6.1.3 Clipping

Modulation Macros for clipping the values to specified ranges. The clipping threshold inputs are non-triggering.

### 6.1.4 Saturator

Signal saturators ('soft clippers') There are some 'hard saturators', which ensure that the output signal level does not exceed a certain threshold and some 'soft saturators', which do not have a saturation threshold.

### 6.1.5 Convert

Dedicated mathematical Macros for conversion between different value scales (e.g. pitch/frequency, gain/decibel etc.).

### 6.1.6 Memory

Scalar and array storage handling Macros. Latches, array read/write, z^-1.

### 6.1.7 Flow

Additional router Macros, which employ a threshold input instead of a BoolCtl one. The routing is done depending on how the input signal compares to the threshold value.

### 6.1.8 Event

Macros specifically focusing on the processing of events (value + triggering). Filtering duplicates, raw clock handling, delaying the events.

### 6.1.9 Logic

Boolean value handling. The Boolean values are represented by integer type signals.

The built in BoolCtl signal type does not implement true signals in the same sense as scalar types, because it does not transmit triggering information. A related issue is that they do not support classical Boolean operations like AND and OR. If Boolean algebraic computations are required, the *Logic* Macros can be used instead. Particularly they contain:

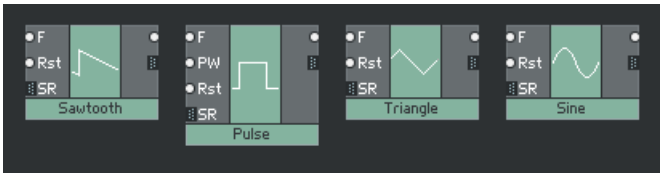
- Comparison operations producing logical signals.
- Classical Boolean AND, OR, NOT and XOR.
- Dedicated float and integer signal routers controlled by a logical signal.
- Conversion to and from BoolCtl.

## 6.2 Audio

At the root of the *Library > Audio* menu there are a number of low-level audio processing Macros (delays, crossfaders, panners). Further audio processing Macros are contained in the submenus.

### 6.2.1 Oscillator

#### Classical analog waveforms




Classical analog waveform Macros.

The inputs are the following:

- *F*: The oscillator frequency in Hz. Negative frequencies can be specified as well.
- *PW*: The width of the pulse waveform.
  - 0 means 0% width
  - 1 means 100% width

- 0.5 means 50% width (default)

- *Rst*: Oscillator restart trigger. The value of the restart event specifies the restart phase:
  - 0 restarts to the beginning of the cycle
  - 1 restarts to the end of the cycle (which is the same as the beginning, since the waveform is fully periodic)
  - 0.5 restarts to the middle of the cycle
  - 0.3 restarts to the 30% of the cycle
  - Values different by a whole integer restart to the same position, e.g. 1.3 restarts to the 30% of the cycle.

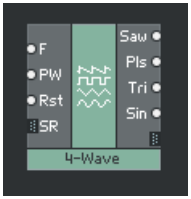
The classical analog waveform oscillators (as well as their slave versions) have a latency of 1 sample. This may need to be taken into account if you need sample-precise control of your Structures.

The Rst input is intended for occasional oscillator restarts, e.g. in response to the MIDI gate signal and should not be used to implement audio oscillator sync. For oscillator sync use the slave versions of the same oscillators.

The SR input is the standard SR Bundle connector explained in section ↑4.9, SR and CR Buses.

The upper output of the oscillators delivers the oscillator signal. The lower (Bundle) output is for connection of the slave oscillators.

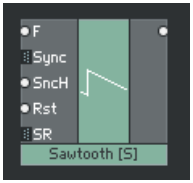
A special *4-Wave* oscillator Macro delivers all 4 waveforms simultaneously. The waveforms produced by this oscillator are phase locked to each other.



The 4-Wave oscillator.

### 'Slave' Oscillators

The 'slave' oscillators are the audio-synced versions of the ordinary oscillators. Compared to the ordinary oscillators, they have two additional inputs:

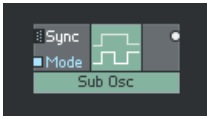



A 'slave' sawtooth oscillator.

- *Sync*: Can be connected to the lower (Bundle) output of a master oscillator to produce the audio sync effect. Only ordinary (non-slave) oscillators can serve as master oscillators.
- *SncH*: Controls the sync hardness:
  - 1 = hard sync
  - 0 = no sync
  - 0 to 1 = various degrees of soft-sync

### Sub Oscillator

The sub oscillator is a special kind of 'slave' pulse oscillator, which is phase-locked to the master.




Sub oscillator.

The idea of the sub oscillator is that it changes its output from -1 to 1 or back whenever there is a transition (a cycle completion) in the master, thereby producing a square waveform one octave below the master oscillator's frequency. By leaving out some of the master transitions out, further sub oscillator modes are available:

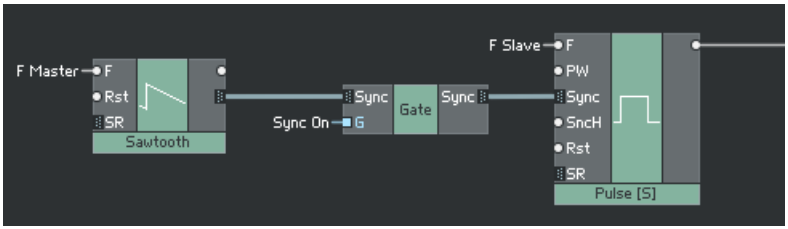
- 0 = -1 octave, 50% pulse width (use every master transition)
- 1 = -2 octaves, 50% pulse width (use every other master transition)

- 2 = -2 octaves, 75% pulse width (use 1st, 4th, 5th, 8th etc. master transitions)

The sub oscillator is fully driven by the master oscillator. Therefore it stops whenever the master is stopped.

## Sync Gate

The *Sync Gate* Macro allows to interrupt the sync signal from the master to a slave or a sub oscillator.



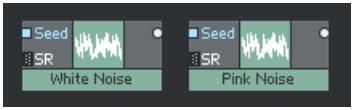

Using a sync gate to stop the audio sync.

The signal at the G input should be either 0 or 1. In the 0 state the sync signal does not come through, effectively disabling the synchronization of the slave to the master.

A sub oscillator would be completely stopped if the gate is off.

## Noise Oscillators

These generate different kinds of noise.




Noise oscillators.

If several identical noise oscillators are used, then by default they will produce fully identical signals. This can be avoided by connecting different constants to their *Seed* inputs, which then initializes their internal (pseudo) random number generators with different values.

## 6.2.2 Filter

The Core Macro library contains a selection of ZDF filters. The filters are built using the *ZDF toolkit*, which is a part of the library (see section ↑6.4.2, ZDF Toolkit).

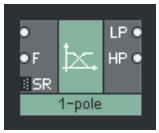
> The filters built using the ZDF toolkit are optimized for Structure readability, not for CPU efficiency. A number of filters have been redone with CPU efficiency in mind. These can be told by the "(opt)" suffix (standing for optimized) at the end of their names.

### Linear Filters

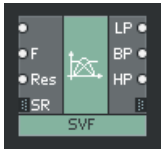
Linear filters only change the frequency spectrum of the signal and do not introduce any distortion. A number of these filters can resonate but cannot selfoscillate (at least not reliably). The maximum resonance setting of these filters, where applicable, is therefore limited to (approximately) 1.

The most important Macros implementing the linear filters are the following.

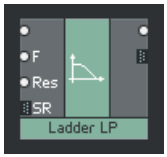
- *1-pole > 1-pole*: A multimode linear 1-pole filter producing a lowpass and a highpass signals with a -6dB/octave rolloff.



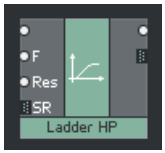

- *2-pole > SVF*: A multimode 2-pole resonating filter producing a lowpass, a bandpass and a highpass output. The rolloff of lowpass and highpass is -12dB/octave. The rolloff of the bandpass is -6dB/octave on each of the two slopes (intuitively, although not fully correctly, one could imagine as each of the filter's poles being responsible for a -6dB/octave rolloff, so for the band pass one pole is responsible for the -6dB/octave slope to the left of the filter's center frequency and the other pole is responsible for the -6dB/octave slope to the right of the filter's center frequency). Modal pickups (see below) can be used to obtain further modes.



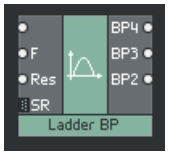
- *4-pole > Ladder LP*: A 4-pole resonating lowpass filter, emulating the classical transistor ladder filter design. The rolloff is -24dB/octave. At higher resonance settings this filter naturally produces a drop in the bass frequencies. This drop can be avoided by using the modal pickups (see below).




- *4-pole > Ladder HP*: A 4-pole resonating highpass filter. The rolloff is -24dB/octave. The *Ladder LP* modal pickups (see below) can be used with *Ladder HP*, producing 'flipped' filter shapes (highpass instead of lowpass etc.).




- *4-pole > Ladder BP*: A 4-pole resonating bandpass filter. The rolloff of each of the two solpes is -12dB/octave. The filter does not provide the modal pickup connector, but two further modes with lower pole counts (BP3 and BP2) are provided directly by the Module outputs. At zero resonance the BP3 mode has a -12dB lowpass rolloff and -6dB highpass rolloff, while BP2 has both rollofs at -6dB/octave




The *F* input specifies the filter cutoff frequency in Hz.

The *Res* input, if available, controls the filter's resonance. The range is from 0 to approximately 1. In fact, the maximum resonance value is clipped slightly below 1, because at too high resonance settings linear filters may produce very loud resonance peaks. For higher resonance settings use nonlinear filters (see below).

The resonance control of the filter Macros is implemented by linear mapping of the resonance parameter to the internal filter's parameters (such as feedback amount or damping). This does not necessarily provide the best resonance control curves. Use control signal shapers (such as those found in *Math Mod > Shapers*) to change the resonance control curve. E.g. an SVF resonance control knob can be shaped with a *Hyp Shaper* whose bending parameter is set to 0.3. This will make the resonance control of the SVF LP feel similar to the one of the Ladder LP.

## Nonlinear Filters

Nonlinear filters, besides the purely filtering effect, apply some nonlinear processing to the signal, causing it to saturate internally in the filter. As a result, the resonance of such filters can be driven to excessive values (above 1) without causing the filter to 'explode'. At these excessive resonance settings the filters may produce sound even in the absence of the input signal, which is referred to as the selfoscillation effect.

The most important nonlinear filters in the library are the following.

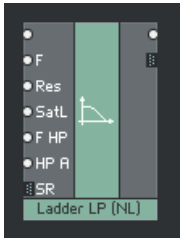
- *2-pole > TSK LP (NL)*: Nonlinear transposed Sallen–Key lowpass filter. In the linear range (low signal levels) this filter sounds identically to the SVF and therefore can be thought of as a nonlinear counterpart of the latter. The *SatL* input controls the internal saturation level of the filter. The smaller is the saturation level setting, the stronger the filter will saturate (default value is 4).


*TSK LP (NL)* (and other TSK filters) have dedicated modal pickup Macros.

The library also contains a nonlinear SVF, but *TSK LP (NL)* is the recommended option.

- *4-pole > Ladder LP (NL)*: A nonlinear counterpart of *Ladder LP*. The *SatL* input controls the internal saturation level of the filter. The smaller is the saturation level setting, the stronger the filter will saturate (default value is 1). The *F HP* and *HP A* inputs control the cutoff (in Hz) and the amount (from 0 to 1) of the internal highpass filtering of the feedback signal. One particular effect of this highpass filtering is prevention of the loss of

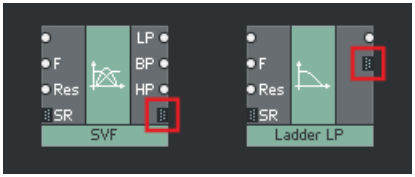
bass frequencies. When used together with normalized DC gain pickups (such as Ladder LP -> LPn), it can produce an extra bass boost.




In order to better balance the behavior of the saturating nonlinear filters in the low-resonance and selfoscillation it is recommended to connect an additional saturator (see section ↑6.1.4, Saturator) at the filter's output.

## Modal Pickups

With many filter designs (including Ladder, SVF and TSK) a large number of different modes can be picked up from the filter. In order to keep the number of outputs low, many filter Macros in the library provide only the main mode (or a small number of 'main modes') directly as their outputs. The remaining modes can be picked up from the dedicated 'modal Bundle' output:

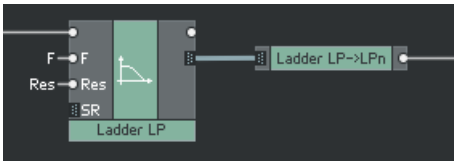



Modal Bundle outputs of SVF and Ladder LP filters.

The picking up can be done by using the modal pickup Macros, located in the *Pickup* submenus of the respective menus. E.g. modal pickups for the 2-pole filters are located in *Audio > Filter > 2-pole > Pickups*. The pickup Macros are categorized as follows.

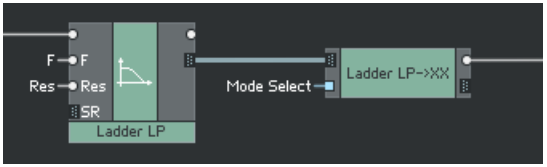
- **SVF pickups:** Can be used with linear and nonlinear 2-pole SVF filters.
- **TSK LP pickups:** Can be used with linear and nonlinear TSK LP filters.

- **Ladder LP pickups:** Can be used with linear and nonlinear Ladder LP filters. The same pickups can also be 'abused' with the Ladder HP filter, producing 'flipped' modes (e.g. HP modes instead of LP modes etc.).




A modal pickup picking up a normalized-DC-gain lowpass mode of a Ladder LP.

There is a special "XX" pickup category. These pickups do not pick up a specific mode, but rather allow dynamic control over which specific mode is being picked up. The control is performed via a selector input. The specific meanings of the selector values can be taken from the infos of the respective pickup Macros.




Dynamic modal pickup.

The upper output of such dynamic pickup Macros provides the picked up modal signal.

> The lower Bundle output of "XX" pickups provides the modal mixing coefficients for the selected mode. These coefficients can be particularly used by the TF modal pickup Macros from the TF Toolkit (see section ↑6.4.3, TF toolkit).

### Multinotch

Contains multinotches with low notch count and a comb filter.

### Butterworth

Butterworth filters are a classical signal processing filter type. They are intended (by design) to statically cut a frequency band from a signal, therefore they do not have any resonance setting. Nevertheless, their parameters can be modulated, as with the other filters in the library.

The same category also contains Butterworth-based crossovers. For even pole counts the Linkwitz–Riley crossover design is used to avoid the +3dB peak at the cutoff (odd pole counts do not suffer from the same problem).

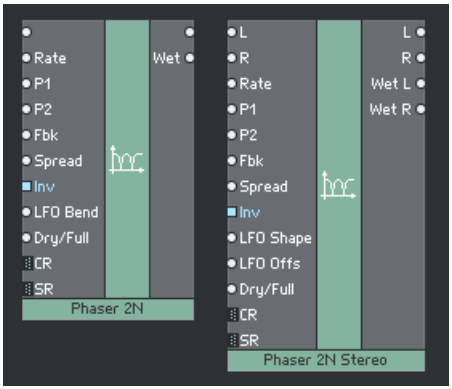
## 6.2.3 FX

A selection of various effects.

### Modulation Effects

The modulation effects include phaser, flanger, chorus, tremolo and autopan. Except for autopan, there are mono and stereo versions of these effects. Most of their control parameters are identical or similar.

The phaser Macros are discussed below as typical examples. Further details about the effect parameters which are different from those of the phasers can be found in the infos of the respective Macros.




Stereo and mono versions of 2-notch phaser.

The following inputs are available:

- *L, R* (stereo) or the top nameless input (mono): Input signal.
- Rate: The rate of internal LFO in Hz.
- *P1, P2*: The boundary pitches of the modulation range. The center cutoff frequency of the internal multinotch filter is modulated within this pitch range.

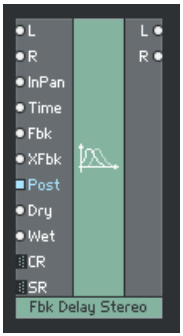
- *Fbk*: Feedback amount. Range -1 to 1. The feedback setting affects the width of the notches, positive values making the notches wider, negative values making them smaller.
- Spread: Adjust the spreading of the notches. Negative values increase the notch density, positive values spread them wider apart.
- *Inv*: The notch inversion mode switch. Swaps the positions of the peaks and the notches. 0 = off, 1 = on.
- *LFO Bend* (effects with exponential LFO): Controls the bending of the exponential LFO segments. 0 = linear, +1/-1 = rectangular jumps.
- *LFO Shape* (effects with triangle-sine morphable LFO): Controls the morphing amount from the triangle LFO shape to the sine LFO shape.
- *Dry/Full*: Controls the mixing amount of the dry and wet signals. 0 = dry signal only, 1 = 'strongest sounding' mixture of dry and wet signals.

The following outputs are available:

- *L, R* (stereo) or the top nameless output (mono): Output signal.
- *Wet L, Wet R* (stereo) or *Wet* (mono): Pure wet signal. To be used with the 'send' style of effect usage, when several sounds are sent to the same effect with different 'send' amounts, producing a common 'wet' signal which is then returned back to the entire mix.

## Delay Effects

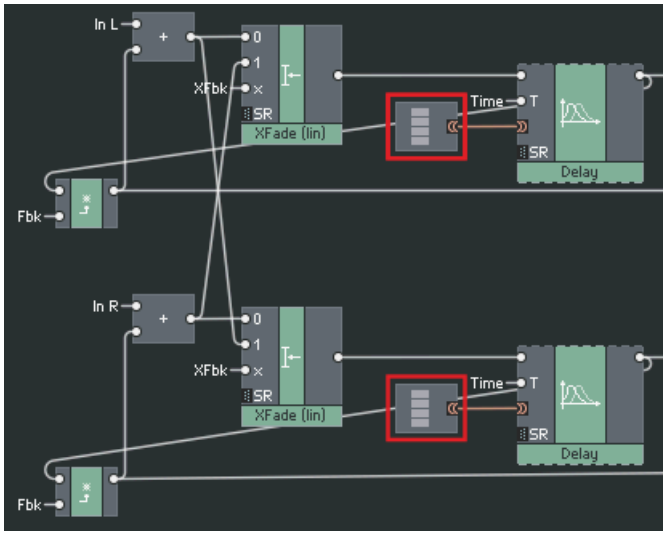
The delay effects are using audio delays to produce echo sounds. A stereo feedback delay Macro illustrates:




Stereo feedback delay.

The following inputs are available:

- *L, R*: Signal inputs.
- *InPan*: Controls stereo balance of the input signal. Range -1..1.
- *Time*: Specifies the delay time in seconds. Maximum delay time at 192kHz sampling rate is 1 sec. At lower sampling rates the maximum time is larger. It can be further increased by modifying the delay buffers inside the Macro:



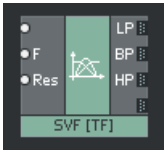

- *Fbk*: Feedback amount. Range -1..1.
- *XFbk*: Cross-channel (left/right) feedback amount. Range -1..1.
- *Post*: The 'post-feedback-gain' output mode. If this input is set to 1, the output signal is picked up after the feedback gain has been applied (if the feedback setting is zero, there will be no wet output signal in this mode).
- *Dry*: Amount of dry signal to be sent to the outputs
- *Wet*: Amount of wet signal to be sent to the outputs.

The following outputs are available:

- *L, R* : Output signal.

### 6.2.4 Transfer Functions

A set of Macros for computing the transfer functions (more precisely, frequency responses) of the filters is located under *Library > Audio > Filters > Tranfer Func*. These are the counterparts to the respective filters. They follow one and the same set of conventions. The *SVF* transfer function Macro illustrates:



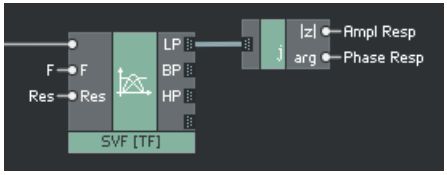

An SVF [TF] Macro.

The inputs and the outputs of a TF Macro are roughly identical to those of the respective filter Macro with some differences.

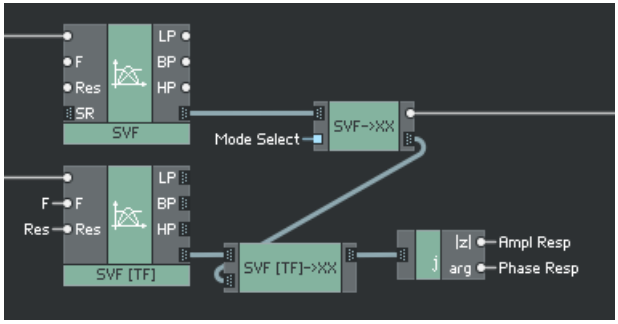
- The upper input of a TF Macro receives a signal frequency value in Hz instead of the audio signal. The Macro will output the value of the filter's frequency response at this frequency.

  In order to build a graphical display of filter's amplitude or phase response, send an iterated range of signal frequency values to this input and pick up the respective output values.

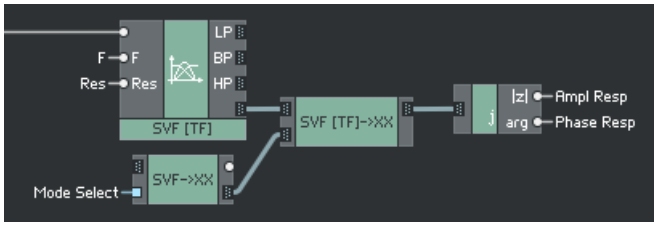
- Other inputs of a TF Macro receive the parameter values identical to those of the respective filter Macro.
- The 'audio signal' outputs of the TF Macro (those corresponding to the audio signal outputs of the filter) produce the frequency response value at the frequency specified by the upper input. These are complex values which can be converted into the amplitude and phase response parts by using the *to Polar* Macro from the TF toolkit (see section ↑6.4.3, TF toolkit):



- The bottom 'modal Bundle' output contains a Bundle of modal TF values, which can be picked up by a dedicated TF pickup Macro. Except for the allpass mode, the modal SVF frequency response pickup is done by using the *SVF [TF] to XX* Macro. The *SVF [TF] to XX* Macro does not have an integer modal selector input, in place of it is expecting a modal coefficients Bundle from the SVF to XX selector:




The *SVF to XX* Macro in the above picture does not need to be connected to an audio filter:




The TF Macros compute the ideal analog frequency responses, not taking into account the discrete-time warping of the frequency axis.

Some of audio filters do not have TF counterparts. However, often there *is* a TF counterpart, which is not immediately obvious. Below are some examples.

- There are no dedicated TF Macros for nonlinear filters (since frequency responses in unstable range do not really make sense anyway). Use the corresponding linear TF Macros.
- The frequency response of TSK filters is identical to those of SVF, so simply use an SVF TF Macro to obtain TSK frequency response.

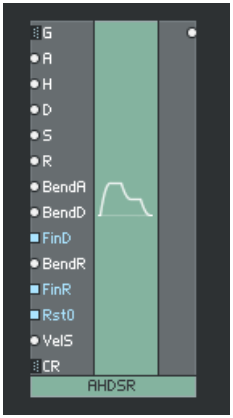
- The *Ladder LP [TF]* Macro has the F HP and HP A inputs available only for the Ladder LP (NL) Macro. To obtain the frequency response of a lowpass ladder filter without a highpass in the feedback these inputs can be simply left disconnected.

# 6.3 Control

## 6.3.1 Envelope

A small selection of prebuilt envelopes of varying complexity. If the selection is not sufficient, the existing envelopes can be easily extended by the builder, because they are built using the Envelope toolkit (see section ↑6.4.1, Envelope Toolkit).

The envelopes follow one and the same set of conventions. The AHDSR (attack-hold-decay-sustain-release) envelope is used here as an example to illustrate these conventions:

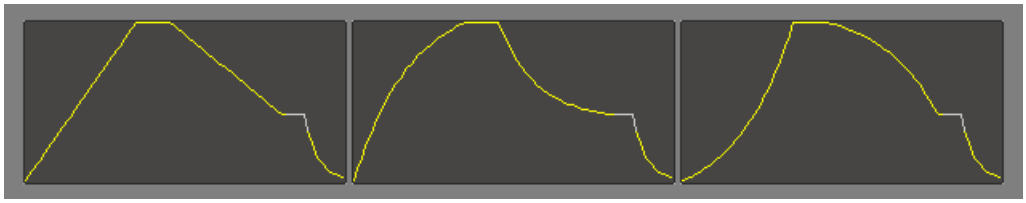



AHDSR envelope.

The envelope inputs are the following.

- G: Should be connected to a gate Bundle (see below for the explanation of the Core's gate Bundle conventions). It is responsible for triggering the attack (upon the 'gate on' event) and release (upon the 'gate off') modes of the envelope and for sending the keypress velocity information, if any.

- A, H, D, S, R: The 'main' control parameters. A, H, D and R specify the time constants for the attack, hold, decay and release stages. S specifies the sustain level, which normally should be in the range 0 to 1, since the attack peak level is always 1.
- *BendA*, *BendD, BendR*: The bending of the attack, decay, and release stage curves. The bending applies only to the finite-time curves (see the finite-time mode explanation below). The bend parameter range is -10 to 10. Zero bending produces linear curves, negative bending produces 'slowing down' curves, positive bending produces 'speeding up curves'. The following pictures illustrate the attack and decay shapes for zero, negative and positive bendings respectively:



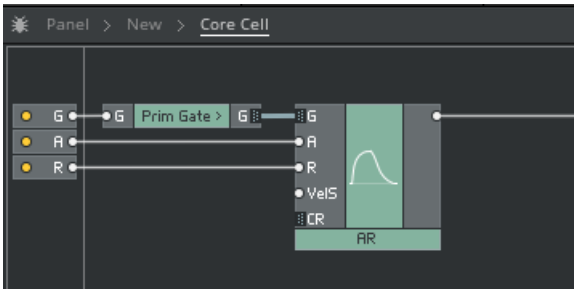
- *FinD, FinR*: The finite-time mode of the decay and release stages. When the mode is set to 1 (finite), the respective stage has the duration as specified by the D or R input. When the mode is set to 0 (infinite), the respective stage decays infinitely to its target value (sustain level or zero).
- *Rst0*: The zero-reset mode of the envelope. When set to 1 the attack always starts from zero level. When set to 0 the attack starts from the current level of the envelope.
- *VelS*: Velocity sensitivity, the range is 0 to 1. If set to 0, the envelope ignores the velocity information in the gate Bundle completely. If set to 1, the envelope scales its level according to the velocity (provided the velocity information is present in the Bundle, see the discussion of the gate Bundle below). At intermediate settings the envelope partially scales its level according to the velocity.

The CR input is the standard CR Bundle connector explained in section ↑4.9, SR and CR Buses.

## Gate Bundle

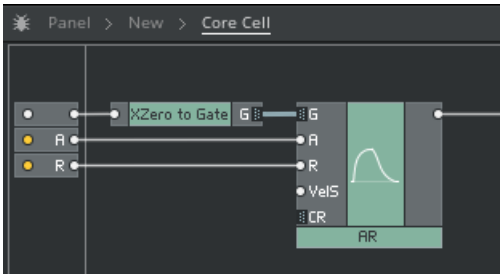
The envelope gate connections are employing a special gate Bundle. The Bundle internally transmits the information about the gate on/off events and the associated velocities.

The gate Bundle can be obtained by converting from a Primary level gate event (forwarded to Core via an event-mode Core Cell port). The conversion is implemented by the *Gate from Prim* Macro (*Library > Control > Envelope > Gate > Gate from Prim)* and includes the gate on/off events and the 'on' velocity (no 'off' velocity):



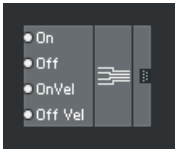

AR envelope controlled by a Primary gate event signal.

The *XZero to Gate* Macro (*Library > Control > Envelope > Gate > XZero to Gate)* sends gate events in response to positive and negative zero crossings of the input signal (positive zero crossings produce 'gate on' events, negative zero crossings produce 'gate off' events):




AR envelope controlled by zero crossings of the incoming audio signal.

Internally the gate Bundle consists up to 4 fibers:




A Bundle Pack for all 4 fibers of the gate Bundle.

The conventions for the gate Bundle are the following.

- *On*: Transmits the 'gate on' trigger. The consumers should ignore the value of this fiber. The producers should always send zero values on this fiber.
- *Off*: Optional fiber. Transmits the 'gate off' trigger. The consumers should ignore the value of this fiber. The producers should always send zero values on this fiber. If this fiber is absent, it means that the 'gate off' trigger events are never sent.
- *On Vel*: Optional fiber. If present, specifies the current 'on velocity'. The standard range is positive numbers less than or equal to 1, however, larger values are allowed as well. Generally speaking, this value should be sent only simultaneously with 'gate on' events (or sent during the initialization, if the value is constant). If an 'on velocity' event comes at any other moment, the consumers may respond to it or ignore it until the next 'gate on'. Velocity-sensitive consumers should treat the missing "On Vel" filber as a value of 1.
- *Off Vel*: Optional fiber. If present, specifies the current 'off velocity'. The convention is the same as for the 'on velocity', except that the 'off velocity' events should be sent together with the 'gate off' events.

A number of preconfigured Bundle *Pack* and *Unpack* Modules with different numbers of fiber definitions/pickups are available under *Library > Control > Envelope > Gate*.

The *Add Gate Defaults* Macro (*Library > Control > Envelope > Gate > Add Gate Defaults)* automatically adds Off, On Vel and Off Vel to the Bundle, in case these fibers are missing (the added velocities are equal to 1). It is intended to be used in implementations of gate Bundle consumers and other Structures.

When constructing a gate Bundle, never leave the triggering fibers On and Off disconnected. Disconnected fibers imply a zero constant, which will send a trigger event upon initialization. Simply leave the optional fiber away from the Bundle, or connect it to a *No Event* Macro if you do not want any events sent.

### 6.3.2 LFO

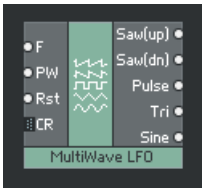
A number of LFOs of classical shapes. These are intended to be used as modulation sources, not as audio signal sources. Particularly, the LFO Macros do not contain any anti-aliasing.


Various LFOs available in the library.

The inputs are similar to those of the oscillators (see section ↑6.2.1, Oscillator).

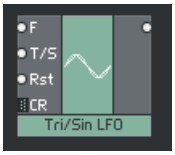
> The CR input is the standard CR Bundle connector explained in section ↑4.9, SR and CR Buses.

The *MultiWave LFO* provides the same waveforms (except the parabolic one) phase-locked to each other.




MultiWave LFO.

The *Tri/Sin LFO* can be smoothly morphed between the triangular and sinusoidal shapes. The T/S input controls the morphing (0 = triangle, 1 = sine, other values in between specify intermediate shapes).

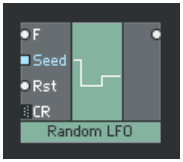



Tri/Sin LFO.

The *Exp LFO* is generating analog-like slowing down exponential curves. The slowdown amount varies from a purely linear segment to a discontinuous jump, thereby morphing between triangular and rectangular shapes at the zero and maximum settings. Instead of slowing down shapes, the speeding up shapes can also be generated. The *Bend* parameter controls the segment shape (0 = linear, 0..-1 = slowdown, -1 = rectangular, 0..1 = speedup, 1 = rectangular). The *PW* parameter controls the 'pulse width'.


Exp LFO.

The *Random LFO* is delivering the signal which would be produced by a white-noise driven S&H. The *Seed* parameter is similar to the one of the noise oscillators (see section ↑6.2.1, Oscillator).




Random LFO.

The triangle, sine and parabolic LFO shapes are also delivered in stereo versions. Compared to mono versions, these have an additional *Offs* input, which controls the offset between the left- and the right-channel phase-locked LFO shapes. At zero Offs both left and right channel signals are identical. For Offs between 0 and 0.5 the left channel LFO is 'preceding' the right channel one ('movement from left to right'). For Offs between 0 and -0.5 the right channel is 'preceding' the left one ('movement from right to left'). At ±0.5 the channels have opposite phases (both +0.5 and -0.5 settings have identical effect). The Offs value can exceed the ±0.5 range, so that at ±1 the channels are again in sync and so on.


Stereo LFOs.

### 6.3.3 Smoother

Please refer to section ↑4.9.4, CR rate change for the details of using smoothers.

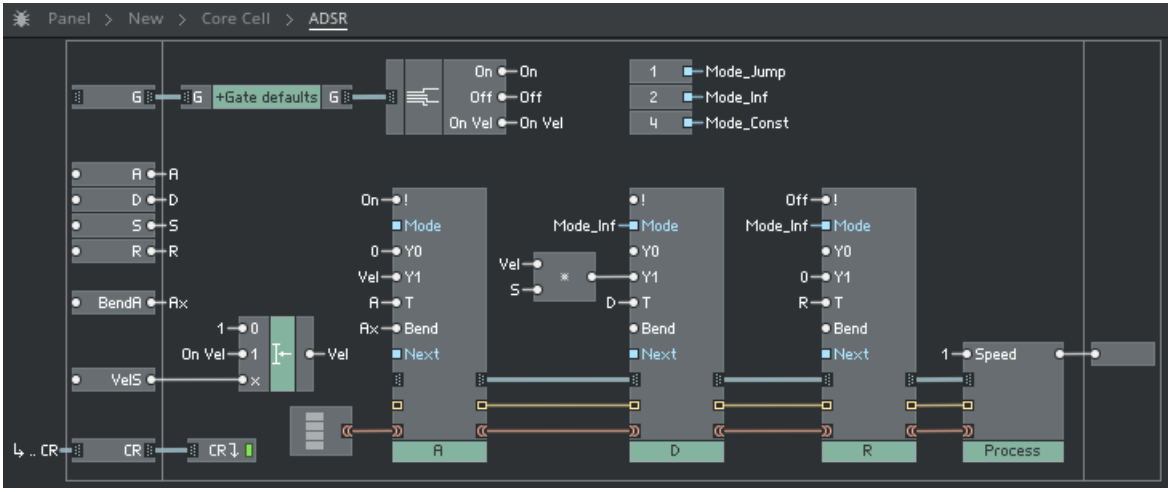
## 6.4 Toolkits

### 6.4.1 Envelope Toolkit

The envelope toolkit (*Library > Control > Envelope > Toolkit*) contains the building blocks of the envelopes in the library. In fact there are only two Macros (*Stage* and *Process*) in the toolkit and each envelope is built using these two Macros, where the *Process* Macro is used exactly once in each given envelope. Therefore in order to build a custom envelope one could simply start off with the existing envelope and duplicate the Stage Macro within that envelope as necessary.

> In fact this duplication approach should be even preferred, because due to the development of the library, different versions of the *Stage* and *Process* Macros may become incompatible to each other.

The library implementation of the ADSR envelope illustrates:




The internal Structure of an ADSR envelope.

The "A", "D" and "R" Macros are simply the renamed copies of the *Stage* Macro. There is no dedicated Macro for the sustain stage, because in this envelope the decay stage is infinitely decaying. Thus there is really no sustain stage, only the sustain level as the target level of the decay stage.

The *Process* Macro on the right is the one which generates the real envelope signal, the *Stage* Macros only being responsible for setting the stage parameters. The upper input of the *Process* Macro specifies the speed coefficient for the envelope. This can be useful if the envelope toolkit is used to build an LFO, otherwise this coefficient should be set to 1, so that the stage durations are exactly as specified by the respective stage's parameters.

The three parallel connection paths at the bottom are used to connect the stages together in their natural order. The *Process* Macro must be at the end of the chain. For the first stage Macro these three inputs must be left disconnected, except for the lowest of them. This should be connected to a 64-bit precision float array. The number of elements of the array must be equal (or at least equal) to the number of the stage Macros times 6.

The other inputs of the stage Macros are as follows.

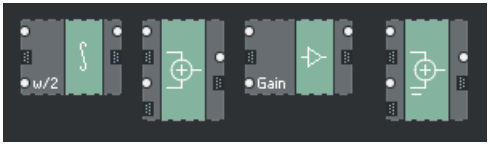
- **!**: This is the explicit triggering input of the stage. An event at this input will force the envelope to immediately jump to this stage. Notice that the 'Gate On' event is connected to the triggering input of the attack stage, while the 'Gate Off' event is connected to the triggering input of the release stage.
- **Mode**: A bitwise OR of stage modal flags:
  - **1** = jump mode. If specified, this flag causes the stage to start (when explicitly triggered or automatically switched to) at the level specified by the Y0 input. Otherwise the stage starts at the current output level of the envelope (no jump).
  - **2** = infinite mode. If specified, this flag causes the stage to infinitely decay from the starting level (according to the jump mode and Y0 settings) to the end level specified by Y1. The infinite decay is exponential, the decay characteristic time being controlled by the T input.
  - **4** = constant mode. If specified, this flag causes the stage to stay at its starting level (according to the jump mode and Y0 settings) infinitely. The segment end level specified by Y1 and duration specified by T are ignored.
- **Y0**: The starting level of the stage. Used only if the jump mode is specified.
- **Y1**: The end level of the stage. Used unless the constant mode is specified.
- **T**: The segment duration in seconds. In the infinite mode this input specifies the segment's characteristic time. In the constant mode this input is ignored.
- **Bend**: The exponential bending of the segment. Does not apply to infinite or constant segments. Refer to section ↑6.3.1, Envelope for more details about this parameter.
- **Next**: The zero-based index of the 'next' segment (the segment which should be switched to after this segment runs out). If disconnected, assumes the 'natural' next segment, according to the order of connection of the stage Macros.

The parameters can be modulated in real time. Particularly the modulation of *Y1*, *T* and *Bend* parameters of the currently running segment will cause realtime changes of the envelope behavior.

## 6.4.2 ZDF Toolkit

The ZDF toolkit (*Library > Audio > Filters > ZDF Toolkit*) contains the building blocks of the filters in the Macro library. The purpose of the toolkit is to allow the implementation of ZDF (zero delay feedback) filters in an intuitively readable manner.

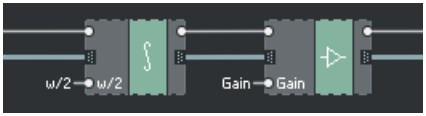
The following building blocks correspond to the fundamental elements of the continuous-time ('analog') filter block diagrams:




ZDF toolkit Macros implementing the fundamental continuous time audio processing primitives.

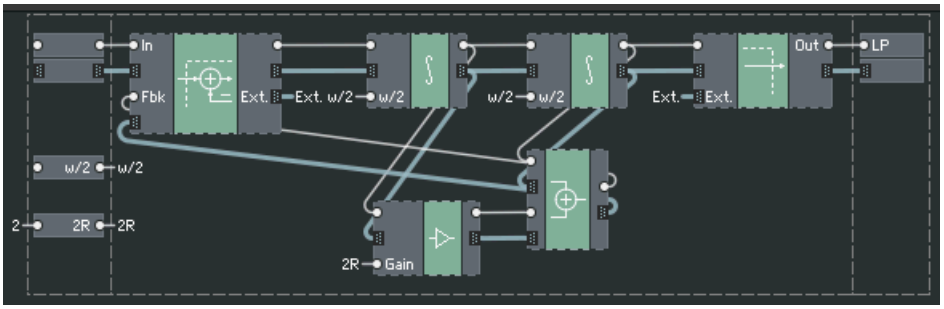
- **Integrator:** A fundamental 'analog memory element'. Roughly corresponds to the $z^{-1}$ blocks in the digital filter block diagrams, but is not functionally equivalent to the latter. The w/2 input should be connected to the prewarped half-cutoff value
- **Adder:** Adds to ZDF signals together.
- **Gain:** Multiplies a ZDF signal by a specified gain factor. The gain input is non-triggering, therefore this gain Macro is similar to the modulation multiplier.
- **Subtractor:** Subtracts one ZDF signal from another. A convenience shortcut for a combination of an adder and a -1 gain.

Each 'audio signal connection wire' in the ZDF toolkit needs to be implemented by two parallel wires:




A single audio path implemented by two parallel ZDF wires.

Each feedback loop needs to be implemented by a single **nonsolid** Macro. The following SVF Structure illustrates.




A nonsolid Structure implementing the main part of a lowpass SVF.

The guidelines are the following.

- Each Macro must implement exactly one feedback loop. Any other feedback loops (if existing) should be contained in the nested Macros built according to the same guidelines.
- The Macro may have exactly one audio input. This input must be directly connected to an *Input -Fbk* (as in the picture above) or an *Input +Fbk* Macro from the ZDF toolkit. This *Input ±Fbk* Macro is the point where the input signal is 'injected' into the feedback loop.

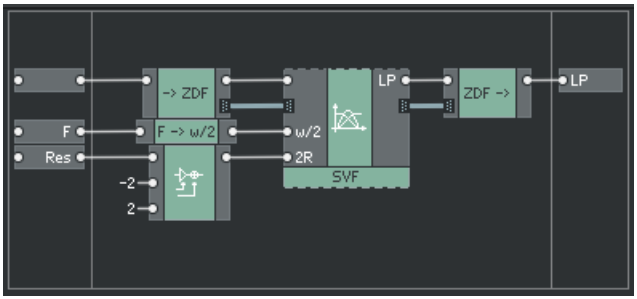
  The difference between the two Macros is simply that *Input -Fbk* inverts the feedback signal. One could use a ZDF gain Macro to invert the signal instead.

- The audio path inside the feedback loop may split and merge back (e.g. in the picture above the path splits after the first integrator and is merged back at the adder). However no further feedback loops are allowed, except in the nested Macros built according to the same guidelines.

  Sometimes this limitation can be worked around by treating two feedback loops as one loop 'nested' into the other. However, there are situations where the inner feedback loop is 'instantaneously unstable' (see the instantaneous instability discussion below) on its own, and only the presence of the outer feedback loop stabilizes it back. In this case such nesting decompositions will not be handled correctly by the ZDF toolkit, as it requires each feedback loop to be stable (in the instantaneous sense only) on its own, without outer stabilization factors. This is e.g. the reason why a nontransposed version of the Sallen–Key filter cannot be (directly, without additional manual 'hacks') implemented by the ZDF toolkit.

- The Macro may have several outputs. However each output needs to be 'implemented' by a dedicated *Output* Macro from the ZDF toolkit. The Ext input of the *Output* Macro needs to be connected to the Ext output of the *Input* Macro of the Structure.

The outermost Macro of a ZDF implementation should be (normally) a solid Macro and implement the conversion between the ordinary and ZDF audio signals (using *to ZDF* and *from ZDF* Macros from the toolkit):




The outermost Structure of a ZDF lowpass SVF implementation.

Besides the audio conversion, this Structure is using an *F to w/2* Macro from the ZDF toolkit to convert the cutoff in Hz to a prewarped half-cutoff for the ZDF integrators. It also converts the resonance parameter into the damping gain 2R.

The above guidelines are not an absolute must and can be in principle violated, as long as the real restrictions are not violated: each feedback loop must have only a single input point implemented by an *Input ±Fbk* Macro and correctly employ the ZDF *Output* Macros. The guidelines simply provide a framework for following these hard restrictions.

## Saturators

In order to build filters which can self-oscillate one needs to introduce saturating nonlinearities into the filters. The ZDF toolkit provides two different versions of tanh saturator and a saturating integrator:

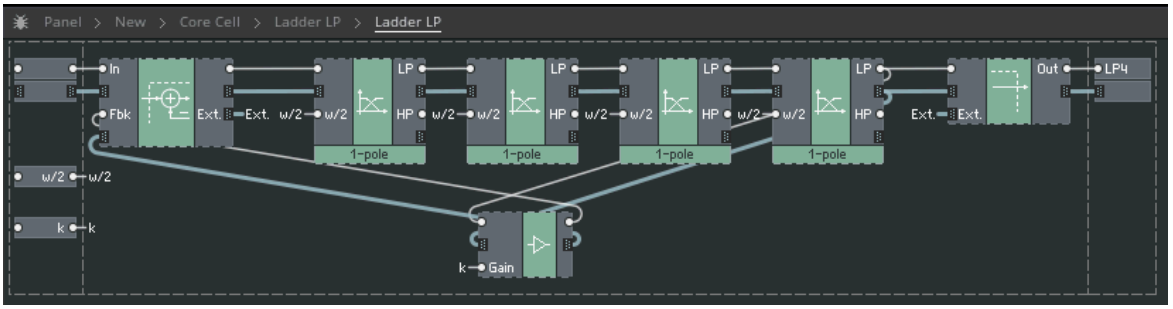

Saturating ZDF primitives.

The L input controls the saturation level (the horizontal asymptote of the saturation curve).

## Filter Primitives

Often filters are used as building blocks for more complex filters. The *Filter Primitives* submenu of the ZDF toolkit contains some of such low-level filters.

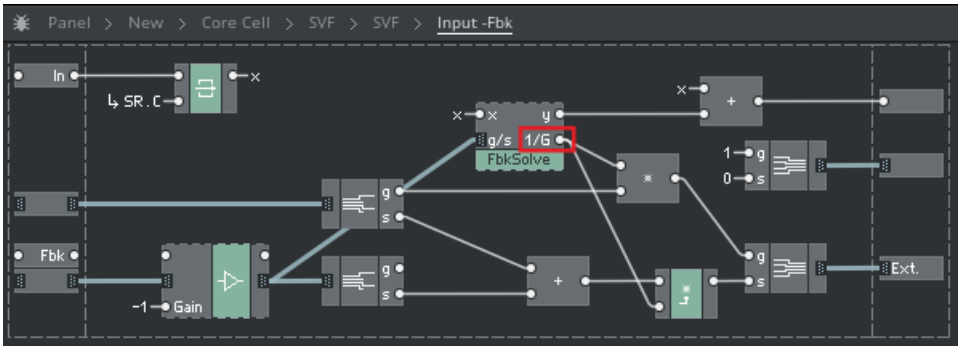
The following example illustrates how a ladder filter is built from four 1-pole filters.




Ladder filter Structure.

## Instantaneous Instability

At certain extreme parameter settings (typically, in cases of excessive positive feedback) the routine application of the ZDF approach will not deliver correct results anymore. The mathematical reasons for this can be found in the corresponding literature. What this means practically is that the denominator of the zero delay feedback equation becomes zero or negative, which can be checked by monitoring the denominator value inside the feedback solver, which is located inside *Input ±Fbk* Macros:




The feedback solver inside the Input -Fbk Macro. The denominator value is available at the 1/G port of the solver.

The instantaneous instability normally happens far in the seloscillation range of the filters or in some untypical designs or usage cases. Some of the examples are the following.

- Ladder filter with excessive unclipped negative resonance settings (since the ladder filter employs an inverted feedback, negative resonance means positive feedback).
- SVF with excessive unclipped resonance settings (in the selfoscillation range the internal damping of the SVF is negative, which together with the inversion of the feedback signal results in a positive feedback).
- An attempt to implement a non-transposed Sallen–Key filter by feedback loop decomposition. The Sallen–Key filter has two feedback points. Since each feedback point needs to be implemented by a dedicated *Input ±Fbk* Macro, and since there can be only one such Macro per feedback loop, it is not implementable in a straightforward manner using the ZDF toolkit. One could attempt a workaround, by decomposing it into two feedback loops, one nested into the other. However, the inner feedback loop employs positive feedback, which at certain (not excessive at all) settings can become instantaneously unstable on its own. It is the presence of the outer loop which stabilizes the inner feedback again. The instantaneous instability of the inner loop results in the ill-conditioning of the entire Structure, and the filter may produce INFs and NaNs at its output and in its state as the result.

  This problem with the non-transposed Sallen–Key filter is not the limitation of the ZDF approach as such, but the one of the toolkit. In principle, with some manual 'hacking' the toolkit also could allow a non-transposed Sallen–Key filter implementation.

It is left to the responsibility of the Structure builder to clip the filter parameters, if necessary, so that the instantaneous instability does not occur.

### 6.4.3 TF toolkit

The 'transfer function' toolkit (located under *Library > Audio > Filter > Tranfer Func > TF Toolkit*) contains the building blocks of the transfer function Macros in the library and can be used to modify the existing transfer function Macros or build the new ones.

The absolute majority of the Macros in the toolkit are simply implementing the complex math operations. The complex values themselves are represented by a Bundle containing *Re* and *Im* fibers.

The special 'input' and 'output' Macros of the toolkit are the following Macros.

- **s**: This Macro takes the specified frequency *f* and the specified cutoff frequency *F* and produces an imaginary value *j f / F*. This complex value specifies the point on the *s*-plane where the (unit cutoff) transfer function needs to be evaluated.


- **to Polar**: This Macro converts a complex value *z* to its absolute magnitude |*z*| and argument arg *z*. Normally used to obtain amplitude and phase responses from the frequency response value.