REAKTOR CORE

Tutorial / Benutzerhandbuch

Wenn Sie diesen Text lesen, bedeutet das wahrscheinlich, dass Sie diese Software gekauft haben. Es sind Menschen wie Sie, die es uns ermöglichen, weiterhin großartige neue Produkte zu entwickeln und zu pflegen. Dafür vielen Dank!

Die Autoren dieses Handbuchs: NATIVE INSTRUMENTS und Len Sasso
Übersetzung: Christoph Laue

Germany

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

USA

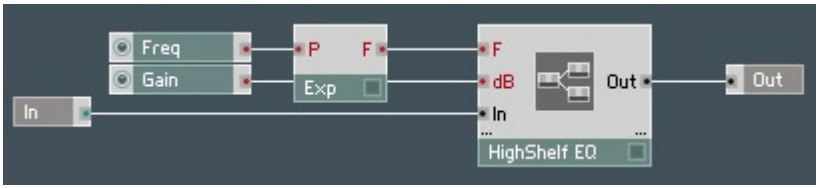
Native Instruments USA, Inc.
5631 A Hollywood Boulevard
Los Angeles, CA 90028
USA
info@native-instruments.com
www.native-instruments.com

Inhalt

1. Erste Schritte in REAKTOR Core

1.1. Was ist REAKTOR Core

REAKTOR Core ist eine neue Funktionsschicht innerhalb von REAKTOR mit einem neuen, anderen Funktionsumfang. Weil es auch noch eine ältere Funktionsschicht gibt, werden wir diese fortan *Core Level* und *Primary Level* nennen. Wenn wir von einer "Primary-Level-Struktur" sprechen, bezeichnet dies die innere Struktur eines Instruments oder eines Macros, aber nicht die eines Ensembles.Die Funktionen von REAKTOR Core sind nicht direkt kompatibel mit denen von REAKTORs Primary Level. Daher müssen zwischen diesen Ebenen Schnittstellen geschaffen werden. Diese Arbeit übernehmen die so genannten *Core Cells.*Core Cells existieren innerhalb von Primary-Level-Strukturen und gleichen in Aussehen und Verhalten den eingebauten Modulen des Primary Level. Hier sehen Sie eine Beispiel-Struktur, die eine *HighShelf EQ Core Cell* verwendet. Diese unterscheidet sich von dem eingebauten Modul auf dem Primary Level durch die Art, wie sie auf Frequenz- und Boost-Regler reagiert:

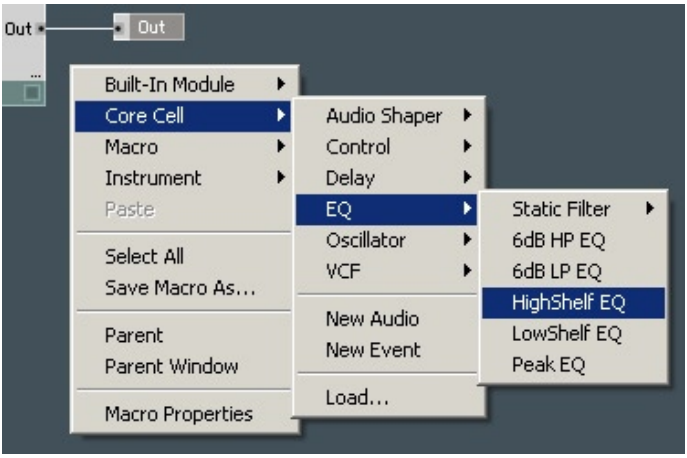


Innerhalb der Core Cells finden sich REAKTOR-Core-Strukturen, die eine effiziente Möglichkeit zur Verfügung stellen, maßgeschneiderte Low-Level-DSP-Funktionen zu implementieren und auch größere Signalbearbeitungs-Strukturen zu entwickeln. Wir werden uns diese Strukturen später noch genauer ansehen. Obwohl eine der Spezialitäten von REAKTOR Core die Fähigkeit zum Aufbauen von Low-Level-DSP-Strukturen ist, beschränkt sich das Einsatzgebiet nicht darauf. Für den Fall, dass Sie kein DSP-Experte sind (und auch nicht vorhaben, einer zu werden), stellen wir Ihnen eine Bibliothek mit vorgefertigten Modulen zur Verfügung, die Sie innerhalb der Core-Strukturen auf ähnliche Weise verbinden können, wie Sie es vom Primary Level kennen. Außerdem gibt es eine Sammlung von vorgefertigten Core Cells, die Sie sofort in Primary-Level-Strukturen einsetzen können.

In Zukunft werden wir bei NATIVE INSTRUMENTS den Schwerpunkt weniger auf die Entwicklung neuer Primary-Level-Module legen. Stattdessen verwenden wir nun unsere neue REAKTOR-Core-Technik und liefern die Module in Form von Core Cells. Sie finden in der Core Cell Library bereits einen Satz neuer Filter, Hüllkurven, Effekte etc.

1.2. Wie Sie Core Cells verwenden

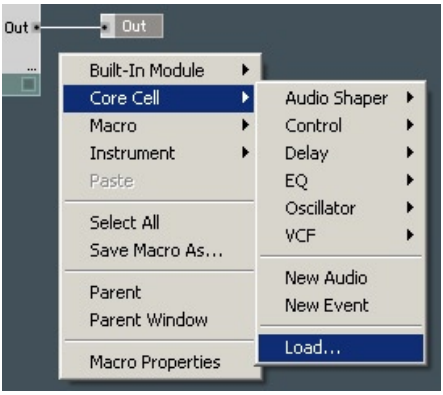
Auf die Core Cell Library können Sie von Primary-Level-Strukturen aus zugreifen, indem Sie einen Rechts-Klick in den Hintergrund ausführen und das Untermenü *Core Cell* ausklappen:



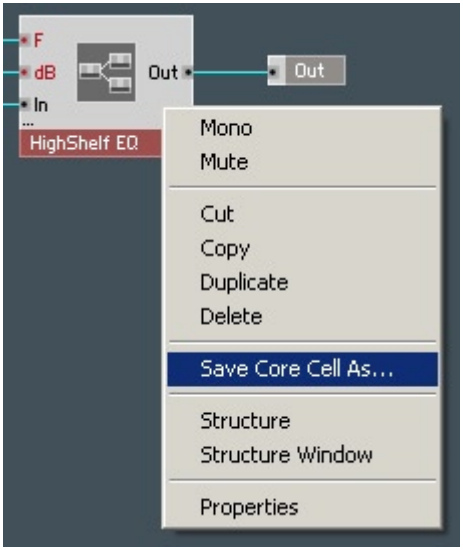
Wie Sie sehen, enthält die Liste alle möglichen Arten von Core Cells, die Sie auf dieselbe Art verwenden können wie die eingebauten Primary-Level-Module.

Eine wichtige Beschränkung gilt beim Einsatz von Core Cells: Sie dürfen Core Cells nicht innerhalb von Event-Loops verwenden. Jeder Event-Loop, der durch eine Core Cell entsteht, wird von REAKTOR blockiert.

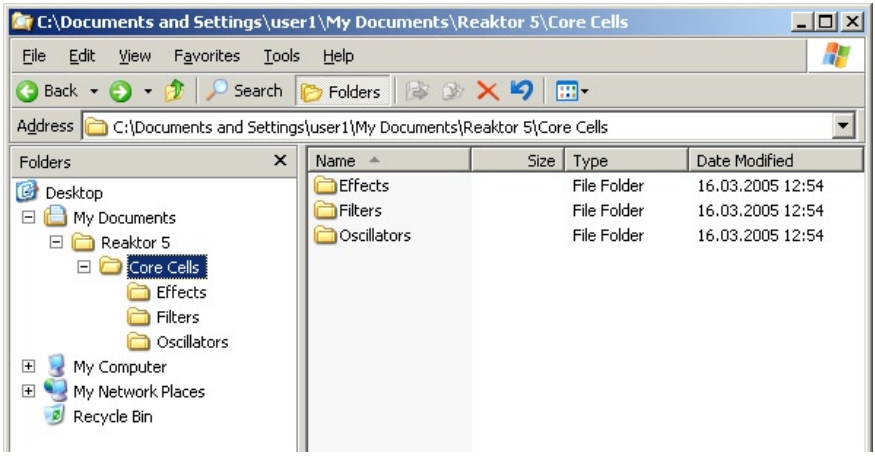
Sie können auch Core Cells einsetzen, die nicht in der Library enthalten sind. Verwenden Sie dazu den Befehl *Load...* aus dem Menü *Core Cell:*



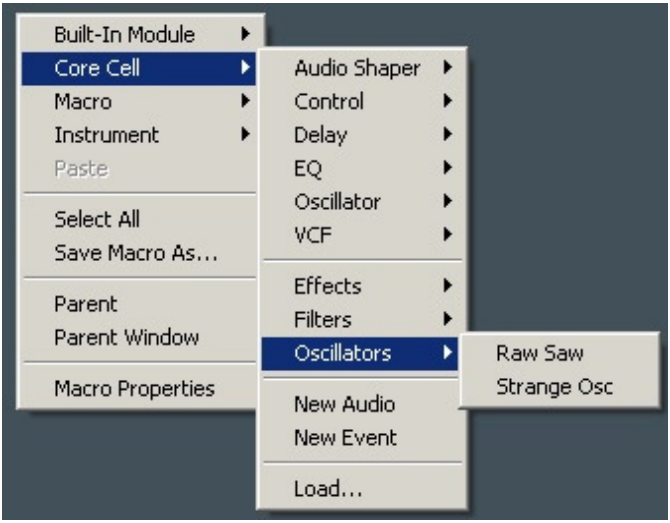
Core Cells, die Sie selbst erstellt oder verändert haben, möchten Sie vielleicht speichern. Führen Sie dazu einen Rechts-Klick auf eine Core Cell aus und wählen Sie aus dem Menü den Eintrag *Save Core Cell As*:



Sie können Ihre eigenen Core Cells auch dem Menü hinzufügen, sodass Sie nicht den Umweg über den Befehl *Load* nehmen müssen, um sie aufzurufen. Dazu legen Sie Ihre Core Cells in das Unterverzeichnis *"Core Cells"* in Ihrem User-Library-Ordner. Es empfiehlt sich aber, die Core Cells in Gruppen zu organisieren, anstatt sie einfach in das Unterverzeichnis "Core Cells" zu legen. Hier sehen Sie ein Beispiel für eine solche Sortierung in Gruppen:



"My Documents\REAKTOR 5" ist der User-Library-Ordner im obigen Beispiel. Auf Ihrem Computer kann dieser Ordner anders heißen, je nachdem, was Sie während der Installation von REAKTOR angegeben oder nachträglich in REAKTORs Voreinstellungen geändert haben. Innerhalb dieses User-Library-Ordeners sollten Sie aber auf jeden Fall einen weiteren Ordner namens *"Core Cells"* finden. Falls nicht, müssen Sie diesen Order von Hand anlegen.
Nun sehen Sie auf der Abbildungg innerhalb des Ordners *"Core Cells"* drei weitere Ordner namens *"Effects", "Filters"* und *"Oscillators"*. In diesen Ordnern befinden sich Core Cells, die im User-Bereich des Menüs *Core Cell* (unterhalb des obersten Trennstrichs) angezeigt werden:



Der Inhalt des Menüs ergibt sich aus dem Inhalt des User-Library-Ordners. Dieser wird beim Start von REAKTOR einmal gescannt. Wenn Sie neue Dateien in diesen Ordner oder in einen seiner Unterordner legen, müssen Sie REAKTOR beenden und erneut starten, damit diese neuen Core Cells im Menü erscheinen.

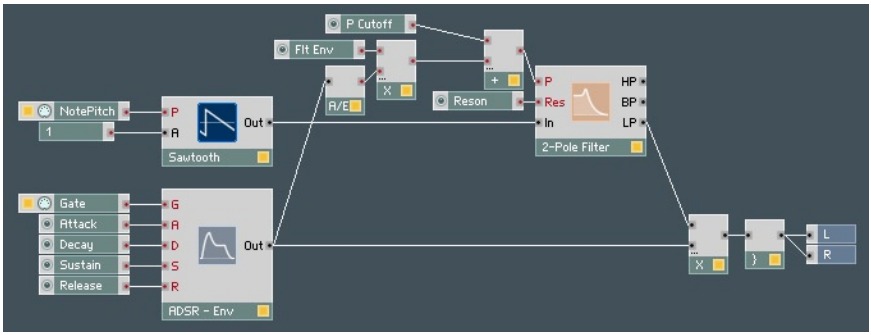
Leere Ordner erscheinen nicht im Menü. Damit ein Ordner im Menü angezeigt wird, muss er mindestens eine Datei enthalten.

Legen Sie unter keinen Umständen Ihre eigenen Dateien in der System Library ab! Das System-Library-Verzeichnis kann sich bei der Installation

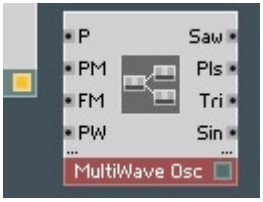
von Updates ändern oder sogar komplett ausgetauscht werden, und dann wären alle Ihre Dateien verloren. Die User Library ist der richtige Ort für alle Erweiterungen, die nicht Bestandteil der Software selbst sind.

1.3. Der Einsatz von Core Cells in der Praxis

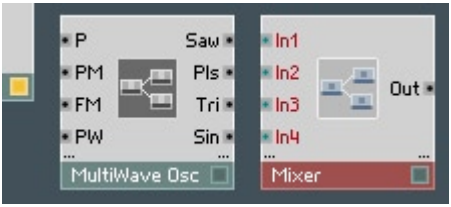
Wir werden nun ein REAKTOR-Instrument nehmen, das ausschließlich aus Primary-Level-Modulen besteht, und es modifizieren, indem wir einige Core Cells einsetzen. Um das Beispiel nachzuvollziehen, finden Sie im Ordner *Core Tutorial Examples* in REAKTORs Installationsverzeichnis das Ensemble *One Osc.ens* und öffnen Sie es. Dieses Ensemble besteht aus nur einem Instrument mit der folgenden internen Struktur:



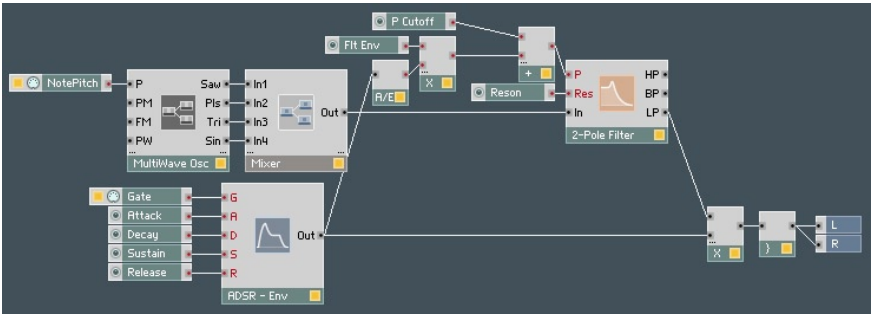
Wie Sie sehen, handelt es sich um einen sehr einfachen subtraktiven Synthesizer mit einem Oszillator, einem Filter und einer Hüllkurve. Wir werden zunächst den Oszillator durch einen anderen, leistungsfähigeren ersetzen. Dazu klicken Sie in den Hintergrund und wählen aus dem Menü *Core Cell > Oscillator > MultiWave Osc:*



Die wichtigste Funktion dieses Oszillators ist, dass er gleichzeitig analoge Wellenformen erzeugt, die in der Phase gekoppelt sind. Wir werden den Mix dieser Wellenformen anstelle des einzelnen Sägezahn-Oszillators verwenden. Glücklicherweise gibt es schon ein fertiges Mixer-Modul: *Insert Macro > Classic Modular > 02 – Mixer Amp > Mixer – Simple – Mono:*

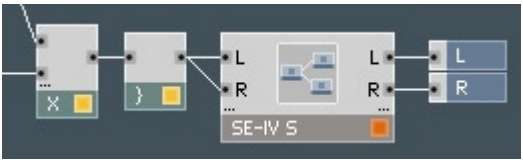


Lassen Sie uns nun den Mixer und den Oszillator verbinden und mit dieser Kombination den Sägezahn-Oszillator ersetzen:

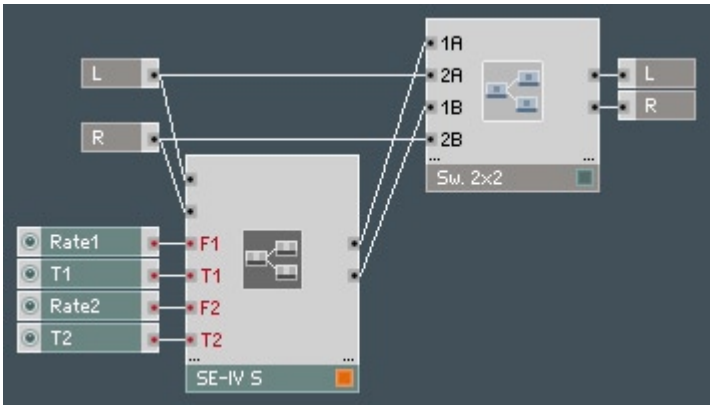


Schalten Sie in den Panel View. Sie können nun die vier Fader des Mixers verwenden, um das Mischungsverhältnis der Wellenformen zu variieren.
Wir werden noch eine weitere Veränderung an dem Instrument vornehmen – lassen Sie uns einen auf REAKTOR Core basierenden Chorus-Effekt hinzufügen. Wir sagen hier übrigens "auf REAKTOR Core basierend", weil der Chorus selbst zwar als Core Cell angelegt ist, der Teil mit den Bedienelementen (Panel Controls) aber immer noch die Funktionen des Primary Level nutzt. Das kommt daher, dass REAKTOR Core keine Möglichkeit zum Definieren eigener Bedien-Panels enthält – die Panels müssen Sie auf dem Primary Level erstellen.

Wählen Sie aus dem Menü *Insert Macro > Building Blocks > Effects > SE-IV Chorus* und setzen Sie das Chorus-Modul nach dem Voice Combiner ein:

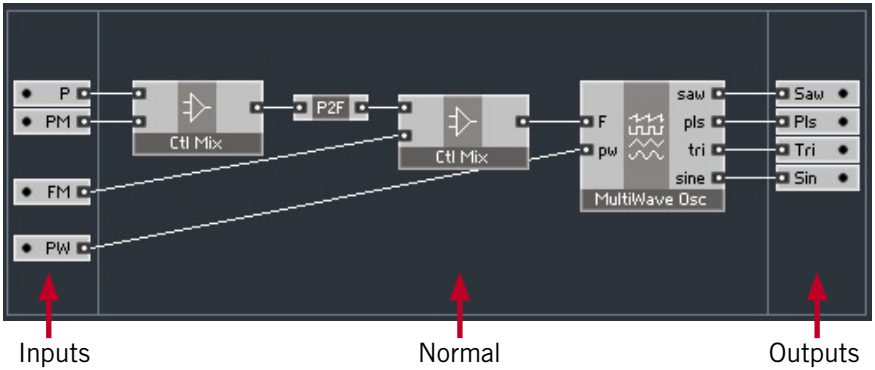


Wenn Sie sich das Innere des Chorus ansehen, erkennen Sie die Core Cell und die Panel Controls:



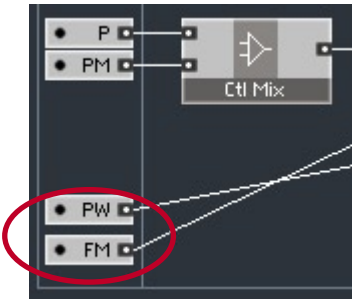
1.4. Grundlegende Bearbeitung von Core Cells

Nun erfahren Sie einige Dinge über das Bearbeiten von Core Cells. Wir fangen mit etwas Einfachem an, zum Beispiel mit der Anpassung einer existierenden Core Cell an Ihre besonderen Anforderungen.
Doppelklicken Sie zuerst das Modul *MultiWave Osc*, um sein Inneres freizulegen:

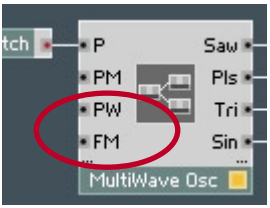


Was Sie nun sehen, ist eine REAKTOR-Core-Struktur. Die drei durch vertikale Linien voneinander abgeteilten Bereiche stehen für die unterschiedlichen Modul-Arten: Eingänge (links), Ausgänge (rechts) und normale Module (Mitte).
Die Eingänge und Ausgänge können Sie nur vertikal verschieben, und ihre relative Anordnung bestimmt immer die Reihenfolge, in der sie außerhalb des Modul erscheinen. Die normalen Module dagegen können Sie in alle Richtungen verschieben. So können Sie durch einfaches Bewegen die Anordnung in

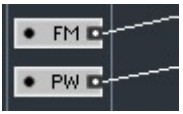
der äußeren Ansicht verändern. Probieren Sie das doch aus, indem Sie den Eingang *FM* unter den Eingang *PW* schieben:



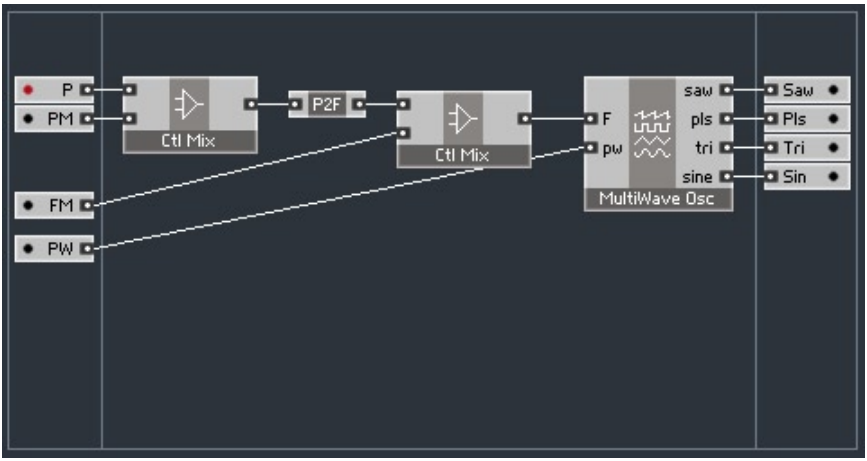
Sie können nun einen Doppelklick in den Hintergrund ausführen, um sich aufwärts zur Außenansicht der Primary-Level-Struktur zu bewegen und die geänderte Eingangskonfiguration zu sehen:



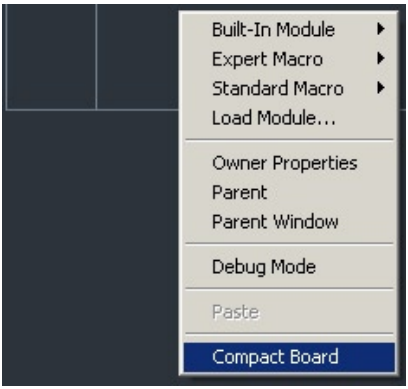
Lassen Sie uns auf die Core-Ebene zurückkehren – und vergessen Sie nicht, die ursprüngliche Anordnung der Eingänge wieder herzustellen:



Wie Sie wahrscheinlich schon bemerkt haben, erhöhen die drei Bereiche der Core Struktur automatisch die Feldgröße, um alle Module beinhalten zu können. Da Sie aber nicht wieder automatisch schrumpfen, können diese Bereiche unötig groß werden:



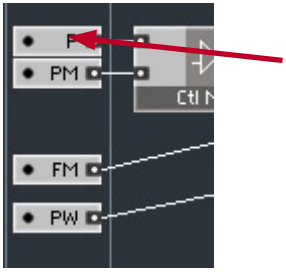
Sie können die Fläche wieder auf das passende Maß reduzieren, indem Sie einen Rechts-Klick in den Hintergrund ausführen und aus dem Menü den Eintrag Compact Board auswählen:



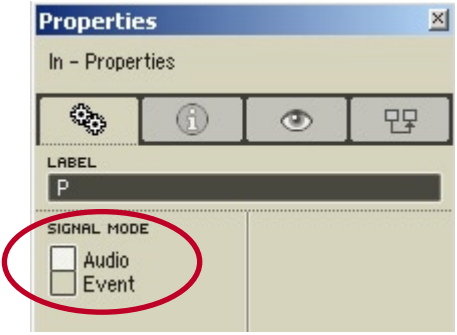
Nun, da wir gelernt haben, wie man Dinge bewegt, und sogar wissen, wie man die Port-Reihenfolge einer Core Cell verändert, wollen wir einige weitere Optionen ausprobieren.

Bei einer Core Cell, die *Audio-Ausgänge* besitzt, ist es möglich, den Typ der Eingänge zwischen Audio und Event (dazu später mehr) umzuschalten. Im obigen Beispiel haben wir das Modul *MultiWave Osc* verwendet, dessen Ein- und Augänge alle vom Typ Audio sind. Trotzdem brauchen wir sie in unserem speziellen Fall wirklich nicht als Audio-Ports, weil das einzige Ding, das mit dem Oszillator verbunden ist, ein Pitch-Drehregler ist. Wäre es da nicht viel günstiger, einige Ports auf den Typ Event umzuschalten und dadurch CPU-

Leistung einzusparen? Die offensichtliche Antwort ist: Na klar, das wäre es. Also lassen Sie uns das tun. Sie sollten zumindest die Eingänge *P* und *PM* in den Event-Modus umschalten, denn dies bewirkt die größte Entlastung der CPU. Doppelklicken Sie dazu auf das Port-Modul P, um das Properties-Fenster für diesen Port zu öffnen:



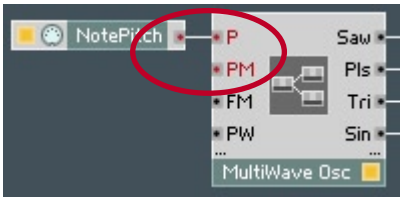
Schalten Sie (falls notwendig) das Properties-Fenster in die Ansicht Function, indem Sie auf die Schaltfläche  klicken. Sie sollten nun die Eigenschaften für den Signal Mode sehen:



Wählen Sie im Ausklapp-Menü den Eintrag *Event*. Beachten Sie, wie daraufhin der große schwarze Punkt links im Input-Modul seine Farbe von Schwarz in Rot ändert und dadurch anzeigt, dass sich der Eingang nun im Event-Modus befindet (was besser zu erkennen ist, wenn der Port nicht mehr ausgewählt ist – klicken Sie einfach irgendwo anders hin):

Klicken Sie nun auf das Input-Modul *PM*, um es auszuwählen und auch in den Event-Modus umzuschalten. Wenn Sie möchten, können Sie die beiden verbleibenden Eingänge benfalls auf Event setzen. Nachdem Sie damit fertig sind, können Sie einen Doppelklick in den Hintergrund der Struktur ausführen,

um auf das Primary Level zurückzukehren und zu sehen, wie sich die Farbe der Ports geändert hat und wie die CPU-Last sinkt.

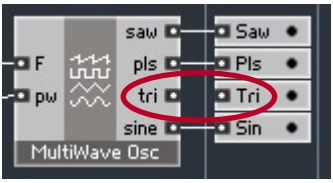


Manchmal ist es nicht sinnvoll, einen Port von einem Typ auf den anderen umzuschalten. Es ergibt zum Beispiel keinen Sinn, einen Eingang, der ein echtes Audio-Signal empfängt (das tatsächlich einen Klang enthält, nicht nur ein Kontroll-Signal wie etwa eine Hüllkurve, die mit Audio-Rate arbeitet) in den Event-Modus zu schalten. Abgesehen davon, dass das nicht nur sinnlos ist, kann es auch die Funktion des Moduls ruinieren. Ein anderer Fall, in dem es sich nicht empfiehlt, die Betriebsart der Ports zu ändern, ist gegeben, wenn ein Port tatsächlich auf Events reagieren soll. Dies kann zum Beispiel ein Event-Trigger-Signal von einer Hüllkurve sein (wie etwa Gate-Inputs von REAKTORs Primary-Level-Hüllkurven). Wenn Sie einen solchen Eingang in den Audio-Betrieb schalten, wird er nicht mehr korrekt arbeiten.
Neben den Fällen, in denen es offensichtlich sinnlos ist, den Port-Typ zu ändern, gibt es Fälle, in denen es zwar sinnvoll wäre, in denen die Module aber trotzdem nicht korrekt arbeiten, wenn Sie den Port-Typ umschalten. Diese Fälle sind allerdings entweder recht speziell oder sie sind das Ergebnis einer fehlerhaften Implementation des Moduls. Normalerweise sollte das Umschalten der Ports also funktionieren. Deshalb kommen wir nun zu folgenden Umschalt-Regel:

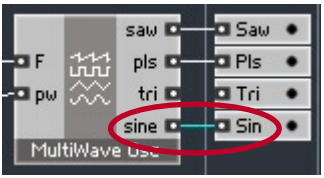
> In einer gut gestalteten Core Cell kann ein mit Audio-Rate arbeitender Eingang ohne Probleme in den Event-Modus umgeschaltet werden. Ein Event-Eingang kann nur dann in den Audio-Modus umgeschaltet werden, wenn er nicht auf Informationen des Typs “Trigger” wartet.

Ein anderer Trick, den Sie verwenden können, um CPU-Leistung zu sparen, geht so: Trennen Sie die Verbindungen von Ausgängen, die Sie nicht verwenden. Dadurch schalten Sie nicht benötigte Teile der REAKTOR-Core-Struktur ab. Auch dies nehmen Sie im Inneren vor – Verbindungen außerhalb der Struktur haben keine Auswirkungen auf die Verbindungen der Elemente innerhalb der Core-Struktur. Nehmen wir an, wir möchten in unserem Beispiel nicht alle vier Ausgänge, sondern nur Sägezahn und Puls verwenden. Wir können uns nun in das Modul *MultiWave Osc* bewegen und dort die nicht benötigten Ausgänge

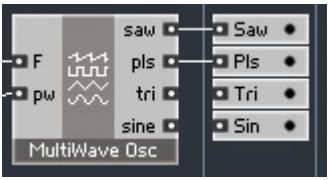
trennen. Das Trennen von Verbindungen ist in REAKTOR Core einfach: Klicken Sie auf den Eingangs-Port der Verbindung, ziehen Sie die Maus mit gedrückter Maustaste an eine Stelle, an der sich kein Ausgang befindet, und lassen Sie dann die Maustaste los. Lassen Sie uns mit dem Ausgang "Tri" beginnen – klicken Sie auf den *Eingangs*-Port des Ausgangs "Tri" und ziehen Sie die Maus mit gedrückter Taste auf einen leeren Bereich des Hintergrunds.



Es gibt noch eine weitere Möglichkeit, Verbindungen zu löschen. Klicken Sie dazu auf das Kabel zwischen dem Ausgang "sine" des Moduls *MultiWave Osc* und dem Ausgang "Sin" der Core Cell, sodass diese Verbindung ausgewählt ist (das erkennen Sie an der Farbe):

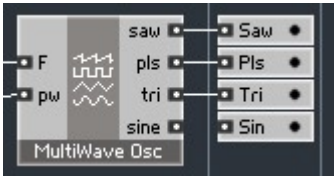


Sie können nun die Taste *Delete* drücken, um das Kabel zu löschen:



Nachdem Sie beide Kabel gelöscht haben, sollte die CPU-Anzeige noch etwas weniger Last anzeigen.

Was aber, wenn Sie es sich anders überlegen und sich entscheiden, die abgetrennten Ausgänge doch zu verwenden? Es ist einfach, die Verbindungen wieder herzustellen. Klicken Sie dazu auf einen Eingang oder Ausgang, den Sie anschließen wollen, ziehen Sie mit gedrückter Maustaste die Maus auf die "Gegenstelle", zu der die Verbindung führen soll, und lassen Sie dort die Taste los. Klicken Sie zum Beispiel auf den Ausgang "tri" des Moduls *MultiWave Osc* und ziehen Sie die Maus zum Ausgang "Tri". Die Verbindung ist wieder da:



Natürlich haben wir noch lange nicht alle Tuning-Möglichkeiten für Core Cells behandelt, aber das soll für den Einstig genügen. Sie lernen im Verlauf dieses Handbuchs noch viele weitere Optionen kennen.

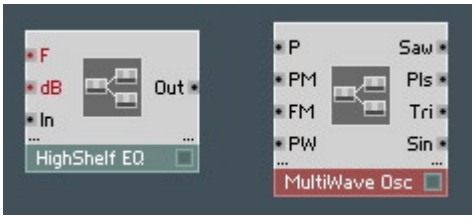
2. Der Einstieg in REAKTOR Core

2.1. Die Core-Cell-Typen Event und Audio

Core Cells existieren in zwei Arten: Event und Audio. Event-Core-Cells können nur Event-Signale des Primary Level an ihren Eingängen empfangen und als Antwort auf solche Signale nur Primary-Level-Event-Signale an ihren Ausgängen bereitstellen. Audio-Core-Cells sowohl Event- als auch Audio-Signale empfangen, aber nur Audio-Signale ausgeben. Zur Übersicht hier eine Tabelle:

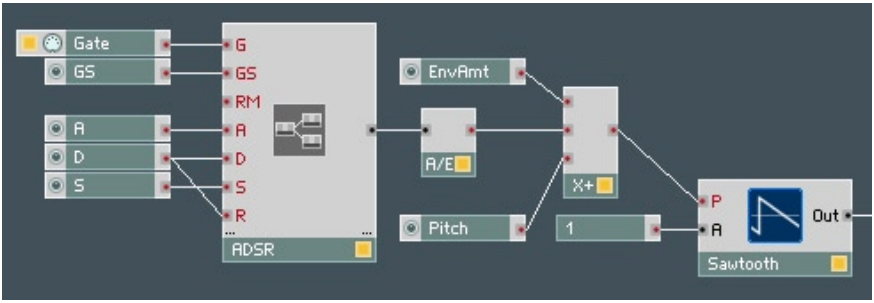
Art	Eingänge	Ausgänge	Clock Src
Event	Event	Event	Ausgeschaltet
Audio	Event/Audio	Audio	Eingeschaltet

Daher können Audio Cells Oszillatoren, Filter, Hüllkurven, Effekte und andere Dinge enthalten, während sich Event Cells nur zur Verarbeitung von Event-Signalen eignen. Die Module *HighShelf EQ* und *MultiWave Osc*, die Sie ja schon kennen, sind Beispiele für Audio-Core-Cells (was Sie auch daran erkennen, dass sie Audio-Ausgänge besitzen):



Und hier sehen Sie ein Beispiel für eine Event-Core-Cell:

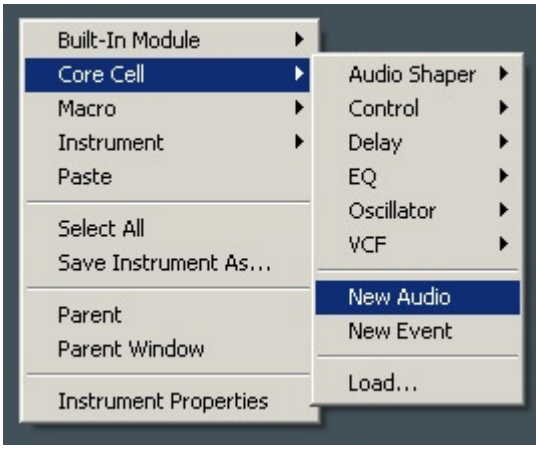
Dieses Modul ist ein parabolischer Former für Kontroll-Signale, wie er zum Beispiel beim Implementieren von Velocity-Kurven oder beim Modellieren von LFO-Signalen zum Einsatz kommen kann.
Wir haben ja bereits festgestellt, dass Core Cells vom Typ Event auf die Abarbeitung von Event-Aufgaben beschränkt sind. Weil Clock-Quellen (Clock Src) in ihnen abgeschaltet sind (siehe obige Tabelle), können sie keine eigenen Events erzeugen. Deshalb können sie nicht zum Implementieren von Module wie Event-getakteten LFOs oder Hüllkurven dienen. Wenn Sie jemals etwas derartiges vorhaben, verwenden Sie am besten eine Core Cell vom Typ Audio, deren Ausgangssignal Sie von einem Audio-nach-Event-Konverter des Primary Level auf Event-Taktung umsetzen lassen:



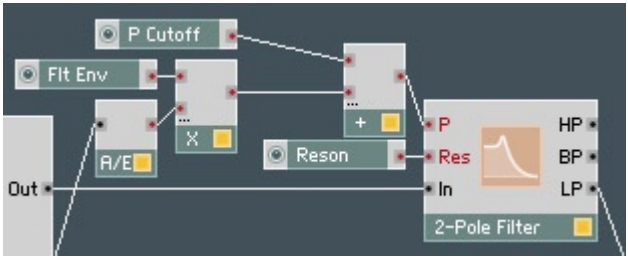
Die obige Struktur verwendet eine Audio-Core-Cell mit einer ADSR-Hüllkurve, deren Ausgangssignal auf Event-Takt umgesetzt wird, um einen Oszillator zu modulieren.

2.2. Wie Sie Ihre erste Core Cell erstellen

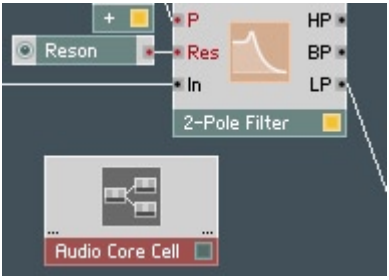
Sie können neue Core Cells erzeugen, indem Sie einen Rechts-Klick in den Hintergrund einer Primary-Level-Struktur ausführen und aus dem Menü *Core Cell > New Audio oder (für Event-Core-Cells) Core Cell > New Event:*



Wir werden nun eine Core Cell von Grund auf neu bauen, und zwar in demselben Ensemble *One Osc.ens*, mit dem Sie schon gespielt haben. Wir werden dabei die modifizierte Version des Ensembles mit dem neuen Oszillator und Chorus verwenden, die wir schon zusammengeschraubt haben. Wenn Sie diese Version nicht gespeichert haben, macht das auch nichts – Sie können alle Schritte auch an der Original-Version des Ensembles *One Osc.ens* nachvollziehen.
Wie Sie in diesem Ensemble sehen können, modulieren wir das Filter am Eingang P, der nur Event-Signale entgegennimmt. Wir verwenden nicht die FM-Version desselben Filters, weil diese erstens ein schlechteres Verhalten bei hohen Cutoff-Frequenzen zeigt und zweitens die lineare Skalierung am Eingang *FM* bei der Modulation durch eine Hüllkurve generell zu weniger musikalischen Ergebnissen führt (was oft nicht ganz korrekt als "langsame Hüllkurven" bezeichnet wird):

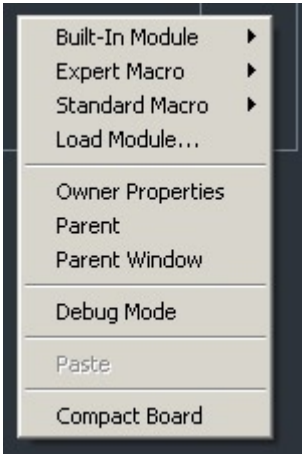


Weil wir am Event-Eingang modulieren wollen, müssen wir die Hüllkurve zunächst in ein Event-Signal umwandeln. Dazu verwenden wir einen Audio-nach-Event-Konverter (A/E). Dadurch wird die Rate unseres Kontroll-Signals ziemlich niedrig. Wir könnten natürlich auch einen Konverter mit einer deutlich höheren Taktung verwenden (der dann aber auch deutlich mehr CPU-Leistung verbraucht), aber wir werden stattdessen das Filter durch ein anderes ersetzen, das wir als Core Cell aufbauen. Alternativ hätten wir auch ein fertiges Filter aus der bestehenden Core-Cell-Library nehmen können, aber dann hätten wir uns um das Vergnügen gebracht, unsere erste eigene REAKTOR-Core-Struktur zu erstellen. Also gehen wir den nicht den einfachen Weg.
Wir beginnen mit dem Erzeugen einer neuen Core Cell vom Typ Audio, also wählen Sie *Core Cell > New Audio*. Es erscheint eine leere Audio-Core-Cell:

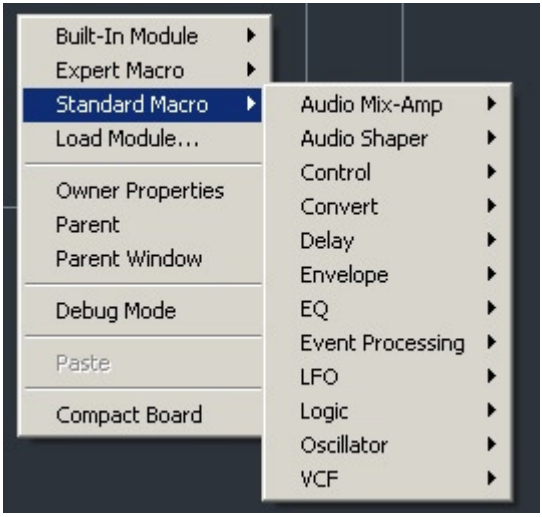


Doppelklicken Sie auf die neue Core Cell, um die innere REAKTOR-Core-Struktur zu sehen – die offensichtlich leer ist. Wie Sie sich bestimmt erinnern, sind die drei Bereiche für Eingänge (Input), Ausgänge (Output) und normale Module gedacht:

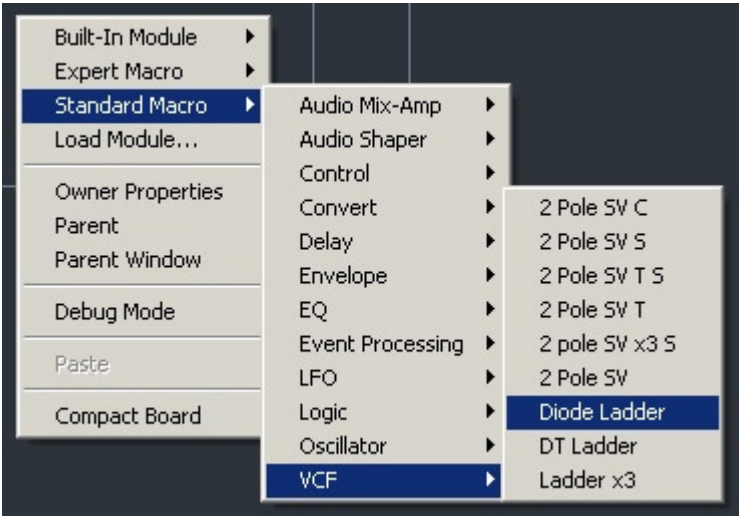
Aufgemerkt: Wir werden jetzt unsere erstes Modul in eine Core-Struktur einsetzen! Führen Sie dazu im Bereich für die normalen Module (Mitte) einen Rechts-Klick aus, um das Menü “Module Creation” aufzurufen:



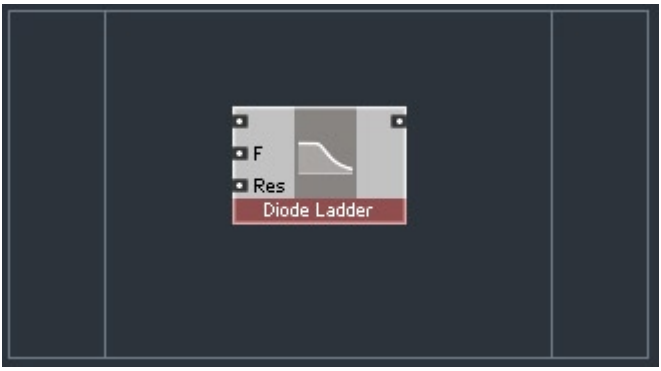
Das erste Untermenü heißt *Built In Module* und bietet Zugriff auf die eingebauten Module von REAKTOR Core, die für wirkliche Low-Level-Jobs gedacht sind und später noch behandelt werden.
Das zweite Untermenü heißt *Expert Macro* und enthält Macros, die gemeinsam mit den eingebauten Modulen für den Low-Level-Bereich zuständig sind. Das überspringen wir also auch erst einmal. Das dritte Untermenü heißt *Standard Macro* – hier scheinen wir richtig zu sein:



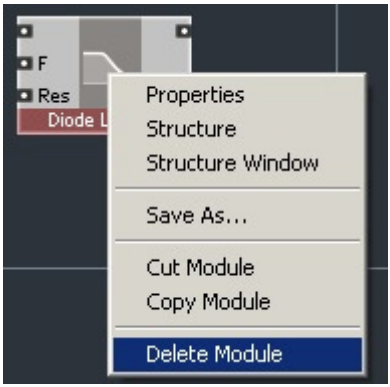
Die Abteilung *VCF* sieht ganz vielversprechend aus; schauen wir mal hinein:



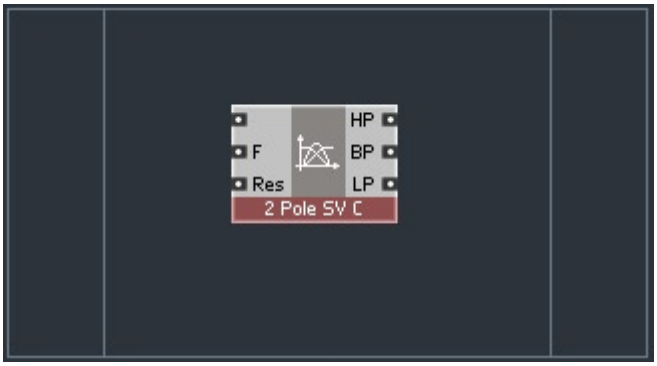
Sollten wir vielleicht mal *Diode Ladder* probieren?
Machen wir das doch einfach:



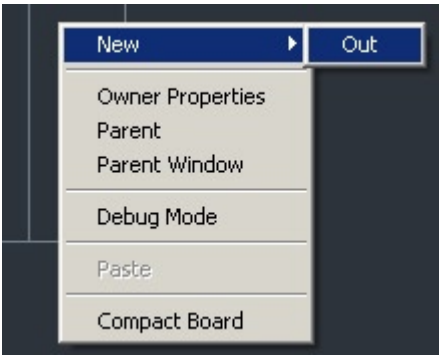
Nun, das war vielleicht nicht die allerbeste Idee,weil *Diode Ladder* wahrscheinlich ganz anders klingt als das Primary-Level-Filter, das wir ersetzen wollen. Die Dioden-Leiter *Diode Ladder* ist nämlich mindestens ein vierpoliges Filter (24 dB/ Oktave), während das Filter, das wir ersetzen wollen, ein zweipoliges Modell (mit 12 dB/ Oktave) ist. Lassen Sie uns das Modul *Diode Ladder* also wieder löschen. Dazu haben Sie zwei Möglichkeiten. Eine ist, einen Rechts-Klick auf das Modul auszuführen und aus dem Menü dem Eintrag *Delete Module* zu wählen:



Die andere Möglichkeit ist, das Modul durch einen Klick darauf auszuwählen und dann einfach die Taste *Delete* zu drücken.
Nachdem wir die *Diode Ladder* gelöscht haben, lassen Sie uns das Filter-Modul *2 Pole SV C* aus der Abteilung *VCF* einsetzen:

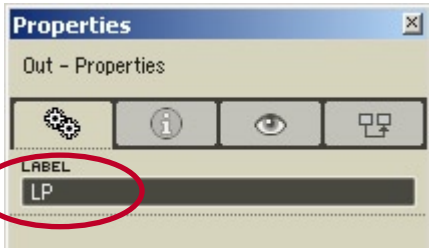


Dies ist ein zweipoliges so genanntes State-Variable-Filter (Zustandsvariablenfilter), das dem Filter, das wir ersetzen wollen, ziemlich ähnlich ist (es gibt Unterschiede, aber die sind ziemlich subtil). Wichtig ist für uns vor allem, dass wir dieses Modul mit Audio-Rate modulieren können. Dazu brauchen wir offenbar noch einige Eingänge und einige Ausgänge für unsere Core Cell.
Um genau zu sein, brauchen wir wahrscheinlich nur einen Ausgang – für das *LP*-Signal. Um diesen Ausgang zu erzeugen, führen Sie einen Rechts-Klick im Ausgangs-Bereich der Core Cell aus:

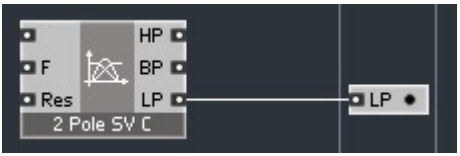


Sie können hier nur eine Art von Modulen erzeugen, also wählen Sie die einfach aus. Die Struktur sollte dann so aussehen:

Doppelklicken Sie das frisch erzeugte Output-Modul, um das Properties-Fenster zu öffnen (wenn es nicht schon offen ist).
Tippen Sie “LP” in das Feld “Label”:

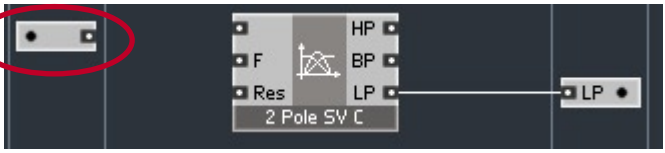


Nun verbinden Sie den Ausgang *LP* des Filters mit dem Output-Modul:

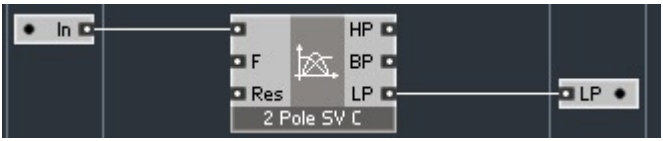


Lassen Sie uns nun mit den Eingängen beginnen. Der erste Eingang wird ein Audio-Signal-Input. Führen Sie einen Rechts-Klick in den Hintergrund des

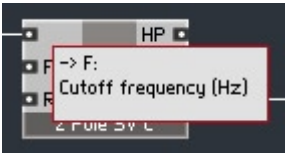
Eingangs-Bereichs aus und wählen Sie aus dem Menü *New > In*:



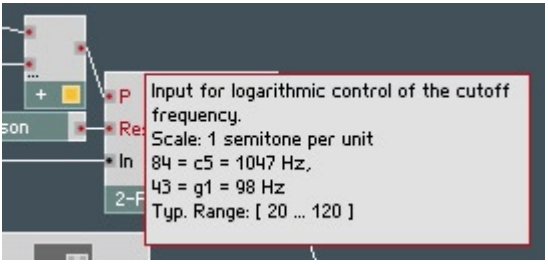
Der Eingang ist bereits vom richtigen Typ – wie Sie an dem großen schwarzen Punkt im Input-Modul sehen können, handelt es sich um einen Audio-Eingang. Sie können diesen Eingang auf dieselbe Weise, auf die Sie den Ausgang soeben in "LP" umbenannt haben, in "In" umbenennen. Verbinden Sie das Eingangs-Modul dann mit dem ersten Eingang des Filter-Moduls:



Der zweite Eingang ist eine etwas kompliziertere Angelegenheit. Wie Sie sehen können, heißt der zweite Eingang des REAKTOR-Core-Filters "F". Das steht für "Frequency", zu deutsch "Frequenz". In der Tat sehen Sie, wenn Sie den Mauszeiger eine Weile auf diesem Eingang parken (und Sie den Schalter  aktiviert haben), den Text "Cutoff frequency (Hz)":



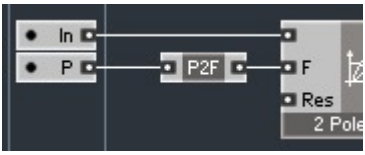
Wie wir bereits wissen, wird die Cutoff-Frequenz unseres Primary-Level-Filters von einem Eingang namens "P" kontrolliert. Wie der Info-Text erklärt, verwendet dieses Signal eine Halbton-Skala:



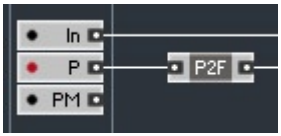
Wir müssen also offenbar eine Konvertierung von Halbtönen in Hertz durchführen. Das können wir entweder auf dem Primary Level (unter Verwendung

des Moduls Expon. (F)) oder innerhalb unserer REAKTOR-Core-Struktur tun. Weil wir gerade lernen, REAKTOR-Core-Strukturen zu bauen, wählen wir natürlich die zweite Möglichkeit. Führen Sie einen Rechts-Klick in den Hintergrund des normalen Bereichs aus und wählen Sie aus dem Menü *Standard Macro > Convert > P2F:structure*:

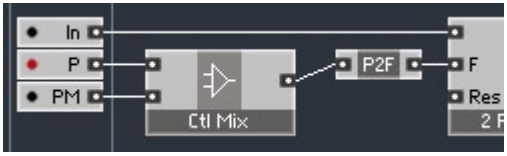
Wie der Name bereits andeutet (und auch der Info-Text verrät), konvertiert dieses Modul zwischen "P"- und "F"-Skalen – genau, was wir brauchen. Lassen Sie uns also ein zweites Eingangs-Modul namens "P" erzeugen und es unter Verwendung des Moduls *P2F* an den Eingang "F" des Filters anschließen:



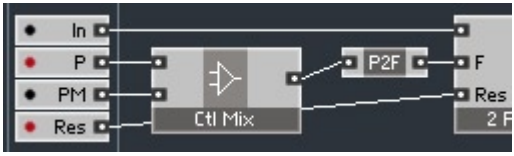
Das sollte reichen – aber Moment mal, wir haben ja einen Drehregler namens "P Cutoff" in unserem Instrument, der den Basis-Cutoff-Wert des Filters bestimmt und dann dem Modulations-Signal aus der Hüllkurve hinzugefügt wird, das wir auf dem Primary Level in ein Event-Signal konvertieren müssen, um es in den Audio-Eingang "P" des Filters leiten zu können.
Diese Konvertierung ist jetzt natürlich nicht mehr notwendig – wir können das Modul *A/E* entfernen und das Audio-Signal direkt in den Audio-Eingang "P" unseres neuen Filters einspeisen. Mit dieser Vorgehensweise haben wir zwar kein Problem, aber wir suchen ja die sportliche Herausforderung und nehmen einen anderen Weg.
Lassen Sie uns also unseren Eingang "P" im Event-Modus belassen und einen anderen Modulations-Eingang im Audio-Modus verwenden. Wenn Sie sich an die Bemerkung über "langsame Hüllkurven" erinnern, fällt Ihnen sicher auf, warum wir dieses Modul lieber "PM" als "FM" nennen und die Modulation in einer Halbton-Skala ("Pitch") stattfinden lassen. Genauso geschieht dies gegenwärtig in unserem Instrument – wir addieren unser Hüllkurven-Signal und das "P Cutoff"-Signal und leiten die Summe in den Eingang "P".
Also schalten wir unseren Eingang "P" in den Event-Modus um (Sie wissen ja schon, wie das geht) und fügen einen weiteren "PM"-Eingang hinzu, der sich offensichtlich schon im Audio-Modus befindet:



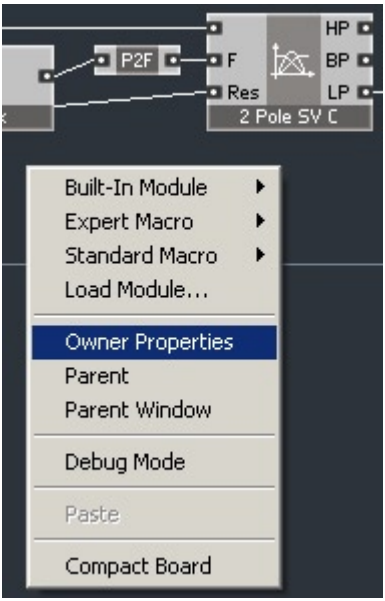
Als Anwender des Primary Level von REAKTOR würden Sie wahrscheinlich erwarten, dass wir die beiden Signale nun addieren. Das könnten wir tatsächlich tun, aber in REAKTOR Core gehört Add zu den Low-Level-Modulen und setzt die Kenntnis einiger grundlegender Arbeitsprinzipien der Low-Level-Funktionen von REAKTOR Core voraus. Diese Prinzipien sind nicht so besonders komplex und werden auch später in diesem Handbuch noch behandelt, aber im Moment brauchen Sie sie einfach nicht zu kennen. Verwenden Sie statt des Add-Moduls also einfach einen Mixer für Kontroll-Signale, zum Beispiel *Standard Macro > Control > Ctl Mix:*



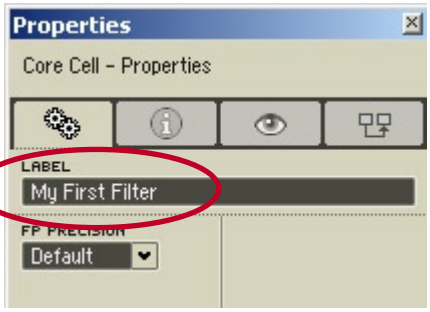
Der letzte Eingang, den wir brauchen, ist ein Resonanz-Eingang, der aber nicht mit Audio-Rate arbeiten muss. Wir wählen also eine Event-Version:



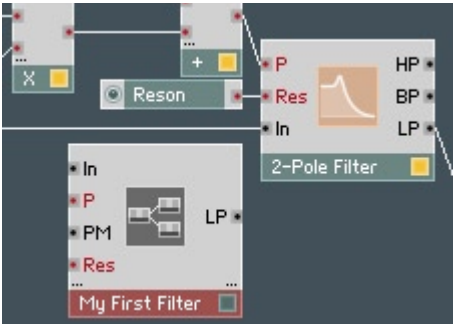
Eine weitere Sache, die wir erledigen müssen, ist, unserer Core Cell einen Namen zu geben. Um die Eigenschaften der Core Cell aufzurufen, können Sie einfach in den Hintergrund klicken, wenn das Properties-Fenster bereits geöffnet ist. Falls das Fenster nicht geöffnet ist, müssen Sie einen Rechts-Klick auf den Hintergrund ausführen und den Eintrag *Owner Properties* aus dem Menü wählen:



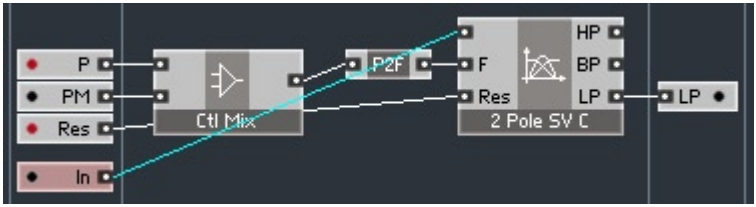
Sie können nun einen Namen in das Feld “Label” eintippen:



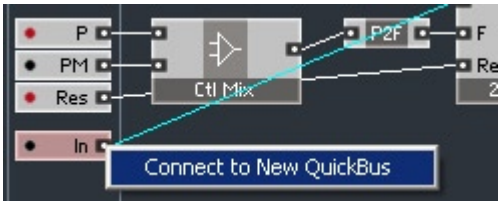
Doppelklicken Sie in den Hintergrund, um das Ergebnis zu sehen:



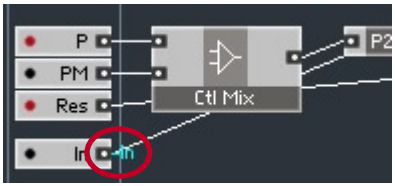
Wow, das sieht ja schon gut aus – abgesehen davon, dass der Audio-Signal-Eingang oben in der Core Cell sitzt, während er im Primary-Level-Filter unten angeordnet ist. Nun, das ist kein großes Problem, und wenn Sie es ändern wollen, ist das auch kein besonderer Aufwand; Sie wissen ja schon, wie das geht. Aber machen wir es doch einfach zusammen, dann lernen Sie unterwegs noch eine neue Funktion kennen.
Wir gehen also wieder zurück in die Core Cell und ziehen zunächst den Audio-Signal-Eingang ganz nach unten:



Das sollte schon die gewünschte Wirkung zeigen, abgesehen davon, dass ein diagonal über die ganze Struktur verlaufendes Kabel nicht besonders hübsch aussieht. Nun, das werden wir jetzt ändern.
Führen Sie einen Rechts-Klick auf das Eingangs-Modul "In" aus und wählen Sie aus dem Menü den Eintrag *Connect to New QuickBus:*



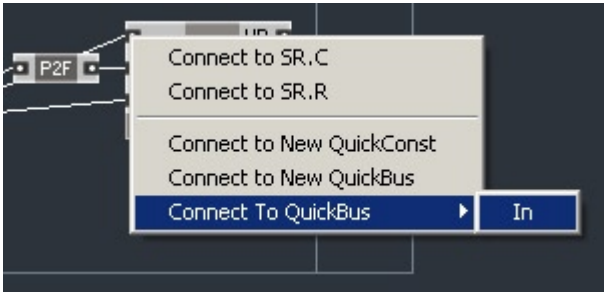
Sie sollten nun dies hier sehen:



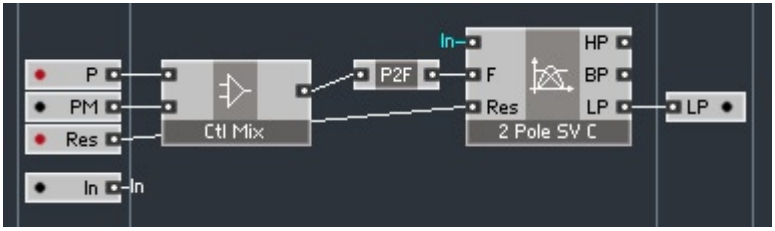
Außerdem sollte sich das Properties-Fenster öffnen und die Eigenschaften des "QuickBus" anzeigen, den Sie gerade erzeugt haben. Die nützlichste der hier angebotenen Eigenschaften der QuickBus-Verbindung ist natürlich der Name, den Sie ändern können. Die anderen Eigenschaften sind eher etwas für Fortgeschrittene, also lassen Sie jetzt erst einmal alles, wie es ist. Sie können später das Properties-Fenster öffnen, indem Sie auf den QuickBus klicken.
Obwohl Sie Ihren QuickBus umbenennen könnten, ist es wahrscheinlich

günstiger, Sie behalten den Namen bei, denn der verweist auf den Eingang, der mit dem QuickBus verbunden ist. Die QuickBusse werden lokal innerhalb der Struktur verwaltet, also brauchen Sie sich keine Sorgen darüber zu machen, falls eine benachbarte oder eingebettete Struktur einen QuickBus mit demselben Namen enthält.

Als nächstes sollten Sie einen Rechts-Klick auf den oberen Eingang des Filter-Moduls 2 Pole SV C ausführen und aus dem Menü *Connect to QuickBus > In* wählen:

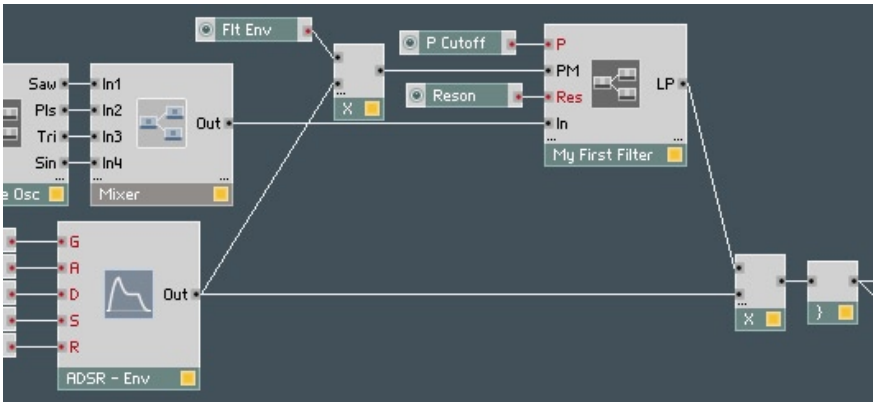


Im oben abgebildeten Menü ist “In” nichts anderes als der Name des QuickBus’, mit dem Sie eine Verbindung herstellen möchten. Sie wollen keinen neuen QuickBus erzeugen, sondern eine Verbindung mit einem bestehenden QuickBus erzeugen, und das machen Sie hier. Danach sollte die Struktur so aussehen:



Anstelle des hässlichen diagonalen Kabels haben wir nun zwei nette Referenzen, deren Beschriftung besagt, dass sie mit einem QuickBus namens “In” verbunden sind.

Jetzt können wir auf das Primary Level zurückkehren und die Struktur für die Verwendung unseres neuen Filters, das wir gerade gebaut haben, modifizieren. Die Module *Add* und *A/E* können Sie löschen. Unser Ergebnis sollte dann so aussehen:

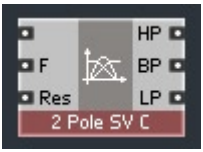


Verbraucht deutlich mehr CPU-Leistung, nicht wahr? Nun, vergessen Sie nicht, dass dieses Filter mit Audio-Rate in einer Tonhöhen-Skala moduliert wird. Wenn Ihnen das nicht gefällt, können Sie immer noch zu der alten Struktur zurückkehren oder das Filter-Modul *Multi 2 pole FM* auf dem Primary Level ("langsame Hüllkurven", erinnern Sie sich?) verwenden – aber wir hoffen, dass Sie es mögen. Falls nicht, gibt es noch ein paar andere Filter mit neuen Funktionen, die Ihnen vielleicht besser gefallen. Und wenn Ihnen keins der neuen REAKTOR-Core-Filter gefällt, können Sie immer noch eine ganze Ladung anderer REAKTOR-Core-Module ausprobieren.

2.3. Audio- und Kontroll-Signale

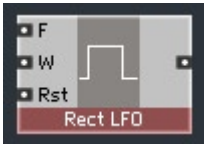
Bevor wir weitermachen, müssen wir uns zunächst eine besondere Konvention ansehen, welche die *Standard Macros* der REAKTOR-Core-Library betrifft. Die Module, die Sie in diesem Bereich finden, lassen sich am besten mit den verschiedenen Signal-Typen beschreiben: Audio, Control, Event und Logic. Wir werden Event- und Logic-Signale etwas später behandeln und uns zunächst auf die erstgenannten beiden Typen konzentrieren.
Audio-Signale sind offensichtlich Signale, die Audio-Informationen enthalten. Zu diesen gehören Signale, die an den Ausgängen von Oszillatoren, Filtern, Verstärkern und Delays abgegriffen werden. Module wie Filter, Verstärker, Sättiger (Saturators), Delays und so weiter nehmen an ihren Eingängen normalerweise Audio-Signale entgegen, die dann verarbeitet werden.
Kontroll-Signale transportieren keine Audio-Informationen, sondern werden nur verwendet, um einige Module zu kontrollieren. So geben zum Beispiel die Ausgänge von Hüllkurven und LFOs keinen Sound aus, und auch Keyboard-Pitch- und Velocity-Signale enthalten keine Audio-Informationen; sie alle können aber dazu eingesetzt werden, beispielsweise die Cutoff-Frequenz oder Resonanz eines Filters, eine Delay-Zeit oder andere Parameter zu kontrollieren. Entsprechend sind die Cutoff- und Resonanz-Eingänge eines Filters oder der Time-Input eines Delays darauf eingerichtet, Kontroll-Signale zu empfangen.

Hier sehen Sie ein Beispiel mit dem REAKTOR-Core-Filter-Modul, das Sie schon kennen:



Der obere Eingang des Filters nimmt das Audio-Signal, das gefiltert werden soll, entgegen; er erwartet daher ein Signal vom Typ Audio. Die Eingänge *F* und *Res* sind offenbar Eingänge des Typs Control. An den Ausgängen des Filters liegen auf verschiedene Arten gefilterte Audio-Signale an, also sind diese Ausgänge alle vom Typ Audio.
Ein Sägezahn-Oszillator-Modul andererseits hat nur einen einzelnen Kontroll-Eingang (für die Frequenz) und einen einzelnen Audio-Ausgang:

Und wenn wir uns das Modul *Rect LFO* ansehen, stellen wir fest, dass es zwei Kontroll-Eingänge – für die Frequenz *(F)* und für die Pulsweite *(W)* – und einen Kontroll-Ausgang besitzt (weil es oft verwendet wird, um Dinge wie Filter-Cutoff oder VCA-Pegel zu kontrollieren):



Einige Arten der Signalverarbeitung wie zum Beispiel Mixing lassen sich sowohl auf Audio- als auch auf Kontroll-Signale sinnvoll anwenden. In solchen Fällen finden Sie gesonderte Versionen der entsprechenden Macros für beide Signal-Typen. Es gibt zum Beispiel Audio-Mixer und Control-Mixer, Audio-Verstärker und Control-Verstärker und so weiter. Im Allgemeinen ist es keine besonders gute Idee, ein Modul für die Verarbeitung des jeweils anderen Signal-Typs zu missbrauchen – Sie sollten in solchen Fällen schon genau wissen, was Sie tun.

Jetzt, da wir das geklärt haben, würden wir gerne noch anmerken, dass es oft möglich ist, Audio-Signale als Kontroll-Signale aufzubereiten. Das am häufigsten anzutreffende Beispiel für ein solches "Signal-Recycling" ist die Modulation einer Oszillator-Frequenz oder eines Filter-Cutoff-Werts durch ein Audio-Signal. Das ist absolut in Ordnung, weil Sie ja in solchen Fällen das Audio-Signal als Kontroll-Signal verwenden *wollen*. Wir wagen aber die Behauptung, dass Fälle, in denen Sie ein Kontroll-Signal als Audio-Signal verwenden wollen, ziemlich selten sein dürften.

Die Abgrenzung zwischen Signalen der Typen Audio, Control, Event und Logic sollten Sie aber nicht mit der Unterscheidung zwischen Audio und Event auf dem Primary Level verwechseln. Die Klassifikation der Signale auf dem Primary Level in Audio und Event bezieht sich, grob gesagt, auf die "Verarbeitungsgeschwindigkeit" und die Tatsache, dass Audio "schneller" und verarbeitet wird und die CPU stärker belastet. Wie Sie wahrscheinlich auch wissen, gelten auf dem Primary Level für Event-Signale andere Fortpflanzungsregeln als für Audio-Signale. Der Unterschied zwischen Audio-, Kontroll- und Event-Signalen in der Terminologie von REAKTOR Core ist rein semantisch und beschreibt den "Gehalt" oder die "Bedeutung" des Signals, nicht die Art seiner Verarbeitung. Es gibt also keine direkten Entsprechungen zwischen den Nomenklaturen

von Primary Level und REAKTOR Core, aber wir können trotzdem versuchen, dieses Verhältnis zu erklären.

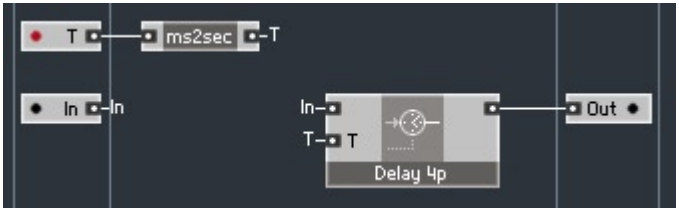
- Ein Primary-Level-Audio-Signal entspricht normalerweise entweder einem REAKTOR-Core-Audio-Signal (z. B. dem Ausgangssignal eines Oszillators oder eines Audio-Filters) oder einem REAKTOR-Core-Kontroll-Signal (zum Beispiel dem Ausgangssignal einer Hüllkurve)
- Ein Primary-Level-Event-Signal ist in der Nomenklatur von REAKTOR Core typischerweise ein Kontroll-Signal. Ein Beispiel für ein solches Signal wäre die Ausgabe eines LFOs, eines Drehreglers oder einer Quelle für MIDI-Pitch- und Velocity-Informationen.
- Manchmal entspricht ein Primary-Level-Event-Signal einem REAKTOR-Core-Event-Signal. Das typische Beispiel dafür ist ein MIDI-Gate (wie versprochen, werden wir die Event-Signale in REAKTOR Core später noch behandeln).
- Manchmal *ähnelt* ein Primary-Level-Event-Signal einem REAKTOR-Core-Logic-Signal, obwohl diese beiden Signal-Typen nicht vollständig kompatibel sind und auf jeden Fall eine Konvertierung zwischen ihnen stattfinden muss (dieses Thema werden wir später noch behandeln). Beispiele für eine solche Ähnlichkeit sind Signale, die von Primary-Level-Modulen wie *Logic AND* verarbeitet werden.

Es ist wichtig, sich vor Augen zu führen, dass Sie beim Festlegen des Typs eines Core-Cell-Eingangs zwischen den Primary-Level-Signal-Typen Audio und Event wählen. nicht zwischen Audio- und Event-Signalen in REAKTOR Core. An den Core-Cell-Ports treffen die beiden Welten aufeinander, deshalb ähneln ihre Bezeichnungen der Primary-Level-Terminologie.

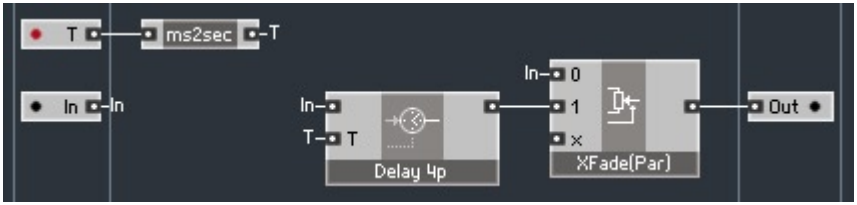
Wir werden nun etwas mehr über dieses Konzept lernen, während wir versuchen, eine Bandecho-Simulation aufzubauen. Dabei bauen zunächst wir ein einfaches digitales Echo, das wir dann erweitern, sodass es einige Funktionen eines Bandechos nachahmen kann. Lassen Sie uns zunächst eine leere Core Cell vom Typ Audio anlegen. Öffnen Sie die Core Cell und geben Sie ihr den Namen “Echo”. Das erste Modul, das wir in die Struktur einbauen, ist ein Delay-Modul. Wir werden ein 4-Point-Interpolations-Delay auswählen, weil das eine bessere Qualität liefert als die 2-Point-Version, und ein nicht-interpolierendes Delay eignet sich nicht für unsere Bandecho-Simulation. Wählen Sie also aus dem Menü *Standard Macro > Delay > Delay 4p:*

Ganz offensichtlich brauchen wir einen Audio-Eingang und einen Audio-Ausgang für unsere Delay-Core-Cell- Wir verwenden eine QuickBus-Verbindung für den Eingang und eine normale Verbindung für den Ausgang:

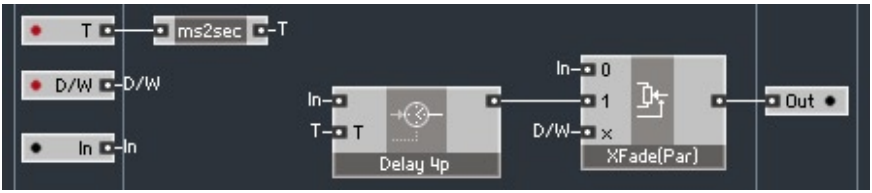
Wir brauchen außerdem einen Event-Eingang, um die Delay-Zeit zu kontrollieren. Dabei müssen wir darauf achten, dass auf dem Primary Level die Delay-Zeit normalerweise in Millisekunden ausgedrückt wird, während die Delay-Macros aus der Library von REAKTOR Core eine Angabe in Sekunden erwarten. Kein Problem für uns, denn wir haben das passende Konverter-Modul zur Hand – wählen Sie aus dem Menü *Standard Macro > Convert > ms2sec:*



Bisher haben wir aber erst ein einzelnes Echo, und dann wäre es auch noch nett, auch das Original-Signal zu hören, nicht nur den Echo-Anteil. Um das Original-Signal an den Ausgang zu leiten, müssen wir es mit dem verzögerten Signal mischen. Weil wir Audio-Signale mischen wollen, müssen wir hier einen Audio-Mixer verwenden (im Unterschied zu dem Control-Mixer, den wir beim Aufbauen der Filter-Core-Cell zum Mischen der Kontroll-Signale verwendet haben). Noch besser können wir einen speziellen Audio-Mixer-Typ verwenden, nämlich einen Crossfader, der eigens dafür entworfen wurde, Kreuzblenden zwischen zwei Signalen durchzuführen: *Standard Macro > Audio Mix-Amp > XFade (par):*

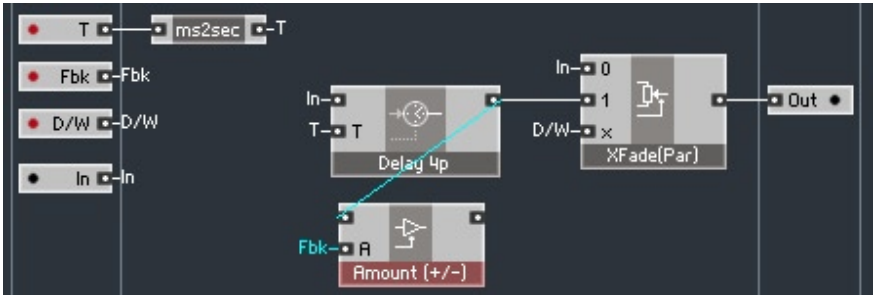


“(par)” steht für parabolisch – das erzeugt eine natürlicher klingende Kreuzblende als ein linear arbeitender Crossfader. Der Control-Input “x” des Crossfade-Macros ist noch nicht angeschlossen, wir werden ihn mit einem neuen Event-Eingang verbinden, der das Mischungsverhältnis zwischen trockenem (unbearbeitetem) und nassem (verzögerten) Signale kontrolliert. Wenn das Kontroll-Signal den Wert 0 annimmt, werden wir nur das Original-Signal hören, wenn es den Wert 1 hat, hören wir nur den verzögerten Signal-Anteil:



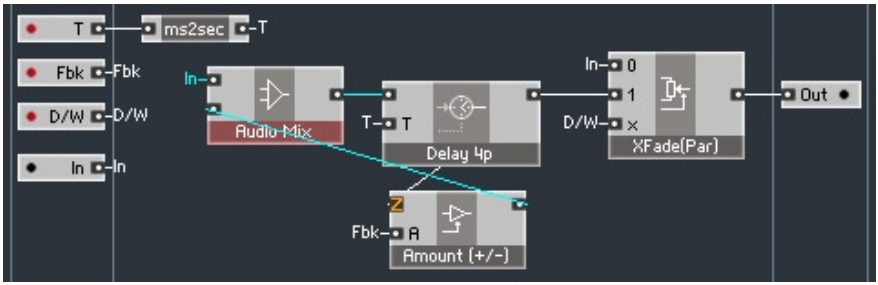
Das ist schon viel besser, denn wir können jetzt das Original-Signal und das Echo hören. Aber es ist immer noch nur ein Echo. Um mehrfache Echos zu erzeugen, müssen wir einen Bruchteil des verzögerten Signals wieder in den Eingang des Delays einspeisen. Dafür müssen wir das verzögerte Signal zunächst abschwächen. Den bekannten Richtlinien entsprechend – verwenden Sie einen Audio-Verstärker, um ein Audio-Signal abzuschwächen – wählen wir *Standard Macro > Audio Mix Amp > Amount.*

Wir nehmen das Modul *Amount*, weil wir den Betrag (Amount) des Signals kontrollieren möchten, der zurück in den Eingang des Delays geführt wird. Außerdem erlaubt uns dieser Verstärker, das Signal durch die Verwendung negativer Werte für den Betrag umzukehren. Im Gegensatz dazu wären wir mit einem Verstärker (dB), der sich eignen würde, die Lautstärke des Signals zu kontrollieren, hier nicht gut bedient, weil dieser nicht die Invertierung von Signalen erlaubt. Wir verbinden also den Amplituden-Kontroll-Eingang des Verstärkers mit einem Event-Eingang, der den Feedback-Anteil kontrolliert:

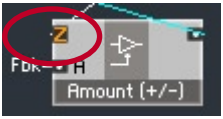


Der vernünftige Bereich für den Feedback-Anteil liegt hier etwa zwischen [0.9..0.9]. Sollten Sie dieses Delay bereits im nächsten Schritt ausprobieren wollen, seien Sie vorsichtig mit dem Feedback-Anteil, weil Sie leicht zu hohe Signalpegel bekommen (es gibt noch keine Sättigung in unserer Schaltung). Wir hätten auch zur Sicherheit einen Feedback Amount Clipper in unsere Core Cell einbetten können, der die Signalspitzen beschneiden würde, aber weil wir später ohnehin noch ein Sättigungs-Modul hinzufügen werden, erscheint das nicht notwendig. Im Gegenteil, Sie könnten mit hohen Feedback-Pegeln experimentieren und würden die Sättigung des Delays hören.

Wir müssen das Feedback-Signal mit dem Eingangssignal mischen. Ein Audio-Mixer *(Standard Macro > Audio Mix Amp > Audio Mix*) ist dafür die beste Wahl:



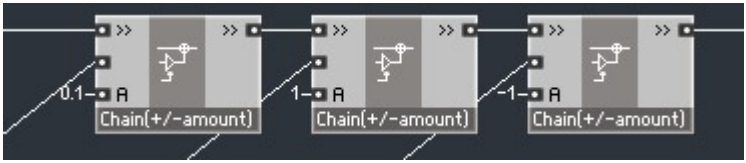
Sie fragen sich vielleicht, was mit dem oberen Eingang des Moduls Amount in der obigen Abbildung passiert ist, das nun ein großes orangefarbenes “Z” zeigt:



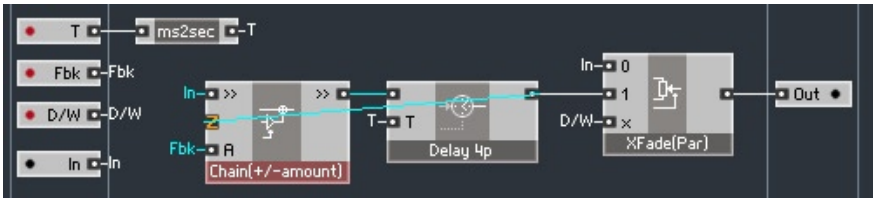
Tatsächlich könnte dieses “Z”, abhängig von der Software-Version und anderen Randbedingungen, auch an anderen Eingängen der Struktur auftauchen, aber das sollte Ihnen nicht allzu viel Kopfzerbrechen bereiten. Dieses Zeichen weist darauf hin, dass ein digitales Feedback in der Struktur auftritt, und ist für das Entwerfen aufwendigerer Strukturen von Bedeutung; dort kann es dem Struktur-Designer wertvolle Hinweise geben.

Für einfache Strukturen wie die oben abgebildete brauchen Sie diesen Hinweis normalerweise nicht zu beachten. Seine Anwesenheit bedeutet nur, dass an dem markierten Punkt eine Verzögerung von einem Sample Länge (ungefähr 0,02 ms bei einer Sampling-Rate von 44,1 kHz, noch weniger bei höheren Sampling-Raten) entsteht. Wir wagen aber zu behaupten, dass Sie es gar nicht bemerken, wenn Ihre Delay-Zeit um 0,02 Millisekunden von dem eingestellten Wert abweicht.
Lassen Sie uns zu unserer Struktur zurückkehren. Diese kann jetzt eine Serie schwächer werdender Echos erzeugen. Das sollte für ein digitales Echo in Ordnung sein, aber wir wollen die Gelegenheit nutzen und Ihnen eine andere Funktion der Library zeigen, mit der Sie Ihre Struktur verkleinern können.
Unter den Audio-Verstärkern finden sich Verstärker, die “Chain” heißen. Diese Verstärker sind in der Lage, ein vorhandenes Signal zu verstärken und es einem anderen “verketteten” Signal hinzuzumischen. Einer dieser Verstärker ist das Modul *Audio Mix Amp > Chain(amount)*, das ähnlich wie das Modul *Amount* arbeitet, aber zusätzlich “verkettetes” Mixing beherrscht:

Das Signal am zweiten Eingang dieses Moduls wird gemäß dem am Eingang “A” anliegenden Betrag abgeschwächt und dem Signal am “Ketten”-Eingang (“>>”) hinzugemischt. Das Signal am Ketten-Eingang wird nicht abgeschwächt. Solche Verstärker können Sie verwenden, um Mixer-Ketten zu bauen, wobei die Verbindungen zwischen den mit “>>” bezeichneten Ports eine Art Mixing-Bus ergeben:



In unserem Fall brauchen wir keinen Mixing-Bus, aber wir können dieses Modul verwenden, um unsere beiden Module *Audio Mix* und *Amount* zu ersetzen. Das zurückgeführte Signal wird genau wie zuvor dem über den Eingang *Fbk* bestimmten Betrag gemäß abgeschwächt und dem Eingangssignal zugemischt:

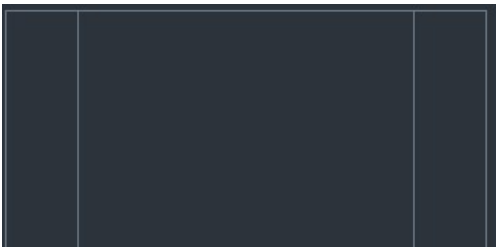


Herzlichen Glückwunsch, Sie haben einen einfachen digitalen Echo-Effekt gebaut! Im nächsten Schritt fügen wir dem etwas Bandecho-Charakter hinzu.

2.4. Wie Sie Ihre ersten REAKTOR-Core-Macros bauen

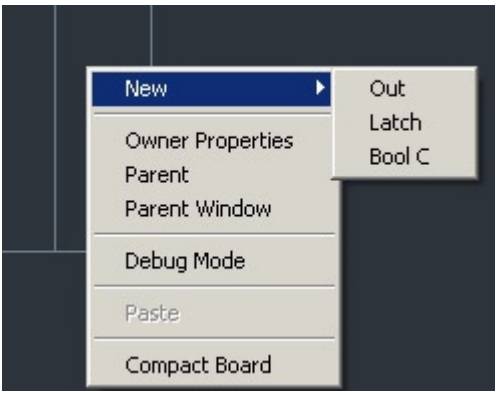
In dem Echo-Effekt, den wir gerade gebaut haben, haben wir das Macro *Delay 4p* aus der Library verwendet, das uns ein digitales Delay von hoher Qualität erzeugt. Hohe Qualität hin oder her, es klingt noch zu digital. Wir könnten es wärmer klingen lassen, indem wir verschiedene Funktionen hinzufügen, die man in einem Bandecho findet, zum Beispiel Sättigung und Flattern. Dann können wir das Macro *Delay 4p* durch ein Bandecho-Macro ersetzen, das wir jetzt bauen werden. Löschen Sie also zuerst das besagte Delay-Macro aus der Struktur. Legen Sie dann ein neues, leeres Macro an. Führen Sie dazu einen Rechts-Klick in den Hintergrund aus und wählen Sie aus dem Menü *Built In Module > Macro*:

Doppelklicken Sie das neue Macro, um hineinzutauchen. Sie sehen eine leere Struktur, ähnlich der, aus der Sie abtauchen:



Es arbeitet auch ähnlich wie das vorherige, aber es gibt einige wichtige Unterschiede. Vor allem war die vorherige Struktur die einer Core Cell, während wir uns nun in der inneren Struktur eines REAKTOR-Core-Macros befinden.

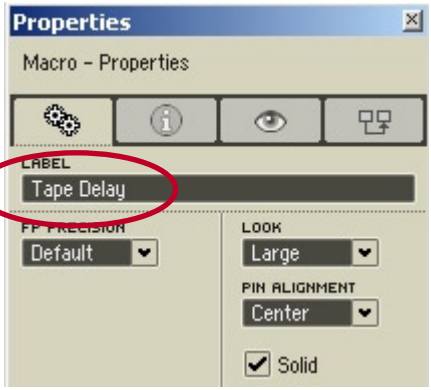
Tatsächlich haben diese Unterschiede mit dem Sortiment der verfügbaren Eingangs- und Ausgangs-Module zu tun. Die sind einfach verschieden:



Die Port-Typen *Latch* und *Bool C* werden viel später in diesem Handbuch erklärt und kommen für fortgeschrittene Aufgaben zum Einsatz. Wir interessieren uns hier nur für den ersten Typ, der einfach "Out" heißt (oder "In", wenn es sich um den Eingangs-Bereich handelt). Dies ist ein vielseitiger Port, der Signale der Typen Audio, Control, Event und Logic entgegennimmt. Genau genommen ist es dem Port egal, welcher Typ von Signal anliegt; der Unterschied zwischen den Signal-Typen ist nur für Sie als Anwender von Bedeutung, weil er beschreibt, wie das Signal verwendet werden soll. Für REAKTOR Core sind alle Signal-Typen gleich. Es gibt auch keinen Unterschied zwischen Audio- und

Event- Ein- und Ausgängen wie in der vorigen Struktur, weil wir es nun nicht mehr mit Signalen von REAKTORs Primary Level zu tun haben – das hier ist REAKTOR Core pur.

Zuerst werden wir unser neues Macro benennen. Das funktioniert genauso wie bei den Core Cells – führen Sie einen Rechts-Klick in den Hintergrund aus und wählen Sie aus dem Menü den Eintrag *Owner Properties*. Im Properties-Fenster tragen Sie in das Feld "Label" einen Namen ein:



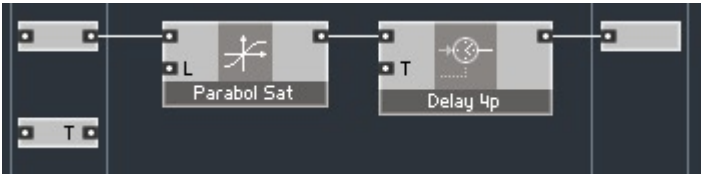
Die übrigen Eigenschaften des Macros betreffen sein Aussehen und die Signalverarbeitung.

Obwohl es Ihnen freisteht, mit den anderen im Properties-Fenster angezeigten Parametern zu experimentieren, sollten Sie das Kästchen *Solid* auf jeden Fall angekreuzt lassen. Auch Änderungen der Einstellungen im Ausklapp-Menü *FP Precision* sollten Sie sich gut überlegen. Die Bedeutung dieser Parameter werden bei den weiterführenden Themen in diesem Handbuch behandelt.

Als nächstes erzeugen wir einen Satz von Eingängen und Ausgängen für unser Macro*Tape Delay*:

Der obere Eingang wird das Audio-Eingangssignal empfangen, der untere den Zeit-Parameter. Sie haben vielleicht die zusätzlichen Ports an der linken Seite der Input-Module bemerkt; die erklären wir ein wenig später.
Als zentralen Bestandteil unseres Macros verwenden wir dasselbe Delay-Modul Delay 4p

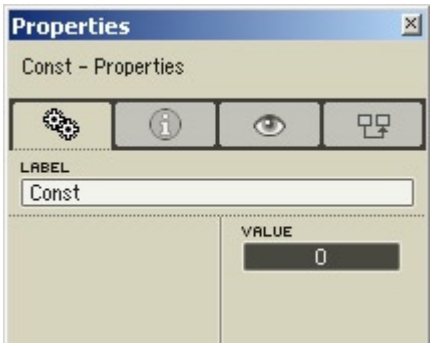
Eine einfache Emulation des Sättigungs-Effekts können wir leicht erzeugen, indem wir ein Sättigungs-Modul vor das Delay schalten. Der Sättiger (Saturator) ist eine Art Signalformer, also suchen wir ihn unter den Audio-Shaper-Modulen (weil er ein Audio-Sättiger ist). Wählen Sie also aus dem Menü *Standard Macro > Audio Shaper > Parabol Sat*:



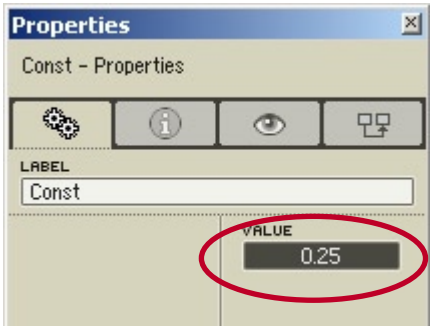
Das Eingangssignal wird nun im Bereich zwischen –1 und +1 gesättigt. Tatsächlich wird dieser Bereich vom Eingang "L" des Sättiger-Moduls kontrolliert; wenn dieser Eingang nicht belegt ist, liegt als Default-Wert 1 an. Das hört sich für Sie vielleicht überraschend an, weil Sie wahrscheinlich gewöhnt sind, dass nicht beschaltete Eingänge so behandelt werden, als ob sie kein Signal empfingen, oder, anders gesagt, als ob sie den Wert 0 empfingen. Nun, das ist nicht wirklich der Fall in REAKTOR-Core-Strukturen – die Module können hier für nicht belegte Eingänge eine besondere Behandlung vorsehen. So ist zum Beispiel für den Eingang "L" unseres Sättiger-Moduls festgelegt, dass er den Default-Wert 1 annehmen soll, wenn er nicht verbunden ist.

Nun wollen wir das Festlegen dieser Default-Werte am Beispiel unseres Eingang "T" lernen. Lassen Sie uns also für den Fall, dass der Eingang "T" nicht belegt ist, bestimmen, dass dieser Eingang so behandelt wird, als ob sein Eingangs-Wert bei 0,25 Sekunden läge. Das ist ganz leicht: Führen Sie einen Rechts-Klick auf diesen Port auf der linken Seite des Eingangs-Moduls "T" aus und wählen Sie aus dem Menü Connect to *New QuickConst*. Sie sollten nun dies hier sehen:

Zusätzlich sollte das Properties-Fenster die Eigenschaften diese Konstante anzeigen (wenn es eine andere Ansicht zeigt, klicken Sie auf die Schaltfläche ):



Tippen Sie in das Feld “Value” den neuen Wert 0.25:



So sollte die QuickConst jetzt in der Struktur aussehen:

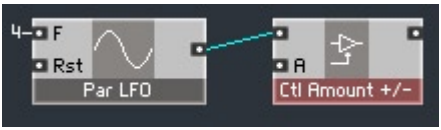
Lassen Sie uns kurz erklären, was wir gerade gemacht haben. Der Port auf der linken Seite des Eingangs-Moduls spezifiziert ein so genanntes “Default-Signal”. Das heißt, dass dieses Default-Signal als Quelle für den Eingang dient, wenn der Eingang nicht verbunden ist (außerhalb des Macros). In unserem Fall wird sich der Eingang “T” des Macros Tape Delay verhalten, als ob er den konstanten Wert 0.25 empfinge, wenn er außerhalb des Macros nicht angeschlossen ist.

Eine Verbindung mit QuickConst ist natürlich nicht die einzige mögliche Verbindung für diesen Default-Signal-Eingang. Sie können ihn auch mit beliebigen anderen Modulen der Struktur verbinden, andere Eingangs-Module eingeschlossen.
Jetzt, da wir eine Sättigung und einen Default-Wert für den Eingang "T" haben, lassen Sie uns den Band-Flatter-Effekt (Flutter) emulieren. Ein einfacher Weg dafür wäre, die Delay-Zeit von einem LFO modulieren zu lassen. Man könnte nun auf der Suche nach dem besten Flutter-Effekt mit verschiedenen LFO-Wellenformen experimentieren, aber wir schlagen hier einfach vor, dass Sie einen der in der Library enthaltenen LFOs verwenden, und zwar *Standard Macro > LFO > Par LFO:*

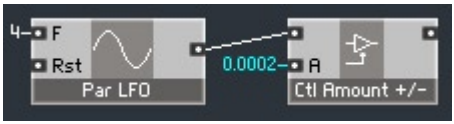
Dies ist ein parabolischer LFO, der ein Signal mit einer Form erzeugt, die einer Sinus-Welle ähnelt; er verbraucht allerdings weniger CPU-Leistung. Am Eingang "F" dieses Moduls muss ein Signal anliegen, das die Oszillations-Rate vorgibt. Wir können hier wieder QuickConst verwenden. Eine Rate von 4 Hz scheint vernünftig, also versuchen wir das mal:

Der Eingang "Rst" wird benötigt, um den LFO neu zu starten; diese Funktion brauchen wir noch nicht.

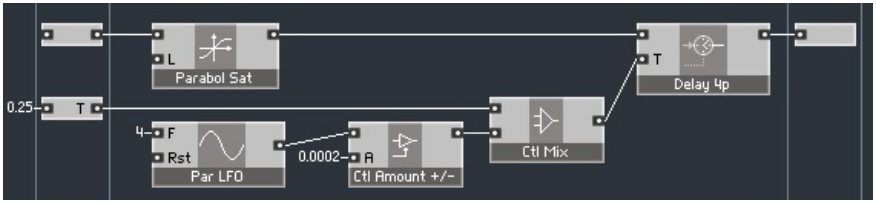
Jetzt müssen wir einen Modulations-Grad festlegen, indem wir den Ausgang des LFOs skalieren; dessen Signal variiert zurzeit im Bereich zwischen –1 und +1, und das ist viel zu viel. Wenn Sie sich erinnern, dass wir es hier mit Kontroll-Signalen zu tun haben, wissen Sie natürlich, dass wir hier entsprechend ein Amount-Modul für Kontroll-Signale verwenden müssen, ähnlich dem Verstärker-Modul *Amount*, das wir für Audio verwendet haben. Wählen Sie also *Standard Macro > Control > Ctl Amount:*



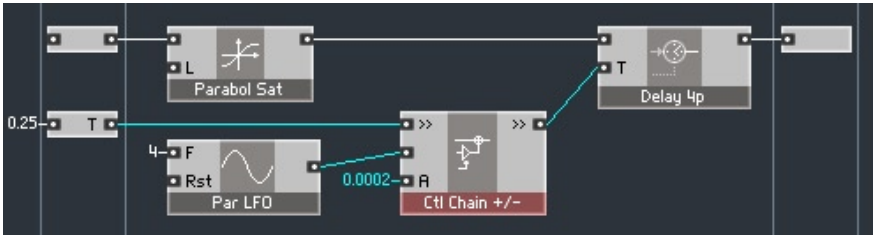
Eine Modulations-Amplitude von 0.0002 sollte gut passen, also skalieren wir das Signal auf diesen Betrag:



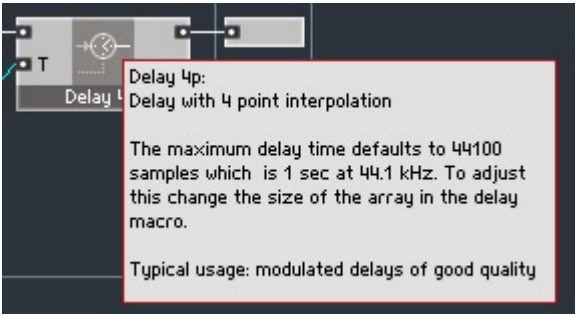
Abschließend können wir die beiden Kontroll-Signale (eins aus dem Eingang "T" und eins aus dem Modul *Ctl Amount*) mischen und sie in den Eingang "T" des Delay-Moduls einspeisen. Das bereits bekannte Modul *Ctl Mix* kann hier wieder zum Einsatz kommen:



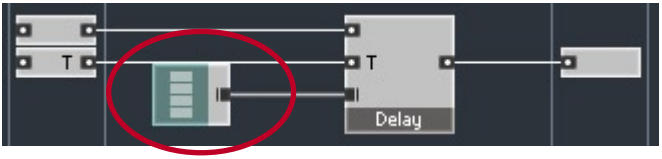
Tatsächlich haben wir aber einen ähnlichen "Ketten"-Typ von Kontroll-Mixer, wie wir ihn für Audio-Signale kennen gelernt haben. Wir könnten diesen Mixer verwenden, um die Module *Ctl Amount* und *Ctl Mix* auf ganz ähnliche Weise zu ersetzen, wie wir das im Audio-Pfad gemacht haben. Wählen Sie also *Standard Macro > Control > Ctl Chain*:



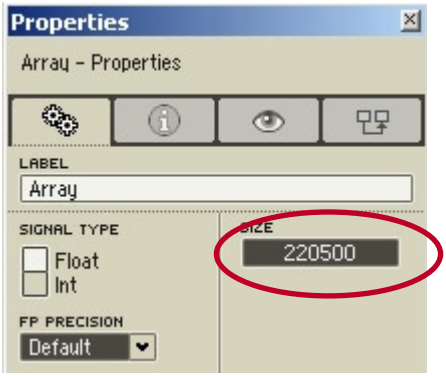
Ein letzter Schliff für unser Macro – wir werden nun die Puffergröße (Buffer Size) für unser Delay ändern, welche die maximal mögliche Delay-Zeit definiert. Wenn Sie den Mauszeigerüber das Macro *Delay 4p* halten (und der Schalter  aktiv ist), können Sie im Hinweis-Text lesen, dass die Default-Puffergröße einer Sekunde Verzögerung bei 44,1 kHz entspricht:



Lassen Sie uns diesen Betrag auf 5 Sekunden erhöhen. Wir multiplizieren also 44100*5 = 220500 (nebenbei bemerkt: da jedes Sample 4 Byte verbraucht, ergibt sich daraus, dass 220500 Samples * 4 Byte = 882000 Byte und damit fast 1 MByte belegen). Doppelklicken Sie auf das Macro *Delay 4p*:

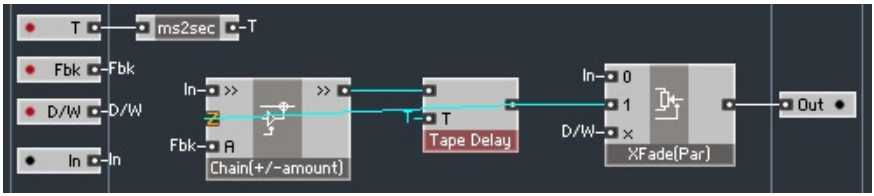


Das Modul links ist das Delay-Buffer-Modul. Doppelklicken Sie es (oder führen Sie einen Rechts-Klick aus und wählen Sie aus dem Menü den Eintrag *Show Properties*), um seineEigenschaften zu bearbeiten. Klicken Sie auf die Schaltfläche  sodass Sie in der zugehörigen Ansicht die Eigenschaft *Size* sehen. Setzen Sie diesen Wert auf 220500 Samples:

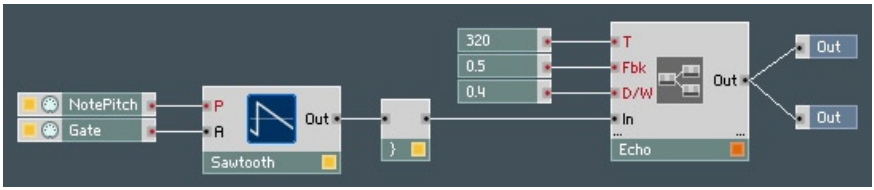


Wie Sie gerade gesehen haben, braucht ein Delay-Puffer von 5 Sekunden schon fast 1 MByte Arbeitsspeicher. Seien Sie also vorsichtig, wenn Sie die Delay-Puffergröße ändern – besonders dann, wenn Sie Delays in polyphonen Bereichen der Struktur verwenden, wo die Größe der Puffer mit der Anzahl der Stimmen multipliziert wird.

Jetzt können wir uns aus dem Macro *Delay 4p* hinaus bewegen, das Macro *Tape Delay* verlassen, das wir gerade erzeugt haben (doppelklicken Sie in den Hintergrund), und uns um die äußeren Verbindungen kümmern:



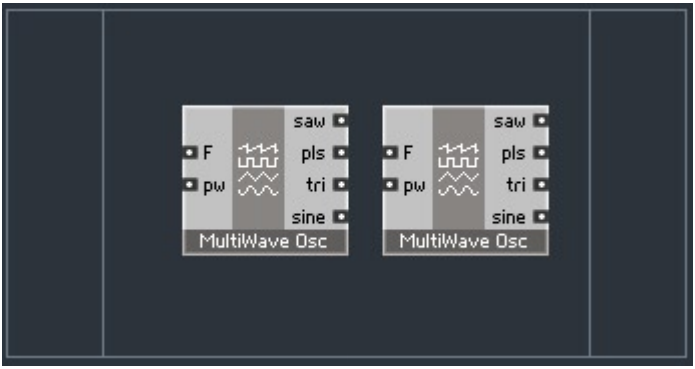
Wenn Sie das noch nicht getan haben, sollten Sie das Echo-Modul, das wir gebaut haben, jetzt ausprobieren. Hier ist eine Test-Struktur von REAKTORs Primary Level, so einfach wie möglich (beachten Sie, dass das Echo-Modul auf *mono* eingestellt ist):



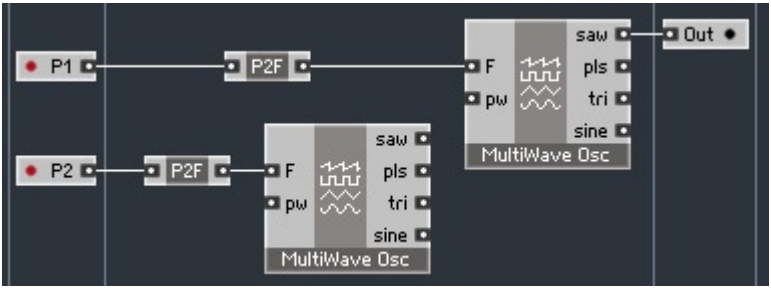
Sie können diese Struktur auf viele Arten erweitern, zum Beispiel, indem Sie Regler für die Kontrolle der Echo-Parameter vorsehen oder indem Sie einen echten Synthesizer als Signal-Quelle vorsehen.

2.5. Wie Sie Audio als Kontroll-Signal verwenden

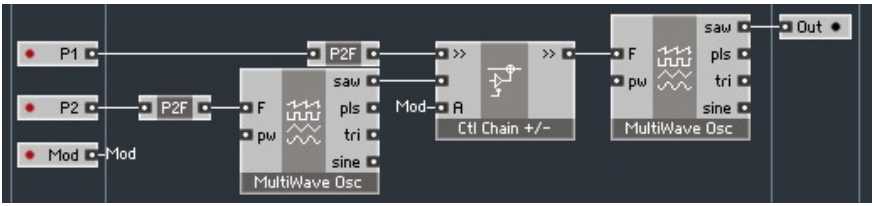
Wir haben bereits erwähnt, dass es möglich ist, Audio-Signale als Kontroll-Signale zu nutzen. Dafür sollten wir uns jetzt ein Beispiel ansehen. Dafür werden wir eine Core Cell mit einem Paar von Oszillatoren, von denen einer den anderen moduliert, erzeugen. Lassen Sie uns dafür zwei Module des Typs *Multiwave Osc* verwenden:



Wir brauchen Pitch-Kontrolle für beide Oszillatoren und einen Audio-Ausgang für den zweiten Oszillator, den wir ja hören wollen. Lassen Sie uns also die erforderlichen Eingänge und Ausgänge erzeugen:

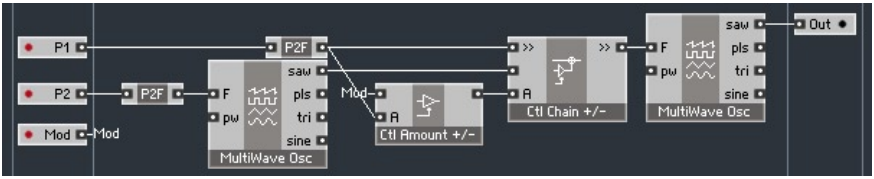


Jetzt nehmen wir das Ausgangssignal des linken Oszillators und verwenden es, um die Frequenz des rechten Oszillators zu modulieren:



Der Eingang *Mod* kontrolliert den Grad der Modulation.
Beachten Sie, dass wir das Modulations-Signal nach dem Konverter-Modul *P2F* mischen, sodass die Modulation entlang einer Frequenz-Skala stattfindet; es ist aber auch möglich, in einer Tonhöhen-Skala (Pitch) modulieren zu lassen..)

Tatsächlich ist es aber viel günstiger, den Modulations-Grad im Verhältnis zur Grund-Frequenz der Oszillation zu skalieren:



Wenn Sie sich nun die oben abgebildete Struktur hinsichtlich der vorhandenen Kontroll- und Audio-Signale ansehen, werden Sie feststellen, dass alle Signale in dieser Struktur außer den Ausgangssignalen der Oszillatoren Kontroll-Signale sind. Die von den beiden von den Oszillatoren ausgegebenen Signale sind offensichtlich vom Typ Audio. Wie auch immer, wir "missbrauchen" das Ausgangssignal des linken Oszillators als Kontroll-Signal, und zwar an dem Punkt, an dem wir es in das Mixer-Modul *Ctl Chain* einspeisen

2.6. Event-Signale

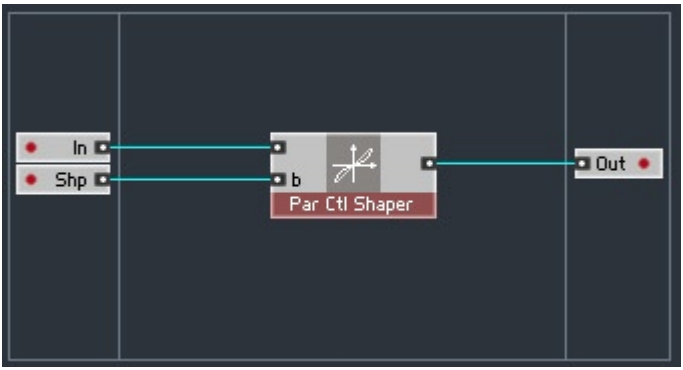
Wie wir schon festgestellt haben, hat der Begriff "Event-Signal" mehrere Bedeutungen. Sie sollten das Konzept der Event-Signale auf REAKTORs Primary Level ja schon kennen. Für ein Primary-Level-Event-Signal gibt es mehrere grundlegend verschiedene Verwendungsmöglichkeiten.

Eine Möglichkeit ist, es einfach als Kontroll-Signal zu verwenden (z. B. LFO-Ausgangssignal oder das von einem Drehregler gesendete Signal), einfach weil es weniger CPU-Last erzeugt als ein Audio-Signal auf dem Primary Level. In diesem Fall hätten Sie wahrscheinlich auch ein Audio-Signal verwenden können und damit dieselbe Wirkung erzielt. Eine andere Möglichkeit, in der Audio nicht als Alternative infrage kommt, ergibt sich dagegen, wenn Sie nicht nur an dem *Wert* interessiert sind, den das Signal transportiert, sondern auch der *Zeitpunkt*, zu dem der neue Wert geliefert wird, von Bedeutung ist – mit anderen Worten, *wann* das *Event* gesendet wird. Ein Beispiel dafür ist ein Hüllkurven-Gate-Signal (Envelope Gate) auf dem Primary Level: Die Hüllkurve wird in dem Moment getriggert, in dem das Event am Gate-Eingang *ankommt*.

Wenn wir von den Signal-Typen Audio, Control, Event und Logic in REAKTOR Core sprechen, reden wir nicht wirklich über technisch unterschiedliche Typen von Signalen (technisch sind alle Signale in REAKTOR Core gleich), sondern über unterschiedliche Einsatzbereiche der Signal-Typen. Wie Sie nun wissen, können Sie ein Primary-Level-Event-Signal als Control-, Event- oder Logic-Signal verwenden (während ein Primary-Level-Audio-Signal als Audio- oder Control-Signal verwendet werden kann).

Wir haben bereits gelernt, Primary-Level-Event-Signale in REAKTOR-Core-Strukturen einzuspeisen und sie als Kontroll-Signale zu verwenden. Die Event-Eingänge der *Audio-Core*-Cell mit dem Filter, die wir in einem früheren Kapitel

gebaut haben, sind ein gutes Beispiel dafür. Es gibt aber auch Fälle, in denen Sie eine Event-Core-Cell verwenden würden, um einige Primary-Level-Event-Signale zu verarbeiten. Hier ist ein Beispiel für eine Event-Core-Cell, in die ein Control-Shaper-Macro eingepackt ist:

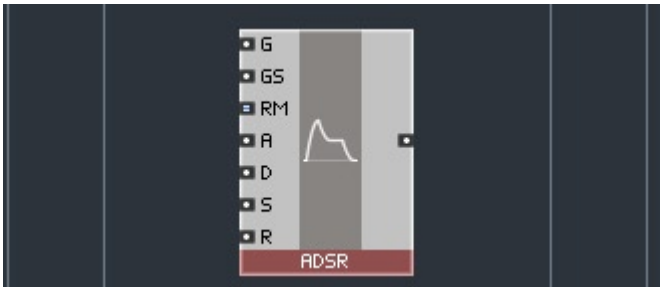


Dieser Control-Shaper empfängt ein Kontroll-Signal mit Event-Rate vom Primary Level (z. B. ein MIDI-Velocity-Signal oder ein LFO-Signal), verbiegt es gemäß dem Parameter “Shp” und stellt das Ergebnis am Ausgang bereit.

Eine wichtige Beschränkung der Event-Core-Cells haben wir früher schon kurz angesprochen: Innerhalb von Event-Core-Cells sind alle Clock-Quellen unbrauchbar. Das heißt, dass sowohl Oszillatoren und Filter als auch Hüllkurven und LFOs innerhalb dieser Core Cells nicht arbeiten. Diese Module sind wirklich darauf beschränkt, Events von REAKTORs Primary Level zu empfangen, sie zu verarbeiten und sie nach draußen zu schicken, genau wie in dem oben beschriebenen Beispiel.

Alternativ können Signale, die außerhalb aus den Primary-Level-Signalen gewonnen werden, als echte Event-Signale innerhalb von REAKTOR-Core-Strukturen verwendet werden. Wir werfen im Folgenden einen Blick auf ein paar einfache Fälle, in denen Events innerhalb von REAKTOR Core genutzt werden.

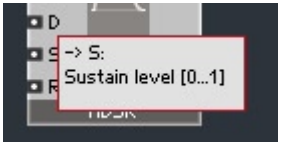
Der erste Fall wäre, eine Hüllkurve in einer REAKTOR-Core-Struktur zu verwenden. Wie Sie sicher unter Berücksichtigung der oben genannten Beschränkung schon vermuten, muss dies eine Audio-Core-Cell sein. Erzeugen Sie also eine neue Core Cell vom Typ Audio und bestücken Sie sie mit dem Modul *Standard Macro > Envelope > ADSR:*



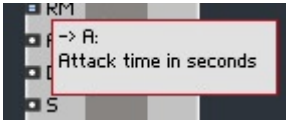
Der obere Eingang der Hüllkurve (“G”) ist ein Gate-Eingang, der ziemlich ähnlich arbeitet wir die Gates der Primary-Level-Hüllkurven. Das heißt, dieses Gate öffnet oder schließt die Hüllkurve als Reaktion auf ankommende *Events*. Kein Problem, wir erzeugen einen Event-Eingang für unsere Core Cell:

Dieser Eingang wird die vom Primary Level ankommenden Gate-Events in REAKTOR-Core-Events übersetzen.

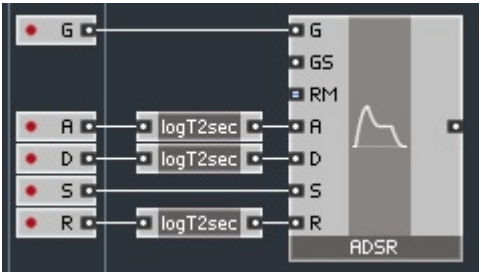
Lassen Sie uns nun die Eingänge A, D, S und R betrachten. Der Eingang “S” (Sustain-Pegel) arbeitet ähnlich wie auf dem Primary Level. Das heißt, er erwartet, dass das ankommende Signal im Bereich zwischen 0 und 1 liegt:



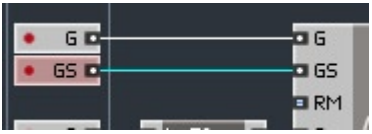
Die Eingänge A, D und R erwarten hingegen – im Unterschied zu den Primary-Level-Hüllkurven – als Eingangswert eine Zeitdauer in Sekunden



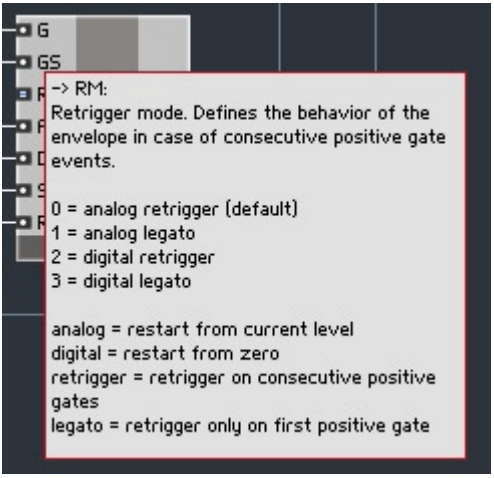
Dieses Problem können wir mit dem Konverter-Macro *Standard Macro > Convert > logT2sec* lösen, das die Hüllkurven-Dauer der Primary-Level-Hüllkurven in Sekunden konvertiert:



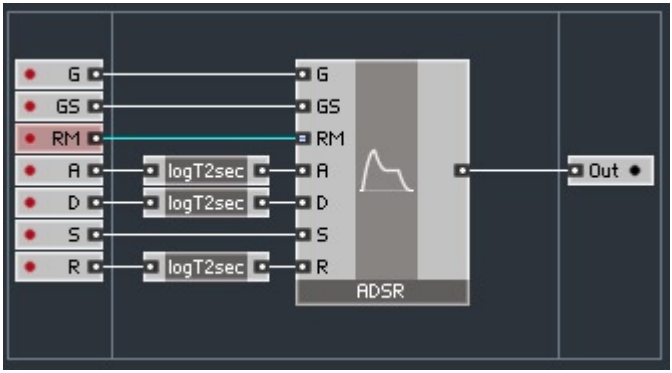
Ungeachtet der Tatsache, dass alle Eingänge in der obigen Struktur sich im Event-Modus befinden, produziert nur der erste dieser Eingänge ein Event-Signal, während die anderen Eingänge Event-Signale erzeugen.
Unsere Hüllkurve hat immer noch zwei unbenutzte Ports. Der Port *GS* bestimmt über den Betrag der Empfindlichkeit (Sensitivity) des Gates. Ist der Wert 0, ignoriert die Hüllkurve den Gate-Pegel und läuft immer mit voller Amplitude. Wenn der Gate-Pegel den Wert 1 annimmt, zeigt er die maximale Wirkung, wie auf REAKTORs Primary Level. Wir können diesen Betrag von außen mit einem weiteren Eingang kontrollieren:



Der Port *RM* bestimmt über den Retrigger-Modus für die Hüllkurve:

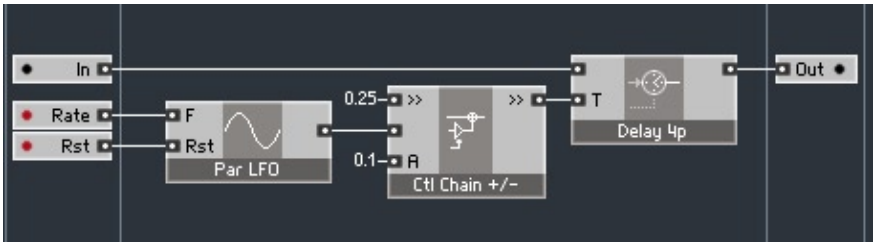


Das Aussehen des Ports “RM” unterscheidet sich von dem der anderen Ports, weil dieser Port Integer-Werte am Eingang erwartet. Wenn wir uns zum Beispiel einen typischen Eingang für eine Attack-Zeit ansehen, kann dieser eine Attack-Dauer von 1 s, von 1,5 s oder von 0,2 s erwarten. Im Gegensatz dazu erwartet der Retrigger-Modus-Port “RM” nicht wirklich Werte wie 1,2 oder 3,1. Dies wird durch das andersartige Aussehen verdeutlicht – aber es bedeutet nicht, dass wir keine normalen Signale an diesen Port anlegen dürfen. Wir können also einfach einen weiteren Event-Eingang verwenden:



Wenn die von außen kommenden Werte nicht ganzzahlig sind (und damit nicht dem Integer-Format entsprechen), werden sie auf den nächsten Integer-Wert gerundet. Das heißt zum Beispiel, ein Wert von 1,2 wird genauso behandelt wie der Wert 1.

Lassen Sie uns nun einen Blick auf ein anderes Beispiel für echte Event-Signal-Nutzung werfen:



Die obige Struktur stellt eine Art Pitch-Modulations-Effekt bereit. Der Effekt wird von einem Delay erzeugt, dessen Zeit um den Basis-Wert 250 ms um 100 ms nach oben und unten variiert. Der Rate dieser Variation wird über das Eingangs-Modul Rate kontrolliert, das die Rate des modulierenden LFOs

(in Hz) bestimmt – das ist ein reines Kontroll-Signal. Das Eingangs-Modul Rst liefert ein echtes Event-Signal, das Sie verwenden können, um den LFO neu zu starten. Der ankommende Wert spezifiziert die Restart-Phase, wobei 0 den LFO am Anfang des Durchlaufs zurücksetzen würde, 0.5 in der Mitte des Laufs und 1 am Ende. Sie können das ausprobieren, indem Sie einen Taster anschließen, der einen bestimmten Wert an dieses Eingang sendet.

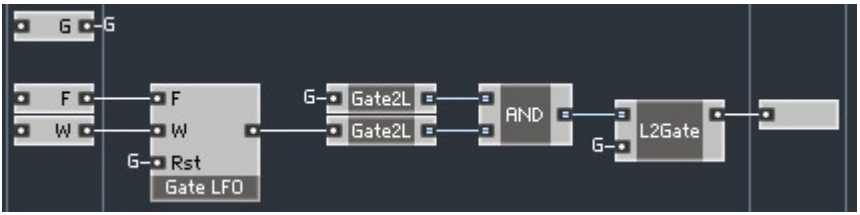
2.7. Logic-Signale

Nun, da Sie sich mit Kontroll- und Event-Signalen auskennen, ist es Zeit, eine andere Art kennen zu lernen, auf die Sie Signale in REAKTOR Core verwenden können: als *Logic-Signale*. Hier ist ein Beispiel für ein Modul, das Logic-Signale verarbeitet:

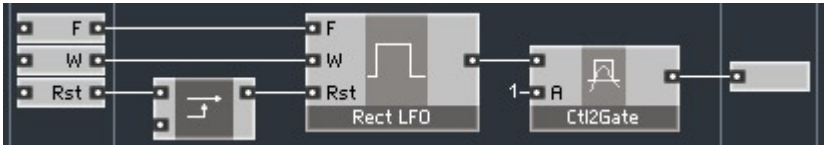
Wie Sie sehen könne, sind die Ports dieses Moduls vom Typ Integer, wie der *RM*-Eingang der Hüllkurve, die Sie schon kennen. Das hat damit zu tun, dass Logic-Signale generell nur Integer-Werte transportieren, ja, sie transportieren sogar nur die Werte 0 und 1.
Der Wert 1 steht dabei für den Zustand "wahr" des Logic-Signals, der Wert 0 für den Zustand "falsch". Die Bedeutung von "wahr" und "falsch" müssen vom Anwender festgelegt werden. Es könnte zum Beispiel ein Logic-Signal geben, das Auskunft darüber gibt, ob ein bestimmtes Gate geöffnet ist:

Im obigen Fall prüft das Macro *Gate2L* das ankommende Gate-Signal und erzeugt die Ausgabe "wahr" (1), wenn das Gate offen ist, und "falsch" (0), wenn es geschlossen ist.

Sie können dann die Logic-Signale verwenden, um "logische Operationen" auszuführen. Zum Beispiel könnten Sie einen Gate-Prozessor bauen, der ein regelmäßiges "getaktetes Gate" (Clocked Gate) über ein MIDI-Gate anwendet:



Die Module *Gate2L, AND* und *L2Gate* sind Logic-Module, die Sie im Menü *Standard Macro > Logic* finden. Das Modul *Gate LFO* ist ein Macro, das wir für diesen Prozessor gebaut haben. Es erzeugt ein Gate-Signal, das sich in einem regelmäßigen Intervall öffnet und schließt.
Das Eingangs-Gate und der Ausgang des LFOs sind mit den Konverter-Modulen *Gate2L* verbunden, die das Gate-Signal in Logic-Signale konvertieren, indem sie offenen Gates den Wahrheitswert "wahr" und geschlossenen Gates den Wahrheitswert "falsch" zuweisen. Das Modul *AND* gibt nur dann ein Signal mit dem Wert "wahr" aus, wenn sich beide Gates zur selben Zeit im offenen Zustand befinden. Mit anderen Worten ist der Ausgangs-Wert des Moduls *AND* dann – und nur dann – "wahr", wenn der Anwender eine Taste drückt und zur selben Zeit der LFO ein offenes Gate ausgibt. Das bedeutet, dass, solange der Anwender eine Taste drückt, alternierend die Werte "wahr" und "falsch" am Ausgang des *AND*-Moduls anliegen, wobei die LFO-Rate die Geschwindigkeit der Wechsel zwischen den beiden Werten bestimmt. Die Ausgabe des Moduls AND wird dann zurück in das Gate-Signal konvertiert; die Amplitude des Gate-Signals wird am Gate-Eingang gewonnen, sodass der Gate-Pegel unverändert beibehalten wird.
Hier sehen Sie die Struktur unseres Macros *Gate LFO*:



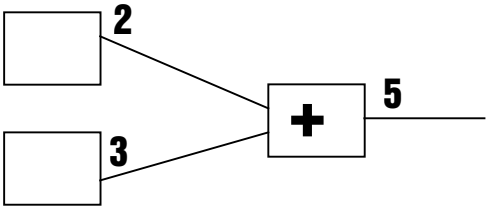
Der Eingang "F" definiert die Rate der Gate-Wiederholungen, der Eingang "W" die Dauer des Zustands "Gate offen" (bei 0 sind sie 50 % der Gate-Periode geöffnet, bei –1 sind sie 0 % der der Periode offen und bei 1 sind sie 100 % offen). Der Eingang "Rst" startet den LFO auf ankommende Events hin neu (wodurch der LFO jedes Mal neu startet, wenn ein Gate-Event am Haupt-Gate-Eingang ankommt).
Das Modul, das an den Eingang "Rst" des Moduls *Rect LFO* angeschlossen ist, heißt *Value*; Sie finden es unter *Standard Macro > Event Processing*. Es stellt sicher, dass der LFO in der 0-Phase neu gestartet wird, indem es alle Werte ankommender Events durch den Wert des unteren Eingangs ersetzte, der 0 ist. Das Ausgangssignal des LFOs wird in ein Gate-Signal konvertiert, und zwar mit dem Modul *Ctl2Gate*, das Sie ebenfalls unter *Standard Macro >Event Processing* finden.

Wie Sie sich bestimmt erinnern, arbeiten LFOs nicht im Inneren von Event-Core-Cells; wenn Sie diese Struktur also ausprobieren willen, müssen Sie eine Audio-Core-Cell verwenden.

3. Grundlagen von REAKTOR Core: Das Core-Signal-Modell

3.1. Werte

Die meisten Ausgänge der REAKTOR-Core-Module erzeugen Werte (Values). Einen Wert "erzeugen" bedeutet, dass dem Ausgang zu jedem beliebigen Zeitpunkt ein Wert zugeordnet ist. Dieser Wert ist für die Module verfügbar, deren Eingänge mit diesem Ausgang verbunden sind.
Im folgenden Beispiel erhält ein Addierer-Modul (Adder) die Werte 2 und 3 von den anderen beiden Modulen, deren Ausgänge mit seinen Eingängen verbunden sind, und erzeugt am Ausgang den Wert 5.

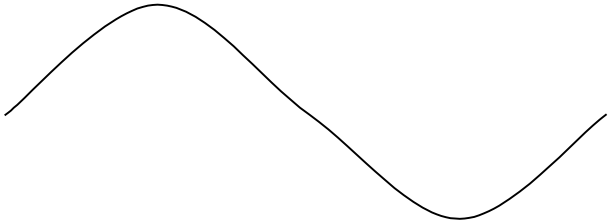


Wenn Sie eine Parallele zur Hardware-Welt ziehen wollen, können Sie sich die Werte als Signal-Pegel (Spannungen) vorstellen, besonders, wenn Sie mit relativ großformatigen Modulen wie Oszillatoren, Filtern, Hüllkurven und so weiter hantieren. Werte sind allerdings nicht darauf beschränkt, die Aufgaben von Steuer-Spannungen zu übernehmen – sie können bei der Implementierung beliebiger Verarbeitungs-Algorithmen eingesetzt werden, nicht nur für die Modellierung von Spannungen.

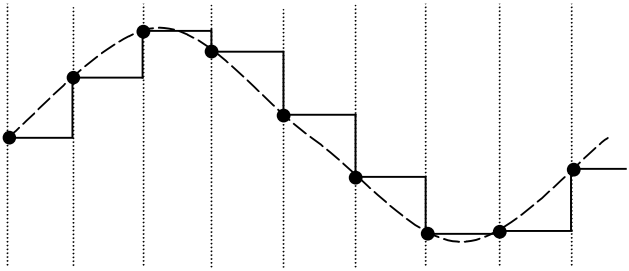
3.2. Events

Die Zeit ist in der digitalen Welt nicht fortlaufend, sondern diskret. Das wahrscheinlich bekannteste Beispiel dafür ist, dass eine digital gespeicherte Aufnahme nicht die vollständige Information eines Audio-Signals enthält, das sich über die Zeit verändert, sondern in regelmäßigen Abständen zu diskreten Zeitpunkten aus dem Signal entnommene "Proben" speichert. Die Anzahl dieser Zeitpunkte pro Sekunde trägt den berühmten Namen *Sampling-Rate*.

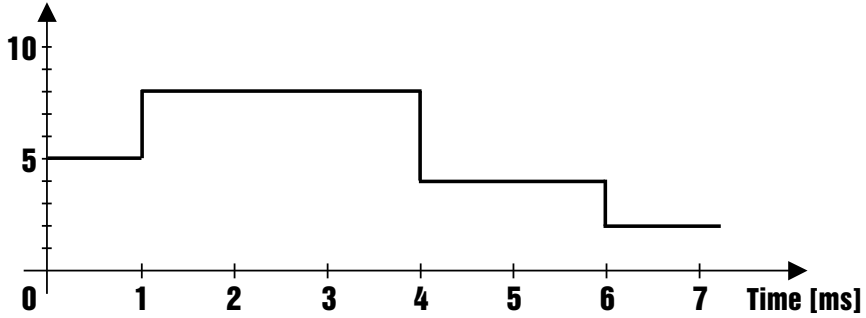
Hier sehen Sie eine Darstellung eines fortlaufenden Signals:



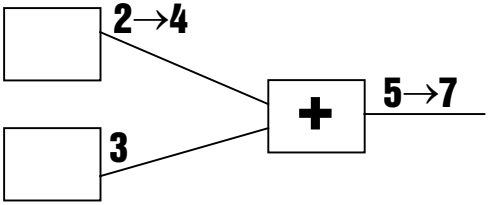
Und hier ist seine digitale Repräsentation:



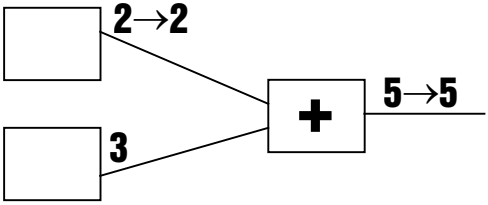
Das bedeutet für uns, dass die Ausgänge unserer Module in der digitalen Welt ihre Werte nicht fortlaufend verändern können. Auf der anderen Seite müssen wir uns auch nicht darauf beschränken, unseren Ausgängen Werteänderungen "in regelmäßigen Abständen zu diskreten Zeitpunkten" abzuringen, das heißt, wir müssen nicht in der gesamten Struktur mit derselben Sampling-Rate arbeiten. Weiterhin brauchen wir in manchen Bereichen unserer Strukturen überhaupt keine Sampling-Rate zu verwenden, was bedeutet, dass unsere Wertänderungen nicht "in regelmäßigen Abständen zu diskreten Zeitpunkten" stattfinden müssen.
Nehmen wir zum Beispiel an, dass der Ausgang unseres Addierers zum Zeitpunkt Null den Wert 5 hat. Die erste Änderung könnte nach 1 ms stattfinden, die zweite nach 4 ms und die dritte nach 6 ms:



Auf dem obigen Bild können Sie die Veränderungen des Ausgangssignals unseres Addierers im Zeitverlauf zwischen 0 und 7 ms sehen. In dem Moment, in dem der Ausgang seinen Wert ändert, erzeugt er einen Event. *Event* bedeutet, dass der Ausgang von einer Veränderung seines Zustands "berichtet", namentlich darüber, dass er einen neuen Werts angenommen hat.
Im folgenden Beispiel hat das obere linke Modul seinen Ausgangs-Wert von 2 auf 4 verändert, was ein Event erzeugt. Als Reaktion darauf wird auch das Addierer-Modul seinen Ausgangs-Wert verändern und an seinem Ausgang ebenfalls ein Event erzeugen.



Alternativ dazu hätte das obere linke Modul auch ein neues Event mit demselben Wert, den das alte Event hatte, erzeugen können. Der Addierer hätte trotzdem als Reaktion darauf ein neues Event erzeugt, anscheinend ohne seinen Ausgangs-Wert zu verändern:



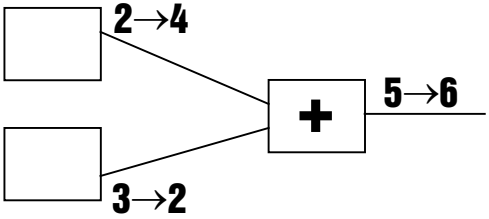
Der neue Wert, der am Ausgang anliegt, muss sich nicht zwangsläufig von dem alten Wert unterscheiden. In jedem Fall ist aber Erzeugung eines Events die einzige Möglichkeit für einen Ausgang, seinen Wert zu ändern.

Wie Sie an den bisherigen Beispielen gesehen haben, wird ein Event, der an einem Ausgang eines Moduls ankommt, von den "stromabwärts" angeschlossenen Modulen wahrgenommen, die dann als Reaktion darauf weitere Events erzeugen (erinnern Sie sich daran, dass der Addierer als Reaktion auf ein ankommendes Event ein Event an seinem Ausgang erzeugt). Diese neuen Events werden dann ihrerseits wieder von den mit den Ausgängen verbundenen Modulen bemerkt und pflanzen sich immer weiter fort, bis die Vermehrung aus einem der Gründe, die wir später noch behandeln werden, zum Stillstand kommt.

Die Events in REAKTOR Core sind nicht identisch mit den Events auf REAKTORs Primary Level – ihr Verhalten folgt anderen Regeln, die wir im Folgenden erklären.

3.3. Gleichzeitige Events

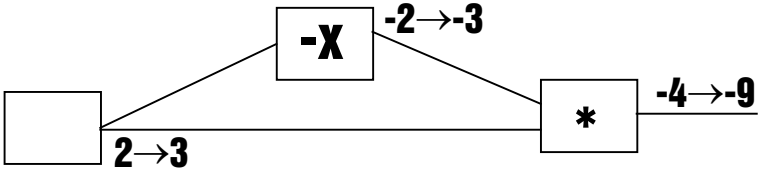
Stellen Sie sich folgende Situation vor: Beide Module auf der linken Seite des vorigen Beispiels erzeugen *gleichzeitig* einen Event.



Dies ist eine der Schlüsselfunktionen des Event-Modells in REAKTOR Core – die Events können gleichzeitig an unterschiedlichen Stellen auftreten. In dieser Situation kommen die den beiden links angeordneten Modulen entstammenden Signale auch gleichzeitig an den Eingängen des Addierers an. Das führt dazu, dass der Addierer als Reaktion darauf genau *ein* Ausgangs-Event erzeugt.

Das ist nicht dasselbe wie auf REAKTORs Primary Level, wo Events nicht gleichzeitig stattfinden können und das Addierer-Modul *Add* (im Event-Modus) in einer solchen Situation zwei Ausgangs-Events erzeugt.

Natürlich werden die Events in Wirklichkeit nicht vom oberen und unteren Modul gleichzeitig erzeugt, weil beide Module von derselben CPU berechnet werden und die CPU zu einer beliebigen Zeit immer nur ein Modul verarbeiten kann. Für uns wichtig ist aber die Tatsache, dass diese Events logisch gleichzeitig stattfinden; das heißt, dass die Modul, die sie empfangen, sie als gleichzeitig behandeln. Jetzt werden wir uns ein anderes Beispiel für gleichzeitige Event-Fortpflanzung ansehen.



In dem obigen Beispiel sendet das Modul ganz links ein Event, das seinen Ausgangs-Wert von 2 auf 3 erhöht. Das Event wird gleichzeitig sowohl an den Inverter (-x) als auch an den Multiplizierer (*) gesendet. Als Reaktion auf das ankommende Event wird der Inverter den neuen Ausgangs-Wert –3 erzeugen. Beachten Sie hier, dass *beide Events logisch gleichzeitig* sind, obwohl das Ausgangs-Event des Inverters ja als Reaktion auf das vom Modul ganz links gesendete Event und daher später als das ankommende Event erzeugt wurde. Dennoch kommen diese Events gleichzeitig im Multiplizierer an, und dieser erzeugt nur einen einzigen Ausgangs-Wert von –9.

Auf dem Primary Level hätten Sie es hier wieder mit zwei Events am Ausgang des Multiplizierer-Moduls *Event Mult* zu tun. Sie könnten außerdem nicht definieren, ob das Event vom Ausgang des Moduls ganz links zuerst an den Inverter oder an den Multiplizierer geschickt würde (was für unsere Beispiel-Struktur allerdings nicht wichtig ist).

Im Allgemeinen können Sie die folgende Regel verwenden, um zu bestimmen, ob zwei bestimmte Events gleichzeitig sind oder nicht:

Alle Events, die ihren Ursprung im selben Event haben (also als Reaktion auf denselben Event gesendet wurden), sind gleichzeitig. Alle als Reaktion auf eine beliebige Anzahl von an unterschiedlichen Ausgängen, aber zur selben Zeit gesendeten (und somit gleichzeitigen) Events gesendeten Folge-Events sind ebenfalls gleichzeitig.

Das letzte Beispiel zeigt den Sinn der Gleichzeitigkeit von Events. In diesem Fall eliminieren wir die überflüssige Verarbeitung des zweiten Events durch den Multiplizierer, die zusätzlich die CPU belastet hätte. Ohne solche Maßnahmen würde die Anzahl der Events in größeren Strukturen unkontrolliert anwachsen, es sei denn, der Entwickler der Struktur hätte besonders darauf geachtet, die Anzahl doppelter Events niedrig zu halten.
Neben der Einsparung von Rechenzeit führt dieses Konzept zu einem anderen Ansatz beim Entwurf von Strukturen und besonders bei der Entwicklung von Low-Level-DSP-Algorithmen. Sie werden diese Unterschiede besser verstehen und nachvollziehen können, sobald Sie mit der Konstruktion eigener Strukturen angefangen haben.

3.4. Verarbeitungsreihenfolge

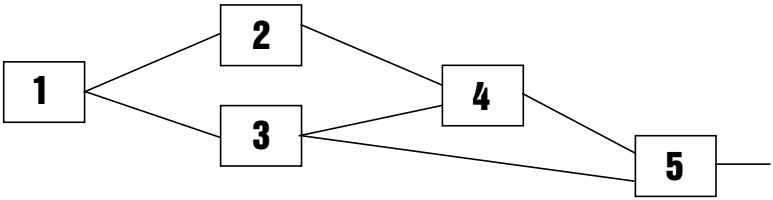
Wie Sie in den vorigen Beispielen schon gesehen haben, reagieren die "stromabwärts" im Signalfluss angeordneten Module, sobald ein Modul ein Event sendet. Daraus könnte man schließen, dass die Module definitiv nicht gleichzeitig verarbeitet werden, obwohl sie logisch gleichzeitige Events erzeugen. Man könnte weiter annehmen, dass es für jede mögliche Verbindung sinnvoll wäre, das weiter "stromaufwärts" gelegene Modul vor dem "stromabwärts" gelegenen zu verarbeiten. Wenn Sie nun diese Vermutungen angestellt haben – nun, dann haben Sie Recht.
Die allgemeine Regel für die Verarbeitungsreihenfolge von Modulen lautet:

> Wenn zwei verbundene Module "gleichzeitige" Events verarbeiten, wird das im Signalfluss weiter "stromaufwärts" gelegene Modul zuerst verarbeitet. Wenn die Events nicht gleichzeitig sind, entspricht die Verarbeitungsreihenfolge der Module natürlich der Reihenfolge, in der die Events verarbeitet werden.

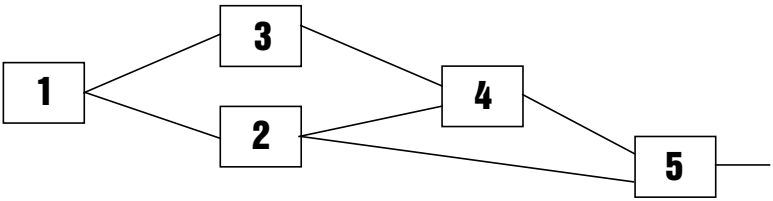
Aus der obigen Regel folgt, dass bei "Einbahnstraßen"-Verbindungen (immer aufwärts oder immer abwärts) von einem Modul zum anderen eine definierte Verarbeitungsreihenfolge existiert , nach der das aufwärts sendende Modul zuerst verarbeitet wird.

> Wenn zwischen den Modulen kein Einbahnstraßen-Pfad besteht, ist die relative Verarbeitungsreihenfolge der Module untereinander undefiniert (für gleichzeitige Events). Das bedeutet, dass diese Reihenfolge beliebig ist und sich jederzeit ändern kann. Beim Entwerfen von Strukturen sollten Sie also darauf achten, dass solche Situationen nur für Module eintreten, deren Verarbeitungsreihenfolge unwichtig ist. Das gilt normalerweise automatisch, solange keine OBC-Verbindungen (siehe unten) beteiligt sind.

Hier sehen Sie ein Beispiel; die Ziffern zeigen die Verarbeitungsreihenfolge der Module an:

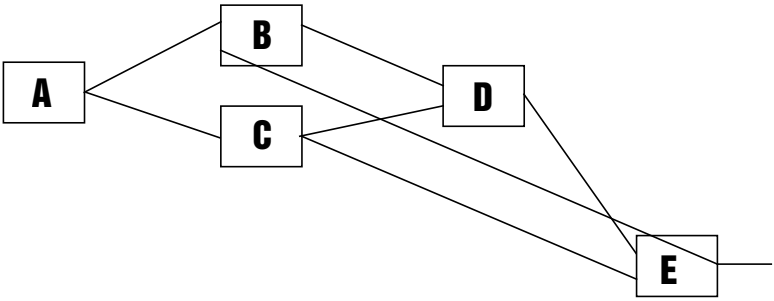


Eine alternative, ebenso gültige Verarbeitungsreihenfolge sehen Sie hier:



Es gibt keine Möglichkeit, vorherzusehen, welche dieser beiden Varianten die Software wählen wird. Glücklicherweise ist die relative Reihenfolge der Module wirklich unwichtig, solange Sie keine der OBC-Verbindungen verwenden, von denen unten die Rede ist.

Die oben genannten Regeln für die Verarbeitungsreihenfolge gelten nicht, wenn in Strukturen Feedback vorkommt, weil sich in diesem Fall für kein Paar von Modulen in der Feedback-Schleife bestimmen lässt, welches der beiden Module "stromaufwärts" des anderen liegt. Auf die Probleme bei der Behandlung solcher Feedback-Situationen auf die zugehörige Verarbeitungsreihenfolge gehen wir später noch genauer ein.



Für die oben abgebildete Struktur ist es nicht möglich, zu bestimmen, ob beispielsweise das Modul B stromaufwärts des Modul liegt oder umgekehrt. Anscheinend existiert eine aufwärts gerichtete Verbindung von D nach B, aber es besteht ebenso eine Aufwärts-Verbindung von B nach D (über E).

3.5. Event-Core-Cells im Rückblick

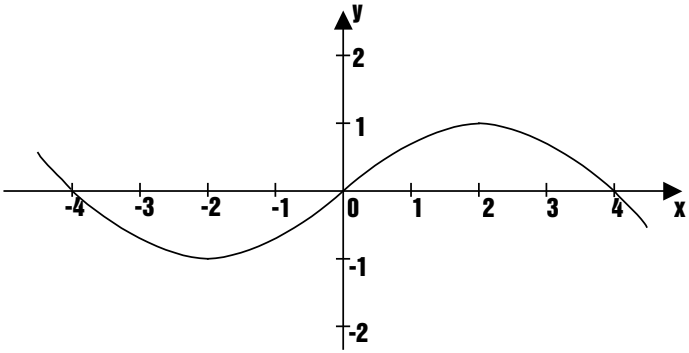
Lassen Sie uns unter Berücksichtigung des eben beschriebenen Event-Konzepts von REAKTOR Core einen Blick auf die Event-Core-Cells werfen.

Wie Sie wissen, besitzen Event-Core-Cells Event-Eingänge und Event-Ausgänge. Diese Eingänge und Ausgänge sind die Schnittstellen zwischen dem Primary Level und der REAKTOR-Core-Ebene. Sie erfüllen diesen Zweck, indem sie Konvertierungen zwischen den Primary-Level-Events und den REAKTOR-Core-Events und umgekehrt durchführen. Die Regeln dieser Konvertierungen lauten wie folgt:

Event Inputs senden als Reaktion auf von außen kommende Primary-Level-Events REAKTOR-Core-Events an das Innere der Struktur Weil die Primary-Level-Events von außen nicht gleichzeitig an den Eingängen ankommen können, finden die intern erzeugten Events ebenfalls nicht gleichzeitig statt.

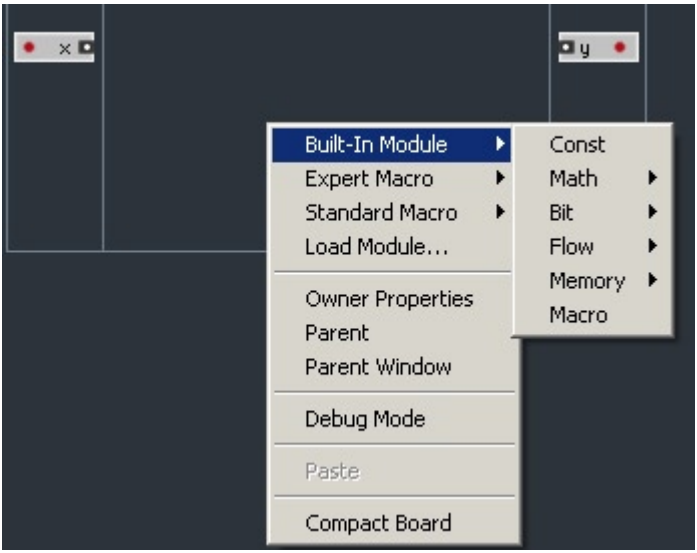
Event Outputs senden als Reaktion auf interne REAKTOR-Core-Events Primary-Level-Events aus der Struktur heraus. Obwohl REAKTOR-Core-Events gleichzeitig an mehreren Ausgängen stattfinden können, können die korrespondierenden Primary-Level-Events nicht gleichzeitig gesendet werden. Deshalb werden im Fall von gleichzeitigen REAKTOR-Core-Events die zugehörigen Primary-Level-Events nacheinander gesendet, wobei *die oberen Ausgänge immer vor den darunter gelegenen Ausgängen senden..*

Als nächstes werden wir das in der Praxis nachvollziehen.
Lassen Sie uns mal versuchen, ein Event-Verarbeitungs-Modul zu bauen, das eine Signal-Formung gemäß der Formel y = 0.25*x*(4-|x|) durchführt.
Der Graph dieser Funktion sieht folgendermaßen aus:

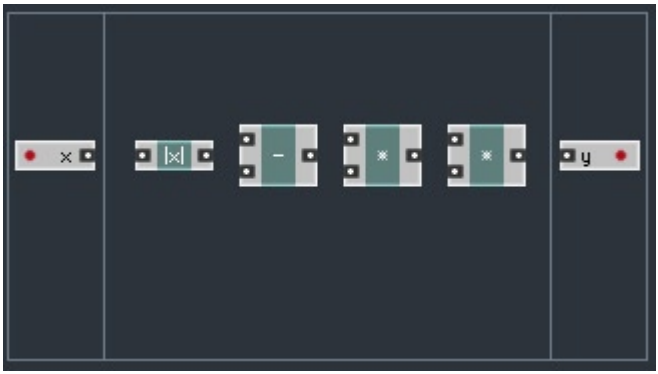


Wir beginnen mit dem Erzeugen einer neuen Event-Core-Cell mit einem Eingang und einem Ausgang, die wir gemäß unserer Formel “x” und “y” nennen:

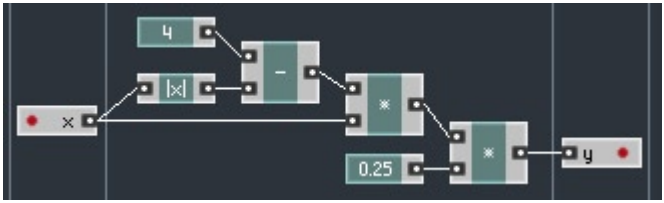
Lassen Sie uns nun die Struktur entwerfen, die diese Formel berechnet. Wir müssen zunächst Module für “|x|” (absoluter Wert) und “-” (Subtraktion) sowie zweimal “*” (Multiplikation) im normalen Bereich anlegen. Hierbei handelt es sich nicht um REAKTOR-Core-Macros, sondern um echte, eingebaute Module von REAKTOR Core. Um eingebaute Module in REAKTOR-Core-Strukturen einzusetzen, führen Sie einen Rechts-Klick in den Hintergrund des normalen Bereichs aus und wählen aus dem Menü das Untermenü *Built In Module*:



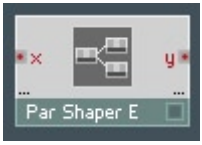
Sie finden alle oben genannten Module im Untermenü *Built In Module > Math*:



Wir brauchen zwei konstante Werte: 0.25 und 4. Wir könnten natürlich genau wie früher QuickConsts verwenden, aber es gibt auch die Möglichkeit, über Built In Module > Const ein echtes Konstanten-Modul einzusetzen (dessen Wert Sie wie bei QuickConst im Properties-Fenster festlegen können):

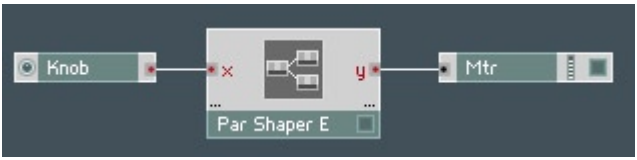


Natürlich bringt der Einsatz von Const-Modulen in unserem besonderen Fall keinen Vorteil gegenüber der Verwendung von QuickConsts, aber manchmal wollen Sie es vielleicht doch nutzen. Wenn zum Beispiel dieselbe Konstante an mehrere Eingänge übergeben werden soll, kann es vorteilhaft sein, das Modul Const zu verwenden, weil Sie dann nur eins davon brauchen und den Wert an einer zentralen Stelle kontrollieren können.
Wie auch immer, die oben abgebildete Struktur erledigt den Job, das Signal in der beschriebenen Weise zu formen, ziemlich gut. Wir können also unserem Modul einen Namen geben und auf das Primary Level zurückkehren:

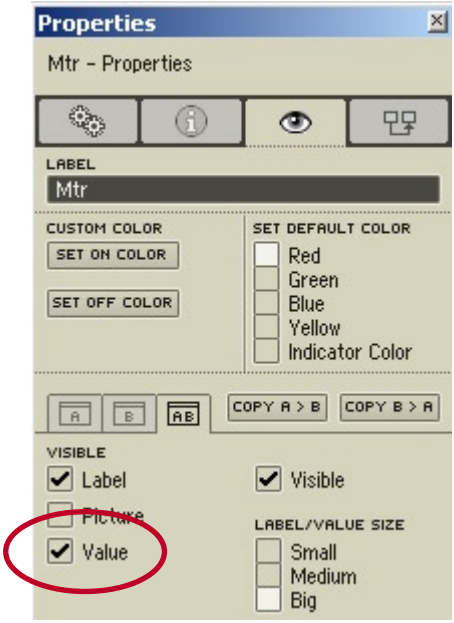


Jetzt lassen Sie 's uns ausprobieren. Setzen Sie den Wert für die Stimmenzahl des REAKTOR-Instruments auf 1, sodass wir leichter eine Werte-Anzeige (Meter-Modul) verwenden können.

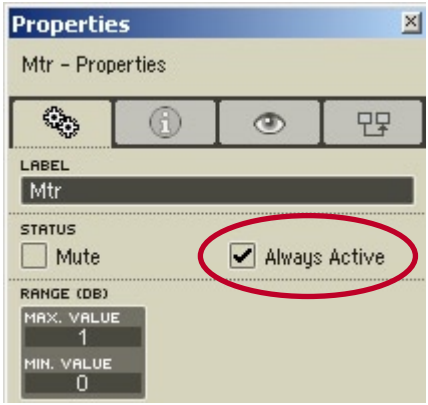
Erzeugen Sie einen Drehregler und eine Werte-Anzeige und verbinden Sie diese mit dem Eingang und dem Ausgang Ihres Moduls:



Legen Sie die Eigenschaften für den Drehregler und die Werte-Anzeige fest. Vergessen Sie dabei nicht, die Werte-Anzeige so einzustellen, dass sie ihren Wert anzeigt,…



...und das Kästchen "Always Active" anzukreuzen:



Bewegen Sie nun den Drehregler und beobachten Sie, wie sich die Ausgangs-Werte verändern.

Shaper-Struktur, die wir gebaut haben, sollte sich perfekt für das Formen von Kontroll-Signalen eignen, aber sie hat noch einen kleinen Makel, der die Event-Verarbeitung betrifft. Wir werden später auf dieses Problem zurückkommen und es dann auch gleich beheben.

4. Strukturen mit internem Zustand

4.1. Clock-Signale

Die Art, wie eine Core Cell ein ankommendes Event behandelt, bleibt vollkommen dem Modul überlassen. Normalerweise verarbeitet ein Modul den ankommenden Wert auf irgendeine Weise, aber es könnte ihn auch einfach ignorieren. Der typische Fall einer solchen Behandlung sind *Clock Inputs* (Takt-Eingänge).
Ein Beispiel für ein Modul mit einem Clock-Eingang ist das Modul *Latch*. Dies ist kein eingebautes Modul, sondern ein Macro, aber es eignet sich trotzdem perfekt, um daran das Clock-Prinzip zu erklären.
Latch hat zwei Eingänge: einen für den Wert und einen für die Clock.

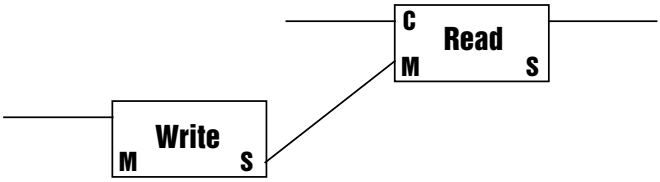
Der Werte-Eingang (oben) schreibt als Reaktion auf einen ankommenden Event den anliegenden Wert in den internen Speicher des Latch-Moduls; es wird nichts an den Ausgang gesendet. Der Clock-Eingang schickt als Reaktion auf ein ankommendes Event den letzten gespeicherten Wert an den Ausgang.

> Der Clock-Eingang ignoriert normalerweise (außer, wenn es ausdrücklich anders festgelegt ist) den Wert des ankommenden Events und reagiert nur auf die Tatsache, dass ein Event ankommt.

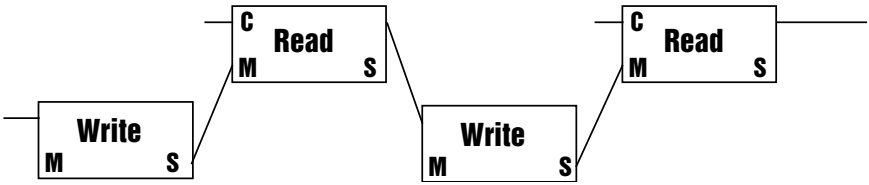
Weil wir uns hier gerade mit Clock-Signalen befassen, nicht mit Latches, folgen die Beispiele zur Verwendung des Moduls *Latch* später an anderer Stelle.
Weil es Module mit Clock-Eingängen gibt, sollte klar sein, dass einige Signale in der Struktur keine Nutz-Werte (oder in diesem Fall "nützlichen" Werte) transportieren. Einige Signale werden sogar eigens für den Einsatz als Clock-Quelle erzeugt. Wir können diese Signale *Clock-Signale* nennen.
Ein Beispiel für ein Clock-Signal ist die Sampling-Rate-Clock. Diese produziert für jedes neue Audio-Sample, das erzeugt werden soll, ein Event, sodass die "Uhr" bei einer Sampling-Rate von 44,1 kHz eben 44100 Mal pro Sekunde "ticken" würde. Der Wert dieses Signals hat keine Bedeutung, ist nicht für eine Auswertung gedacht und (in der derzeitigen Implementation) immer Null.

4.2. Object Bus Connections

Die bereits oben erwähnten *Object Bus Connections* (OBC, Objekt-Bus-Verbindung) sind eine spezielle Art von Verbindungen zwischen den Modulen. Eine OBC zwischen zwei Modulen besagt, dass sich diese beiden Module irgendein internes Objekt teilen. Der häufigste Fall von Modulen, die eine OBC nutzen, sind die Speicher-Module *Read* und *Write,* die sich gemeinsamen Speicher teilen, wenn sie via OBC verbunden sind.
Die Funktion des Moduls *Write* besteht darin, einen an seinem Eingang ankommenden Wert in den über OBC gemeinsam genutzten Speicher zu schreiben. Die Aufgaben des Moduls *Read* ist es, auf ein am Eingang "C" eingehendes Clock-Signal hin den OBC-Speicher auszulesen; den ausgelesenen Wert stellt das Modul dann an seinem Ausgang bereit.



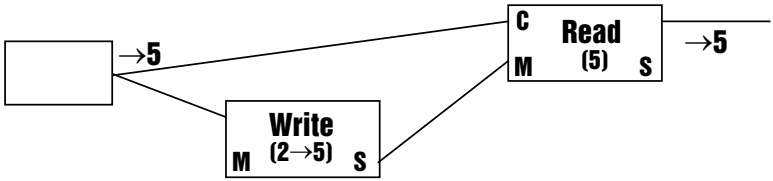
Die obige Struktur stellt die Funktionalität des Macros Latch dar (und es handelt sich in der Tat um die interne Struktur des Latch-Macros). Die Anschlüsse *M* und *S* der Module *Read und Write* sind vom Typ *Latch OBC*. Der Anschluss M ist der Master-Connection-Eingang, der Anschluss S der Slave-Connection-Eingang. Der Master-Eingang des Moduls Read ist mit dem Slave-Ausgang des Moduls *Write* verbunden (das andere Master-Slave-Paar ist nicht belegt) Deshalb teilen sich in dieser Struktur die Module *Write* und *Read* den gemeinsamen Speicher.
In der nächsten Struktur sehen Sie zwei Paare von Modulen der Typen *Write* und *Read*. Jedes Paar besitzt seinen eigenen Speicher. Beachten Sie, dass die Verbindung in der Mitte (vom Ausgang von *Read* zum Eingang von *Write*) nicht vom Typ *OBC* ist.



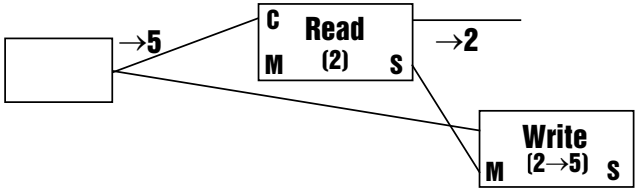
Sie könnten sich nun fragen, worin der Unterschied zwischen Master- und Slave-Betrieb besteht. Nun, hinsichtlich der Besitzanteile am gemeinsamen Objekt (in diesem Fall Speicher) gibt es keine Unterschiede. Allerdings besagt eine Regel, die Sie in einem früheren Kapitel gelernt haben, dass bei der Verarbeitung "gleichzeitiger Events" "stromaufwärts" gelegene Module vor den weiter "stromabwärts" im Signalfluss gelegenen Modulen verarbeitet werden. Deshalb werden in den beiden letztgenannten Beispielen die Write-Module vor den als Slave angebundenen Read-Modulen verarbeitet, was ganz offensichtlich nicht dasselbe ist wie die umgekehrte Reihenfolge.

Die relative Reihenfolge der Verarbeitung von via OBC verbundenen Modulen folgt denselben Regeln wie die Verarbeitung anderer Module: Die weiter "stromaufwärts" gelegenen Module werden zuerst verarbeitet.

Lassen Sie uns nun zwei verschiedene Fälle betrachten. In beiden Fällen ist der Ausgangszustand des Speichers der Wert 2, und dasselbe Event von der Wertigkeit 5 wird sowohl an das *Write*- als auch an das *Read*-Modul geschickt. In dem einen Fall ist das Modul *Write* der Master, im anderen Fall ist *Read* der Master.



Oben sehen Sie die Struktur für den ersten Fall. Das Modul auf der linken Seite schickt einen Event mit dem Wert 5, der zuerst im *Write*-Modul eintrifft und dieses veranlasst, den neuen Wert 5 in den gemeinsamen Speicher des Modul-Paars *Write* und *Read* zu schreiben. Nun kommt der Event im *Read*-Modul an, wird als Clock-Event ausgewertet und triggert den Lese-Vorgang, im Verlauf dessen der gerade gespeicherte Wert 5 ausgelesen und zum Ausgang geschickt wird. Dies ist die Funktion, die das Macro *Latch* aus der Macro-Library von REAKTOR Core zur Verfügung stellt.
Nun betrachten Sie die zweite Struktur:



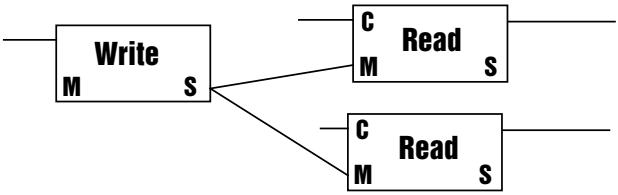
Hier haben wir die entgegengesetzte Situation. Das Clock-Event trifft zuerst im *Read*-Modul ein und sendet den Wert 2 an den Ausgang. Erst nach dem Lesen kommt das Event am Eingang des Moduls *Write* an und setzt den gespeicherten Wert auf 5. Diese Struktur implementiert die Funktion eines Z^{-1}-Blocks (ein Sample Verzögerung), die in der DSP-Theorie viel verwendet wird. In der Tat liegt in diesem Fall der Ausgangs-Wert immer einen Schritt hinter dem Eingangs-Wert zurück

---

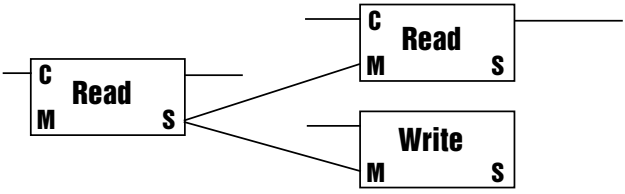
Wie erwähnt, implementiert die obige Struktur die Z^{-1}-Funktionalität. Bevor Sie aber selbst solche Strukturen aufbauen können, müssen Sie einige andere wichtige Dinge wissen – also lesen Sie weiter.

Wenn Sie mehr als zwei Module via OBC verbunden haben, bedeutet dies, dass alle diese Module dasselbe Objekt teilen. Hier ist es sehr wichtig zu wissen, ob eine bestimmte Reihenfolge von Schreib- und Lese-Vorgängen von Bedeutung ist, und wie diese Reihenfolge gegebenenfalls aussehen sollte.

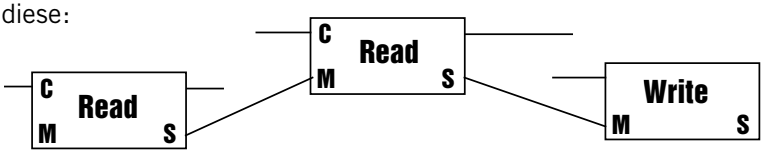
In der folgenden Struktur beispielsweise ist die relative Reihenfolge der beiden Lese-Vorgänge nicht definiert, aber sie erfolgen beide nach dem Schreib-Vorgang, sodass alles in Ordnung sein sollte:



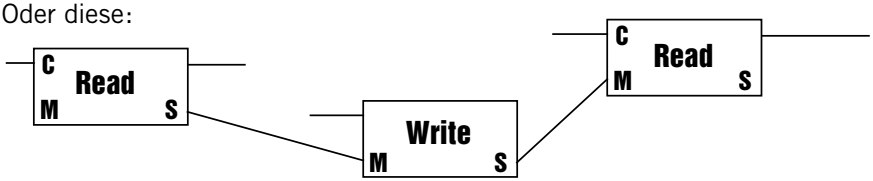
In der nächsten Struktur ist die relative Reihenfolge des Schreib-Vorgangs und des zweiten Lese-Vorgangs undefiniert, sodass dies eine möglicherweise gefährliche Struktur ist, die Sie auf jeden Fall vermeiden sollten:



Eine bessere Möglichkeit, die obige Struktur umzusetzen, wäre wahrscheinlich diese:



Oder diese:



Auch wenn Sie annehmen, dass an manchen Stellen die relative Reihenfolge der Schreib- und Lese-Vorgänge irrelevant ist, schadet es nicht, eine bestimmte Reihenfolge festzulegen – im Gegenteil, es macht die Sache sicherer.

Die relative Reihenfolge der Schreib-Vorgänge ist wichtig. Die relative Reihenfolge der Lese-Vorgänge ist nicht von Bedeutung, solange ihre Anordnung im Verhältnis zu den Schreib-Vorgängen definiert bleibt.

Verbindungen des Typs OBC sind nicht mit normalen Signal-Verbindungen kompatibel. Weiterhin sind Verbindungen des Typs OBC, die mit unterschiedlichen Typen von Objekten korrespondieren (z. B. verschiedene Fließkomma-Genauigkeiten bei der Speicherverwaltung) nicht miteinander kompatibel. Anschlüsse inkompatiblen Typs lassen sich nicht miteinander verbinden, d. h., Sie können einen normalen Signal-Ausgang nicht mit einem OBC-Eingang verbinden.

4.3. Initialisierung

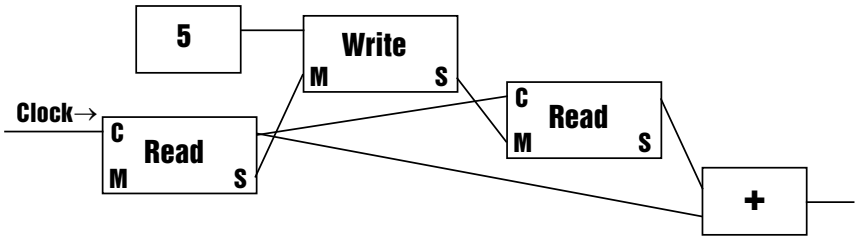
Wenn wir anfangen, mit Objekten zu arbeiten, die einen internen Zustand annehmen (im Fall von *Read* und *Write* ist der gemeinsame Speicher dieser Objekte ihr interner Zustand), ist es wichtig, den *Anfangs-Zustand* (Initial State) einer gegebenen Struktur zu kennen. Wenn wir zum Beispiel einen Wert aus dem Speicher lesen wollen (wozu wir das *Read*-Modul verwenden), bevor etwas in den Speicher geschrieben wurde, müssen wir uns fragen, welcher Wert wohl in diesem Fall übergeben würde. Und wenn uns dieser Default-Wert nicht passt, wie können wir ihn ändern?

Diesen Fragen widmet sich der Initialisierungs-Mechanismus von REAKTOR Core. Die Initialisierung von REAKTOR-Core-Strukturen wird wie folg durchgeführt:

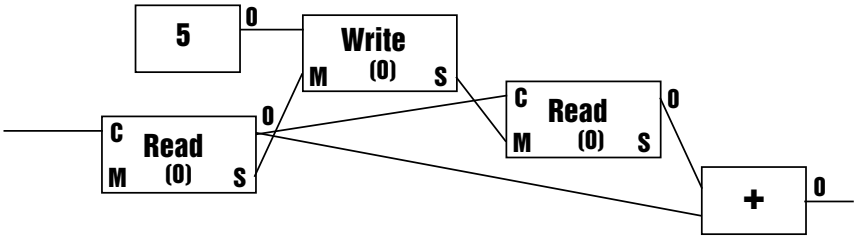
- Zuerst werden alle Zustands-Elemente auf Default-Werte initialisiert, normalerweise auf Nullen. Besonders alle gemeinsam genutzten Speicher und alle Ausgangs-Werte der Module werden auf Null gesetzt, außer in den im Handbuch beschriebenen Ausnahmefällen.
- Anschließend wird ein *Initialisierungs-Event* gesendet, und zwar gleichzeitig von allen *Initialisierungs-Quellen*. Die Initialisierungs-Quellen umfassen die meisten Module, die keinen Eingang besitzen: Const-Module (einschließlich *QuickConsts*), Core-Cell-Eingänge (typischerweise) und einige andere. Die Quellen senden ihre Anfangs-Werte während des Initialisierungs-Events, z. B. würden Konstanten-Module ihre Werte schicken und Core-Cell-Eingänge würden die Anfangswerte senden, die sie von der umgebenden Primary-Level-Struktur empfangen haben.

Wenn ein Modul eine Initialisierungs-Event-Quelle ist, finden Sie Informationen darüber im Modul-Referenz-Abschnitt zu diesem Modul. Wenn das Modul keine Initialisierungs-Event-Quelle ist, behandelt es das Initialisierungs-Event genau wie jedes andere ankommende Event. Die *allermeisten* Initialisierungs-Quellen sind Module, die keine Eingänge haben.

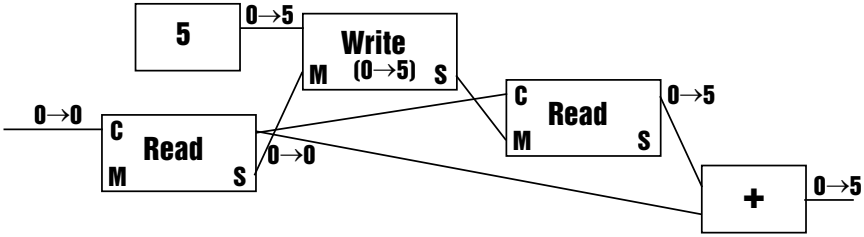
Lassen Sie uns am folgenden Beispiel betrachten, wie die Initialisierung vor sich geht:



Dies ist ein Teil der Struktur; das *Read*-Modul links ist mit einer Clock-Quelle verbunden, die auch ein Initialisierungs-Event sendet (wie es Clock-Quellen normalerweise tun sollten).
Anfänglich sind alle Signal-Ausgänge und der interne Zustand der Lesen-Schreiben-Lesen-Kette (Read-Write-Read) auf Null gesetzt

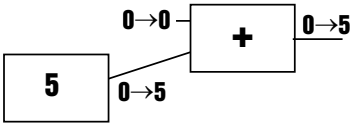


Dann senden die Clock-Quelle und das Konstanten-Modul 5 gleichzeitig ein Initialisierungs-Event.



Das Modul *Read* (links) wird vor dem Modul *Write* verarbeitet, sodass das Clock-Event dort eintrifft, bevor der neue Wert in den Speicher geschrieben wurde; also ist die Ausgabe dieses Moduls Null. Dann wird der Wert vom Modul *Write* in den Speicher geschrieben. Nun wird das zweite *Read*-Modul angetriggert, was einen Wert von 5 an seinem Ausgang erzeugt. Schließlich wird das Addierer-Modul verarbeitet und ermittelt aus den anliegenden Werten eine Summe von 5.

Wie Sie sich erinnern, werden nicht angeschlossene Eingänge in REAKTOR Core behandelt, als hätten sie den Wert Null (sofern dies nicht von einem bestimmten Modul anders festgelegt wird). Genauer gesagt werden unbeschaltete Eingänge wie Null-Konstanten behandelt. Das bedeutet, dass diese Eingänge genauso Initialisierungs-Events empfangen wie Eingänge, an denen ein echtes Konstanten-Modul mit dem Wert Null angeschlossen ist.

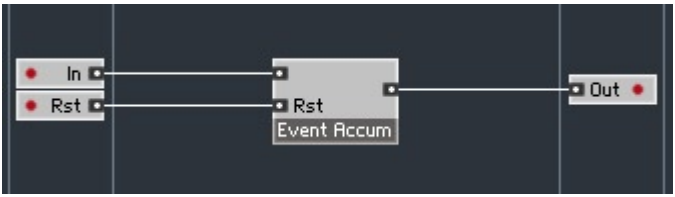


Oben sehen Sie einen Addierer mit einem nicht verbundenen Eingang und einem Eingang, der mit einem Konstanten-Modul verbunden ist. Dieser Addierer empfängt zwei gleichzeitige Initialisierungs-Events, und zwar eins von der an unbeschalteten Eingängen anliegenden Default-Konstante Null und eins über eine echte Verbindung zu einem Konstanten-Modul.

Nicht angeschlossene Eingänge, die keine Signal-Eingänge sind (und die man deshalb ohnehin nicht mit einer Null-Konstante verbinden kann), können eine besondere Bedeutung bekommen. Zum Beispiel kann ein nicht beschalteter Master-Eingang eines Write-Moduls bedeuten, dass die gemeinsam genutzte Speicher-Kette hier beginnt und sich in den Modulen, die an den Slave-Ausgang angeschlossen sind, fortsetzt.

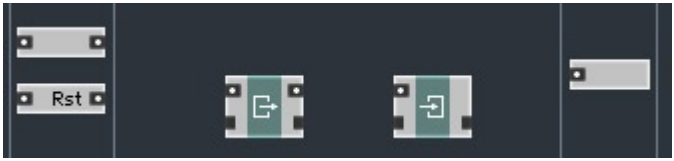
4.4. Wie Sie einen Event-Akkumulierer bauen

Ein Event-Akkumulierer, wie wir ihn nun bauen werden, braucht zwei Eingänge, und zwar einen für die Event-Werte, die akkumuliert werden sollen, und einen zum Rücksetzen des Akkumulierer-Moduls auf Null. Es gibt außerdem einen Ausgang, der die Gesamtsumme der akkumulierten Events ausgibt.
Wir werden dieses Modul in Form eines Core-Macros anlegen. Ein solches Macro wäre auch leicht innerhalb einer Event-Core-Cell zu verwenden:



Uns so sieht das Innere unseres Macros aus:

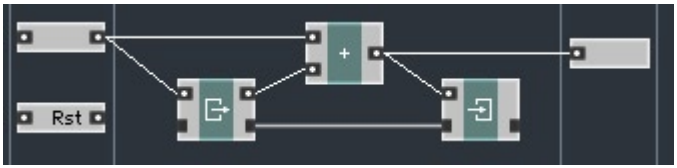
Offensichtlich braucht das Akkumulierer-Modul einen internen Zustand, in den es den aktuellen akkumulierten Wert schreiben kann. Wir werden also die Module *Read* und *Write* verwenden, um die Akkumulierer-Schleife zu bauen. Sie finden diese Module unter *Built In Module > Memory:*



Das Modul, das Sie links im Bild sehen (mit einem Pfeil in Auswärts-Richtung), ist das Modul *Read*, das Modul rechts (mit dem nach innen weisenden Pfeil) ist das Modul *Write*

.

Als Reaktion auf ein ankommendes Event soll die Akkumulierer-Schleife den alten Wert nehmen und den neuen Wert hinzufügen. Wir müssen also ein *Read*-Modul einsetzen, um den alten Zustand auszulesen, dem ermittelten Wert mit einem Addierer den neuen Wert hinzufügen und ein *Write*-Modul verwenden, um den Wert wieder zu speichern.



Beachten Sie, dass das Modul *Read* vom ankommenden Event getaktet wird und dass das via OBC angeschlossene *Write*-Modul natürlich stromabwärts angeordnet ist, weil wir ja erst schreiben wollen, nachdem wir gelesen haben.

Im Normalfall sollte die oben abgebildete Struktur sofort arbeiten, und zwar sollte sie die ankommenden Werte sammeln (akkumulieren) und die Gesamtsumme am Ausgang ausgeben. Was noch fehlt, sind eine Reset-Funktion zum Zurücksetzen und eine Schaltung, die den korrekten Anfangs-Zustand sicher stellt.

Lassen Sie uns zunächst die Reset-Schaltung bauen. Weil wir uns in der REAKTOR-Core-Welt bewegen, können die Eingänge “In” und “Rst” gleichzeitig Events senden, also müssen wir das im Allgemeinen (wenn wir ein allgemein verwendbares REAKTOR-Core-Macro bauen wollen) berücksichtigen.

Lassen Sie uns also annehmen, dass die Eingänge “In” und „Rst” gleichzeitig jeweils ein Event erzeugen. Was soll in diesem Fall geschehen? Ist es logisch betrachtet günstiger, den Reset durchzuführen, bevor das akkumulierte Event verarbeitet wird, oder danach? (Dieser Unterschied ist ganz ähnlich wie der Unterschied zwischen den Funktionen *Latch* und Z^{-1}, die sich nur in ihrer relativen Verarbeitungsreihenfolge von Signal-Eingang und Clock-Eingang unterscheiden.)

Wir schlagen die Latch-Variante vor, weil dieses Modul sehr viel in REAKTOR-Core-Strukturen, verwendet wird und deshalb ein derartiges Verhalten eher der Intuition entspricht. In einem Latch kommt das Clock-Signal logisch später an als das Werte-Signal. In unserem Fall sollte das Reset-Signal logisch nach dem akkumulierten Signal eintreffen (und Zustand und Ausgang auf Null setzen).
Dies bedeutet, dass wir irgendwie das Ausgangssignal des Akkumulierers mit einem Anfangs-Wert überschreiben müssen. Um das zu erreichen, brauchen wir ein neues Konzept, das wir nun diskutieren wollen.

4.5. Event Merging

Sie haben schon verschiedene Methoden gesehen, mit denen sich zwei verschiedene Signale in REAKTOR Core kombinieren lassen, darunter arithmetische Operationen und einige andere. Was uns aber bislang fehlt, ist eine Möglichkeit, Signale sortiert zu mischen.
Dieses im Englischen *Merging* genannte sortierende Mischen ist kein Addieren. Merging bedeutet, dass wir von allen ankommenden Signalen den jeweils letzten Wert nehmen, anstatt sie zu summieren. Um die Signale zu mischen, die Sie mischen müssen, verwenden Sie das Modul *Merge*.
Lassen Sie uns mal schauen, wie das funktioniert. Stellen Sie sich vor, wir haben ein *Merge*-Modul mit zwei Eingängen. Der anfängliche Ausgangs-Wert (vor dem Initialisierungs-Event) ist natürlich Null, wie für die meisten Module:

Nun kommt ein Event mit dem Wert 4 am zweiten Eingang des Moduls an:

Das Event passiert das Modul und erscheint am Ausgang. Nun hat der Ausgang des *Merge*-Moduls den Wert 4.
Dann kommt ein weiteres Event mit dem Wert 5 am ersten Eingang an:

Das Event passiert ebenfalls das Modul und erscheint am Ausgang, wodurch sich dessen Wert in 5 ändert.Nun kommen zwei Events mit den Werten 2 und 8 gleichzeitig an beiden Eingängen an.

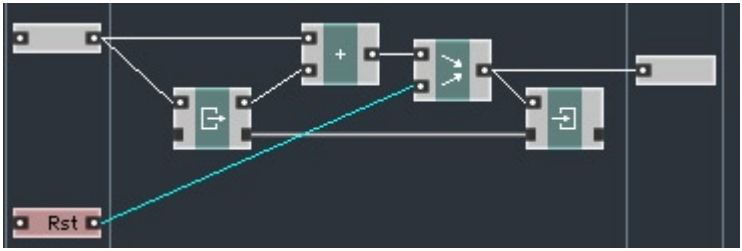
Und hier kommen wir zu einer besonderen Regel für unser Merge-Modul:

> Events, die gleichzeitig an den Eingängen eines Merge-Moduls ankommen, werden in der Reihenfolge der Eingangs-Nummerierung verarbeitet. Trotzdem wird nur ein Ausgangs-Event erzeugt, weil ein Ausgang in REAKTOR Core nicht mehrere Events gleichzeitig erzeugen kann.

Im oben beschriebenen Fall bedeutet dies, dass das Event am zweiten Eingang nach dem Event am ersten Eingang verarbeitet wird, wodurch der Wert 2 mit dem Wert 8 überschrieben wird. Dieser Wert 8 liegt dann am Ausgang an.

4.6. Event-Akkumulierer mit Reset und Initialisierung

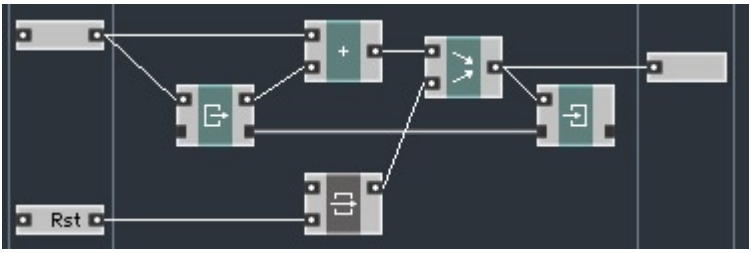
Um also die gewünschte Reset-Funktion zu erreichen, müssen wir die Ausgaben des Addierers mit einem Anfangs-Wert überschreiben. Dafür können wir ein *Merge*-Modul verwenden (das Sie im Untermenü *Built In Module > Flow* finden). Die einfachste Möglichkeit wäre, den zweiten Eingang des *Merge*-Moduls mit dem Eingangs-Modul "Rst" zu verbinden:



Jetzt wird der Reset-Event direkt an das *Merge*-Modul übergeben und überschreibt die Ausgabe des Addierers, falls das akkumulierte Event zur selben Zeit eintreffen sollte. Von da aus gelangt das Reset-Event an den Ausgang und in den internen Zustand des Akkumulierers.
In der oben abgebildeten Struktur wird der am Eingangs-Modul "Rst" auftretende Wert als neuer Wert des Akkumulierers verwendet. Vielleicht ist das keine schlechte Idee, aber dann ist es eigentlich keine "Reset"-Funktion, sondern eher eine "Set"-Funktion, wie sie im Standard-Event-Akkumulierer-Modul von REAKTOR verwirklicht wurde. Wenn wir eine echte Reset-Funktion entwickeln wollen, sollten wir ausschließlich Null-Werte in den Zustand schreiben, unabhängig davon, welcher Wert am Eingangs-Modul "Rst" anliegt. Wir müssen also immer dann einen Null-Wert an das Modul *Write* schicken, wenn am Eingang "Rst" ein Event auftritt.

Das Senden eines Events mit einem bestimmten Wert als Reaktion auf einen ankommenden Event ist eine ziemlich normale Sache in REAKTOR Core, und wir würden zu diesem Zweck das Macro *Latch* aus der REAKTOR-Core-Library verwenden. Sie finden es unter *Expert Macro > Memory > Latch:*

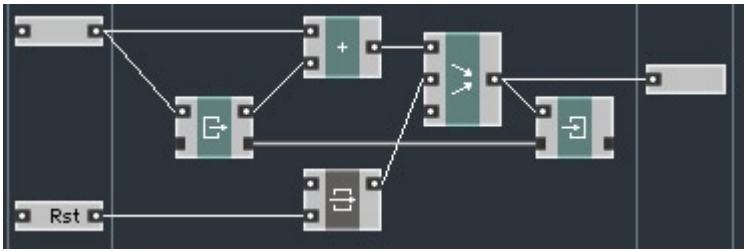
Wie bereits erwähnt, besitzt das Latch-Modul einen Werte-Eingang (oben) und einen Clock-Eingang (unten). Wir müssen das Eingangs-Modul "Rst" mit dem Clock-Eingang verbinden, um das Senden von Events an den Ausgang des Latch-Moduls zu triggern. Außerdem müssen wir ein Konstanten-Modul mit dem Wert Null an den Werte-Eingang des Latch-Moduls anschließen, weil wir die ganze Zeit Events mit dem Wert Null an diesen Eingang senden wollen. Oder wir erinnern uns daran, dass nicht angeschlossene Eingänge so behandelt werden, als wäre ihr Wert konstant Null (sofern nicht anders angegeben) und lassen den Werte-Eingang des Moduls Latch einfach offen:



Jetzt sollte die Reset-Funktion arbeiten wie gewünscht.

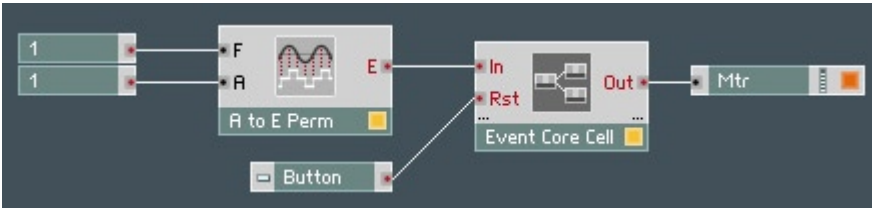
Schließlich müssen wir noch für eine korrekte Initialisierung sorgen. Es ist auch eine Frage, welches die korrekte Initialisierung ist. Lassen Sie uns deshalb einen Blick darauf werfen, wie die oben abgebildete Struktur initialisiert wird.
Nehmen wir an, dass das Initialisierungs-Event von den Eingängen "In" und "Rst" der obersten Ebene der Core-Cell-Struktur und von impliziten Null-Konstante am leeren Werte-Eingang des Latch-Moduls gleichzeitig gesendet wird. In diesem Fall wird das über den Eingang "Rst" angetriggerte Modul *Latch* den Wert Null an den zweiten Eingang des Moduls *Merge* übergeben und damit überschreiben, was auch immer am ersten Eingang von *Merge* ankommt. Somit wird der Wert Null in den internen Zustand geschrieben und an den Ausgang gesendet – perfekt!
Es gibt trotzdem noch ein kleines Problem. Es könnte nämlich sein, dass das Initialisierungs-Event nicht an einem oder beiden Ports ankommt. Ein Grund dafür könnte sein, dass das Initialisierungs-Event nicht einmal am korrespondierenden Eingang der Event-Core-Cell angekommen ist, oder dass dieses Macro Teil einer komplizierteren REAKTOR-Core-Struktur ist, die nicht auf allen Leitungen Initialisierungs-Events erhält (wir lernen später noch, was in solchen Fällen zu tun ist). Wir müssen also eine letzte und endgültige Modifikation vornehmen, um sie etwas vielseitiger aussehen zu lassen.

Setzen Sie dafür im Properties-Fenster des Moduls *Merge* die Anzahl der Eingänge auf 3.

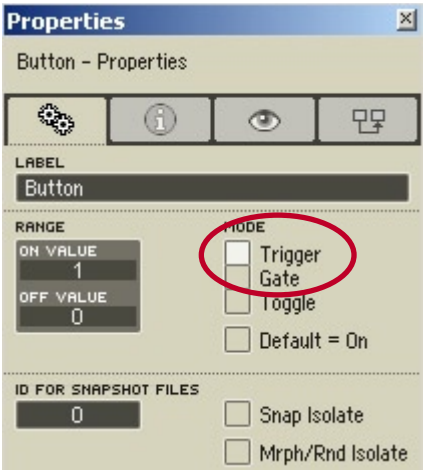


Jetzt sendet auch dann, wenn am "Rst"-Eingang kein Signal ankommt, die implizite Konstante am (nicht beschalteten) dritten Eingang des Moduls *Merge* ein Initialisierungs-Event , das für die korrekten Ausgabe- und Anfangs-Werte sorgt.

Lassen Sie uns mal schauen, wie das funktioniert. Wir schlagen vor, dass Sie die folgende Primary-Level-Struktur bauen und darin das erzeugte Modul *Event Accum* verwenden:



Die Stimmen-Anzahl des Instruments sollten Sie auf 1 setzen; das Anzeige-Modul Mtr sollten Sie so einstellen, dass es einen Wert anzeigt und immer aktiv ist, wie im vorigen Beispiel beschrieben. Der Schalter sollte sich im Trigger-Modus befinden.



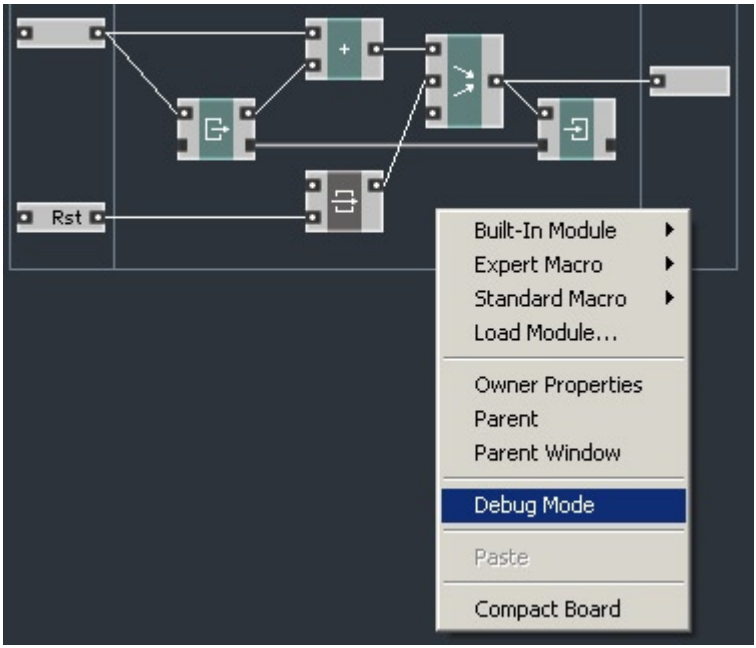
Schalten Sie nun in die Panel-Ansicht und schauen Sie zu, wie sich die Werte in Schritten von 1 pro Sekunde erhöhen und wie sie auf Knopfdruck zurückgesetzt werden.

Wir werden diese Gelegenheit nutzen, um Ihnen den ebug-Modus von REAKTOR Core vorzustellen. Wie Sie wahrscheinlich schon bemerkt haben, können Sie in REAKTOR Core nicht wie auf dem Primary Level den Wert am Ausgang eines Moduls durch Parken des Mauszeigers auf diesem Ausgang anzeigen lassen. Das ist ein unglücklicher Seiteneffekt der internen Optimierung von

REAKTOR Core, der dazu führt, dass die Werte von REAKTOR-Core-Strukturen nicht von außen verfügbar sind

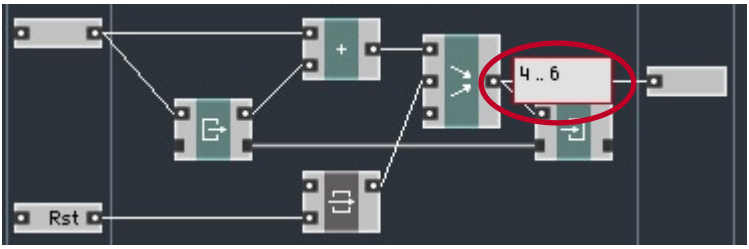
Weil wir Sie schon über diesen Umstand klagen hören, haben wir uns für einen Kompromiss entschieden: Sie können die Optimierung für eine bestimmte Core-Struktur abschalten und dann die Ausgangs-Werte sehen. Lassen Sie uns diesen Debug-Modus an der Struktur ausprobieren, die wir gerade gebaut haben.

Führen Sie einen Rechts-Klick in den Hintergrund aus und wählen Sie aus dem Menü den Eintrag *Debug* Mode:

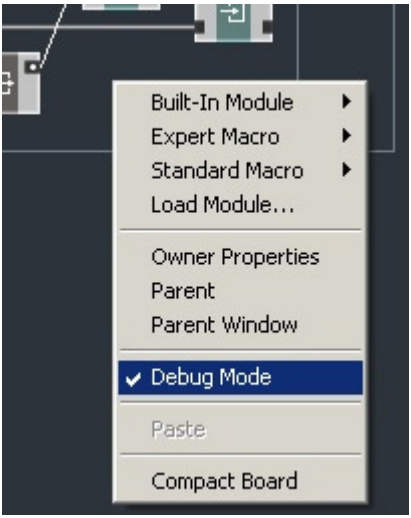


(Sie können dieselbe Funktion auch über die Schaltfläche  in der Werkzeugleiste erreichen).

Wenn Sie nun den Mauszeiger über einen bestimmten Ausgang führen, werden Sie die Werte (oder den Wertebereich) dieses Ausgangs sehen:

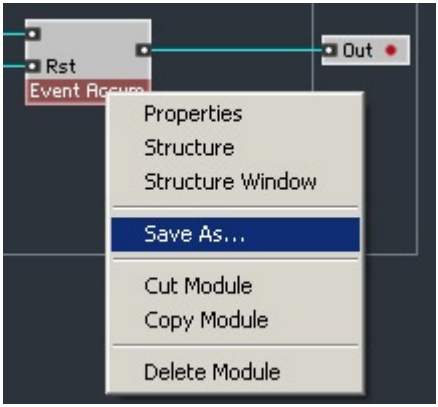


Sie können den Debug-Modus wieder abschalten, indem Sie denselben Eintrag aus dem Menü noch einmal wählen (oder denselben Schalter  ein zweites Mal drücken):

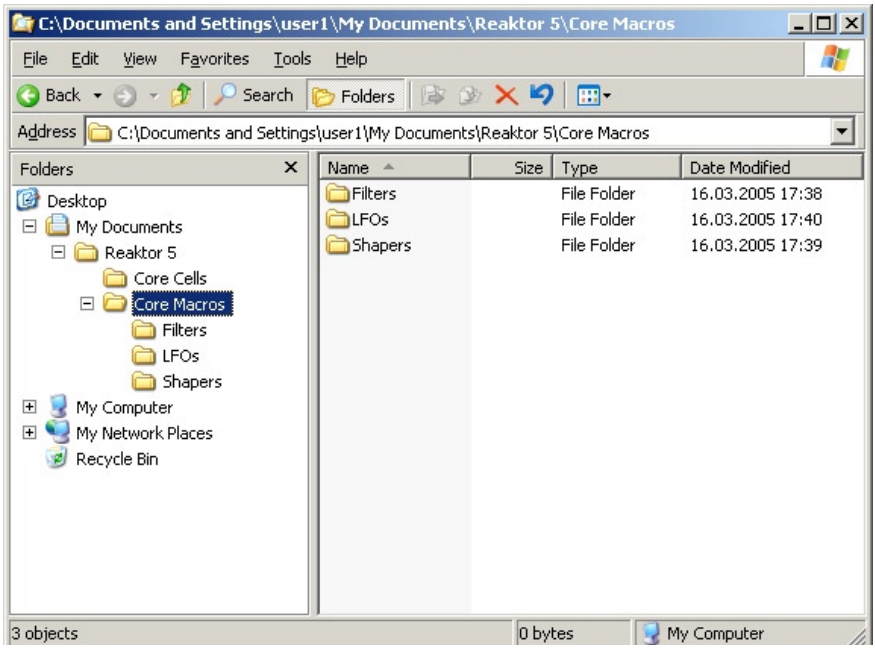


Alternativ dazu wird der Debug-Modus automatisch abgeschaltet, wenn Sie diese Struktur verlassen, sodass Sie ihn für andere Strukturen bei Bedarf wieder einschalten müssen.

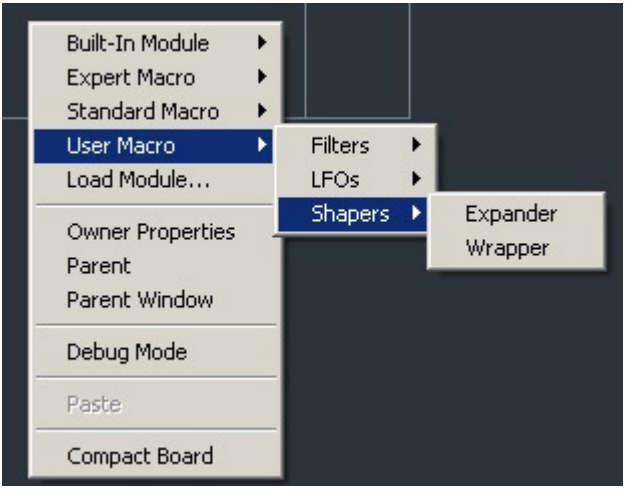
Nach dem "Debugging" unseres REAKTOR-Core-Macros können wir uns überlegen, es für zukünftige Einsätze als separate Datei zu speichern. Dazu führen Sie einen Rechts-Klick af das Macro aus und wählen aus dem Menü den Eintrag *"Save As..."*:



Wie bei Core Cells haben Sie die Möglichkeit, Ihre eigenen Macros in das Menü aufzunehmen. Dazu müssen Sie die Macros in das Unterverzeichnis “Core Macros” im User-Library-Verzeichnis von REAKTOR legen:



Wenn in diesem Ordner *“Core Macros”* oder in einem seiner Unter-Ordner irgendwelche Dateien gefunden werden, erscheint ein neues Untermenü im “Rechts-Klick-Menü”:

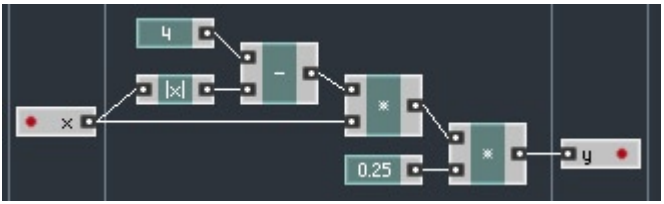


Wie für den Ordner “Core Cells” gelten auch für den Ordner “Core Macros” einige Beschränkungen:

- Leere Ordner werden im Menü nicht angezeigt.
- Legen Sie niemals Ihre eigenen Dateien in die System-Library von REAKTOR, sondern bewahren Sie sie in Ihrem User-Library-Ordner auf.

4.7. Die Reparatur des Event-Shaper-Moduls

Jetzt können wir genauer darauf eingehen, was an der Struktur des Event-Shaper-Moduls, das wir in einem früheren Kapitel gebaut haben, falsch ist:



Das Problem ist das Initialisierungs-Event. Wenn Sie sich ansehen, wie die Initialisierung der obigen Struktur erfolgt, werden Sie folgendes bemerken:

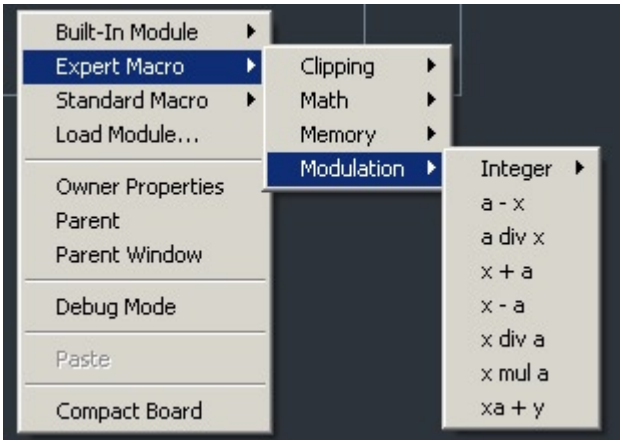
- Der Eingang “x” sendet seine Initialisierungs-Event ab oder sendet sie nicht, abhängig davon, ob er von der umgebenden Primary-Level-Struktur ein Initialisierungs-Event empfängt (dies ist die Ini tialisierungs-Event-Regel für die Core-Cell-Event-Eingänge)

- Die Konstanten 4 und 0.25 senden immer ein Initialisierungs-Event. Dadurch wird in dem Fall, dass am Eingang des Event-Shapers kein Initialisierungs-Event stattfindet (aus welchen Gründen auch immer), der Ausgang des Shapers immer noch das Event von der letzten Multiplikations-Stufe empfangen und dieses Event an die umgebende Primary-Level-Struktur weitergeben.

Obwohl das für die Verarbeitung von Kontroll-Signalen ganz in Ordnung sein mag (weil im Fall einer fehlenden Eingangs-Initialisierung der Eingang den Wert Null annimmt und das Event für die Initialisierung des Ausgangs immer noch gesendet wird), ist es nicht gerade das, was man intuitiv von einem Event-Verarbeitungs-Modul erwarten würde. Ein etwas besser nachvollziehbares Verhalten wäre, wenn das Modul ein Ausgangs-Event nur als Reaktion auf ein ankommendes Event schicken würde.

Unser Problem besteht also darin, dass die beiden Konstanten-Module Events zur falschen Zeit senden dürfen (das heißt, wenn kein Eingangs-Event anliegt). Als Lösung schlagen wir vor, die Subtraktions- und Multiplikations-Module, die Konstanten an ihren Eingängen empfangen, durch ihre Gegenstücke vom Typ *Modulation* zu ersetzen.

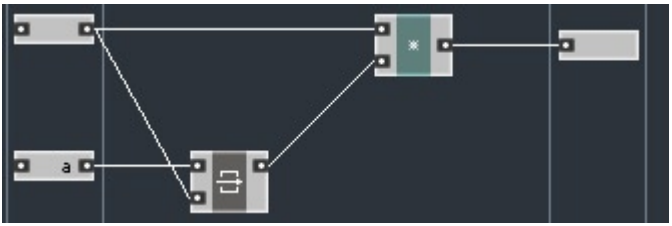
Die "Modulations-Macros" sind eine Gruppe von Modulen in der REAKTOR-Core-Library, die Sie unter *Expert Module > Modulation* finden:



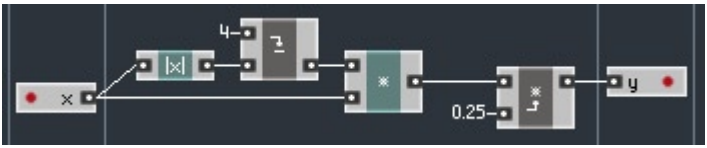
Der Name "Modulation" ist eigentlich nicht ganz korrekt, aber er spiegelt die Fähigkeit dieser Macros wider, ein Signal zu verwenden, um ein ande-

res zu modulieren (was Sie am leichtesten verstehen, wenn wir später in den Low-Level-Strukturen Kontroll-Signale verwenden, um Audio-Signale zu manipulieren). Die meisten dieser Macros kombinieren zwei Signale, von denen eines der "Träger" (oder Carrier) und das andere der "Modulator" ist. Anders als die eingebauten arithmetischen REAKTOR-Core-Module erzeugen die "Modulations-Macros" nur als Reaktion auf ein Event am "Carrier"-Eingang hin eine Ausgabe. Die Events am "Modulator"-Eingang triggern den Rechenvorgang nicht an.

Die interne Implementation der Modulations-Macros ist sehr einfach: Sie speichern einfach das Modulator-Signal zwischen, wobei das Latch-Modul vom Carrier getaktet wird. Hier sehen Sie ein Beispiel für die Macro-interne Struktur eines "Modulations-Multiplizierers" (wobei der Eingang "a" der Modulator ist):



Wir ersetzen also das Subtrahierer-Modul durch das Modulations-Macro a - x und das zweite Multiplizierer-Modul durch das Modulations-Macro *x mul a*. So sieht die Struktur nach dem Ersetzen aus (wir haben auch die Const-Module durch QuickConst ersetzt, aber das ist unwichtig):



Sie können die Modulations-Eingänge von Modulations-Macros normalerweise an ihren Icons (Symbole auf dem Modul) erkennen. Eine Modulator-Eingangs-Position erkennen Sie an der Position des Pfeils auf dem Modul. Im Fall des Subtrahierer-Moduls ist der Pfeil oben, deshalb ist der Modulations-Eingang auch oben. Beim Multiplizierer-Modul ist der Pfeil unten, entsprechend liegt auch der Modulations-Eingang unten. Sie werden auch bemerken, dass der Ausgang dieser Module jeweils dem Carrier-Eingang gegenüber liegt, was einen zusätzlichen Anhaltspunkt bietet.

Schließlich können Sie noch den Mauszeiger über das Modul und seine Eingänge bewegen und die zugehörigen Hinweis-Texte lesen.

In der obigen Struktur werden keine Events gesendet, bis ein Event am Eingang der Core Cell auftritt:

- Das Modul |x| wird direkt vom Core-Cell-Eingangs-Event getriggert.
- Das nachfolgende Subtrahierer-Modul wird nur vom Ausgang des Moduls |x| getriggert, der nur als Reaktion auf das Eingangs-Event ein Event sendet; QuickConst hat keine Trigger-Funktion.
- Der erste Multiplizierer wird entweder vom Ausgang des Subtrahierer-Moduls oder vom Core-Cell-Eingangs-Event angetriggert, aber wir haben bereits gesehen, dass diese beiden Events nur gleichzeitig auftreten.
- Der zweite Multiplizierer wird nur vom ankommenden Event und nicht von QuickConst getriggert.

So, nun legt unsere Struktur ein etwas besser nachvollziehbares Verhalten an den Tag.

5. Audio-Verarbeitung im Kern

5.1. Audio-Signale

Es gibt keinen besonderen Signal-Typ für Audio-Signale in REAKTOR Core. Audio wird in REAKTOR Core als Event behandelt, das sich hinsichtlich seiner Struktur nicht von beliebigen anderen Events unterscheidet. Der Unterschied besteht vielmehr darin, dass entlang von Audio-Signalwegen die Events normalerweise "in regelmäßigen Abständen zu diskreten Zeitpunkten" erzeugt werden, und zwar abhängig von der "Sampling-Rate".

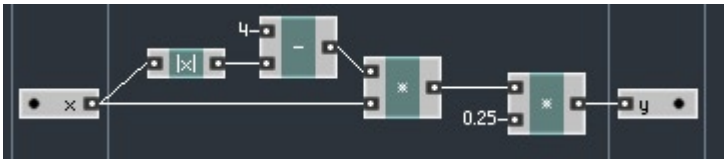
Um Events in regelmäßigen Abständen (oder, was diese Sache angeht, irgendwelche Events) zu erzeugen, brauchen wir eine Event-Quelle. Ähnlich wie bei Event-Core-Cells, bei denen die Eingänge der Module die Event-Quellen waren, sind auch bei Audio-Core-Cells die Eingänge die Event-Quellen. Allerdings steht uns hier ein zusätzlicher Eingangs-Typ zur Verfügung:

Audio Inputs senden regelmäßig und mit der in den Sampling-Rate-Einstellungen der umgebenden Primary-Level-Struktur eingestellten Rate REAKTOR-Core-Events an das Innere der Struktur. Die Events werden gleichzeitig von allen Audio-Eingängen einer Core-Cell-Struktur gesendet.
Die Audio-Eingänge senden auch das Initialisierungs-Event an die Core-Cell-Struktur. Dieses Event wird unabhängig davon gesendet, was gerade in der umgebenden Primary-Level-Struktur passiert. Auf jeden Fall hängt der Wert, den diese Eingänge während der Initialisierung senden, vom äußeren Initialisierungs-Vorgang ab.

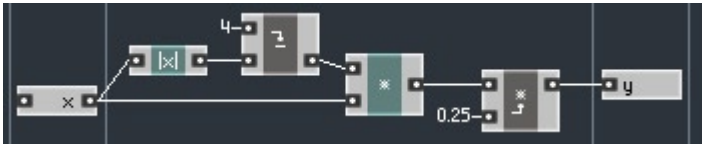
Es gibt auch einen Ausgangs-Typ, der *anstelle* von Event-Ausgängen verwendet werden muss.

Audio Outputs liefern den letzten aus dem Inneren der REAKTOR-Core-Struktur empfangenen Wert an die umgebende Primary-Level-Struktur. Weil die Audio-Ausgänge auf dem Primary Level keine Events senden, werden keine Events nach außen geschickt.

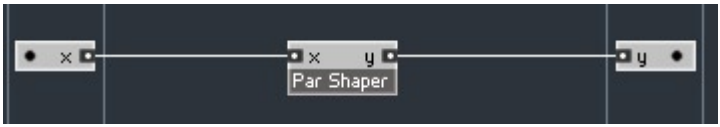
Jetzt werden wir denselben Shaper, den wir für Events gebaut haben, für Audio-Signale nachbauen. Dafür erzeugen Sie zuerst eine Audio-Core-Cell. Grundsätzlich können wir genau dieselbe Struktur verwenden, nur dass wir anstelle von Event-Eingängen und –Ausgängen die Audio-Variante wählen müssen:



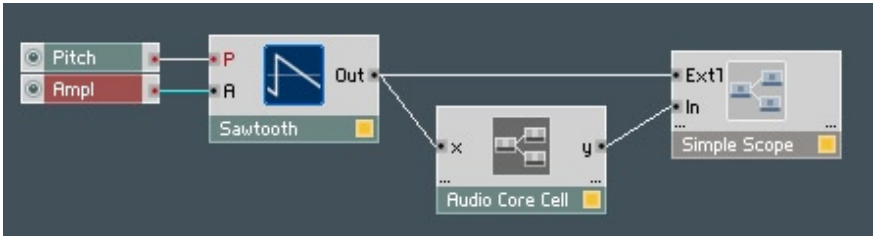
Sie fragen sich vielleicht, warum wir in diesem Fall nicht die Modulations-Macros verwenden. Nun, weil wir hier ein Audio-Signal verarbeiten und Audio-Signale immer ein Initialisierungs-Event senden, sind wir hier auf der sicheren Seite. Wenn Sie wollen, können Sie aber auch Modulations-Macros verwenden, das macht keinen Unterschied.
Wir könnten auch versuchen, die oben abgebildete Struktur in ein Macro zu verpacken, das wir in verschiedenen REAKTOR-Core-Struktur sowohl für Audio- als auch für Event-Verarbeitung verwenden können. In diesem Fall sollten wir besser Modulations-Macros verwenden, weil wir ja nicht im Voraus wissen, welche Art von Signalen von diesem Modul verarbeitet wird:



Und das ist die innere Struktur der Audio-Core-Cell in diesem Fall:



Um sie auszuprobieren, schließen wir einen Sägezahn-Oszillator und ein Oszilloskop an. Ein Oszilloskop finden Sie unter *Insert Macro > Classic Modular > 00 Classic Modular – Display > Simple Scope* (aus einer Primary-Level-Struktur). Vergessen Sie auch nicht, die Anzahl der Stimmen für das Instrument auf 1 zu begrenzen.



Wir verwenden den externen Trigger für das Oszilloskop, um bei hohen Verzerrungsgraden eine bessere Synchronisation zu erzielen (der Schalter *Ext* auf dem Oszilloskop-Panel muss aktiv sein, damit das funktioniert). Ändern Sie den Bereich des Reglers *Ampl* auf einen Wert zwischen 0 und 5, damit Sie die formende Wirkung unserer Schaltung sehen können.

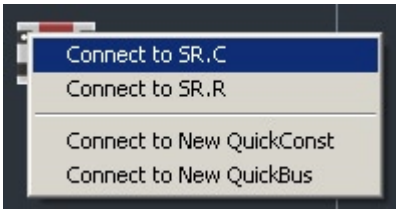
5.2. Sampling Rate Clock Bus

Es sieht ganz so aus, als würden zumindest einige Funktionen zum Aufbauen von Audio-Strukturen noch fehlen. Eine könnte uns zum Beispiel erlauben, Audio-Core-Cells ohne Audio-Eingänge zu entwerfen. Wir können solche Core Cells natürlich bauen, aber woher bekommen wir dann die Quelle für die Audio-Events? Die zweite Funktion brauchen wir, weil viele DSP-Algorithmen Kenntnis über die derzeitige Sampling-Rate haben müssen. Wir werden Ihnen diese fehlenden Funktionen im Folgenden vorstellen.
In jeder REAKTOR-Core-Struktur ist eine besondere Art von Verbindungen möglich, die “Sampling Rate Clock Bus” heißt. Dieser Bus überträgt zwei Signale: die Clock und die Rate.

Clock ist eine Signal-Quelle, die Events regelmäßig mit Audio-Sampling-Rate sendet. Wie bei allen Standard-Audio-Signalen der Fall, sendet sie außerdem ein Initialisierungs-Event.. Der Wert dieser Events ist zurzeit immer Null, aber im eigentlich sollte jede Struktur, die dieses Signal verwendet, die Werte ignorieren, weil diese sich in Zukunft möglicherweise ändern.

Rate ist eine Signal-Quelle, deren Wert immer der aktuellen Audio-Sampling-Rate in Hz entspricht. Die Events werden von dieser Quelle während der Initialisierung und bei jeder Änderung der Sampling-Rate gesendet.

Sie können auf den Sampling-Rate-Bus zugreifen, indem Sie auf einem Signal-Eingang einen Rechts-Klick ausführen und aus dem Menü den Eintrag *Connect to SR.C* auswählen, um den Eingang mit einem Clock-Signal zu verbinden. Um den Eingang mit einem Rate-Signal zu verbinden, wählen Sie den Eintrag *Connect to SR.R*.



Die Verbindung wird direkt neben dem Eingang angezeigt:

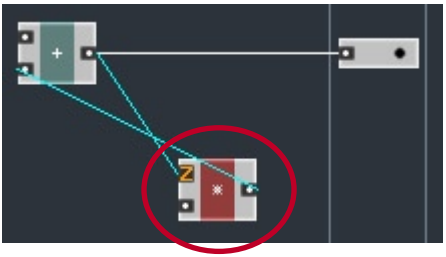
Der Sampling Rate Clock Bus arbeitet innerhalb von Event-Core-Cells nicht.

5.3. Verbindungs-Feedback

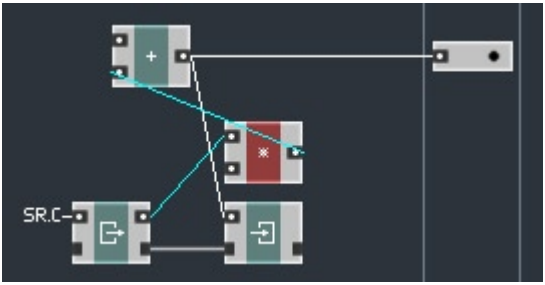
Wie Sie bereits gesehen haben, lassen sich die Regeln für die Verarbeitungsreihenfolge nicht anwenden, wenn innerhalb der Struktur Feedback-Verbindungen existieren. Deshalb müssen wir ein paar weitere Regeln aufstellen, die über den Umgang mit solchen Rückkopplungen bestimmen.

Die Hauptregel ist: REAKTOR-Core-Strukturen können nicht mit Feedback umgehen.
Das stimmt natürlich nicht ganz – Sie können Feedback-Verbindungen in REAKTOR Core anlegen. Aber weil die Engine von REAKTOR Core keine Strukturen mit Feedback verarbeiten kann, wird sie das Feedback *auflösen*. "Feedback auflösen" bedeutet, dass Ihre Struktur so verändert wird (und zwar intern, Sie werden es auf dem Bildschirm nicht sehen), dass keine Rückkopplung mehr auftritt.
Nur für den Fall, dass Sie es noch nicht wussten: In der digitalen Welt ist ein Feedback ohne Verzögerung nicht möglich. Normalerweise entsteht eine Verzögerung von mindestens einem Sample im digitalen Audio-Signalweg. Und

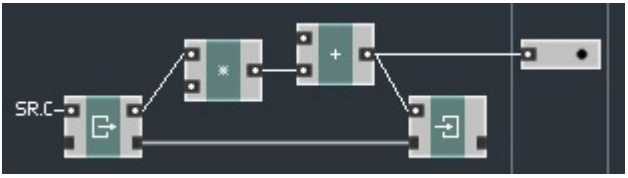
deshalb setzt die Engine von REAKTOR Core beim Auflösen der Feedback-Situation ein Modul (Z^-1) ein, das einen Versatz von einem Sample Länge in den Feedback-Pfad einfügt.Bekanntlich wird eine Stelle, an der das Modul Z^-1 implizit seine Arbeit verrichtet, durch das große orangefarbene Z gekennzeichnet, das Sie an dem betreffenden Port sehen:



Wir haben schon eine Struktur auf der Grundlage der Module *Read* und *Write* gesehen, in der die Z^-1-Funktion zum Einsatz kam. Lassen Sie uns eine solche Konstruktion mit *Read* und *Write* in unsere Struktur einbauen. Wir werden sie an das Kabel anschließen, an dem die Feedback-Auflösung stattfindet:



Also, erst schreiben, dann lesen (und beachten Sie, dass das Modul *Read* via SR.C getaktet wird, um sicherzustellen, dass der Lese-Vorgang einmal pro "Audio-Tick" stattfindet). Deshalb liegt der gelesene Wert immer ein Audio-Sample hinter dem geschriebenen zurück. Und schon ist da kein Feedback mehr in der Struktur. Sehen Sie's nicht? Okay, lassen Sie uns die Module ein bisschen zurechtrücken (und dabei keine einzige Verbindung verändern):



Sehen Sie's jetzt? Na klar.

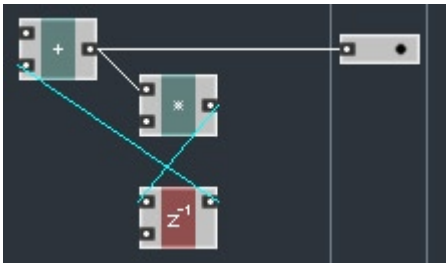
Also beseitigt das Einfügen eines Z^{1}-Moduls formal die Rückkopplung aus der Struktur, belässt sie aber logisch betrachtet darin (durch die Verzögerung von einem Audio-Sample).

Tatsächlich ist die interne Struktur eines Macros des Typs Z^-1 etwas komplizierter und besteht nicht nur aus einem Paar von *Read*- und *Write*-Modulen. Wie es genau beschaffen ist und warum das so ist, lernen Sie im nächsten Abschnitt.

Sie haben keine Kontrolle darüber, an welcher Stelle die automatische Feedback-Auflösung stattfindet. Sie tritt an einer beliebigen Signal-Verbindung innerhalb der Feedback-Schleife auf. Es ist nicht einmal gesagt, dass die Auflösung immer an dieser bestimmten Verbindung auftritt – in der nächsten Version der Software kann das anders sein, aber es kann auch schon ausreichen, die Struktur an einer anderen Stelle zu ändern oder sie neu von der Festplatte zu laden.
Diese automatische Feedback-Auflösung ist für Strukturen gedacht, bei denen es nicht von Bedeutung ist, wo genau die Auflösung stattfindet. Es kann nämlich vorkommen, dass solche Feedback-Strukturen von Anwendern erstellt werden, die sich noch nicht gut genug mit digitaler Signal-Verarbeitung auskennen, um das Rückkopplungs-Problem zu verstehen. Die automatische Feedback-Auflösung erlaubt diesen Anwendern immer noch, brauchbare Ergebnisse zu erzielen.

Wenn Sie die Punkte, an denen in Ihrer Struktur eine Feedback-Auflösung stattfinden soll, genau bestimmen wollen, können Sie dafür gezielt Z^-1-Module in die Struktur einsetzen. Diese Module werden die Rückkopplung formal entfernen, eine Auflösung durch REAKTOR Core ist dann nicht mehr notwendig.

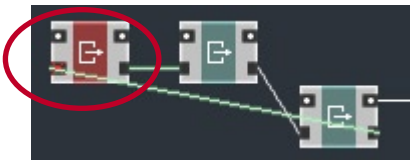
Hier sehen Sie eine Version der oben abgebildeten Struktur mit einem Z^-1-Macro (das Sie im Untermenü *Expert Macro > Memory* finden):



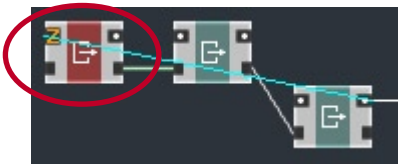
Wie Sie sehen können, ist das große orangefarbene Z nun verschwunden. Beachten Sie auch, dass der Punkt, an dem der Versatz von einem Sample erzeugt wird, anders angeordnet ist als bei der automatischen Feedback-Auflösung: Die automatische Auflösung fand in der Verbindung zwischen dem Ausgang des Addierers und dem Eingang des Multiplizierers statt, unsere neue Version löst die Rückkopplung zwischen dem Ausgang des Multiplizierers und dem Eingang des Addierers auf.

Die Bedeutung des zweiten Eingangs am Modul Z^-1 wird später noch erklärt. Im Normalfall lassen Sie diesen Eingang einfach unbeschaltet.

Feedback in OBC und anderen Typen von Nicht-Signal-Verbindungen (die wir noch kennen lernen werden) ist vollkommen sinnlos und deshalb nicht erlaubt. Sofern Feedback-Schleifen auftreten, in denen keine Kabel-Verbindungen vorkommen, wird eine dieser Verbindungen als ungültig markiert und als nicht vorhanden betrachtet. Die Markierung “ungültig” ist ein großes rotes X auf dem betreffenden Port:



Auf der anderen Seite sind Feedback-Schleifen mit gemischten Arten von Verbindungen , die normale Signal-Kabel enthalten, vollkommen in Ordnung. Diese Rückkopplungen werden normal aufgelöst, wobei die Auflösung in einem der normalen Signal-Kabel stattfindet:

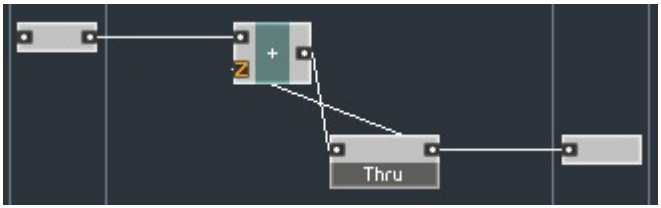


Alles in allem bedeutet dies, das Nicht-Signal-Verbindungen nie von der Feedback-Auflösung betroffen sind, außer wenn Sie komplette Nicht-Signal-Rückkopplungen erzeugen, was nun wirklich überhaupt keinen Sinn ergibt.

5.4. Feedback um Macros herum

Macros werden hinsichtlich der Feedback-Auflösung im Allgemeinen genauso behandelt wie die eingebauten Module.
Lassen Sie uns ein Macro betrachten, das einfach das ankommende Signal durchlässt. Das hier ist die interne Struktur eines solchen Macros, das wir einfach *Thru* nennen:

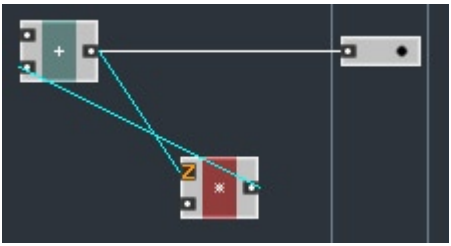
Nun nehmen wir an, wir bauen eine Feedback-Struktur, in der wir dieses Macro verwenden:



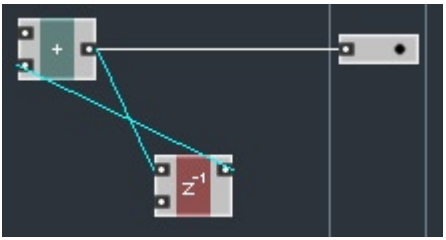
Die Feedback-Schleife verläuft durch zwei Kabel in der obigen Struktur und durch ein weiteres Kabel *innerhalb* des Macros. Nur, wo findet jetzt die Auflösung statt? Okay, Sie können auf dem Bild oben erkennen, dass sie i*n diesem speziellen* Fall am Eingang des Addierer-Moduls stattfindet, aber wie wir wissen, hätte sie auch irgendwo anders stattfinden können.
Stellen Sie sich für einen Moment vor, dass Thru kein Macro, sondern ein eingebautes Modul wäre. In diesem Fall ist es offensichtlich, dass die Feedback-Auflösung nicht innerhalb dieses Moduls stattfinden könnte, sondern außerhalb geschehen müsste. Nun, wir versuchen unser Bestes, den Macros das Aussehen und Verhalten der eingebauten Module beizubringen. Aus diesem Grund findet *standardmäßig* die Auflösung von Feedback-Schleifen außerhalb des Macros statt. Es ist zwar nicht festgelegt, wo genau sie stattfindet, aber die Stelle liegt auf jeden Fall außerhalb unseres Macros *Thru*.

> Die allgemeine Regel lautet: Die Feedback-Auflösung erfolgt auf der höchsten Struktur-Ebene der Feedback-Schleife

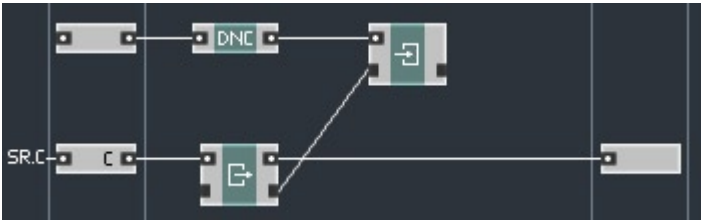
Trotzdem können Sie dieses Verhalten ändern – namentlich können Sie Feedback-Auflösung innerhalb von Macros erlauben. Sie hätten sich tatsächlich schon fragen sollen, wie das Macro Z^-1 die Rückkopplung auflöst, wenn Macros genauso behandelt werden wie eingebaute Module. Schauen Sie sich einmal die folgende Struktur an:



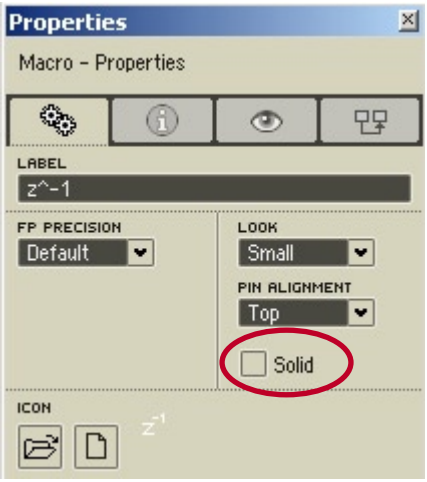
Wenn Macros und eingebaute Module dasselbe sind, sollte sich doch eigentlich nichts ändern, wenn wir den Multiplizierer durch das Macro Z^-1 ersetzen:



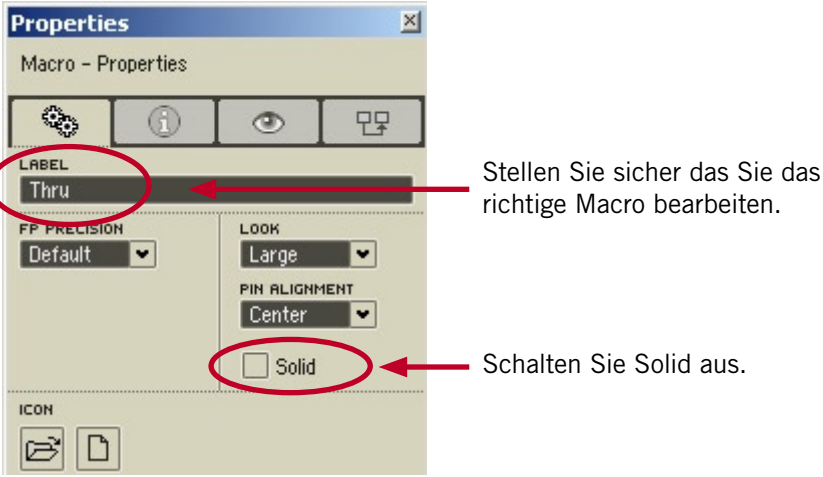
Aber es *gibt* einen Unterschied, weil das implizite Feedback jetzt verschwunden ist. Es muss also etwas Besonderes an diesem Z^-1 Macro sein. Und in der Tat gibt es eine Besonderheit daran. Wenn wir uns dieses Macro von innen ansehen, erkennen wir zur Verwirklichung der Z^-1-Funktion fast dieselbe Struktur wie die bereits früher erwähnte:



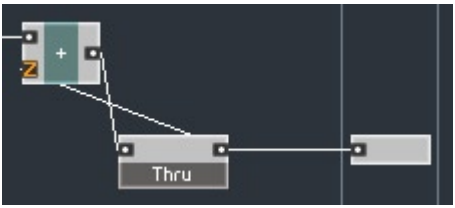
Wie Sie sehen, ist der Clock-Eingang des Macros mit dem internen *Read*-Modul verbunden. Die Default-Verbindung für diesen Eingang ist keine Null-Konstante, sondern die *Audio-Clock*, und genau das wollen wir in den meisten Fällen. Das Modul, das zwischen dem oberen Eingang und dem Modul *Write* angeschlossen ist, sehen wir uns später noch an; vorerst ignorieren Sie es einfach.
Bisher also nichts Besonderes an der Struktur, außer dass sie die Z^-1-Struktur zu enthalten scheint, die wir schon behandelt haben. Trotzdem, woher weiß die Engine von REAKTOR Core, dass diese Struktur dazu da ist, Feedback-Schleifen aufzulösen? Es ist ja offensichtlich, dass die Engine Feedback-Schleifen auflösen kann, aber wie erkennt sie, ob das beabsichtigt ist?
Diese Eigenschaft kontrollieren Sie im Properties-Fenster des Macros, genauer



Diese Eigenschaft verrät der REAKTOR-Core-Engine, ob das Macro für den Zweck der Feedback-Auflösung wie ein "festes" (solid) eingebautes Modul oder transparent erscheinen soll. In 99 % der Fälle werden Sie diese Eigenschaft eingeschaltet lassen. Das sollten Sie tun, weil Sie normalerweise keine implizite Feedback-Auflösung im Inneren Ihrer Macros stattfinden lassen wollen.
Ein Grund dafür ist, dass die Auflösung im Inneren eines Macros unsichtbar wäre, bis Sie in das Macro hineinschauen, sodass einige der impliziten Verzögerungen durch die Z^-1-Funktion unbemerkt geschehen. Zum Beispiel können wir unsere frühere Struktur mit dem Macro *Thru* nehmen und die Eigenschaft *Solid* abschalten (achten Sie darauf, dass Sie die *Solid*-Einstellung für das richtige Macro bearbeiten; das erkennen Sie an dem Text "Thru" im Feld *Label* des Properties-Fensters):



Jetzt sieht die äußere Struktur wahrscheinlich genauso aus (wir sagen "wahrscheinlich", weil man nie sicher sein kann, wo genau die automatische Feedback-Auflösung stattfinden wird):



Aber wenn Sie Ihre Struktur ein wenig verändern, indem Sie den Ausgang mit einem anderen Modul verbinden. könnte sie so aussehen:

Unsere Feedback-Auflösungs-Verzögerung scheint verschwunden zu sein. In einer größeren und unübersichtlicheren Struktur könnten wir leicht übersehen, das eine implizite Verzögerung stattfindet. Wo ist unser Z^-1-Delay hin? Natürlich befindet es sich jetzt innerhalb unseres Macros *Thru* – an dem einzigen Ort in der Struktur, an dem wir es nicht sehen können.:

Ein anderer Grund dafür, die Eigenschaft *Solid* eingeschaltet zu lassen, ist, dass sich in manchen Fällen die Funktion eines Macros ändern kann, wenn es in den Feedback-Pfad eingesetzt wird. Also tun Sie sich selbst den Gefallen und schalten Sie diese Eigenschaft nur dann ab, wenn Sie Macros bauen, die Feedback auflösen sollen. Das wird aber nicht oft vorkommen.
Lassen Sie uns nun zum Modul Z^-1 zurückkehren. Wenn die Eigenschaft *Solid* für dieses Macro abgeschaltet ist, werden die Grenzen des Macros für die Feedback-Auflösung vollkommen durchsichtig. Dadurch wird das Macro Z^-1 nicht wirklich wie ein eingebautes Modul behandelt und ist daher in der Lage, Rückkopplungen in der beschriebenen Weise aufzulösen.

5.5. Denormale Werte

Die Signal-Werte in den Strukturen, die wir während der bisherigen Abschnitte aufgebaut haben, werden im Computer durch einen binären Daten-Typ repräsentiert, den man auf Deutsch *Fließkomma-Zahlen*, auf Englisch *floating point numbers* oder kurz *floats* nennt. Diese Art von Daten erlaubt die effiziente Darstellung eines großen Spektrums von Werten.Die Bezeichnung *Fließkomma-Zahlen* gibt keinen genauen Aufschluss darüber, wie die Zahlen dargestellt werden, sondern beschreibt nur den Ansatz, der für ihre Darstellung gewählt wird, und lässt bei der Umsetzung eine Menge Freiraum für die Details der Implementation.
Die CPUs heutiger Personal Computer verwenden den "IEEE Floating Point Standard". Dieser Standard definiert genau, wie die Fließkomma-Zahlen repräsentiert werden sollten und was bei Rechenoperationen mit diesen Zahlen geschehen soll (z. B., wie mit dem Thema der begrenzten Genauigkeit umgegangen werden soll). Insbesondere besagt dieser Standard, dass für eine Gruppe von besonders kleinen Fließkomma-Werten, die wegen der begrenzten Fließkomma-Genauigkeit nicht auf die "übliche" Weise dargestellt werden können, eine andere Darstellungsform gewählt werden soll. Diese andere Darstellungsform nennen wir "denormale Repräsentation" oder auch kurz "Denormale".

Der Bereich der "denormalen" Werte für 32-Bit-Fließkomma-Werte reicht ungefähr von 10^{-38} bis 10^{-45} und von -10^{-38} bis -10^{-45}. Kleinere Werte als 10^{-45} sind überhaupt nicht darstellbar und werden als Nullen angesehen.

Weil diese Zahlen eine etwas andere Darstellungsform haben als "normale" Zahlen, haben manche CPUs bestimmte Probleme im Umgang mit solchen Zahlen. Besonders Rechenoperationen mit diesen Zahlen können nur sehr viel langsamer durchgeführt werden und dauern bis zu zehnmal solange wie mit "normalen" Zahlen.

Eine typische Situation, in der denormale Zahlen über *längere Zeiträume hinweg* auftreten, ergibt sich durch exponentiell abfallende Werte, wie sie in Filtern, in manchen Hüllkurven oder in Feedback-Strukturen vorkommen. In solchen Strukturen fallen die Signale *asymptotisch* bis auf Null ab, nachdem das Eingangssignal auf den Wert Null geschaltet wurde. *Asymptotisch* bedeutet, dass das Signal versucht, den Wert Null zu erreichen, das aber nie schafft. In dieser Situation können denormale Zahlen auftreten und für relativ lange Zeit in der Struktur verweilen, wodurch sich die CPU-Belastung deutlich erhöht.

Eine andere Situation, in der denormale Zahlen auftreten können, entsteht, wenn Sie die Genauigkeit eines Fließkomma-Werts von einer höheren Genauigkeit (64 Bit) auf eine niedrigere Genauigkeit (32 Bit) umschalten. Auslöser ist hier, dass ein Wert von 10^{-41} bei einer Fließkomma-Genauigkeit von 64 Bit nicht denormal ist, bei 32-Bit-Genauigkeit hingegen schon. Das Verändern der Fließkomma-Genauigkeit diskutieren wir übrigens später.

Lassen Sie uns nun die Modellierung eines 1-Pol-Tiefpassfilters mit einer Cutoff-Frequenz von 20 Hz betrachten. Unsere digitalen Signal-Werte werden analogen Spannungen in Volt entsprechen. Wir stellen uns vor, dass der Pegel des Eingangssignals über einen hinreichend langen Zeitraum gleichwertig mit einer Spannung von 1 Volt wäre. Dann ist die Spannung am Ausgang des Filters ebenfalls gleichwertig mit 1 V. Jetzt ändern wir abrupt die Eingangsspannung auf Null. Die Ausgangsspannung wird gemäß dem folgenden Gesetz abfallen:

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

wobei f_c der Filter-Cutoff in Hz ist, *t* die Zeit in Sekunden und $V_0 = 1\mathrm{V}$ (Anfangs-Spannung).
Dann wird sich die Ausgangsspannung wie folgt verändern:

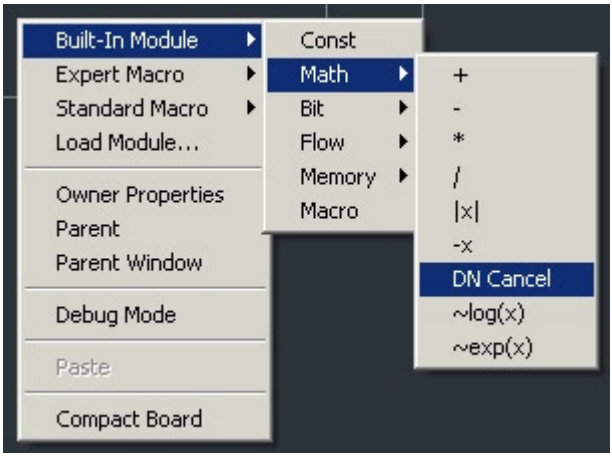
nach 0.5 sec	$V_{out} \approx 10^{-29}$ volt
nach 0.6 sec	$V_{out} \approx 10^{-33}$ volt
nach 0.7 sec	$V_{out} \approx 10^{-38}$ volt
nach 0.8 sec	$V_{out} \approx 10^{-44}$ volt

Oha, die Zahlen zwischen 10^{-38} und 10^{-45} liegen im denormalen Bereich. Also wird in der Zeitspanne ungefähr zwischen 0,7 bis 0,8 Sekunden unsere Spannung durch einen denormalen Wert dargestellt. Und das nicht nur im Inneren des Filters – das Ausgangssignal des Filters wird wahrscheinlich von einigen stromabwärts in der Struktur liegenden Modulen weiter verarbeitet, sodass es zumindest diese paar folgenden Module ebenfalls mit denormalen Werten zu tun bekommen.

Bei einer Sampling-Rate von 44,1 kHz entspricht ein Zeitintervall von 0,1 Sekunden 4410 Samples. Wenn wir voraussetzen, dass die typische ASIO-Puffergröße einige Hundert Samples beträgt, müssen wir bei deutlich höherer CPU-Last mehrere Puffer erzeugen. Falls die CPU-Belastung (pro Puffer-Berechnung) nah genug an 100 % heranreicht oder diesen Wert überschreitet, führt das zu Audio-Aussetzern.

Aus dem obigen Text müssen Sie vor allem einen Schluss ziehen: Denormale Werte sind schlecht für die Echtzeit-Verarbeitung von Audio.

Die Module auf REAKTORs Primary Level sind so programmiert, dass sie auf denormale Werte achten, die in ihrem Inneren auftreten. Dieser Vorsorge-Mechanismus beruht auf Modifikationen an den verwendeten DSP-Algorithmen; diese wurden so verändert, dass sie generell keine denormalen Werte erzeugen. Wenn Sie Ihre eigenen Low-Level-DSP-Strukturen in REAKTOR Core entwerfen, müssen Sie denormale Werte ebenfalls vermeiden. Um Ihnen dabei zu helfen, haben wir das Modul *Denormal Cancel* eingeführt, das Sie im Untermenü *Built In Module > Math* finden:



Das Modul *Denormal Cancel* besitzt einen Eingang und einen Ausgang und versucht, den ankommenden Wert vorsichtig so zu verändern, dass keine Denormalen den Ausgang erreichen:

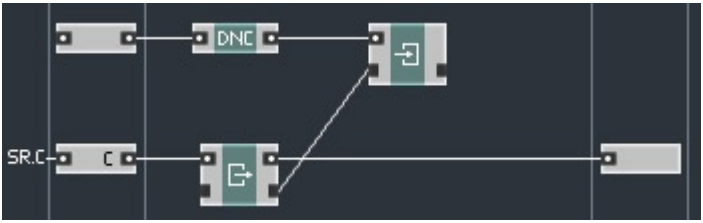
Die Art, wie dieses Modul das Signal modifiziert, ist nicht dauerhaft festgelegt und kann sich von einer Software-Version zur andern verändern oder sogar an zwei Stellen in der Struktur unterschiedlich sein. Zurzeit wird bei der Modifikation dem Eingangs-Wert eine sehr kleine Konstante hinzugefügt. Wegen der damit einhergehenden Einbußen bei der Genauigkeit verändert diese Addition keine Werte, die groß genug sind (so wird ein Wert von beispielsweise 10^{-10} überhaupt nicht verändert). Dieselben Genauigkeits-Verluste machen es andererseits sehr unwahrscheinlich, dass das Ergebnis der Addition ein denormaler Wert ist (und in den meisten Fällen ist das sogar unmöglich).

Wenn aus irgendeinem Grund das Modul *Denormal Cancel* (DN Cancel) in Ihrer Struktur nicht funktioniert, können Sie selbstverständlich Ihre eigene Technik zum Unterdrücken denormaler Werte verwenden. Allerdings kann dabei das Problem auftreten, dass diese Technik, die auf einer Plattform arbeitet, auf der anderen nicht arbeitet, während wir versucht haben, unseren eingebauten *DN-Cancel*-Algorithmus an jede unterstützte Plattform anzupassen. Versuchen Sie also nach Möglichkeit, das Modul zu verwenden. Wir denken sogar darüber nach, alternative Algorithmen in dieses Modul einzubauen – fühlen Sie sich ermutigt, in unserem Support-Forum Ihre Meinung zu diesem Thema zu sagen!

Einige CPUs bieten die Möglichkeit, den IEEE-Standard zu verletzen und die Erzeugung denormaler Zahlen zu unterdrücken, wobei die denormalen Werte zwangsweise auf Null gesetzt werden. Diese Option ist manchmal verfügbar, manchmal nicht. Weil REAKTOR-Core-Strukturen eigentlich plattformunabhängig arbeiten sollen, raten wir Ihnen sehr, bereits in Ihren Strukturen auf eine Unterdrückung denormaler Werte zu achten, selbst wenn Ihr eigener Rechner nicht unter den Auswirkungen dieser Werte leidet.

Weil eine der häufigsten Situationen, in denen denormale Werte auftreten, exponentiell abfallende Feedback-Schleifen sind, und weil die meisten Feedback-Schleifen in der Audio-Verarbeitung exponentiell abfallen (was Filter und Feedback-Schleifen mit Delays belegen), haben wir uns entschlossen, die Denormalen-Unterdrückung in das Standard-Macro Z^-1 einzubauen.

Wie Sie sich erinnern, sieht das Innere dieses Macros folgendermaßen aus:



Es gibt noch eine andere Version dieses Macros namens *Z^-1 ndc*, die keine Denormalen-Unterdrückung durchführt (ndc = no denormal cancel).

Sie können diese alternative Version in Strukturen verwenden, von denen Sie sicher wissen, dass sie keine Denormalen erzeugen (z. B. FIR-Filter):

5.6. Andere böse Zahlen

Denormale Zahlen sind nicht die einzige schlechte Art von Zahlen, die in Strukturen mit internem Zustand und besonders in Feedback-Schleifen kleben bleibt. Es gibt noch ein paar andere Sorten schlechter Zahlen: INFs, NaNs und QNaNs. Wir werden darauf jetzt nicht im Detail eingehen, die entsprechenden Informationen sind aber anderweitig verfügbar, zum Beispiel im Internet. Für uns wichtig ist, wie wir das Auftauchen dieser Zahlen in unseren Strukturen verhindern.

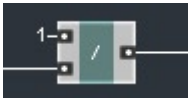
Im Allgemeinen erscheinen solche Zahlen als Ergebnis ungültiger Operationen. Das Teilen durch Null ist der einfachste Fall. Andere Fälle sind Zahlen, die zu groß sind, um in die Fließkomma-Darstellung zu passen (das wäre bei Werten oberhalb von 10^{38} in vollem Ausmaß der Fall), oder die außerhalb des erlaubten Bereichs für eine bestimmte Operation liegen.

Solche Zahlen neigen dazu, in den Strukturen haften zu bleiben, und auf bestimmte Weise sind sie noch viel klebriger als Denormale. Wenn Sie nämlich einen denormalen Wert und einen anderen, nicht denormalen Wert addieren, wird das Ergebnis nicht denormal sein (es sei denn, der andere Wert ist auch extrem klein und liegt dicht an der Grenze zur Denormalität). Wenn Sie andererseits einen Wert zu einem INF addieren, wird das Ergebnis immer noch ein INF sein.

Abgesehen von ihrer Neigung, sich in Strukturen auf ewig (das heißt, bis die Struktur zurückgesetzt wird) festzusetzen, haben diese Zahlen auch die schlechte Angewohnheit, auf manchen CPUs extrem viel Rechenzeit zu fordern. Deshalb sollten wir vorsichtig sein und uns nach Kräften bemühen, die Entstehung dieser Zahlen zu verhindern.

Vorsichtig sein heißt zum Beispiel, dass Sie immer, wenn zwei Zahlen dividiert werden sollen, überprüfen müssen, ob eine Division durch Null möglich ist. Die Initialisierung ist hier von besonderer Bedeutung.

Betrachten Sie zum Beispiel folgendes Element einer Struktur:



Wenn aus irgendeinem Grund das Initialisierungs-Event nicht am unteren Eingang des Dividierer-Moduls ankommt, wird während der Verarbeitung der Initialisierung eine Division durch Null erfolgen. In diesem Fall können Sie darüber nachdenken, ein Delay-Macro aus der Abteilung "Modulation" zu verwenden, oder nach einer zu Ihren jeweiligen Anforderungen passenden Alternative suchen.

5.7. Wie Sie ein 1-Pol-Tiefpassfilter bauen

Ein einfaches 1-Pol-Tiefpassfilter können Sie nach der folgenden rekursiven Gleichung aufbauen:

$$y=b*x+(1-b)*y_{-1}$$

wobei

x das Eingangs-Sample ist,
y das neue Ausgangs-Sample,
y_{-1} das vorige Ausgangs-Sample ist und
b die Koeffiziente, die den Filter-Cutoff festlegt.

Den Wert der Koeffizienten *b* nehmen wir als gleichwertig zur normalisierten zyklischen Cutoff-Frequenz an, die sich nach folgender Formel errechnen lässt:

$$F_c = 2*\pi*f_c / f_{SR}$$

wobei

f_c die gewünschte Cutoff-Frequenz in Hz ist,
f_{SR} die Sampling-Rate in Hz ist,
π etwa 3,14159... entspricht und
F_c der normalisierte zyklische Cutoff (in Radianten, falls Sie das interessiert) ist.

Tatsächlich entspricht die Koeffiziente b dem normalisierten Cutoff nur ungefähr, wobei sich die Abweichung bei hohen Cutoff-Werten vergrößert. Für unsere Zwecke sollte das aber mehr oder weniger in Ordnung sein, besonders, wenn wir keine genaue Einstellung der Cutoff-Frequenz für unser Filter brauchen

---

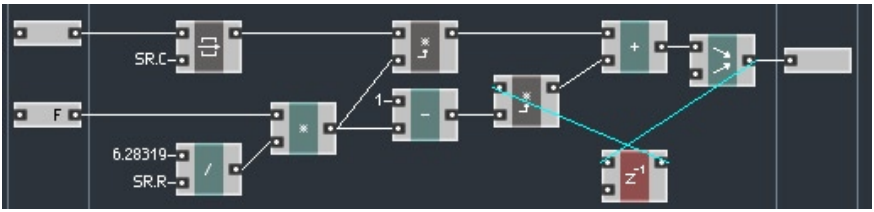
Wir beginnen, indem wir eine Audio-Core-Cell mit zwei Eingängen erzeugen: einen für den Audio-Eingang und eine für den Cutoff. Wir werden in dieser Version des Moduls einen Event-Eingang für den Cutoff verwenden.

Tatsächlich glauben wir, dass es eine gute Angewohnheit ist, REAKTOR-Core-Strukturen als Macros aufzubauen, um ihre leichte Wiederverwendung zu ermöglichen. Deshalb werden wir unser Filter auch als Macro anlegen. Wir erzeugen also ein neues Macro in der Core Cell und bestücken es mit derselben Konstellation von Eingängen und Ausgang:

Lassen Sie uns nun die Schaltung für die Konvertierung der Cutoff-Frequenz in den normalisierten zyklischen Cutoff bauen:

6.28319 ergibt sich aus *2*π*, dieser Wert wird dann durch die Sampling-Rate geteilt, woraus sich der Wert ergibt, der mit der Cutoff-Frequenz multipliziert wird. Wir brauchen keinen Modulations-Multiplizierer, weil "F" logisch ein Kontroll-Signal-Eingang ist, sodass wir die anfängliche Multiplikation auch dann durchführen dürfen, wenn kein Initialisierungs-Event am Eingang "F" anliegt.

Wir führen die Division vor der Multiplikation durch, weil die Division die CPU relativ stark belastet und sich die Sampling-Rate auch nicht so oft ändert. Wenn sich nur die Cutoff-Frequenz ändert, werden keine Events an das Dividierer-Modul geschickt und die Division wird nicht durchgeführt. Dies ist eine der Standard-Optimierungen, die Sie als Designer von REAKTOR-Core-Strukturen vornehmen können.



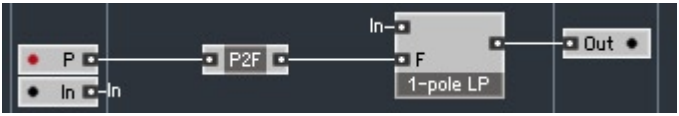
Das Audio-Eingangssignal wird für den Fall zwischengespeichert, dass Events asynchron zur Standard-Audio-Clock eintreffen. Das wäre in einer Core-Cell-Struktur, in der Audio-Eingänge bekanntlich ihre Events immer zur richtigen Zeit senden, nicht notwendig, aber in einem allgemein nutzbaren Core-Macro ist das eine sehr gute Maßnahme.
Zwei Modulations-Multiplizierer verhindern, dass Events am Eingang (die dort, allgemein gesagt, jederzeit eintreffen können), die Berechnung in der Feedback-Schleife triggern. Hier sollte auch klar werden, warum diese Macros "Modulations-Macros" heißen; in diesem Fall wird das aus dem Cutoff gewonnene Signal verwendet, um die Verstärkung im Feedback-Pfad zu modulieren.

Die Zwischenspeicherung (Latching) ist eine Standard-Technik in REAKTOR Core, die dafür sorgt, dass ankommende Events nicht zur unpassenden Zeit Berechnungen triggern. Latching ist auch in Form von Modulations-Macros und anderen Situationen weit verbreitet.

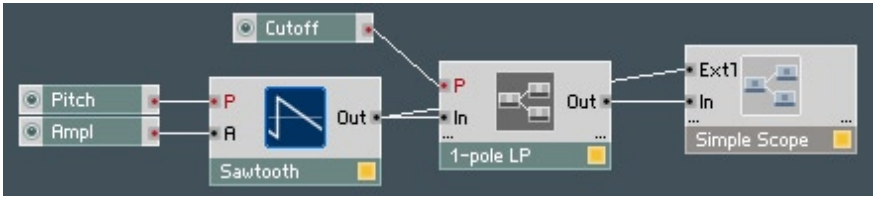
Das Modul Z^-1 speichert der vorigen Ausgangs-Wert und sendet automatisch auf jeden Audio-Tick hin ein Event mit dem vorigen Ausgangs-Wert als Inhalt. Es achtet auch auf mögliche denormale Werte, die andernfalls auftreten könnten. Wenn Sie sich mit DSP auskennen, sollte Ihnen auffallen, dass die Struktur ziemlich ähnlich aussieht wie die herkömmlichen DSP-Filter-Diagramme.

Das Merge-Modul am Ausgang des Addierers stellt sicher, dass der Zustand des Filters nach der Initialisierung immer noch Null ist, auch wenn das Eingangssignal zu diesem Zeitpunkt einen anderen Wert als Null führt.

Vergessen Sie nicht, den Tonhöhe-nach-Frequenz-Konverter *P2F* in die Core Cell einzubauen; wenn Sie dieses Modul platziert haben, sind wir bereit für den Test:



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



Den Cutoff-Regler sollten Sie auf einen Bereich zwischen 0 und 100 oder so ähnlich einstellen. Vorsicht vor zu hohen Cutoff-Werten! Wegen der zunehmenden Abweichung der Filter-Koeffizienten bei hohen Cutoff-Werten wird das Filter bei solchen hohen Werten instabil.

Ein besseres Filter-Design sollte wenigstens die Cutoff-Werte auf den Bereich beschneiden, in dem das Filter stabil arbeitet. Für unseren Fall hätten wir das durch Beschneiden der Koeffizienten b auf den Bereich zwischen 0 und 0,99 oder ähnliche Werte erreicht. Die Technik für das Beschneiden von Werten wird später im Text noch beschrieben.

Das hier sollten Sie nun im Panel sehen:

Bewegen Sie den Cutoff-Regler und beobachten Sie, wie sich die Form des Signals verändert.

6. Bedingte Verarbeitung

6.1. Event-Routing

Die Events in REAKTOR Core müssen nicht immer dieselben festgelegten Pfade benutzen – es gibt auch die Möglichkeit, diese Pfade dynamisch zu verändern. Sie können diese Änderung der Signalwege mit dem Modul *Router* erreichen, das Sie unter *Built-In Module > Flow > Router* finden:

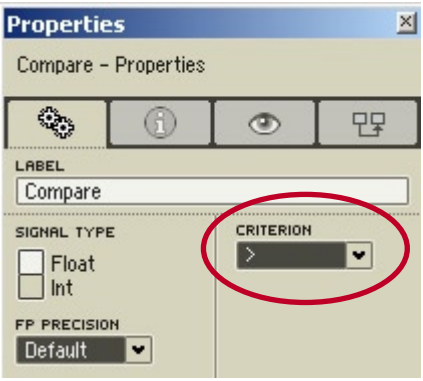
Das Modul *Router* nimmt Events an seinem Signal-Eingang (unten) entgegen und leitet entweder auf seinen Ausgang 1 (oben) oder seinen Ausgang 0 (unten). Auf welchen der Ausgänge die Events geleitet werden, hängt vom aktuellen Zustand des Router-Moduls ab, der über den Eingang *Ctl* (oben) gesteuert wird.
Der Eingang *Ctl* akzeptiert eine Verbindung eines neuen Typs, die weder mit normalen Signalen noch mit OBCs kompatibel ist. Dieser Signal-Typ heißt "Boolean Control" (*BoolCtl*). Das BoolCtl-Signal kann einen von zwei Zuständen annehmen: wahr oder falsch (an oder aus, 1 oder 0). Wenn das BoolCtl-Signal den Zustand "Falsch" hat, werden die Events auf den Ausgang 0 geleitet.

Die BoolCtl-Signale (und andere Kontroll-Signale) unterscheiden sich deutlich von den normalen Signalen in REAKTOR Core: Sie übertragen keine Events und können deshalb von sich aus keine Verarbeitung triggern.

Um einen Router zu kontrollieren, braucht man offensichtlich eine Kontroll-Signal-Quelle. Die gebräuchlichste wäre das Vergleicher-Modul (Compare), das Sie unter *Built In Module > Flow > Compare* finden:

Dieses Modul vergleicht zwei ankommende Signale und gibt das Ergebnis in Form eines BoolCtl-Signals aus. Das obere Eingang wird als links des Vergleichs-Symbols befindlich angenommen, der untere Eingang befindet sich logisch rechts davon. So erzeugt ein Modul mit dem Symbol ">" ein BoolCtl-Signal mit der Wertigkeit "wahr", wenn der Wert am oberen Eingang höher ist als der Wert am unteren Eingang. Sie können das Vergleichs-Kriterium im

Properties-Fenster des Moduls ändern:

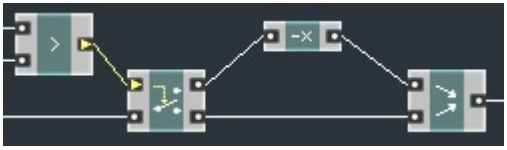


Die verfügbaren Kriterien sind:

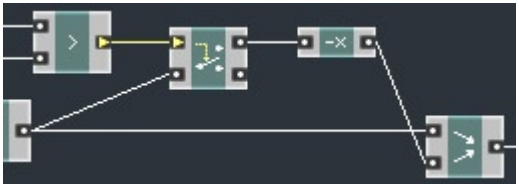
=	gleich
!=	nicht gleich (≠)
<=	kleiner gleich (≤)
<	kleiner
>=	größer gleich (≥)
>	größer

Es ist natürlich möglich, mehrere Router an dasselbe Vergleicher-Modul anzuschließen. Die Router werden in diesem Fall zeitgleich ihre Zustände wechseln.

Das Router-Module spaltet den Event-Pfad in zwei Zeige. Ziemlich oft werden diese Pfade wieder zusammengeführt:



Abhängig vom Ergebnis des Vergleichs wird die obige Struktur das Signal entweder invertieren oder es unberührt lassen. Eine alternative Implementation dieser Struktur ist möglich:



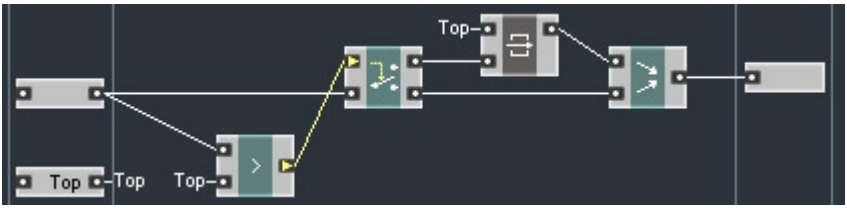
In dieser Version ist der Ausgang 0 des Routers nicht angeschlossen. Deshalb arbeitet der Router wie ein Gate, das die Events nur dann durchlässt, wenn es sich im Zustand "wahr" befindet. Das invertierte Signal kommt dann am zweiten Eingang des Moduls *Merge* an, wobei es den nicht invertierten Wert überschreibt, der immer am ersten Eingang ankommt. Wenn der Router sich im Zustand "falsch" befindet, empfängt der Inverter kein Event und sendet demnach auch kein Signal an den zweiten Eingang des Moduls *Merge*, sodass das Original-Signal unverändert den Ausgang des Moduls *Merge* erreicht.

Die Zweige werden meistens mit einem Merge-Module wieder vereint. Theoretisch würden sich dafür aber auch andere Module eignen, z. B. arithmetische Module wie Addierer, Multiplizierer und so weiter.

Router behandeln das Initialisierungs-Event genau wie jedes andere Event. Deshalb kann man durch die Verwendung von Router-Modulen das Initialisierungs-Event ausfiltern, sodass es in bestimmten Bereichen der Struktur nicht auftauscht.

6.2. Wie Sie einen Signal-Beschneider bauen

Lassen Sie uns nun eine REAKTOR-Core-Macro-Struktur bauen, die das ankommende Audio-Signal oberhalb eines bestimmten Pegels abschneidet:

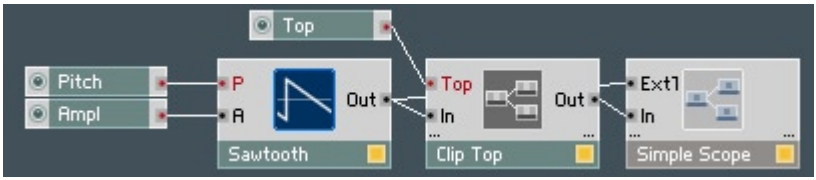


Wenn das Eingangssignal nicht oberhalb der festgelegten Schwelle liegt, wird es auf den Ausgang 0 des Routers geleitet und gelangt durch das Modul *Merge* unverändert zum Ausgang der Struktur. Wenn das Signal die Schwelle überschreitet, wird es auf den Ausgang 1 geleitet, wo es das Latch-Modul triggert, das den Schwellwert an das Merge-Modul sendet. Dasselbe passiert bei der Initialisierung.

Beachten Sie, dass diese Struktur ihre Ausgabe nicht als Reaktion auf Änderungen des Schwellwerts ändert. Stattdessen wird der neue Schwellwert für das nächste am Signal-Eingang ankommende Event

und alle nachfolgenden Events verwendet.. Dieses Verhalten ähnelt in gewisser Weise dem Verhalten der Modulations-Macros, bei denen Veränderungen am Modulator ebenfalls keine Ausgangs-Events erzeugen. similar to a modulation macros behavior, where modulator changes do not result in output events.

Hier sehen Sie eine Test-Struktur für das Beschneider-Modul, das wir unter Verwendung einer Audio-Core-Cell gebaut haben:



Und das hier sollten Sie in der Panel-Ansicht sehen:

Tatsächlich finden Sie eine Anzahl solcher "Modulations"-Beschneider-Macros im Menü *Expert Macro > Clipping*.

6.3. Wie Sie einen einfachen Sägezahn-Oszillator aufbauenr

Wir wollen nun einen einfachen Sägezahn-Oszillator bauen, der eine Sägezahn-Wellenform mit der Amplitude 1 und einer festgelegten Frequenz erzeugt. Wir werden den folgenden Algorithmus verwenden: "Erhöhe den Ausgangssignal-Pegel mit konstanter Geschwindigkeit und lasse ihn in dem Moment, in dem er größer als 1 wird, um 2 abfallen."

Man könnte nun einwenden, dass wir auch den Pegel auf –1 zurücksetzen könnten, anstatt das Signal um 2 abfallen zu lassen, aber das

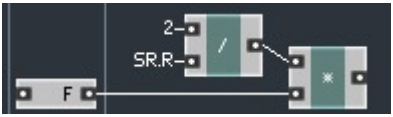
ist im Allgemeinen eine schlechte Idee, weil wir dann die geforderte Oszillator-Frequenz nicht genau beibehalten können.

Die Geschwindigkeit des Anwachsens definiert die Oszillator-Frequenz durch die folgende Gleichung:

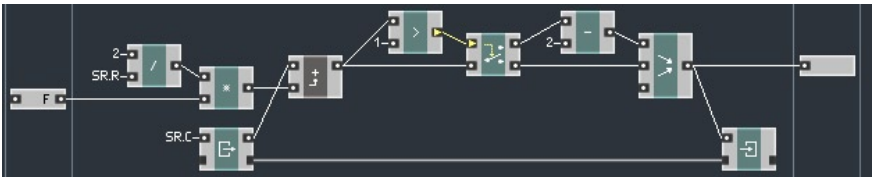
$$d = 2f / f_{SR}$$

wobei d Pegel-Erhöhung mit jedem Audio-Sample ist, f die Oszillator-Frequenz und f_{SR} die Sampling-Rate.

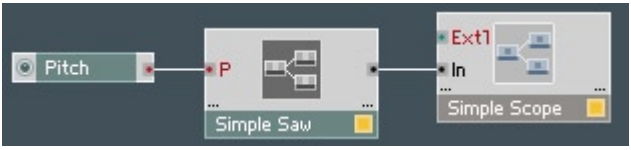
Zuerst bauen wir die Schaltung für die Berechnung der Geschwindigkeit des Anwachsens:



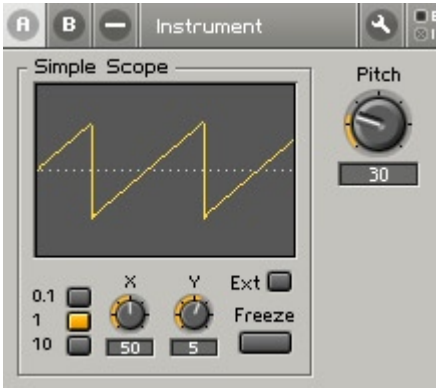
Jetzt brauchen wir eine Erhöher-Schleife (Increment Loop). Es ist also Zeit, ein Modul-Paar aus *Read* und *Write* zu verwenden, genau wie wir es im Akkumulierer gemacht haben:



Das Modul *Read* triggert die Pegel-Erhöhung während jedes Audio-Events. Die Summe der alten Pegel und die Erhöhung werden dann mit dem Schwellwert 1 verglichen und je nach Ergebnis entweder direkt an den Ausgangs-Zwischenspeicher geleitet oder in den umhüllenden Schaltkreis geschickt.
Der dritte Eingang des Moduls *Merge* stellt die Initialisierung des Oszillators mit dem Wert Null sicher. In der Theorie hätte das Modul, das 2 vom Signal-Pegel subtrahiert, eigentlich ein Modulations-Macro sein sollen, aber das soll uns hier nicht weiter stören, denn Merge überschreibt das Ergebnis der Initialisierung sowieso.
Hier ist die vorgeschlagene Test-Struktur (vergessen Sie nicht das Konverter-Modul *P2F* innerhalb der Core Cell):



Und hier ist die Panel-Ansicht:



7. Weitere Signal-Typen

7.1. Fließkomma-Signale

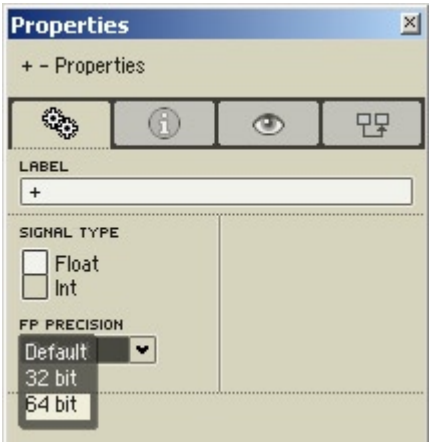
Der Signal-Typ, der bei der digitalen Signalverarbeitung auf modernen Personal Computern am häufigsten vorkommt, ist der Typ Fließkomma (Floating Point, kurz Float). Mit Fließkomma-Zahlen lassen sich Werte bis 10^{38} (im 32-Bit-Modus) oder sogar 10^{308} (im 64-Bit-Modus) ausdrücken. So praktisch das ist, einen Haken gibt es: die begrenzte Genauigkeit. Im 64-Bit-Modus ist die Genauigkeit zwar höher, aber immer noch beschränkt.

> Die Begrenzung der Genauigkeit von Fließkomma-Werten hat technische Gründe: Wenn diese Werte nicht begrenzt würden, wäre eine unendlich große Menge Speicher für ihre Verarbeitung erforderlich. Sie kennen das Problem vielleicht vom Umgang mit Zahlen wie π, die Sie auch nicht in voller Länge auf ein endliches Blatt Papier schreiben können. Selbst wenn Ihnen die komplette Ziffernfolge bekannt wäre (was natürlich bei einer unendlichen Zahl von Stellen unmöglich ist, aber mal angenommen…), geht Ihnen an irgendeiner Stelle das Papier aus. Abgesehen davon würde das Verarbeiten solcher Zahlen auch eine unendlich schnelle CPU erfordern.

Für die Darstellung der Signale und für die Speicherverwaltung, die wir bisher kennengelernt haben, kommen 32-Bit-Fließkomma-Zahlen zum Einsatz. REAKTOR Core bietet Ihnen die Möglichkeit, 64-Bit-Fließkomma-Zahlen zu verwenden, wenn Sie eine höhere Genauigkeit oder einen größeren Wertebereich benötigen (wobei man sich kaum vorstellen kann, dass der Bereich zwischen 10^{-38} und 10^{38} nicht ausreichen soll).

Standardmäßig werden alle Berechnungen in REAKTOR Core mit 32-Bit-Fließkomma-Genauigkeit durchgeführt. Das heißt nicht zwingend, dass die Signale wirklich als 32-Bit-Floats berechnet werden, sondern dass mindestens 32-Bit-Fließkomma-Zahlen bei der Verarbeitung verwendet werden (obwohl mitunter auch 64-Bit-Floats für Zwischenergebnisse verwendet werden).

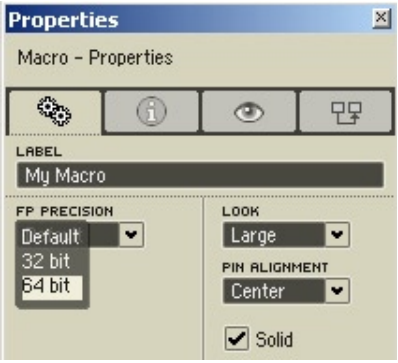
Sie können die Fließkomma-Genauigkeit sowohl für einzelne Module als auch für ganze Macros festlegen. Um die Fließkomma-Genauigkeit für ein einzelnes Modul zu ändern, öffnen Sie das Properties-Fenster des Moduls und wählen Sie einen Eintrag aus dem Menü *FP Precision* (Floating Point Precision):



default	bedeutet, dass die Default-Genauigkeit der aktuellen Struktur verwendet wird, welche das auch immer ist.
32 bit	bedeutet, dass mindestens eine 32-Bit-Genauigkeit verwendet wird.
64 bit	bedeutet, dass mindestens eine 64-Bit-Genauigkeit verwendet wird.

Das Verändern der Genauigkeit für ein Modul führt dazu, dass die Verarbeitung innerhalb dieses Moduls fortan mit der angegebenen Genauigkeit durchgeführt wird und dass der Ausgangs-Wert ebenfalls unter Verwendung des angegebenen Genauigkeit erzeugt wird.

Die Standard-Genauigkeit für ganze Strukturen ändern Sie, indem Sie einen Rechts-Klick in den Hintergrund ausführen und aus dem Menü den Eintrag Owner Properties auswählen, der die Eigenschaften des Eigentümer-Moduls aufruft:



Die hier getroffene Einstellung legt die Genauigkeit für alle Module einschließlich Macros innerhalb der betreffenden Struktur fest, sofern für die Module keine abweichende Genauigkeit spezifiziert oder (im Fall von Macros) keine Standard-Genauigkeit für die eingeschlossenen Strukturen festgelegt ist.

Die normalen Fließkomma-Signale mit 32 und 64 Bit Genauigkeit sind voll miteinander kompatibel und können entsprechend beliebig untereinander verbunden werden. OBC-Signale mit unterschiedlicher Genauigkeit sind nicht miteinander kompatibel (weil es keinen Speicherinhalt geben kann, der gleichzeitig in 32 Bit und in 64 Bit vorliegt). Außerdem werden im Fall von OBC-Signalen die Einstellungen “Default”, “32 bit” und “64 bit” als unterschiedlich und deshalb inkompatibel behandelt, weil die effektive Standard-Genauigkeit durch Ändern der entsprechenden Eigenschaft für eins der übergeordneten Macros verändert werden kann.

Die Eingangs- und Ausgangs-Module der Top-Level-Strukturen von Core Cells senden und empfangen immer im 32-Bit-Fließkomma-Format, weil dieser Signal-Typ für die Event- und Audio-Verbindungen auf REAKTORs Primary Level verwendet wird.

7.2. Integer-Signale

Es gibt noch einen anderen Signal-Typ, der von modernen CPUs umfassend unterstützt wird, und dieser Typ spielt in der digitalen Welt eine noch größere Rolle als Fließkomma-Zahlen. Es handelt sich dabei um den Typ Integer (Ganzzahl). Integer-Zahlen werden mit unendlicher Genauigkeit dargestellt und verarbeitet.

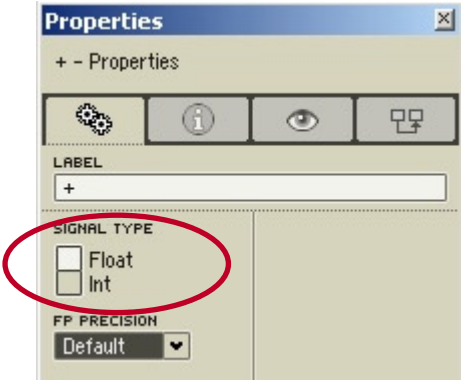
Obwohl die Genauigkeit von Integer-Zahlen unendlich ist, ist der Darstellungsbereich auch für Werte dieses Typs begrenzt. Die Obergrenze für 32-Bit-Integer-Werte liegt oberhalb von 10^9.

> Unendliche Genauigkeit für die Speicherung und Verarbeitung von Integer-Werten ist möglich, weil Werte dieses Typs keine Dezimalstellen nach dem Komma besitzen, sodass man sie mit einer endlichen Anzahl von Stellen beschreiben kann. Schreiben Sie zum Beispiel die Anzahl der Sekunden einer Stunde auf: 3, 6, 0, 0 – fertig. So leicht ist das. Wenn Sie dagegen die Zahl π aufschreiben wollen, schaffen Sie das niemals in voller Länge – 3, 1, 4, 1, Stopp. Nicht ganz vollständig? Okay, lassen Sie uns noch ein paar Stellen aufschreiben: 5, 9, Stopp. Immer noch nicht vollständig. Dieses Spiel können Sie beliebig fortsetzen.. Eine Integer-Zahl dagegen können Sie komplett und präzise aufschreiben: 3600, das reicht.

Während der Typ Fließkomma eine natürliche Wahl für Werte ist, die sich fortlaufend verändern (beispielsweise Audio-Signale), sind Integer-Werte für sich diskret ändernde Werte (z. B. Zähler) die besser passende Wahl.
Viele der Module in REAKTOR Core können Sie in den Integer-Modus schalten, in dem die Module Integer-Signale am Eingang erwarten, diese als Integer (das heißt mit unendlicher Genauigkeit) verarbeiten und am Ausgang das Ergebnis als Integer-Wert bereitstellen. Zu diesen Modulen gehören beispielsweise arithmetische Module wie Addierer, Subtrahierer und Multiplizierer. Es gibt sogar einige Module, die sich nur mit Integer-Werten verwenden lassen.

> Die minimale Wortlänge für Integer-Werte in REAKTOR Core beträgt 32 Bit.

Zwischen den Typen Float und Integer umschalten können Sie (sofern das Modul dies unterstützt) in der Eigenschaft Signal Type im Properties-Fenster des Moduls:



Ein Modul, das in den Integer-Modus geschaltet ist, wird die ankommenden Werte als Integer-Zahlen verarbeiten und Integer-Ausgangs-Werte erzeugen. Dass sich ein Modul im Integer-Modus befindet, erkennen Sie daran, dass seine Signal-Ein- und Ausgänge anders aussehen:

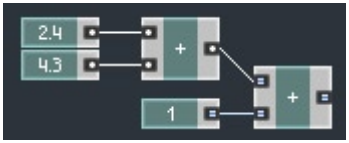
Es gibt keinen Standard-Signal-Typ für Macros. Der Grund dafür ist, dass Sie Strukturen, die Integer-Werte verarbeiten sollen, normalerweise nicht genauso aufbauen würden wie Strukturen, die zur Verarbeitung von Fließkomma-Zahlen gedacht sind, und umgekehrt – das kommt eigentlich nur bei einigen sehr einfachen Strukturen vor.

Integer-Signale können Sie frei mit Fließkomma-Signalen verbinden, allerdings bei solchen Verbindungen zwischen unterschiedlichen Typen eine Signal-Konvertierung durchgeführt, die CPU-Last erzeugt. Diese Konvertierung macht sich auf den zur Zeit der Erstellung dieses Handbuchs aktuellen PCs etwas, auf aktuellen Macs deutlich stärker bemerkbar. OBC-Anschlüsse unterschiedlichen Typs sind überhaupt nicht miteinander kompatibel.

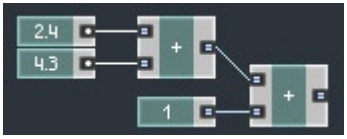
Durch die Signal-Konvertierung können Informationen verloren gehen. Besonders große Integer-Zahlen können nicht exakt durch Fließkomma-Zahlen ausgedrückt werden; ebenso lassen sich Fließkomma-Werte nicht genau als Integer-Zahlen wiedergeben. Große Fließkomma-Werte (größer als der größte darstellbare Integer-Wert) lassen sich gar nicht im Integer-Format darstellen, weshalb die Konvertierung in solchen Fällen undefiniert ist. Während der Konvertierung von Float nach Integer

werden die Werte *näherungsweise auf den nächsten Integer-Wert* gerundet. “Näherungsweise” bedeutet, dass das Ergebnis einer Rundung des Float-Werts 0,5 entweder 0 oder 1 sein kann, wobei Sie ziemlich sicher sein können, dass 0,49 auf 0 abgerundet, 0,51 dagegen auf 1 aufgerundet wird.

Es ist wichtig, sich vor Augen zu führen, dass das Umschalten des Verarbeitungs-Modus‘ auf Integer nicht zu denselben Ergebnissen führt wie die Konvertierung des Fließkomma-Ergebnisses derselben Operation. Lassen Sie uns zur Verdeutlichung ein Beispiel ansehen. Hier addieren wir zwei Fließkomma-Werte, nämlich 2,4 und 4,3. Das Ergebnis ist natürlich 6,7, was nach der Konvertierung in Integer den Wert 7 ergibt. Die Ausgabe der folgenden Struktur lautet also 8:



Wenn wir nun den Modus des ersten Addierers auf Integer umschalten, werden statt der Werte 2,4 und 4,3 die gerundeten Werte 2 und 4 addiert, wodurch sich als Ergebnis 6 ergibt. Die Ausgabe der Struktur lautet also 7:



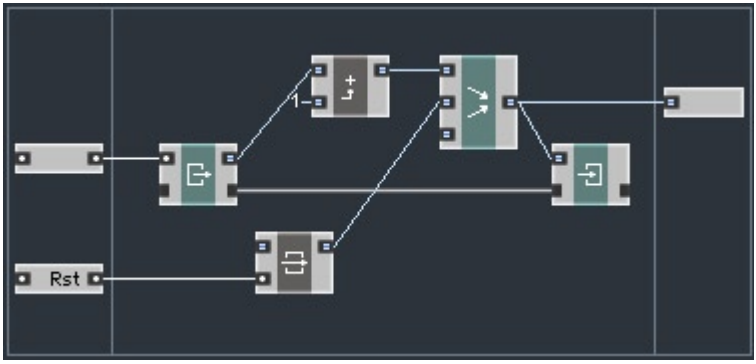
Clock-Eingänge ignorieren die ankommenden Werte vollständig, deshalb sind sie normalerweise immer vom Typ Float. Die Signal-Konvertierung wird nicht für Signale durchgeführt, die nur als Clock dienen:

Hier befindet sich der Clock-Eingang des Moduls Read immer noch im Float-Modus, obwohl das Modul in den Integer-Modus geschaltet wurde (die OBC-Ports sehen immer gleich aus, unabhängig davon, ob sie vom Typ Float oder Integer sind).

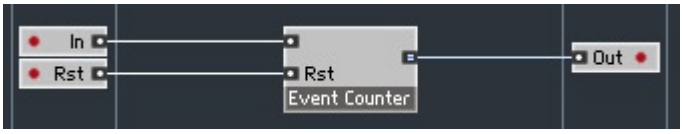
Integer-Feedback wird automatisch auf dieselbe Art aufgelöst wie Fließkomma-Feedback – indem ein Integer-Mode-Modul des Typs Z^-1 eingesetzt wird (wobei hier natürlich keine Denormalen-Unterdrückung notwendig ist).

7.3. Wie Sie einen Event-Zähler bauen

Lassen Sie uns ein Event-Zähler-Modul bauen. Die Funktion dieses Moduls wird der des Event-Akkumulierers sein, aber anstatt die Werte der Events zu summieren, werden wir die Events diesmal nur zählen. Der Signal-Typ Integer scheint eine logische Wahl für den Zählvorgang zu sein:

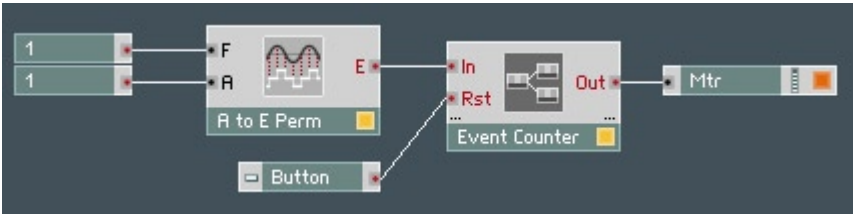


Ausgangs-Modul und alle eingebauten Module befinden sich im Integer-Modus. Das Macro *ILatch* setzt anstelle eines normalen Latch-Macros die Schaltung zurück. Es macht genau dasselbe wie ein Latch (und ist auch im selben Menü zu finden), arbeitet aber mit Integer-Signalen. Außerdem kommt ein Integer-Modulations-Macro zum Einsatz (das Sie unter *Expert Macro > Modulation > Integer* finden). Die beiden Eingänge müssen Sie nicht in den Integer-Modus schalten, sie liefern nur Clock-Signale
Sehen Sie sich nun die Struktur unserer Event-Core-Cell an, die dieses Macro enthält:



Wie Sie sehen, befindet sich der Ausgang dieses Moduls nicht im Integer-Modus (und lässt sich auch gar nicht in diesen Modus versetzen). Das kommt

daher, dass die Core Cell nach außen ein Primary-Level-Modul ist und ein normales Primary-Level-Event vom Typ Float liefern muss. Hier sehen Sie eine Test-Struktur für unser Zähler-Modul:



Und das Panel wird voraussichtlich so aussehen:

7.4. Wie Sie einen Flanken-Zähler bauen

Nun lernen Sie eine Technik für den *Vorzeichen-Vergleich* kennen, die Sie manchmal beim Aufbauen von REAKTOR-Core-Strukturen brauchen werden. "Vorzeichen-Vergleich" beschreibt eine besondere Art des Vergleichens von zwei Zahlen, bei der Sie den Wert der Zahlen ignorieren und nur die jeweiligen Vorzeichen beachten (Plus oder Minus). Natürlich ist auch hier Plus höherwertig als Minus. Hier ist ein Beispiel:

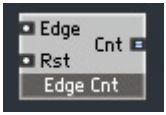
3.1 ist vorzeichengrößer als -1.4
2.1 ist vorzeichengleich mit 5.0
4.5 ist vorzeichengleich mit -2.9

Beachten Sie, dass das Vorzeichen von Null undefiniert ist. Das bedeutet, dass das Ergebnis jedes Vorzeichen-Vergleichs, an dem der Wert Null beteiligt ist, beliebig sein kann.

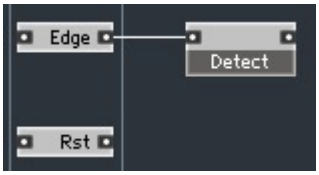
Natürlich hätten Sie den Vorzeichen-Vergleich auch unter Verwendung einiger Vergleicher-Module und einiger Router implementieren können, aber die Lösung hier ist viel effizienter. Sie können den Vorzeichen-Vergleich in REAKTOR-Core-Strukturen nämlich mit dem Modul Compare Sign durchführen (*Built In Module > Flow > Compare Sign*):

Das Modul erzeugt an seinem Ausgang ein Signal des Typs *BoolCtl*, sodass Sie es mit einem Router verbinden können.

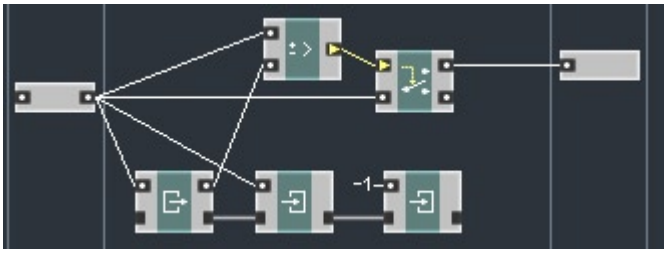
Ein mögliches Einsatzgebiet für ein solches Modul wäre, die steigenden Flanken eines ankommenden Signals zu erkennen. Einen Zähler für diese steigenden Flanken werden wir nun als REAKTOR-Core-Macro aufbauen:



Achten Sie darauf, dass sich der Ausgang im Integer-Modus befindet, weil die Zählung in Ganzzahlen stattfindet. Zuerst brauchen wir ein Flanken-Detektor-Macro, das die erkannten Flanken in ein Event konvertiert:



So könnte unser Detektor-Macro aussehen:

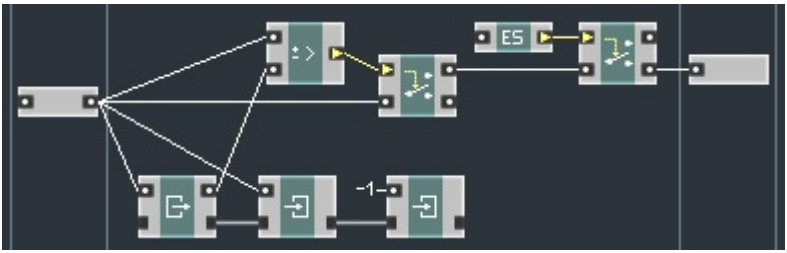


Die OBC-Kette unten enthält den vorigen Wert des Eingangssignals. Wie Sie sehen können, wird der neue Wert gespeichert, nachdem der alte Wert gelesen wurde. Das letzte Write-Modul in der Kette übernimmt die Initialisierung des vorigen Werte-Speichers. Wir initialisieren den Speicher mit -1, sodass der erste positive Wert als steigende Flanke gezählt wird.

Ein Write-Modul am Ende der OBC-Kette ist eine andere Möglichkeit (als Merge), den Speicher zu initialisieren, Dabei muss das Write-Modul als letztes Glied der Kette angeordnet sein, um die von den stromaufwärts liegenden Write-Modulen gespeicherten Ergebnisse überschreiben zu können.

Der vom Modul Sign Comparison kontrollierte Router arbeitet als Event-Gate und lässt nur die Events durch, bei denen ein Vorzeichenwechsel von negativ in positiv erfolgt.
IEs ist nicht eindeutig, ob ein solches Modul während der Initialisierung ein

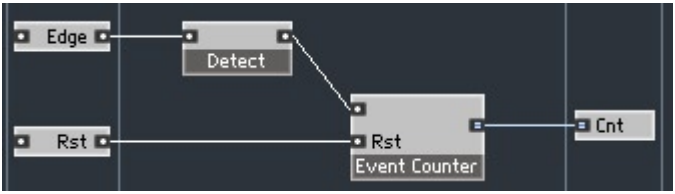
Event sendet oder nicht. Das liegt daran, dass der Speicher zum Zeitpunkt der Verarbeitung des Initialisierungs-Events immer noch den Wert Null hat und das Vorzeichen von Null undefiniert ist. Wir können die Struktur modifizieren, um das Senden eines Events während der Initialisierung zu verhindern:



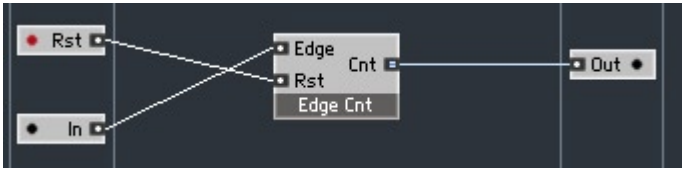
Das Modul *ES Ctl* ist ein "Event-empfindlicher Regler". Das von diesem Modul erzeugte Kontroll-Signal ist nur dann "wahr", wenn am Eingang des Moduls ein ankommendes Event vorhanden ist. Weil dieser Eingang in der obigen Struktur nicht angeschlossen ist und deshalb eine Null-Konstante empfängt, nimmt sein Signal nur zum Zeitpunkt der Initialisierung den Wert "wahr" an. Der zweite Router wird also alle während der Initialisierung auftretenden Events abblocken und alle anderen durchlassen.

Beachten Sie, dass wir es hier mit einem Modul zu tun haben, dessen Ausgang während der Initialisierung kein Event sendet.

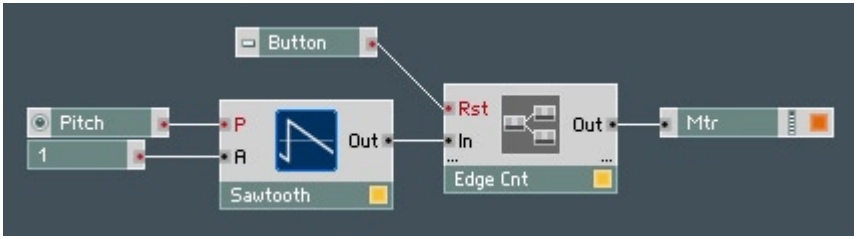
Nun haben wir ein Detektor-Modul, das wir mit der Zähler-Schaltung verbinden können, die wir schon aufgebaut haben:



Die Funktion der obigen Schaltung sollte einleuchten. Das Modul *Detect* sendet jedes Mal ein Event, wenn es eine steigende Flanke erkennt; diese Events zählt das Modul *Evt Cnt.*
Um unsere Schaltung zu testen, fügen wir dieses Macro in eine Audio-Core-Cell ein und zählen die steigenden Flanken einer Sägezahn-Wellenform. Die interne Struktur der Core Cell wird folgendermaßen aussehen:



Und hier die Primary-Level-Test-Struktur:



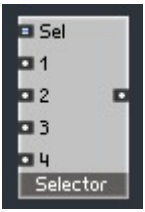
Vergessen Sie nicht, die Eigenschaften für das Modul *Mtr* (Meter) wie in den vorangegangenen Beispielen einzustellen.
Das hier sollten Sie nun in der Panel-Ansicht sehen:

Die Geschwindigkeit der Werteänderungen im Anzeigefeld korrespondiert mit der Frequenz des Oszillators, die Sie mit dem Regler *Pitch* einstellen können. Bei einem Pitch-Wert von Null beträgt die Oszillator-Frequenz ungefähr 8 Hz, sodass sich der Wert im Anzeigefeld mit einer Rate von acht Schritten pro Sekunde ändern sollte.

8. Arrays

8.1. Einführung in das Thema “Arrays”

Lassen Sie uns annehmen, Sie wollen einen Audio-Signal-Wahlschalter bauen, der abhängig von dem am Kontroll-Eingang anliegenden Wert das Signal von einem der vier Audio-Signal-Eingänge entgegennimmt:



Diese Funktion ließe sich mit Router-Module umsetzen, aber es gibt auch noch eine andere Möglichkeit. Wir verwenden nämlich lieber die in REAKTOR Core verfügbaren *Arrays*.
Ein *eindimensionales Array* ist eine *geordnete Sammlung* von Daten-Einträgen *desselben Typs*, die sich über ihre Position in dieser Ordnung (oder diesem *Index*) adressieren lassen. Nehmen wir zum Beispiel die folgende Gruppe von Fließkomma-Zahlen:

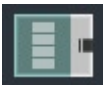
5.2 16.1 -24.0 11.9 -0.5

In REAKTOR Core sind die Array-Element-Indizes Null-basiert; das bedeutet, dass das erste Element des Arrays einen Index von 0 hat. Deshalb ist für unsere Zahlengruppe ein Index von 0 dem Wert 5,2 zugeordnet, ein Index von 1 entspricht 16,1,der Index 2 adressiert –24,0, der Index 3 ist dem Wert 11,9 zugewiesen und Index 4 verweist auf den Wert -0,5.

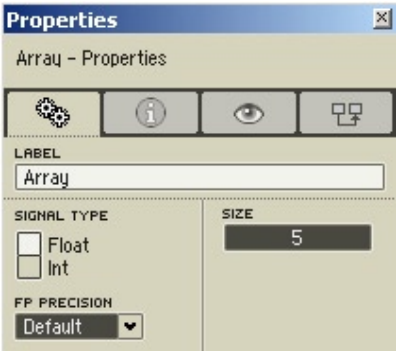
Hier sehen Sie eine tabellarische Darstellung des beschriebenen Arrays:

Index	0	1	2	3	4
Value	5.2	16.1	-24.0	11.9	0.5

Arrays erzeugen Sie in REAKTOR Core durch Verwendung der *Array*-Module (*Built In Module > Memory > Array*):



Ein Array-Modul hat einen einzelnen Ausgang vom Typ *Array OBC*. Die Größe des Arrays (die Anzahl der Elemente) und den Typ der in dem Array aufbewahrten Daten können Sie im Properties-Fenster des Array-Moduls angeben:



Zum Beispiel müssen wir für die oben abgebildete Tabelle mit fünf Elementen im Feld *Size* eine Größe von 5 eintragen. Da es sich bei den Elementen unseres Arrays um Fließkomma-Zahlen handelt, müssen wir außerdem den Signal-Typ Float auswählen

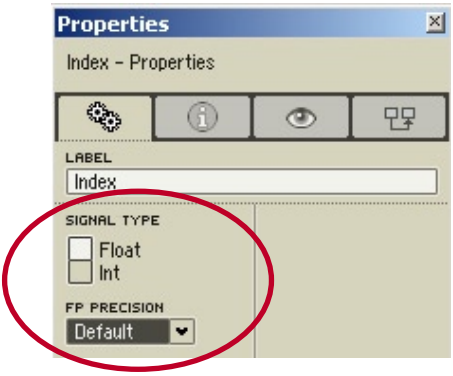
---

Weil Array-Indizes in REAKTOR Core Null-basiert sind, umfasst der Index-Bereich für ein Array der Größe 5 von 0 bis 4 (wie Sie auch in der oben abgebildeten Tabelle sehen können).

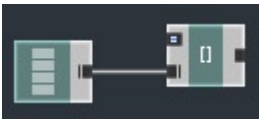
Array-OBC-Signale, die mit Elementen unterschiedlichen Daten-Typs korrespondieren, sind natürlich nicht miteinander kompatibel.

Um ein Array-Element zu adressieren, müssen Sie einen Index anlegen. Dazu stellt REAKTOR Core das Modul Index (*Built In Module > Memory > Index*) zur Verfügung:

Der Master-OBC-Eingang des Index-Moduls (unten) sollte mit dem Slave-Ausgang eines Array-Moduls verbunden sein. Der Verbindungs-Typ des Master-Eingangs sollte mit dem Array-Typ übereinstimmen; Sie können ihn im Properties-Fenster des Moduls Index einstellen:

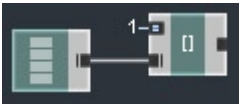


Und nun die Verbindung:



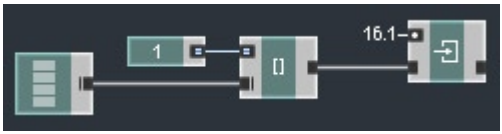
Der obere Eingang des Index-Moduls ist immer vom Typ Integer und nimmt den Index-Wert entgegen. Hier sehen Sie, wie wir das Array-Element mit dem Index von 1 adressieren:

Beachten Sie, dass sich das Konstanten-Modul ebenfalls im Integer-Modus befindet (was Sie am Aussehen des Ausgangs-Ports erkennen). Das ist zwar nicht unbedingt notwendig, weil andernfalls eine automatische Konvertierung nach Integer durchgeführt werden würde, aber es sieht besser aus.
Alternativ hätten wir auch eine QuickConst verwenden können:



Der Ausgang des Index-Moduls hat den Typ *Latch OBC*. Das bedeutet, dass Sie Module der Typen *Read* und *Write* (und davon sogar mehrere) mit diesem Ausgang verbinden können. Dabei müssen Sie natürlich darauf achten, dass die Read- und Write-Module auf denselben Daten-Typ gesetzt sind wie die Module *Array* und *Index*.

Hier sehen Sie, wie das Array-Element mit dem Index von 1 auf den Wert 16,1 initialisiert wird:



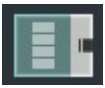
Wenn ein außerhalb des definierten Bereichs liegender Index an das Index-Modul gesendet wird, ist das Ergebnis des Zugriffs auf das Array undefiniert. Die Struktur wird nicht abstürzen, aber es ist in diesem Fall unklar, auf welches Element des Arrays in diesem Fall zugegriffen wird und ob überhaupt ein Zugriff stattfindet. Das bedeutet, dass Sie im Zweifelsfall den Wertebereich des Index' unter Verwendung von Router-Modulen oder Macros aus der Library beschneiden sollten.

8.2. Wie Sie einen Audio-Signal-Wahlschalter bauen

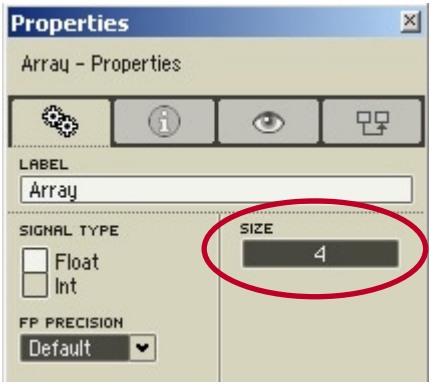
Lassen Sie uns zum Aufbau des Audio-Signal-Wahlschalters zurückkehren:

Hier sehen Sie eine leere interne Struktur für dieses Modul:

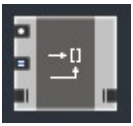
Wir werden ein Array mit vier Fließkomma-Elementen zum Speichern unserer Audio-Signale verwenden:



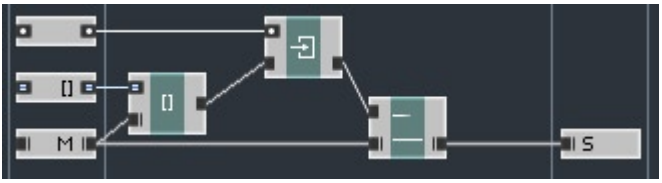
Hier ist das Properties-Fenster des Array-Moduls:



Um die Eingangs-Werte in das Array zu schreiben, sollten wir das Standard-Macro *Write []* (*Expert Macro > Memory > Write []*) verwenden:



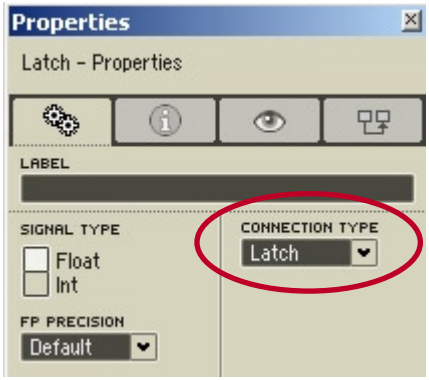
Dieses Macro besitzt ein internes Index-Modul und ein Write-Modul, das mit einem festgelegten Index in das Array-Element schreibt:



Der obere Eingang nimmt natürlich den Wert entgegen, der geschrieben werden soll. Der Eingang "[]" empfängt den Index, an dem der Schreibvorgang stattfinden soll. Der Eingang "M" stellt via OBC die Verbindung zu einem Float-Array mit Standard-Genauigkeit her, der Ausgang "S" ist eine "durchgehende" Verbindung, ähnlich wie bei den anderen OBC-Modulen, etwa *Read* und *Write.*
Der Eingang "M" und der Ausgang "S" gehören zu einer anderen Sorte von Macro-Anschlüssen, die sich von der unterscheidet, die wir bisher verwendet

haben. Solche Ports können Sie über den Eintrag “Latch” aus dem Port-Menü erzeugen (der dritte Eintrag erzeugt einen Macro-Port vom Typ *BoolCtl*):

Die Latch-Ports können Sie für Latch-OBC-Verbindungen (zwischen Modulen der Typen Read und Write) oder für Array-OBC-Verbindungen verwenden. Diese Auswahl können Sie im Properties-Fenster des Ports treffen:



Indem Sie die Verbindungs-Typ auf “Latch” oder “Array” setzen, definieren Sie den OBC-Modus entsprechend. Für die Ports des Macros Write [] müssen Sie hier offensichtlich “Array” einstellen.
Das Modul mit den beiden horizontalen Linien heißt *R/W Order (Built In Module > Memory > R/W Order):*

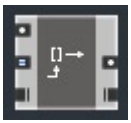
Es lässt nur die Verbindung an seinem Master-Eingang (unten) zu seinem

Slave-Ausgang durch, weiter tut es nichts. Der obere Eingang hat absolut keine Wirkung, weil aber an diesem Eingang eine Verbindung anliegt, beeinflusst er doch die Verarbeitungsreihenfolge der Module. Deshalb *wird alles, was an den Ausgang "S" des Macros angeschlossen ist, nach dem Write-Modul verarbeitet*, was nicht der Fall wäre, wenn das Modul *R/W Order* in dieser Struktur fehlen würde.

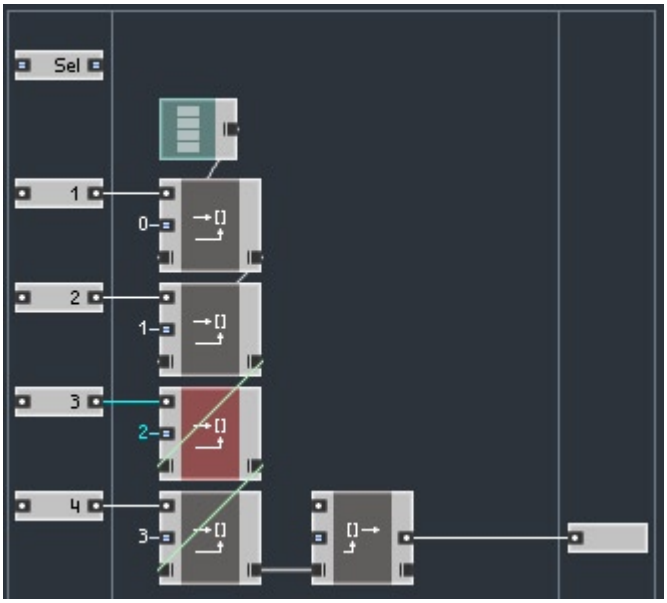
Wenn das Modul *R/W Order* fehlen würde, wäre die Funktion des Macros *Write []* nicht sehr zuverlässig oder nachvollziehbar, weil Sie als Anwender natürlich erwarten, dass alles an den Ausgang "S" des Macros *Write []* Angeschlossene nach diesem Macro verarbeitet wird. Im Allgemeinen entsteht dieses Problem nur bei Verbindungen des Typs OBC; setzen Sie in diesem Fall R/W-Order-Module an den entsprechenden Stellen in den Macros, die Sie entwerfen, ein.

Das Modul *R/W Order* besitzt, ähnlich wie die OBC-Ports, eine Eigenschaft namens *Connection Type*. Bei diesem Modul kontrolliert die Eigenschaft *Connection Type* nur den Typ der Ports "M" und "S", der "Sidechain"-Eingang befindet sich immer im Modus "Latch". Weitere Details finden Sie im Anhang in der Beschreibung des Moduls *R/W Order.*
Lassen Sie uns nun eine Schaltung aufbauen, die die Eingangssignale in das Array schreibt:

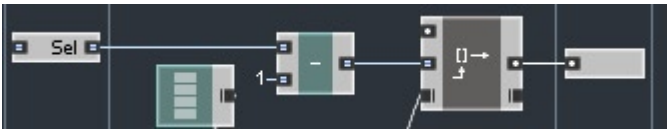
Die vier Module des Typs *Write []* sorgen dafür, dass die ankommenden Audio-Werte in das Array geschrieben werden. Nun brauchen wir einen Schaltkreis, um einen der vier Werte auszulesen. Dafür schlagen wir das Macro *Read []* (zu finden unter *Expert Macro > Memory > Read []*) vor:



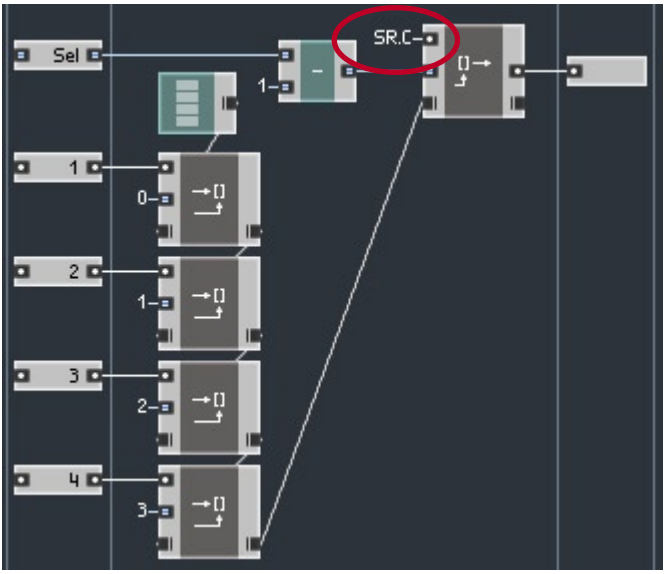
Dieses Macro liest ein Array-Element aus, dessen Index über den Integer-Eingang in der Mitte bestimmt wird. Der obere Eingang ist der Clock-Eingang für den Lese-Vorgang – er sendet als Reaktion auf ein ankommendes Event den gelesenen Wert an den oberen Ausgang des Moduls. Die unteren Ports sind natürlich Master- und Slave-Array-Verbindungen.
Aber womit verbinden wir nun den Master-Eingang? Anscheinend können wir ihn nicht direkt mit dem Array-Modul verbinden, weil der Lese-Vorgang nach allen Schreib-Vorgängen erfolgen muss (weil sonst eine Verzögerung von einem Sample entstehen kann oder auch nicht, alles in allem ist das jedenfalls ziemlich unzuverlässig). Wir können den Master-Eingag aber auch nicht direkt mit einem der Module des Typs *Write []* verbinden, weil das unser Problem nicht lösen würde. Daher schlagen wir vor, dass Sie die Write-[]-Module in Reihe schalten, sodass Sie das Modul *Read []* mit dem Ausgang des letzten Moduls vom Typ *Write []* verbinden können.



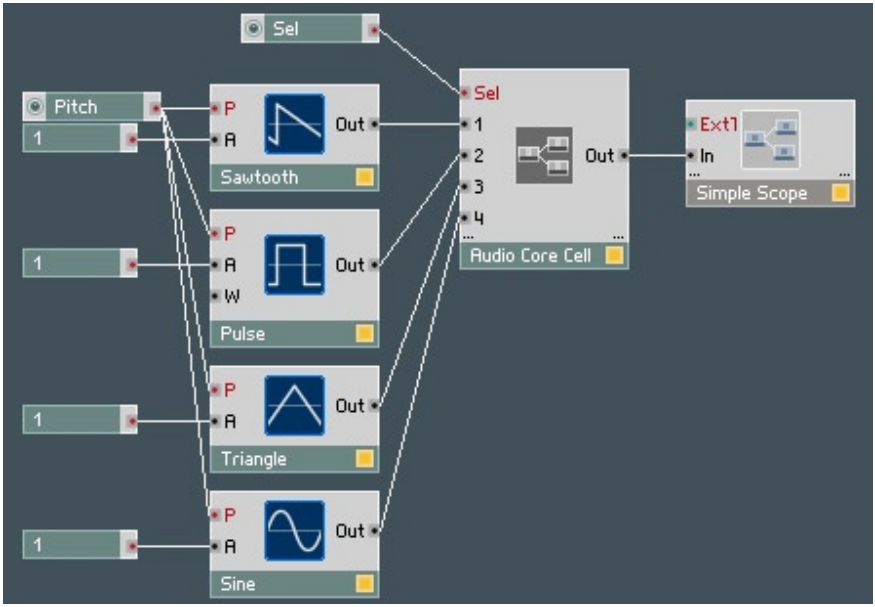
Und wo schließen wir den Index-Eingang des Moduls *Read []* an? Wenn wir wollen, dass unser Auswahl-Wert im Bereich zwischen 1 und 4 liegt, müssen wir den Wert des Eingangs “Sel” um 1 verringern. Beachten Sie, dass wir hier eine Integer-Subtraktion durchführen (weil “Sel” nur ein Kontroll-Eingang ist, brauchen wir hier wirklich kein Modulations-Macro):



Zu guter Letzt müssen wir noch das Read-Modul durch die Sampling-Rate-Clock takten lassen (weil das hier ja ein Audio-Signal-Wahlschalter sein soll):

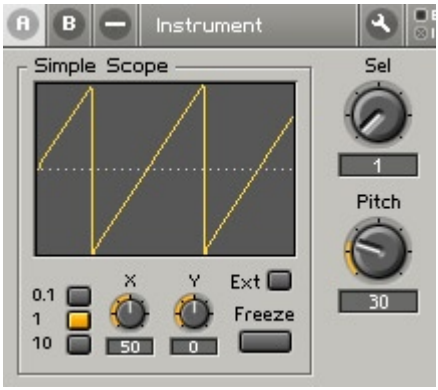


Normalerweise hätten wir auch auf das Beschneiden des Eingangs-Werts am Eingang “Sel” auf den korrekten Bereich achten sollen, aber der Einfachheit halber verzichten wir hier darauf.
Hier sehen Sie eine geeignete Test-Struktur (das Macro ist in eine Audio-Core-Cell verpackt):



Der Drehregler “Sel” ist so eingestellt, dass er zwischen den vier Werten von 1 bis 4 umschaltet.

Nun schalten Sie in die Panel-Ansicht um und sehen Sie sich an, wie sich die Wellenform abhängig von der Stellung des Drehreglers ändert:

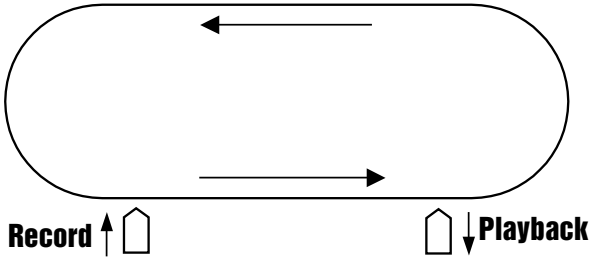


8.3. Wie Sie ein Delay aufbauen

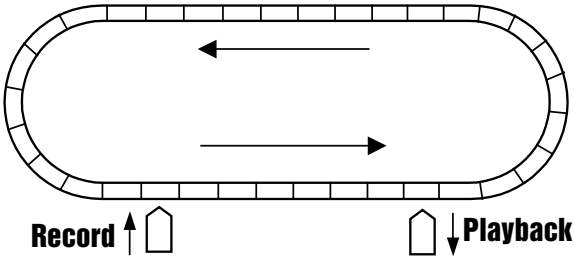
Nun, da wir etwas Erfahrung mit Arrays haben, lassen Sie uns ein einfaches Audio-Delay-Macro aufbauen. Das Modul sollte wie folgt aussehen:

Noch besser ist, wenn es so aussieht (wofür wir die Eigenschaft *Port Alignment* des Macros in "top" ändern müssen):

Der Eingang "T" nimmt die Delay-Zeit in Sekunden entgegen.
Wenn Sie sich einmal ein analoges Bandecho-Gerät öffnen, sehen Sie darin eine Tonbandschleife in Kombination mit einem Aufnahmekopf (Record) und einem Wiedergabekopf (Playback). Genau genommen gibt es auch noch einen Löschkopf, aber um unser Beispiel einfach zu halten, nehmen wir an, dass der Aufnahmekopf sowohl für das Aufnehmen als auch für das Löschen zuständig ist.

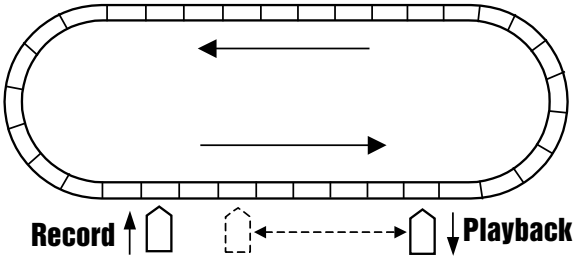


Wenn wir nun diese Konstruktion in digitaler Form simulieren wollen, brauchen wir eine Art digitaler Bandschleife. Wegen der diskreten Natur der digitalen Welt nimmt die digitale "Bandschleife" eine endliche Zahl von Audio-Samples auf. Diese Samples werden mit der Audio-Sampling-Rate aufgenommen und wiedergegeben. In dem folgenden Bild nimmt jedes Segment der Bandschleife ein Audio-Sample auf:



Die nächstliegende Wahl für eine digitale Bandsschleife ist ein Array, wobei die Größe des Arrays der Gesamtanzahl der in der Bandschleife aufgenommenen Samples entsprechen muss.

Bei einem analogen Bandecho hängt die Delay-Zeit von zwei Parametern ab: von dem (unveränderlichen) Abstand zwischen dem Aufnahme- und dem Wiedergabekopf und von der (veränderlichen) Bandgeschwindigkeit. Diese Lösung hat technische Gründe – es ist viel einfacher, die Laufgeschwindigkeit des Tonbands zu verändern, als den Abstand zwischen den Köpfen. Im Fall unserer digitalen Simulation ist es allerdings umgekehrt, weil eine Variation der Bandgeschwindigkeit einer Sampling-Raten-Konvertierung zwischen dem "digitalen Tonband" und dem Ausgang entsprechen würde, während wir den Abstand zwischen den "Köpfen" relativ einfach verändern können – also machen wir das:



Es gibt noch einen weiteren Unterschied zur analogen Welt: In der analogen Welt bewegt sich die Bandschleife. Wenn wir unser digitales Tonband bewegen wollen würden, müssten wir *mit jedem Audio-Takt-Signal* alle Array-Elemente auf ihre Nachbarpositionen kopieren, was ziemlich aufwendig wäre. Wir werden also stattdessen die Köpfe bewegen.

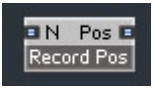
Aus dem oben Gesagten können wir ableiten, dass wir für unsere Bandecho-Simulation folgende Bauteile brauchen:

array	– um unsere “digitale Bandschleife” zu simulieren
write index	– dies ist unser Aufnahmekopf
read index	– dies ist unser Wiedergabekopf

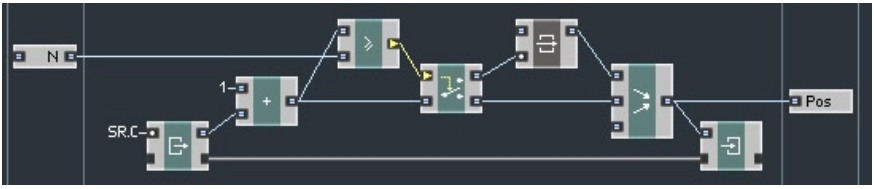
Schreib- und Lese-Index werden sich Sample für Sample durch das Array bewegen. In dem Moment, in dem einer von beiden das Ende des Arrays erreicht, soll der betreffende Index auf den Anfang des Arrays zurückgesetzt werden (was dem Zusammenkleben der offenen Enden eines Tonband-Streifens zu einer Schleife entspricht). Der Versatz zwischen der Schreib- und der Lese-Position ist von der Delay-Zeit (in Samples) abhängig.

Diese Technik ist beim Programmieren ziemlich gebräuchlich und heißt dort “Kreispuffer” oder “Ringpuffer”.

Wir beginnen mit der Konstruktion unseres “Aufnahmekopfs”. Diese Funktion ähnelt stark dem Sägezahn-Oszillator, den wir schon kennen, abgesehen davon, dass die Berechnungen im Integer-Modus stattfinden. Werte werden mit jedem Audio-Tick um 1 erhöht; der Bereich der Ausgangs-Werte ist mit *0* bis *N-1* festegelegt, wobei *N* die Größe des Arrays ist. Lassen Sie uns den Schaltkreis, der den Schreib-Index berechnet, in ein Macro namens “Record-Pos” verpacken:

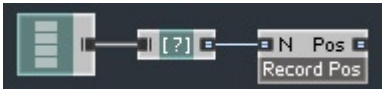


Der Eingang “N” sollte die Anzahl der Elemente im Array entgegennehmen, der Ausgang “Pos” gibt die aktuelle Schreib-Position (Index) aus. Dies ist eine Möglichkeit, dieses Macro aufzubauen (sehen Sie sich zum Vergleich noch einmal den Sägezahn-Oszillator an):

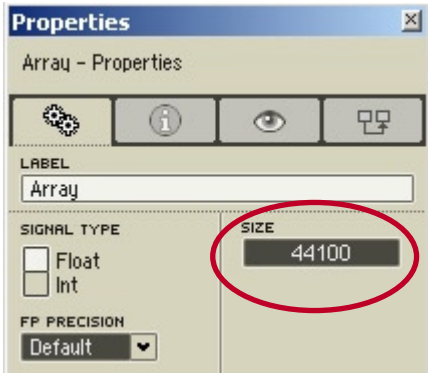


Beachten Sie, dass das Vergleicher-Modul auf “>=”. eingestellt ist. Das war für den Sägezahn-Oszillator nicht von Bedeutung (da konnten wir sowohl “>=” als auch “>” verwenden), aber in Integer-Berechnungen ist das normalerweise wichtig. Verwenden Sie also die Bedingung “>=” um sicherzustellen, dass

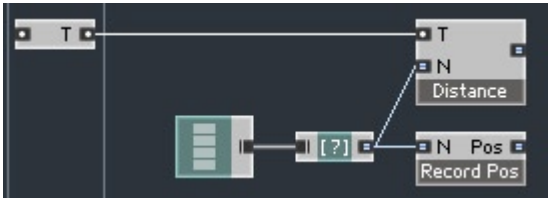
der Schreib-Index niemals den Wert *N* erreicht.
Auf der oberen Ebene erzeugen Sie ein Array-Modul und verbinden es über das Modul *Size []* (das Sie unter *Built In Module > Memory > Size []* finden und das die Größe des Arrays meldet) mit dem Macro "RecordPos":



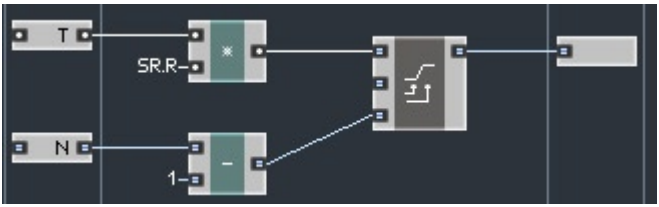
Die Eigenschaft *Size* des Arrays können Sie im Properties-Fenster auf den Wert 44100 einstellen. Dies ermöglicht uns ein Delay mit einer Länge von bis zu einer Sekunde (genau genommen ein Sample weniger) bei einer Sampling-Rate von 44,1 kHz:



Jetzt müssen wir den Schreib-Index berechnen. Das werden wir machen, indem wir zwei neue Macros aufbauen. Das erste Macro wird die gewünschte Delay-Zeit in eine Entfernung in Samples konvertieren:



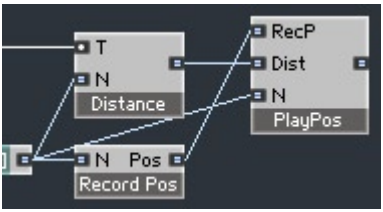
Das geschieht, indem die Zeit in Sekunden mit der Sampling-Rate in Hz multipliziert wird. Wir sollten auch nicht vergessen, das Ergebnis passend zu beschneiden; dafür verwenden wir das Macro *Expert Macro > Clipping >IClipMinMax:*



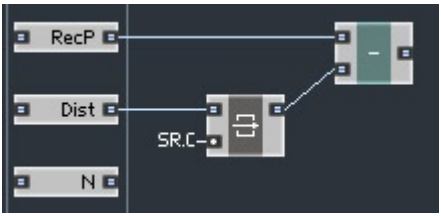
Wir stutzen das Ergebnis auf N-1, weil dies die maximale Distanz zwischen zwei verschiedenen Array-Elementen ist. Beachten Sie, dass die Konvertierung nach Integer hinter der Multiplikation stattfindet.

Alternativ hätten wir den Eingangs-Wert auf einen bestimmten Bereich beschneiden können, was meistens etwas günstiger ist, weil Fließkomma-Werte, die außerhalb des Bereichs der Integer-Darstellungsraums liegen, beliebige Integer-Werte erzeugen können, sodass wir nicht länger eine wirkliche Werte-Beschneidung durchführen.

Jetzt verwenden wir ein anderes Macro, um den Lese-Index aus den Werten von *RecordPos* and *Distance* zu ermitteln:



Offenbar muss die Abspiel-Position um die von *Distance* gespeicherte Anzahl von Samples hinter der Aufnahme-Position liegen, deshalb ziehen wir die eine von der anderen ab:

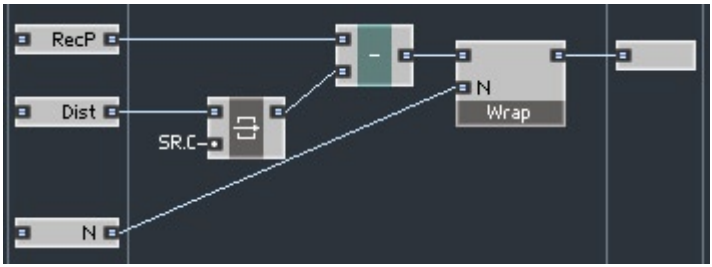


Der Entfernungswert wird zwischengespeichert, weil er einem Kontroll-Signal-Eingang entspringt, der prinzipiell jederzeit Events empfangen kann, und wir wollen ja nicht, dass die Subtraktion zu einem anderen Zeitpunkt stattfindet als beim Eintreffen eines Audio-Clock-Events.

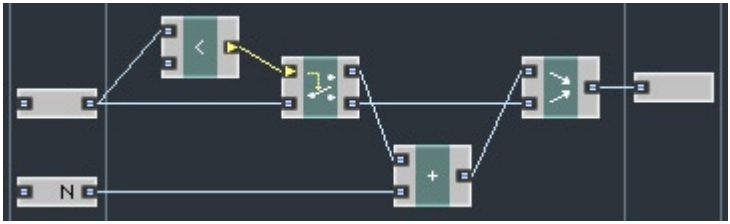
Wenn wir einfach subtrahieren, kann die Differenz allerdings leicht weniger als Null sein. Das kommt daher, dass unser Array keine Schleife ist – seine "Enden" sind nicht miteinander verbunden. Wir müssen das Ergebnis also "umbiegen":

-1 muss zu *N-1* werden,
-2 muss zu *N-2* werden,
-3 muss zu *N-3* werden,
etc.

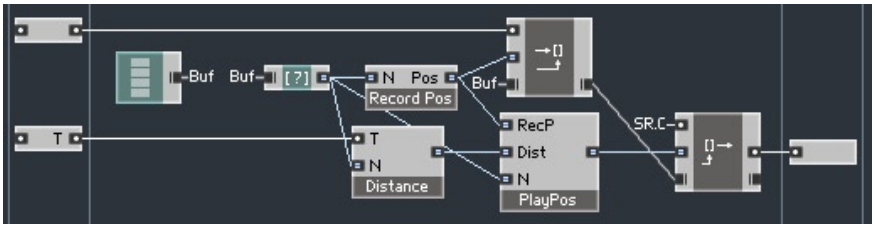
Also setzen wir für dieses "Umbiegen" (Wrapping) ein anderes Macro ein:



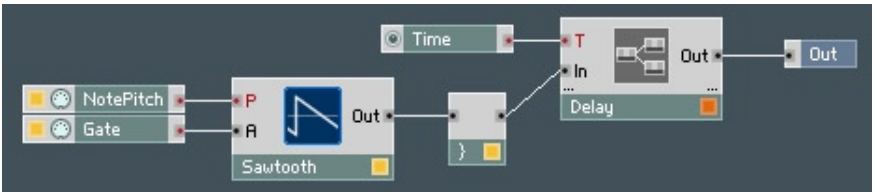
Weil wir wissen, dass die Differenzen nicht kleiner als –N+1 werden können (weil *RecordPos* immer zwischen 0 und N 1 liegt und *Distance* sich zwischen 0 und N-1 bewegt), können wir das Wrapping als einfache Addition von *N* implementieren:



Lassen Sie uns nun zu unserer Top-Level-Struktur zurückkehren. Es scheint, als wir nun Schreib- und Lese-Indizes hätten, also müssen wir nur noch das Schreiben und Lesen durchführen lassen:



Beachten Sie, dass das Lesen nach dem Schreiben erfolgt und dass es von der Sampling-Rate-Clock getaktet wird.
Hier sehen Sie eine geeignete Test-Struktur. Vergessen Sie nicht, den Konverter *ms2sec* in die Delay-Core-Cell einzusetzen und auf monophonen Betrieb einzustellen:



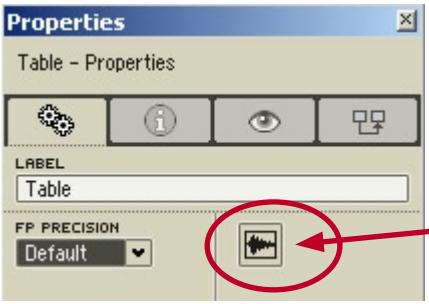
Es ist tatsächlich eine gute Idee, das Delay so früh wie möglich in den monophonen Modus zu schalten, weil es für jede Stimme ungefähr 200 KByte Arbeitsspeicher belegen würde. 44100 Samples, von denen jedes 4 Byte (32 Bit) verwendet:
44100*4 = 176400 Byte, etwas mehr als 172K (ein Kilobyte (KByte) hat 1024 Byte)

Um die obige Struktur zu testen, spielen Sie Noten auf Ihrem MIDI-Keyboard und hören Sie, wie diese Noten um die mit dem Regler “Time” eingestellte Zeitspanne verzögert werden.

8.4. Tabellen

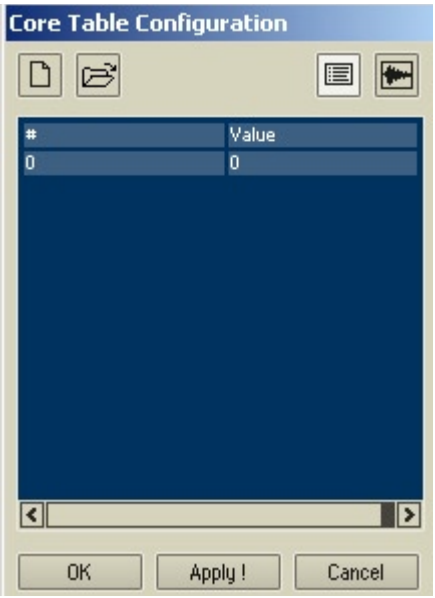
Es gibt noch ein anderes Modul, das dem Array ähnlich ist. Dieses Modul heißt *Table* (Tabelle); Sie finden es unter *Built In Module* > *Memory* > *Table*:

Die Ähnlichkeit ist, dass Tabellen Arrays einer bestimmten Art sind. Der Unterschied ist, dass Sie aus Tabellen nur lesen, aber nichts in die Tabellen hineinschreiben können. Die Werte in einer Tabelle werden im Properties-Fenster des Table-Moduls vorinitialisiert. Um auf die Liste der Werte zuzugreifen, klicken Sie auf den Schalter im Properties-Fenster:



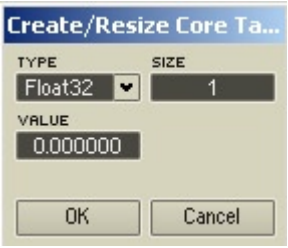
Klicken Sie hier um die Werte zu bearbeiten.

Ein neues Fenster sollte erscheinen:



Was Sie hier sehen, ist eine leere Tabelle. Sie besteht im Moment nur aus einem einzigen Element mit dem Wert Null. Sie können nun entweder manuell (über Ihre Computertastatur) neue Werte eingeben oder Werte aus einer Datei importieren.

Wenn Sie den manuellen Weg gehen wollen, klicken Sie auf die Schaltfläche  button. Es erscheint das folgende Dialogfenster:



Hier müssen Sie den Daten-Typ angeben, der in der Tabelle gespeichert werden soll. Außerdem müssen Sie die Größe der Tabelle bestimmen (die Anzahl der Elemente in der Tabelle) und einen Wert angeben, um alle Elemente der Tabelle zu initialisieren.

Alternativ können Sie die Tabelle aus einer Datei importieren. Die Datei kann in den Audio-Formaten WAV und AIFF, als reiner Text (TXT/ASC) oder als Native Table File (NTF) vorliegen.
Um eine Tabelle aus einer Datei zu importieren, klicken Sie auf die Schaltfläche  .

Es erscheint ein Dateiauswahl-Dialog, in dem Sie eine Datei auswählen können. Anschließend geben Sie in einem weiteren Dialogfenster den Daten-Typ für die Tabellenwerte an.

Lassen Sei uns mal versuchen, eine Tabelle zu verwenden. Wir werden ein Sinus-Oszillator-Macro bauen, das auf einer Tabelle beruht:

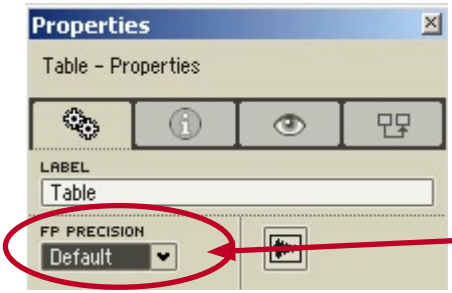
In diesem Macro erzeugen Sie ein Table-Modul:

Und dann initialisieren Sie die Tabelle mit dem Inhalt der Datei *sinetable.txt*, die wir für Sie vorbereitet und im Ordner *"Core Tutorial Examples"* in Ihrem REAKTOR-Installationsverzeichnis abgelegt haben. Es handelt sich dabei um eine Text-Datei, die Werte für eine Periode und ein Sample einer Sinus-Funktion enthält.

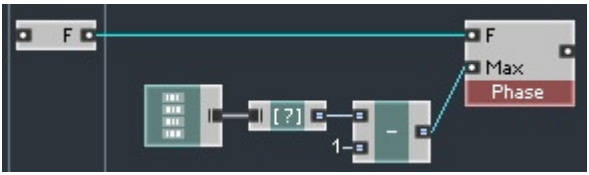
Importieren Sie diese Datei als Werte vom Typ *Float32*:

Sie können sich die geladenen Werte auch in einer Wellenform-Anzeige ansehen.

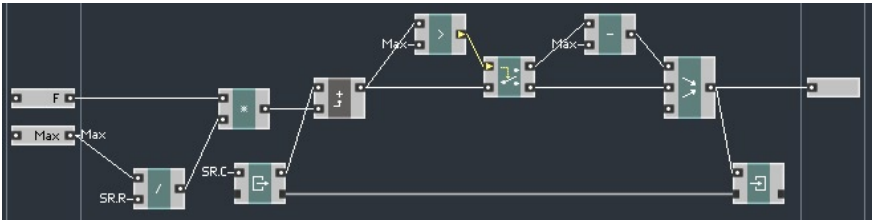
Mit den Schaltflächen  und  können Sie zwischen der Listen- und der Wellenform-Ansicht umschalten.
Klicken Sie nun auf “OK”, um den Dialog zu schließen und die geladenen Werte an die Tabelle zu übergeben.
Im Properties-Fenster des Table-Moduls finden Sie außerdem die Eigenschaft *FP Precision*. Diese Eigenschaft kontrolliert nicht wirklich die Genauigkeit der Werte in der Tabelle (denn die sollten Sie ja schon beim Importieren oder manuellen Eingeben der Werte festlegen), sondern die “formale” Genauigkeit der Ausgangs des Table-Moduls. Normalerweise behalten Sie hier die Einstellung “Default” bei:



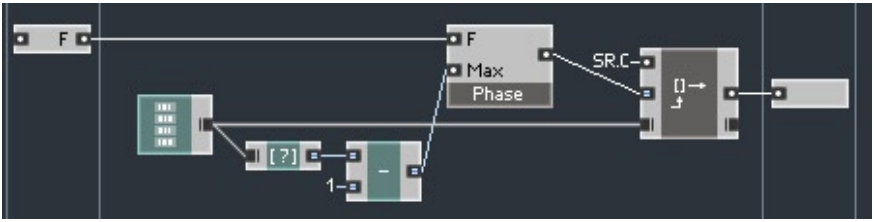
Jetzt haben wir also eine Tabelle und können den Aufbau des Oszillators fortsetzen. Im Kern wird es sich um einen "Phasen-Oszillator" handeln, der ein steigendes Sägezahn-Rampen-Signal von 0 bis zur Größe der Tabelle minus 1 erzeugt



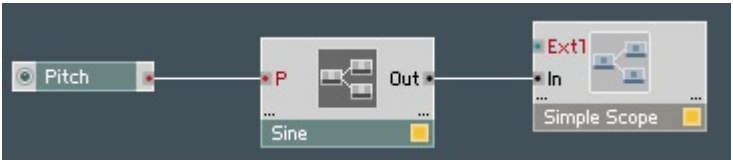
Der Phasen-Oszillator ist ziemlich ähnlich aufgebaut wie ein Sägezahn oder die Aufnahme-Position unseres Delays:



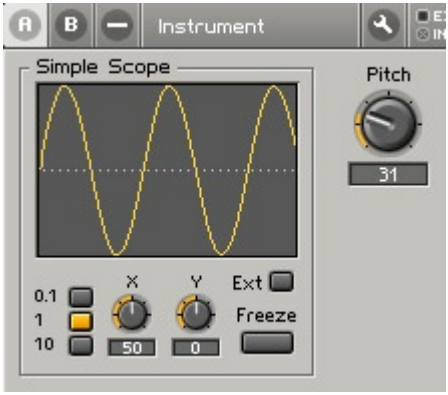
Ein Modul des Typs *Read []*, das mit dem Phasen-Oszillator verbunden ist und von der Sampling-Rate-Clock getaktet wird, greift auf das zugehörige Tabellen-Element zu und gibt seinen Wert aus.



Hier sehen Sie die passende Test-Struktur (und vergessen Sie nicht, ein Konverter-Modul des Typs *P2F* in die Core Cell einzusetzen):



Und hier ist die entsprechende Panel-Ansicht:



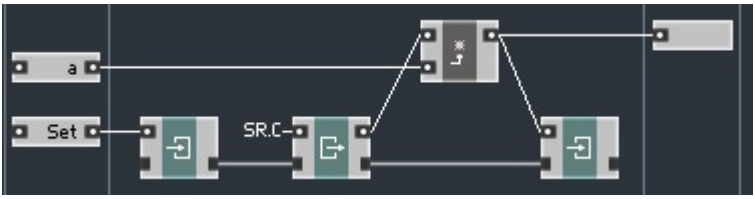
Das ist natürlich kein besonders sauber klingender Sinus, weil wir keine Interpolation zwischen den Werten vorgesehen haben. Wir überlassen es Ihnen, eine interpolierte Version zu bauen.

9. Wie Sie optimale Strukturen aufbauen

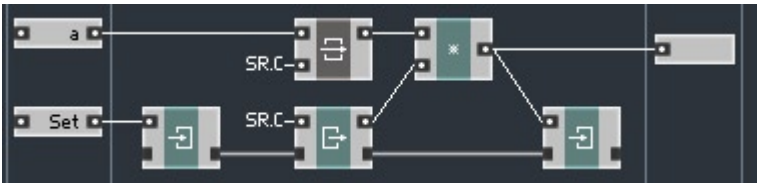
Es gilt die Regel, dass kein Werkzeug von sich aus ideal ist. REAKTOR Core bildet da keine Ausnahme. Das bedeutet natürlich nicht, dass es schlecht ist, ganz im Gegenteil – wir finden REAKTOR Core natürlich klasse. *Nicht ideal* bedeutet einfach, dass Sie einige Dinge darüber wissen müssen, wie man aus dieser Technik die besten Ergebnisse herausholt. Nennen Sie's "Tipps und Tricks" oder wie Sie sonst wollen, hier befassen wir uns jedenfalls mit diesen Dingen.

9.1. Latches und Modulations-Macros

Verwenden Sie Latches und Modulations-Macros an allen Stellen, an denen es geboten erscheint, um sicherzustellen, dass die Events verzögert werden, bis die Werte, die sie transportieren, wirklich verarbeitet werden müssen.
Hier sehen Sie eine Struktur, die ein Modulations-Macro für die Multiplikation in der Audio-Iterationsschleife verwendet. Durch den Einsatz des Modulations-Macros wird verhindert, dass am Eingang "a" ankommende Events die Verarbeitung triggern:



Alternativ dazu könnte man einen expliziten Latch in dieser Struktur verwenden:



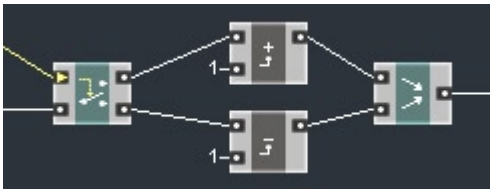
Wir haben in früheren Kapiteln auch verschiedene andere Beispiele für diese Technik kennen gelernt.Die Verwendung von Latches hat sowohl mit der Optimierung der Performance als auch mit der *Korrektheit* Ihrer Strukturen zu tun. Einige typische Fehler beim Entwerfen von Strukturen haben mit dem Senden von Events an bestimmte Module zu ungeeigneten Zeitpunkten zu tun.

Haben Sie keine Angst, dass die Latches Ihre Strukturen ausbremsen – Latches

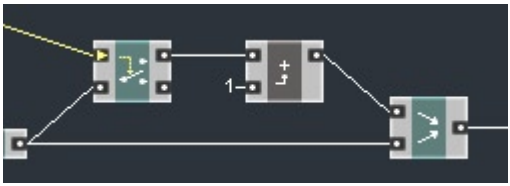
sind nicht sehr anspruchsvoll in Hinblick auf Prozessorleistung, und in manchen Fällen brauchen sie auch *überhaupt keine* Rechenzeit.
Latches sollten Sie beim Filtern von Events generell dem Routing vorziehen, weil sie weniger CPU-Last erzeugen. Verwenden Sie Router nur dann, wenn die Verarbeitungslogik Routing vorschreibt.

9.2. Routing und Merging

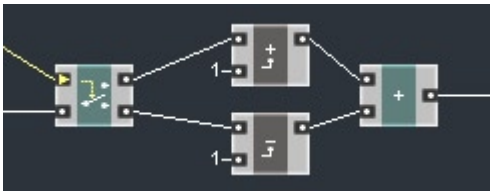
Das Routing kann die CPU je nach Situation und Plattform mehr oder weniger belasten. Wenn Sie Routing vermeiden können, ohne sich dafür andere aufwendige Berechnungen einzuhandeln, dann tun Sie's. Manchmal lässt sich das ES-Ctl-Routing durch die Verwendung von Latches umgehen. Wenn möglich, nehmen Sie diese Gelegenheit wahr. Wenn Sie den Event-Pfad unter Verwendung eines Routers in zwei Zweige aufspalten, ist es eine gute Idee, die an den Ausgängen des Router-Moduls entspringenden Zweige zu mischen:



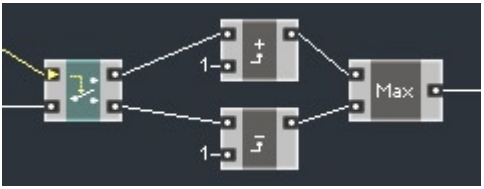
Es ist außerdem klug, das ankommende (ungeteilte) Event dem Router hinzuzumischen:



Das sortierende Mischen (Merging) muss nicht notwendigerweise mit einem *Merge*-Modul durchgeführt werden. Jedes arithmetische oder vergleichbare Modul kann diese Aufgabe übernehmen



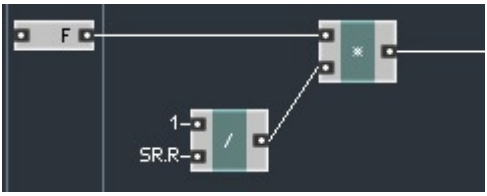
Das Merging kann auch innerhalb eines Macros stattfinden (abhängig von seiner internen Struktur):



Es kann sinnvoll oder sogar notwendig sein, die beiden von verschiedenen Router-Modulen erzeugten Zweige zu mischen, aber behalten Sie in diesem Fall die CPU-Last im Auge.

9.3. Numerische Operationen

Addition, Multiplikation, Subtraktion, Absoluter Wert und Negation sind normalerweise die am wenigsten anspruchsvollen Fließkomma-Operationen. Beim Verarbeiten von Integer-Zahlen belasten Addition, Subtraktion und Negation die CPU am wenigsten. Auch der Absolute Wert ist bei Integer-Berechnungen mehr oder weniger okay. *DN Cancel* entspricht zurzeit einer einfachen Addition, wie Sie sich erinnern werden. Die Division von Float-Werten und die Multiplikation und Division von Integer-Zahlen belasten die CPU allerdings deutlich mehr als die durchschnittlichen Rechenoperationen. Es ist ratsam, dass Sie Ihre Rechenoperationen so gruppieren, dass die anspruchsvollste so selten wie möglich in Zahlen ausgedrückt wird. Wenn Sie zum Beispiel eine normalisierte Frequenz durch Teilen der Frequenz in Hz durch die Sampling-Rate berechnen wollen, ist es sinnvoll, dass Sie zuerst den Kehrwert der Sampling-Rate berechnen und dann das Ergebnis mit der Frequenz multiplizieren:



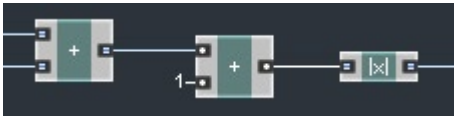
In der obigen Struktur wird die Division nur dann durchgeführt, wenn sich die Sampling-Rate ändert, was ziemlich selten vorkommen sollte. Veränderungen der Frequenz würden nur die Multiplikation triggern.
Betrachten Sie zum Vergleich die simple Implementation derselben Formel:

Hier wird die Division auch bei jeder Änderung der Frequenz durchgeführt.

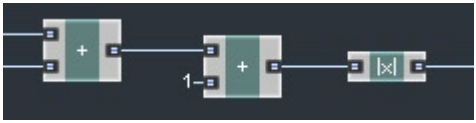
9.4. Konvertierungen zwischen Float und Integer

Vermeiden Sie nach Möglichkeit alle unnötigen Konvertierungen zwischen Fließkomma- und Integer-Zahlen. Abhängig von der Plattform erfordern diese Konvertierungen nämlich viel CPU-Leistung. Notwendige Konvertierungen dürfen Sie natürlich durchführen lassen, keine Frage.

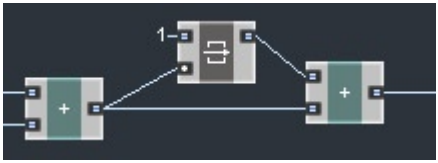
Obwohl die folgende Struktur vielleicht wie vorgesehen arbeitet, finden hier tatsächlich zwei unnötige Konvertierungen zwischen den Zahlen-Formaten Float und Integer statt:



Die erste Konvertierung erfolgt am Eingang des Addierer-Moduls in der Mitte. Das Modul befindet sich im Fließkomma-Modus, empfängt aber ein Integer-Eingangssignal. Deshalb wird eine Konvertierung von Integer nach Float durchgeführt. Die zweite Konvertierung findet am Eingang des Absoluter-Wert-Moduls statt, das im Integer-Modus arbeitet, aber ein Fließkomma-Signal empfängt. Deshalb ist eine Konvertierung von Float nach Integer erforderlich.
Hier sehen Sie eine viel bessere Lösung:



Alle Module sind in den Integer-Modus geschaltet, deshalb finden keine Konvertierungen statt. Clock-Signale sollten normalerweise im Fließkomma-Format vorliegen, aber wenn ein Integer-Signal als Clock auftaucht, ist das auch kein Problem:



Obwohl der Clock-Eingang des Moduls ILatch sich im Fließkomma-Modus befindet, wird er von einem Integer-Signal getaktet. Weil der Wert des Clock-Signals allerdings unerheblich ist, findet keine Konvertierung statt.

Appendix A. REAKTOR Cores Bedienoberfläche

A.1. Core cells

Eine Core Cell erzeugen Sie aus einer Primary-Level-Struktur heraus (außer aus Ensemble-Strukturen), indem Sie einen Rechts-Klick in den Hintergrund ausführen und aus dem Menü *Core Cell > New Audio* oder *Core Cell > New Event* auswählen.
Library-Core-Cells (sowohl aus der System- als auch aus der User-Library) finden Sie im selben Menü *Core Cell*. Sie können Core Cells auch über den Menü-Eintrag *Core Cell > Load*... laden.
Um eine Core Cell zu löschen, wählen Sie die Core Cell aus und drücken die Taste *Delete*. Alternativ können Sie einen Rechts-Klick auf die Core Cell ausführen und aus dem Menü den Eintrag *Delete* auswählen. Sie können auch mehrere ausgewählte Core Cells gleichzeitig löschen.
Um eine Core Cell als Datei zu speichern, führen Sie einen Rechts-Klick auf die Core Cell aus und wählen aus dem Menü den Eintrag *Save Core Cell As*...aus. Um die interne Struktur einer Core Cell zu bearbeiten, doppelklicken Sie auf die Core Cell. Um zurück auf die höhere Ebene zu gelangen, klicken Sie in den Hintergrund.
Um die äußeren Eigenschaften einer Core Cell zu bearbeiten, führen Sie einen Rechts-Klick auf die Core Cell aus und wählen Sie den Eintrag *Properties* aus dem Menü. Wenn das Properties-Fenster bereits geöffnet ist, müssen Sie nur auf die Core Cell klicken, deren Eigenschaften Sie bearbeiten wollen.
Um die inneren Eigenschaften einer Core Cell zu bearbeiten, müssen Sie die innere Struktur der Core Cell freilegen, einen Rechts-Klick in den Hintergrund ausführen und den Eintrag *Owner Properties* aus dem Menü auswählen. Wenn das Properties-Fenster bereits geöffnet ist, müssen Sie nur in den Hintergrund klicken.

A.2. Core-Module und -Macros

Um ein normales Core-Modul oder -Macro zu erzeugen, führen Sie einen Rechts-Klick in den zentralen (und größten) Bereich der Core-Struktur aus und wählen Sie einen der Einträge aus den Menüs *Built In Module, Expert Macro, Standard Macro* oder *User Macro* aus. Sie können Module und Macros auch laden, indem Sie einen Rechts-Klick in den Hintergrund ausführen und aus dem Menü den Eintrag *Load Module*... auswählen.
Ein leeres Macro erzeugen Sie, indem Sie im Menü *Built In Module* den entsprechenden Eintrag auswählen. Um ein Core-Modul oder Core-Macro als Datei zu speichern, führen Sie einen Rechts-Klick auf das betreffende Modul

oder Macro aus und wählen aus dem Menü den Eintrag *Save As…* aus.
Um ein Core-Modul oder Core-Macro zu löschen, wählen Sie es aus und drücken die Taste Delete. Alternativ können Sie einen Rechts-Klick auf das Modul oder Macro ausführen und aus dem Menü den Eintrag *Delete* auswählen. Sie können auch mehrere ausgewählte Module oder Macros gleichzeitig löschen.
Um die interne Struktur eines Core-Macros zu bearbeiten, doppelklicken Sie auf die Core Cell. Um zurück auf die höhere Ebene zu gelangen, klicken Sie in den Hintergrund.
Um die Eigenschaften eines Core-Moduls oder –Macros zu bearbeiten, legen Sie die innere Struktur frei, führen einen Rechts-Klick in den Hintergrund aus und wählen den Eintrag *Owner Properties* aus dem Menü. Wenn das Properties-Fenster bereits geöffnet ist, müssen Sie nur in den Hintergrund klicken.
Sie können die Eigenschaften des Moduls oder Macros auch von außen erreichen, indem Sie einen Rechts-Klick auf das Modul oder Macro ausführen und aus dem Menü den Eintrag *Properties* auswählen. Wenn das Properties-Fenster bereits geöffnet ist, müssen Sie nur auf das Modul oder Macro klicken.

A.3. Core ports

Um einen Core-Port zu erzeugen führen Sie einen Rechts-Klick in den Eingangs-Bereich (links) oder in den Ausgangs-Bereich (rechts) einer Core-Struktur aus. Wählen Sie aus dem Untermenü New einen der verfügbaren Port-Typen aus.
Um einen Core-Port zu löschen, wählen Sie den Port aus und drücken die Taste *Delete*. Alternativ können Sie einen Rechts-Klick auf den Port ausführen und aus dem Menü den Eintrag *Delete* auswählen. Sie können auch mehrere ausgewählte Core-Ports gleichzeitig löschen (einschließlich gemischter Modul-/Port-Auswahlen).

A.4. Core-Strukturen bearbeiten

Um ein Core-Modul zu bewegen, klicken Sie darauf und verschieben Sie das Modul mit gedrückter Maustaste an den gewünschten Platz. Ports können Sie nur in vertikaler Richtung verschieben; die vertikale Reihenfolge bestimmt über ihre Reihenfolge in der äußeren Ansicht.
Um eine Verbindung zwischen dem Eingang eines Moduls und dem Ausgang eines anderen Moduls herzustellen, klicken Sie auf einen der beiden Ports und ziehen Sie das virtuelle "Kabel" zum anderen Port.
Um eine Verbindung zu entfernen, wählen Sie zunächst das Verbindungskabel aus, indem Sie darauf klicken. Drücken Sie dann die Taste *Delete*, um die ausgewählte Verbindung zu löschen. Alternativ können Sie auch das Verbindungskabel aus dem Eingang ziehen und über dem Hintergrund der Struktur loslassen.

Um eine QuickConst zu erzeugen, führen Sie einen Rechts-Klick auf den Eingang eines Moduls aus und wählen Sie den Eintrag *Connect to New QuickConst* aus. Um das Properties-Fenster der QuickConst zu öffnen und auf die Eigenschaften zuzugreifen, klicken Sie auf die QuickConst.
Um einen QuickBus zu erzeugen, führen Sie einen Rechts-Klick auf einen Eingang oder einen Ausgang eines Moduls aus und wählen aus dem Menü den Eintrag *Connect to New QuickBus* aus. Um einen Eingang oder einen Ausgang eines Moduls mit einem bestehenden QuickBus zu verbinden, führen Sie einen Rechts-Klick auf diesen Eingang oder Ausgang aus und wählen aus dem Menü *Connect to QuickBus* einen der verfügbaren Busse aus.

Appendix B. Konzepte von REAKTOR Core

B.1. Signale und Events

In REAKTOR Core kommen Signale der Typen Float und Integer vor. Float-Ports sehen so aus:  . Integer-Ports sehen so aus: .Die Signale pflanzen sich durch die Verbindungen zwischen Ausgängen und den angeschlossenen Eingängen in Form von Events fort. Ein Event ist eine elementare Aktion, die infolge einer Änderung des Werts eines Ausgangs an diesem Ausgang auftritt (wobei in Sonderfällen der Wert auch auf denselben Wert geändert werden kann).Alle Events, die aus derselben Event-Quelle stammen, werden als "gleichzeitig" behandelt. "Gleichzeitig" bedeutet, dass zwei solche Events, die an unterschiedlichen Eingängen desselben Moduls ankommen, dort "gleichzeitig" ankommen.

"Dieselbe Event-Quelle" bedeutet, dass die Events von demselben Ausgang gesendet werden. Unter bestimmten Umständen können aber auch mehrere Ausgänge als "dieselbe Event-Quelle" angesehen werden. Z. B. werden alle Audio-Eingänge und alle Standard-Sampling-Rate-Clock-Verbindungen als eine Event-Quelle behandelt. Während der Initialisierung werden alle Ausgänge, die Events senden, als dieselbe Event-Quelle betrachtet. "Dieselbe Event-Quelle" bedeutet in diesem Zusammenhang allerdings nicht "derselbe Wert", sondern bezieht sich nur auf die "Gleichzeitigkeit". Wenn ein Modul nicht selbst eine Event-Quelle ist, kann es nur durch das Eintreffen eines oder mehrerer Events an seinem Eingang zum Verarbeiten der anliegenden Werte angeregt werden. Im Fall des Eintreffens mehrerer Events wird nur ein Ausgangs-Event erzeugt, weil die Eingangs-Events gleichzeitig eintreffen.

B.2. Initialisierung

Die Initialisierung der Strukturen geht folgendermaßen vor sich: Zuerst werden alle Werte auf Null zurückgesetzt. Dann senden alle Initialisierungs-Quellen gleichzeitig das Initialisierungs-Event. Normalerweise sind diese Quellen Konstanten-Module, Core-Cell-Eingänge (nicht immer) und Clock-Quellen. Das ist es schon.

B.3. Verbindungs-Typ OBC

Der Verbindungs-Typ OBC (Object Bus Connections) bezeichnet Verbindungen zwischen Modulen, die keine Signale senden, sondern belegen, dass die Module einen gemeinsamen Zustand (Speicherplatz) teilen. Am häufigsten kommt der Typ OBC bei Verbindungen zwischen Modulen der Typen *Read* und *Write*, die auf denselben gespeicherten Wert zugreifen, zum Einsatz.

B.4. Routing

Sie können Module des Typs *Router* verwenden, um den Strom von Events zwischen zwei möglichen Pfaden aufzuteilen. Wenn der *Router* einen Ausgangs-Pfad für das ankommende Event auswählt, empfängt der andere Pfad keine Events (was sich besonders darin äußert, dass sich der Wert dieses anderen Ausgangs nicht ändern kann).Das Modul *Router* wird von einer Eingangs-Verbindung des Typs *BoolCtl* kontrolliert. Auf der anderen Seite dieser Verbindung befindet sich typischerweise ein Vergleicher-Modul (*Compare*); manchmal sind einige vermittelnde Modulen wie Macro-Ports des Typs *BoolCtl* zwischen die Module *Compare* und *Router* geschaltet. Durch Verwenden von Modulen des Typs *Router* können Sie Werteänderungen in Teilen Ihrer Struktur gezielt ermöglichen und verhindern.
In den allermeisten Fällen werden Sie die beiden durch das Einsetzen eines Router-Moduls entstehenden Signal-Pfade wieder mischen; hierzu können Sie ein Modul des Typs *Merge* oder auch ein anderes Modul einsetzen. Oftmals können Sie auch einen Pfad dem vor der Aufteilung anliegenden Original-Signal hinzumischen. Aber Sie können hier im Allgemeinen machen, was Sie wollen – behalten Sie nur die Performance Ihrer Struktur im Auge!

B.5. Latching

Latching ist wahrscheinlich die am weitesten verbreitete Technik in REAKTOR Core. Das Prinzip dahinter ist, dass Sie Latch-Module verwenden, um zu verhindern, dass Events zur falschen Zeit gesendet werden. Zum Beispiel wollen Sie wahrscheinlich nicht, dass ein Kontroll-Signal eine Berechnung in der Audio-Schleife der Struktur triggert. Alternativ können Sie auch Macros verwenden, die Sie im Menü unter *Expert Macros > Modulation* finden; es handelt sich bei diesen Macros im Prinzip um die häufigsten Kombinationen von Latches mit einigen arithmetischen Modulen.

B.6. Clocking

Clocks sind Quellen von Events. Das Clock-Event tritt üblicherweise in regelmäßigen Intervallen abhängig von der Clock-Rate auf. Sie brauchen Clocks normalerweise, um eine Vielzahl von Modulen anzutreiben, zum Beispiel Oszillatoren, Filter und so weiter. Die meisten dieser Module brauchen keine Taktung von außen, sondern verwenden eine in Core-Strukturen verfügbare Standard-Clock-Quelle. Diese Quelle ist die Sampling-Rate-Clock, die mit der Standard-Audio-Rate läuft.Beachten Sie, dass in Event-Core-Cells das Clock-Signal nicht verfügbar ist, obwohl eine Verbindung zur Sampling-Rate-Clock möglich ist. Deshalb arbeiten die meisten Oszillatoren, Filter und ähnliche Module nicht in Event-Core-Cells.

Appendix C. Core-Macro-Ports

C.1. In

Empfängt ein ankommendes Event von der Außenseite und leitet es unverändert an seinen inwärts gerichteten Ausgang weiter.

Die Eingangs-Verbindung auf der Innenseite können Sie zum Überschreiben des Default-Werts dieses Ports verwenden.

C.2. Out

Empfängt ein ankommendes Event an seinem innenseitigen Eingang und leitet es unverändert an seinen nach außen gerichteten Ausgang weiter.

C.3. Latch (Eingang)

Empfängt ein ankommendes Event an seinem innenseitigen Eingang und leitet es unverändert an seinen nach außen gerichteten Ausgang weiter.

C.4. Latch (Ausgang)

Schleust eine OBC-Verbindung von der Innenseite eines Macros zur Außenseite des Macros durch.

C.5. Bool C (Eingang)

Schleust eine BoolCtl-Verbindung von der Außenseite eines Macros zur Innenseite des Macros durch.

Die Eingangs-Verbindung auf der Innenseite können Sie zum Überschreiben des Default-Werts dieses Ports verwenden.

C.6. Bool C (Ausgang)

Schleust eine BoolCtl-Verbindung von der Innenseite eines Macros zur Außenseite des Macros durch.

Appendix D. Core-Cell-Ports

D.1. In (Audio-Modus)

Ermöglicht den Zugriff auf das Audio-Signal von außerhalb des Moduls. Sendet regelmäßig (mit der Sampling-Rate), synchron zur globalen Sampling-Rate-Clock.

INITIALIZATION EVENT: sendet ein Initialiserungs-Event. Der Wert wird von der äußeren Initialisierung bestimmt.

D.2. Out (Audio-Modus)

Stellt den am inneren Eingang empfangenen Wert an der Außenseite des Moduls zur Verfügung. Zu jedem Zeitpunkt wird der letzte empfangene Wert an die Außenseite übergeben.

D.3. In (Event-Modus

Konvertiert von außen eintreffende Primary-Level-Events in REAKTOR-Core-Events und leitet sie an die Innenseite weiter.

INITIALIZATION EVENT: sendet ein Initialisierungs-Event, wenn auf der Außen seite ein Initialisierungs-Event empfangen wird.

D.4. Out (Event-Modus)

Konvertiert REAKTOR-Core-Events, die aus dem Inneren ankommen, in Primary-Level-Events und leitet diese an die Außenseite weiter. Wenn mehrere im Event-Modus befindliche Ausgänge gleichzeitig REAKTOR-Core-Events empfangen, werden die korrespondierenden Primary-Level-Events in der vertikalen Reihenfolge der Ausgänge (von oben nach unten) gesendet.

Appendix E. Built-in buses

E.1. SR.C

Sendet regelmäßige Clock-Events mit der Sampling-Rate.
INITIALIZATION EVENT: sendet immer ein Initialisierungs-Event.

E.2. SR.R

Stellt die aktuelle Sampling-Rate in Hz zur Verfügung. Sendet als Reaktion auf Veränderungen der Sampling-Rate Events mit neuen Werten.
INITIALIZATION EVENT: sendet immer ein Initialisierungs-Event mit der anfänglichen Sampling-Rate.

Appendix F. Built-in modules

F.1. Const

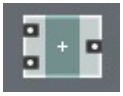
Erzeugt ein Signal mit einem konstanten Wert.
Der Wert wird im Modul angezeigt.

INITIALIZATION EVENT: sendet während der Initialisierung ein Event mit dem festgelegten Wert an den Ausgang. Dies ist das einzige Mal, dass dieses Modul ein Event sendet.

EIGENSCHAFTEN:

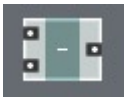
Value der Wert, der an den Ausgang gesendet werden soll

F.2. Math > +



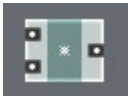
Erzeugt die Summe aus den ankommenden Signalen am Ausgang. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem oder beiden Eingängen gleichzeitig ankommt.

F.3. Math > -



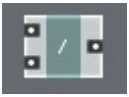
Erzeugt die Differenz aus den ankommenden Signalen am Ausgang (das Signal des unteren Eingangs wird vom Signal des oberen Eingangs subtrahiert). Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.4. Math > *



Erzeugt das Multiplikations-Produkt aus den ankommenden Signalen am Ausgang. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.5. Math > /



Erzeugt den Quotienten aus den ankommenden Signalen am Ausgang (das Signal am oberen Eingang wird durch das Signal am unteren Eingang geteilt). Im Integer-Modus führt dies zu einer Division mit Rest, wobei der Rest verworfen wird. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.6. Math > |x|

Erzeugt den absoluten Wert des ankommenden Signals. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event am Eingang ankommt.

F.7. Math > –x

Erzeugt den invertierten Wert (durch Ändern des Vorzeichens) des ankommenden Signals am Ausgang. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event am Eingang ankommt.

F.8. Math > DN Cancel

Modifiziert das ankommende Signal so, dass das Auftreten denormaler Zahlen verhindert wird. Zurzeit geschieht dies durch Addieren einer sehr kleinen Konstante. Funktioniert nur bei Fließkomma-Zahlen. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event am Eingang ankommt.

F.9. Math > ~log

Berechnet eine Annäherung an den Logarithmus des ankommenden Werts. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event am Eingang ankommt.

EIGENSCHAFTEN:

Base	die Logarithmus-Basis
Precision	die Genauigkeit der Näherung (höhere Genauigkeit erfordert mehr CPU-Leistung)

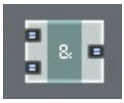
F.10. Math > ~exp

Berechnet eine Annäherung an den Exponenten des ankommenden Werts. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event am Eingang ankommt.

EIGENSCHAFTEN

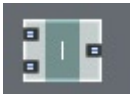
Base	die Exponenten-Basis
Precision	die Genauigkeit der Näherung (höhere Genauigkeit erfordert mehr CPU-Leistung)

F.11. Bit > Bit AND



Führt eine bitweise Zusammenführung (Konjunktion) der ankommenden Signale durch. Arbeitet nur mit Integer-Signalen. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.12. Bit > Bit OR



Führt eine bitweise Trennung (Disjunktion) der ankommenden Signale durch. Arbeitet nur mit Integer-Signalen. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.13. Bit > Bit XOR

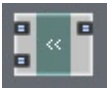
Führt eine bitweise ausschließende Trennung (exklusive Disjunktion) der ankommenden Signale durch. Arbeitet nur mit Integer-Signalen.

Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.14. Bit > Bit NOT

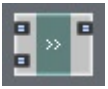
Führt eine bitweise Umkehrung (Invertierung) der ankommenden Signale durch. Arbeitet nur mit Integer-Signalen. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event am Eingang ankommt.

F.15. Bit > Bit <<



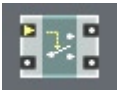
Versetzt den Wert an oberen Eingang bitweise nach links in Richtung der signifikanteren Bits (More Significant Bits). Die Anzahl von Bits, um die der Wert versetzt werden soll, wird vom unteren Eingang bestimmt. Das Ergebnis für $N < 0$ und $N > 31$ ist undefiniert (das heißt, Sie sollten diese Funktion nur verwenden, wenn gilt ($0 \leq N \leq 31$) Arbeitet nur mit Integer-Signalen. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.16. Bit > Bit >>



Versetzt den Wert des oberen Eingangs bitweise nach links in Richtung der weniger signifikanten Bits (Less Significant Bits). Es wird keine Vorzeichen-Erweiterung durchgeführt. Das Ergebnis für $N < 0$ und $N > 31$ ist undefiniert (das heißt, Sie sollten diese Funktion nur verwenden, wenn gilt $0 \leq N \leq 31$). Arbeitet nur mit Integer-Signalen. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

F.17. Flow > Router



Leitet das am Signal-Eingang (dem unteren Eingang) anliegende Signal ab-

hängig vom Zustand des Kontroll-Signals (am oberen Eingang) an einen der beiden Ausgänge weiter. Wenn das Kontroll-Signal den Wert “wahr” hat, wird der Ausgang 0 (der obere Ausgang) beschickt; nimmt das Kontroll-Signal den Zustand “falsch” an, landet das Signal am Ausgang 0 (dem unteren Ausgang). Das Ausgangs-Event wird an genau einen der Ausgänge gesendet, wenn ein Event am Signal-Eingang ankommt.

F.18. Flow > Compare

Erzeugt ein BoolCtl-Signal am Ausgang, welches das Ergebnis des Vergleichs der Eingangs-Werte enthält. Der Wert am oberen Eingang steht auf der linken Seite des Vergleichs-Symbols, der Wert des unteren Eingangs steht rechts (sodass das abgebildete Modul prüft, ob der obere Wert größer ist als der untere Wert).

EIGENSCHAFTEN:

Criterion die Vergleichs-Operation, die durchgeführt werden soll

F.19. Flow > Compare Sign

Erzeugt ein BoolCtl-Signal am Ausgang, welches das Ergebnis des Vergleichs der Eingangs-Werte enthält. Der Wert am oberen Eingang steht auf der linken Seite des Vergleichs-Symbols, der Wert des unteren Eingangs steht rechts (sodass das abgebildete Modul prüft, ob das Vorzeichen des oberen Werts größer ist als das Vorzeichen des unteren Werts).

Der Vorzeichen-Vergleich ist wie folgt definiert:

+ ist gleich +
– ist gleich –
+ ist größer als –

Das Vorzeichen des Werts Null ist undefiniert, sodass beliebige Ergebnisse auftreten können, wenn einer der verglichenen Werte Null ist.

EIGENSCHAFTEN:

Criterion die Vergleichs-Operation, die durchgeführt werden soll

F.20. Flow > ES Ctl

Erzeugt ein BoolCtl-Signal am Ausgang, das die gegenwärtige Anwesenheit eines Events am Eingang anzeigt (was bedeutet, dass das Kontroll-Signal den Wert “wahr” annimmt, wenn zu einem bestimmten Zeitpunkt ein Event am Eingang dieses Moduls auftritt).

F.21. Flow > ~BoolCtl

Erzeugt ein BoolCtl-Signal am Ausgang, das eine Umkehrung des BoolCtl-Eingangs- Signals ist (“wahr” wird zu “falsch” und vice versa).

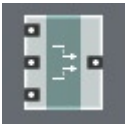
F.22. Flow > Merge

Sendet jedes Mal ein Ausgangs-Event, wenn an einem der Eingänge oder an mehreren Eingängen gleichzeitig ein Event ankommt. Wenn nur ein Eingang zu einem bestimmten Zeitpunkt das Event empfängt, ist der Ausgangs-Wert gleich dem Wert dieses Eingangs-Events. Wenn mehrere Eingänge gleichzeitig ein Event empfangen, wird der Wert des untersten Eingangs (der in diesem Moment empfangenden Eingänge) ausgewählt. Wenn also sowohl der zweite als auch der dritte (von oben aus gesehen) Eingang ein Event empfangen, wird der Wert am dritten Eingang ausgewählt.

EIGENSCHAFTEN:

Input Count Anzahl der Eingänge des Moduls

F.23. Flow > EvtMerge



Die Funktion ähnelt der des Moduls Merge, allerdings werden alle Eingangs-Werte ignoriert. Der Wert des Ausgangs-Events ist undefiniert. Dieses Modul ist dafür gedacht, Signale zu erzeugen, die als Clock verwenden werden. Arbeitet nur im Fließkomma-Modus, weil die Werte ohnehin nicht für eine Weiterverwendung vorgesehen sind.

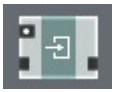
EIGENSCHAFTEN:

Input Count Anzahl der Eingänge des Moduls

F.24. Memory > Read

Liest den gespeicherten Wert aus dem Speicher, der der OBC-Kette zugeordnet ist, zu der dieses Modul gehört. Der Lese-Vorgang findet als Reaktion auf ein Event am oberen Eingang (Clock) statt, der gelesene Wert wird an den oberen Ausgang geschickt. Die unteren Ports sind Master- und Slave-Anschlüsse für die OBC-Anbindung.

F.25. Memory > Write



Schreibt den Wert, der am oberen Eingang ankommt, in den Speicher, der der OBC-Kette zugeordnet ist, zu der dieses Modul gehört. Der Schreib-Vorgang findet als Reaktion auf ein Event am oberen Eingang (Clock) statt. Die unteren Ports sind Master- und Slave-Anschlüsse für die OBC-Anbindung.

F.26. Memory > R/W Order

Dieses Modul führt gar keine Aktion aus. Sie können es in einen Struktur einsetzen, um die Verarbeitungsreihenfolge der via OBC verbunden Module zu kontrollieren. Die OBC-Ports (unten) sind OBC-Master- und Slave-Verbindungen, die intern einfach "durchgeschleift" sind. Der OBC-Eingang (oben) stellt die "Sidechain"-Verbindung her, die es ermöglich, das Modul logisch hinter dem Modul zu platzieren, das an den Sidechain-Eingang angeschlossen ist.
Die Sidechain-Verbindung können Sie nur an normale Module des Typs Latch OBC anschließen. Die Master- und Slave-Ports dagegen können Sie mit Latch- oder Array-OBC-Modulen verbinden, abhängig von den für das Modul R/W Order im Properties-Fenster getroffenen Einstellungen. Auf jeden Fall müssen Signal-Typ und Genauigkeit für alle Verbindungen zu diesem Modul gleich sein (d. h., Sie können nicht den Sidechain-Eingang mit einem Integer-Read-Modul verbinden und gleichzeitig Master und Slave an Fließkomma-Module anschließen).

EIGENSCHAFTEN:

Connection Type Art der “durchgeschleiften” Port-Verbindung (Latch oder Array)

F.27. Memory > Array

Definiert ein Array-Speicher-Objekt. Das Modul selbst führt keine Aktionen aus. Alle Operationen über dem Inhalt des Arrays werden von den Modulen durchgeführt, die an den Ausgang des Arrays angeschlossen sind. Dieser Ausgang ist eine OBC-Slave-Verbindung vom Typ Array.

EIGENSCHAFTEN:

Size Anzahl der Elemente im Array

F.28. Memory > Size [ ]

Übermittelt die Größe des Array-Objekts, das mit dem Eingang verbunden ist. Die Größe ist ein konstanter Integer-Wert

INITIALIZATION EVENT: sendet während der Initialisierung ein Event mit dem Wert der Array-Größe an den Ausgang. Dies ist das einzige Mal, dass dieses Modul ein Event sendet.

F.29. Memory > Index

Ermöglicht den Zugriff auf ein einzelnes Array-Element. Der Zugang erfolgt in Form einer Latch-OBC-Verbindung, die dem Array-Element zugeordnet ist. Die Zuordnung wird durch das Senden eines Events an den oberen Eingang (Index) des Index-Moduls hergestellt und/oder verändert, der *Null-basiert* ist und sich immer im Integer-Modus befindet. Der untere Eingang ist die Master-OBC-Verbindung zum Array. Der Ausgang stellt die Latch-OBC-Verbindung mit dem über den Index-Eingang ausgewählten Array-Element her. Der Basis-Werte-Typ und die Genauigkeit müssen am Eingang und Ausgang der OBC-Verbindung identisch sein; Sie können diese Parameter im Properties-Fenster des Index-Moduls festlegen

F.30. Memory > Table

Definiert ein vorinitialisiertes Read-Only-Array (Tabelle). Alle Operationen über dem Inhalt der Tabelle werden von den Modulen durchgeführt, die an den Ausgang des Table-Moduls angeschlossen sind. Dieser Ausgang ist eine OBC-Slave-Verbindung vom Typ Array.

EIGENSCHAFTEN:

ruft den Editor zum Bearbeiten der Werte der Tabelle auf

FP Precision kontrolliert die formale Genauigkeit der Ausgangs-Verbindung

F.31. Macro

Bildet einen Container für eine im Inneren befindliche Struktur. Die Anzahl der Eingänge und Ausgänge ist nicht festgelegt und wird von der internen Struktur bestimmt.

EIGENSCHAFTEN:

FP Precision kontrolliert die formale Genauigkeit der Ausgangs-Verbindung

Look schaltet zwischen den Ansichten *Large* (Label und Port-Namen sichtbar) und *Small* (Label und Port-Namen unsichtbar) um

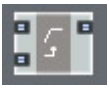
Pin Alignment kontrolliert die Ausrichtung der Ports in der äußeren Ansicht des Macros

Solid kontrolliert die Behandlung des Macros durch die Core-Engine. Wenn diese Option ausgeschaltet ist, sind die Grenzen des Macros transparent für die Feedback-Auflösung und andere Dinge. Lassen Sie diese Option eingeschaltet, außer, Sie wissen wirklich ganz genau, was Sie tun!

Icon  lädt ein neues Icon für das Macro,  löscht das Icon (kein Icon zugewiesen)

Appendix G. Expert macros

G.1. Clipping > Clip Max / IClip Max

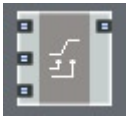



Das Signal am oberen Eingang wird von oben ausgehend gemäß dem am unteren Eingang anliegenden Schwellwert beschnitten. Veränderungen des Schwellwerts erzeugen keine Events.

G.2. Clipping > Clip Min / IClip Min


Das Signal am oberen Eingang wird von unten ausgehend gemäß dem am unteren Eingang anliegenden Schwellwert beschnitten. Veränderungen des Schwellwerts erzeugen keine Events.

G.3. Clipping > Clip MinMax / IClipMinMax

Das Signal am oberen Eingang wird von oben ausgehend gemäß dem am mittleren Eingang anliegenden Schwellwert und von oben ausgehend gemäß dem Schwellwert am unteren Eingang beschnitten. Veränderungen der Schwellwerte erzeugen keine Events.

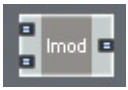
G.4. Math > 1 div x

Berechnet den Kehrwert des Eingangs-Werts.

G.5. Math > 1 wrap

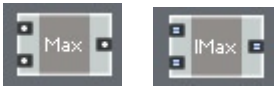
Faltet die ankommenden Werte in den Bereich [-0.5 bis 0.5] (die Faltungsperiode ist 1).

G.6. Math > Imod



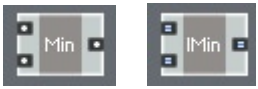
Berechnet den Rest, der bei der Division des oberen Werts durch den unteren Wert übrig bleibt. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

G.7. Math > Max / IMax



Berechnet das Maximum der Eingangs-Werte. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

G.8. Math > Min / IMin



Berechnet das Minimum der Eingangs-Werte. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

G.9. Math > round

Rundet den ankommenden Wert auf die nächste Ganzzahl (Integer). Das Ergebnis der Rundung ist für Werte, die genau in der Mitte zwischen zwei Ganzzahlen liegen, nicht definiert. Das bedeutet, dass z. B. 1,5 entweder auf 1 oder auf 2 gerundet wird.

G.10. Math > sign +-

Gibt entweder 1 oder –1 aus, abhängig vom Vorzeichen des Eingangs (positive Zahlen erzeugen den Wert 1, negative den Wert –1; es wird niemals Null ausgegeben).

G.11. Math > sqrt (>0)

Berechnet näherungsweise die Quadratwurzel des Eingangs-Werts. Arbeitet nur mit Werten größer als 0.·

G.12. Math > sqrt

Berechnet näherungsweise die Quadratwurzel des Eingangs-Werts.

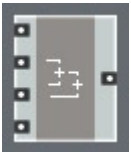
G.13. Math > x(>0)^y

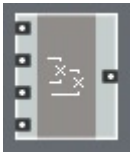
Angenäherte Bestimmung von x^y, wobei gelten muss $x > 0$. Das Ausgangs-Event wird jedes Mal gesendet, wenn ein Event an einem Eingang oder an beiden Eingängen gleichzeitig ankommt.

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

Berechnen die zweite, dritte und vierte Potenz von x.

G.15. Math > Chain Add / Chain Mult





Addiert bzw. multipliziert die Signale in der Reihenfolge von oben nach unten. Das Ausgangs-Event wird gesendet, wenn an einem der Eingänge ein Event auftritt bzw. mehrere Events auftreten.

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

Faltet die ankommenden Werte in den Bereich [-π bis π] (die Faltungsperiode ist 2π).

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

Arcsinus-/Arccosinus-/Arctangens-Approximation.

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

Sinus-/Cosinus-/Tangens-Approximation.

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

Sinus-/Cosinus-/Tangens-Approximation (arbeitet nur im Bereich [-π bis π])

G.20. Math > Trig-Hyp > tan –pi4..pi4

Tangens-Approximation (arbeitet nur im Bereich [-π/4..π/4]).

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

Hyperbolische Sinus-/Cosinus-/Tangens-Approximation.

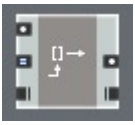
G.22. Memory > Latch / ILatch

Verzögert das Signal am oberen Eingang, bis ein Clock-Event am unteren Eingang ankommt. Wenn beide Events gleichzeitig ankommen, wird das ankommende Signal sofort durchgelassen.

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

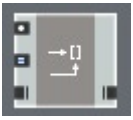
Senden als Reaktion auf ein Clock-Event den letzten Wert, den der obere Eingang empfangen hat, bevor das Clock-Event ankommt. Wenn der Clock-Eingang nicht angeschlossen ist, verwenden beide Versionen des Moduls stattdessen die Standard-Audio-Clock (SR.C) und verzögern das Signal effektiv um ein Sample.Beide Module können automatisch Feedback-Schleifen auflösen, allerdings beherrscht nur das Modul Z^-1 die Denormalen-Unterdrückung. Die Version *Z^-1 ndc* sollten Sie nur an Stellen verwenden, an denen keine Denormalen zu erwarten sind.

G.24. Memory > Read []



Liest als Reaktion auf ein (am oberen Eingang) ankommendes Clock-Event einen Wert aus einem Array mit einem bestimmten Index (der vom mittleren Eingang spezifiziert wird). Der untere Eingang (OBC) stellt die Verbindung zum Array her. Verwenden Sie den OBC-Ausgang des Moduls, um OBC-Ketten für serielle Array-Zugriffe zu erstellen.

G.25. Memory > Write []



Schreibt einen Wert (der am oberen Eingang empfangen wird) in ein Array an einen bestimmten Index (der vom mittleren Eingang spezifiziert wird). Der Schreib-Vorgang wird von einem ankommenden Wert getriggert. Der untere Eingang (OBC) stellt die Verbindung zum Array her.
Verwenden Sie den OBC-Ausgang des Moduls, um OBC-Ketten für serielle Array-Zugriffe zu erstellen.

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

Fügt als Reaktion auf ein ankommendes Event einen Parameter (vom unteren Eingang) dem Signal des oberen Eingangs hinzu. Parameter-Änderungen erzeugen keine Events.

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

Multipliziert als Reaktion auf ein ankommendes Event das Signal (am oberen Eingang) mit einem Parameter (am unteren Eingang). Parameter-Änderungen erzeugen keine Events.

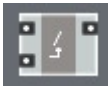
G.28. Modulation > x – a / Integer > Ix – a

Subtrahiert als Reaktion auf ein ankommendes Event einen Parameter (am unteren Eingang) vom Signal (am oberen Eingang). Parameter-Änderungen erzeugen keine Events.

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

Subtrahiert als Reaktion auf ein ankommendes Event das Signal (am unteren Eingang) von einem Parameter (am oberen Eingang). Parameter-Änderungen erzeugen keine Events.

G.30. Modulation > x / a



Dividiert als Reaktion auf ein ankommendes Event das Signal (am oberen Eingang) durch einen Parameter (am unteren Eingang). Parameter-Änderungen erzeugen keine Events.

G.31. Modulation > a / x

Dividiert als Reaktion auf ein ankommendes Event einen Parameter (am oberen Eingang) durch ein Signal(am unteren Eingang). Parameter-Änderungen erzeugen keine Events.

G.32. Modulation > xa + y

Multipliziert das Signal am oberen Eingang mit dem Gain-Parameter (am mittleren Eingang) und addiert das Ergebnis zum Signal am unteren Eingang. Events an einem Signal-Eingang oder an beiden Signal-Eingängen erzeugen einen neuen Ausgangs-Wert, Events am Parameter-Eingang nicht,

Appendix H. Standard macros

H.1. Audio Mix-Amp > Amount

Ermöglicht eine lineare, invertierbare Kontrolle der Amplitude eines Audio-Signals.

A = 0	schaltet das Signal stumm
A = 1	lässt das Signal unverändert
A = -1	invertiert das Signal

Typische Verwendung: Kontrolle der Stärke eines Audio-Feedbacks

H.2. Audio Mix-Amp > Amp Mod

Moduliert die Amplitude des Audio-Signals mit einer bestimmten Stärke (AM) entlang einer linearen Skala.

AM = 1	verdoppelt die Amplitude
AM = 0	lässt das Signal unverändert
AM = -1	schaltet das Signal stumm

Typische Verwendung: Tremolo, Amplitudenmodulation.

H.3. Audio Mix-Amp > Audio Mix

Mischt die beiden Audio-Signale zusammen.

H.4. Audio Mix-Amp > Audio Relay

Schaltet zwischen zwei anliegenden Audio-Signalen um. Wenn x > 0, wird Signal 1 durchgelassen, andernfalls wird Signal 0 durchgelassen.

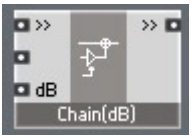
H.5. Audio Mix-Amp > Chain (amount)

Verändert die Amplitude eines Audio-Signals um eine bestimmte lineare Größe (A) und mischt das Signal dem verketteten Audio-Signal (>>) hinzu.

A = 0 schaltet das Signal stumm
A = 1 lässt das Signal unverändert
A = -1 invertiert das Signal

Typische Verwendung: Audio-Mixer-Ketten, Audio-Feedback-Regelung

H.6. Audio Mix-Amp > Chain (dB)



Verändert die Amplitude eines Audio-Signals um einen bestimmten Wert in dB und mischt das Signal dem verketteten Audio-Signal (>>) hinzu.

Typische Verwendung: Audio-Mixer-Ketten

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

Verändert die Amplitude eines Audio-Signals um einen bestimmten Wert in dB.

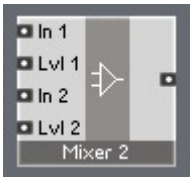
+6 dB verdoppelt die Amplitude
0 dB lässt das Signal unverändert
-6 dB halbiert die Amplitude

Typische Verwendung: Lautstärke-Kontrolle entlang einer dB-Skala

H.8. Audio Mix-Amp > Invert

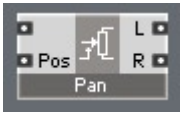
Kehrt die Polarität des Audio-Signals um.

H.9. Audio Mix-Amp > Mixer 2 … 4



Mischt die anliegenden Audio-Signale (In 1, In 2, …) und schwächt dabei ihre Pegel um die angegebenen Werte (Lvl 1, Lvl 2, …) in dB ab.

H.10. Audio Mix-Amp > Pan



Bewegt die ankommenden Audio-Signale im Stereo-Panorama (mit parabolischem Kurvenverlauf). Die Panorama-Position ist folgendermaßen definiert:

-1 ganz links
0 Mitte
1 ganz rechts

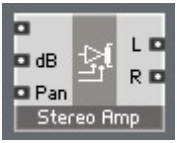
H.11. Audio Mix-Amp > Ring-Amp Mod

Das Carrier-Audio-Signal (am oberen Eingang) wird vom Signal am Eingang "Mod" moduliert. Die Art der Modulation kontrolliert der Eingang "R/A", der sanft zwischen Ringmodulation und Amplitudenmodulation überblendet.

R/A = 0 Ringmodulation
R/A = 1 Amplitudenmodulation

(Für echte Amplitudenmodulation sollte die Amplitude des Modulators den Wert 1 nicht überschreiten).

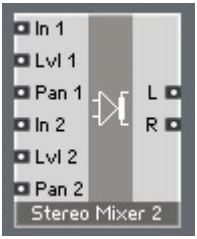
H.12. Audio Mix-Amp > Stereo Amp



Verstärkt ein monophones Audio-Signal um einen bestimmten Wert in dB und platziert es an der angegebenen Position im Stereo-Panorama. Die Panorama-Position ist folgendermaßen definiert:

-1 ganz links
0 Mitte
1 ganz rechts

H.13. Audio Mix-Amp > Stereo Mixer 2 … 4



Mischt die anliegenden Audio-Signale (In 1, In 2, …), schwächt dabei ihre Pegel um die angegebenen Werte (Lvl 1, Lvl 2, …) in dB ab und platziert sie auf den Positionen im Stereo-Panorama (Pan 1, Pan 2, …). Die Panorama-Position ist folgendermaßen definiert:

-1 ganz links
0 Mitte
1 ganz rechts

H.14. Audio Mix-Amp > VCA

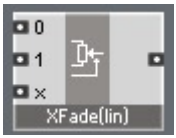
Audio-Verstärker mit direkter linearer Kontrolle über die Amplitude.

A = 0 schaltet das Signal stumm
A = 1 lässt das Signal unverändert

Typische Verwendung: Verbinden Sie die Amplituden-Hüllkurve mit dem Eingang “A”.

Anmerkung: Verwenden Sie für umkehrbare Verstärkung das Modul Audio Amount.

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

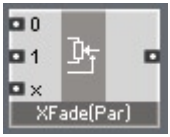


Audio-Crossfader mit linearem Kurvenverlauf

x = 0	nur das Signal 0 ist zu hören
x = 0.5	beide Signale sind gleich stark im Mix vertreten
x = 1	nur das Signal 1 ist zu hören

Anmerkung: Mit einem parabolischen Crossfade erzielen Sie meistens besser klingende Ergebnisse.

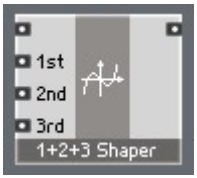
H.16. Audio Mix-Amp > XFade (par)



Audio-Crossfader mit parabolischem Kurvenverlauf. Erzeugt meistens besser klingende Ergebnisse als ein linearer Crossfader.

x = 0	nur das Signal 0 ist zu hören
x = 0.5	beide Signale sind gleich stark im Mix vertreten
x = 1	nur das Signal 1 ist zu hören

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



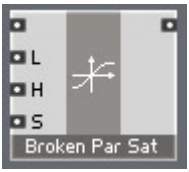
Stellt kontrollierbares Audio-Signal-Shaping zweiter und dritter Ordnung zur Verfügung. Der Eingang “1st” spezifiziert den Anteil des Original-Signals am Ausgangssignal (1= unverändert, 0=kein Anteil). Die Eingänge “2nd” und “3rd” legen den Anteil der Verzerrungen 2. respektive 3. Ordnung fest.

H.18. Audio Shaper > 3-1-2 Shaper

Audio-Signal-Shaper mit veränderlichem Anteilen von Verzerrungen 2. und 3. Ordnung. Der Anteil und die Art der Verzerrung wird über den Eingang "Shp" kontrolliert:

Shp = 0	kein Shaping
Shp > 0	Shaping 3. Ordnung
Shp < 0	Shaping 2. Ordnung

H.19. Audio Shaper > Broken Par Sat



Kaputter parabolischer Sättiger (Saturator). Hat ein lineares Segment um den Nullpegel herum.
Der Eingang "L" bestimmt den Ausgangspegel für die "volle Sättigung" (Default = 1).
Der Eingang "H" bestimmt die Härte (Bereich 0…1). Größere Werte bedingen ein größeres lineares Segment um den Nullpegel herum.
Der Eingang "S" kontrolliert die Symmetrie der Shaping-Kurve (Bereich –1…1). Am Wert 0 ist die Kurve symmetrisch.

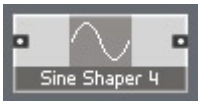
H.20. Audio Shaper > Hyperbol Sat

Einfacher hyperbolischer Sättiger. Der Eingang "L" bestimmt den Ausgangspegel für die "volle Sättigung" (Default = 1). Allerdings erreicht dieser Typ Sättiger niemals die "volle Sättigung".

H.21. Audio Shaper > Parabol Sat

Einfacher parabolischer Sättiger. Der Eingang “L” bestimmt den Ausgangspegel für die “volle Sättigung” (Default = 1).
Anmerkung: Die volle Sättigung wird bei einem Eingangspegel erreicht, der doppelt so hoch ist wie der Wert am Eingang L.

H.22. Audio Shaper > Sine Shaper 4 / 8



Sinus-Shaper 4./8. Ordnung. Der Shaper 8. Ordnung hat eine bessere Sinus-Approximation, braucht aber mehr CPU-Leistung.

H.23. Control > Ctl Amount

Ermöglicht lineare, umkehrbare Kontrolle der Amplitude des Kontroll-Signals.

A = 0	schaltet das Signal ab
A = 1	lässt das Signal unverändert
A = -1	invertiert das Signal

Typische Verwendung: Kontrolle des Modulationsgrads

H.24. Control > Ctl Amp Mod

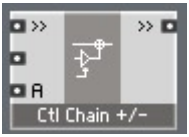
Moduliert die Amplitude des Kontroll-Signals um einen bestimmten Wert (AM) entlang einer linearen Skala.

AM = 1	verdoppelt die Amplitude
AM = 0	keine Veränderung
AM = -1	schaltet das Signal stumm

H.25. Control > Ctl Bi2Uni

Formt ein bipolares Signal mit einem Wertebereich von –1 bis1 in ein unipolares Signal um. Der Eingang “a” kontrolliert den Grad der Umformung: Bei 0 findet keine Veränderung statt, bei 1 wird das Signal zu 100% geändert (Default = 1). Typische Verwendung: Schalten Sie dieses Modul direkt hinter einen LFO, um die Polarität der Modulation einzustellen.

H.26. Control > Ctl Chain



Ändert die Amplitude des Kontroll-Signals um einen bestimmten Wert (A) und mischt das Signal dem verketteten Kontroll-Signal (>>) hinzu.

A = 0	schaltet das Signal ab
A = 1	lässt das Signal unverändert
A = -1	invertiert das Signal

Typische Verwendung: Verwenden Sie dieses Modul zur Kontrolle von Mixer-Ketten

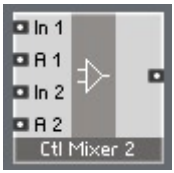
H.27. Control > Ctl Invert

Invertiert die Polarität des Mixer-Signals.

H.28. Control > Ctl Mix

Mischt zwei Kontroll-Signale.

H.29. Control > Ctl Mixer 2



Mischt zwei Kontroll-Signale (In 1, In 2) unter Verwendung bestimmter Gain-Werte (A 1, A 2) zusammen.

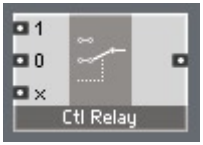
A = 0	kein Signal
A = 1	unverändertes Signal
A = -1	invertiertes Signal

H.30. Control > Ctl Pan

Platziert ein Kontroll-Signal im “Stereo-Panorama”; verwendet eine parabolische Kurve.

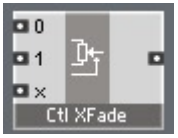
Pos = -1	ganz links
Pos = 0	Mitte
Pos = 1	ganz rechts

H.31. Control > Ctl Relay



Schaltet zwischen zwei Kontroll-Signalen um. Wenn x > 0, wird Signal 1 weitergeleitet; andernfalls wird Signal 0 weitergeleitet.

H.32. Control > Ctl XFade



Führt eine Kreuzblende (Crossfade) zwischen zwei Kontroll-Signalen aus; verwendet eine lineare Kurve.

x = 0 nur Signal 0 kann passieren
x = 0.5 beide Signale sind gleich stark im Mix vertreten
x = 1 nur Signal 1 kann passieren

H.33. Control > Par Ctl Shaper

Wendet eine doppelte parabolische Kurve auf ein Kontroll-Signal an. Das Eingangssignal muss im Bereich zwischen –1 und 1 liegen. Das Ausgangssignal liegt ebenfalls im Bereich zwischen –1 und 1. Der Grad der Beugung des Signals wird über den Eingang "b" kontrolliert (dessen Bereich ebenfalls von –1bis 1 reicht).

b = 0 keine Beugung (linearer Kurvenverlauf)
b = -1 maximal mögliche Beugung auf die x-Achse zu
b = 1 maximal mögliche Beugung auf die y-Achse zu

Sie können diesen Shaper auch für Signale verwenden, deren Wertebereich von 0 bis 1 reicht; in diesem Fall wird nur eine Hälfte der Kurve genutzt.
diesem Fall wird nur eine Hälfte der Kurve genutzt.
Typische Verwendung: Shaping von Velocity-Werten und anderen Kontroll-Signalen

H.34. Convert > dB2AF

Konvertiert ein Kontroll-Signal aus der dB-Skala in einen linearen Amplituden-Verstärkungsfaktor.s

0 dB → 1.0
-6 dB → 0.5
etc.

H.35. Convert > dP2FF

Konvertiert ein Kontroll-Signal aus einem Tonhöhen-Intervall (in Halbtönen) in ein Frequenzverhältnis.

12 Halbtöne → 2
-12 Halbtöne → -2
etc.

H.36. Convert > logT2sec

Konvertiert die auf REAKTORs Primary Level für Hüllkurven verwendete logarithmische Zeiteinteilung in Sekunden.

0 → 0.001 sec
60 → 1 sec
etc.

H.37. Convert > ms2Hz

Konvertiert einen Zeitabschnitt (in Millisekunden) in die entsprechende Frequenz in Hz. Z. B. 100ms → 10 Hz.

H.38. Convert > ms2sec

Konvertiert eine in Millisekunden angegebene Zeitdauer in Sekunden. Z. B. 500ms → 0,5 sec.

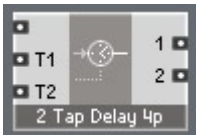
H.39. Convert > P2F

Konvertiert ein Kontroll-Signal aus der Tonhöhen-Skala in die Frequenz-Einteilung. Z. B. Note 69 → 440 Hz.

H.40. Convert > sec2Hz

Konvertiert eine Zeitdauer in Sekunden in die entsprechende Frequenz in Hz. Z. B. 0.1sec → 10 Hz.

H.41. Delay > 2 / 4 Tap Delay 4p



2/4-stufiges Delay mit 4-Punkt-Interpolation. Die Eingänge T1 bis T4 bestimmen die Delay-Zeit in Sekunden für jede der Stufen.
Die maximale Delay-Zeit liegt standardmäßig bei 44100 Samples, beträgt also 1s bei einer Sampling-Rate von 44,1kHz. Um die Delay-Zeit einzustellen, ändern Sie die Größe des Arrays im Delay-Macro.

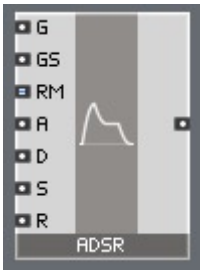
H.42. Delay > Delay 1p / 2p / 4p

Delay, das wahlweise als nicht interpoliertes 1-Punkt-Delay oder als interpoliertes 2-Punkt- respektive 4-Punkt-Delay arbeitet. Der Eingang "T" bestimmt die Delay-Zeit in Sekunden.
Die maximale Delay-Zeit liegt standardmäßig bei 44100 Samples, beträgt also 1s bei einer Sampling-Rate von 44,1kHz. Um die Delay-Zeit einzustellen, ändern Sie die Größe des Arrays im Delay-Macro.
Verwenden Sie die interpolierten Delay-Versionen für modulierte Delays. Für nicht modulierte (mit festen Zeiten versehene) Delays eignen sich nicht interpolierte Delay-Varianten normalerweise besser.

H.43. Delay > Diff Delay 1p / 2p / 4p

Diffusions-Delay, das wahlweise als nicht interpoliertes 1-Punkt-Delay oder als interpoliertes 2-Punkt- respektive 4-Punkt-Delay arbeitet. Der Eingang "T" bestimmt den Diffusionsfaktor.
Die maximale Delay-Zeit liegt standardmäßig bei 44100 Samples, beträgt also 1s bei einer Sampling-Rate von 44,1kHz. Um die Delay-Zeit einzustellen, ändern Sie die Größe des Arrays im Delay-Macro.

H.44. Envelope > ADSR



Erzeugt eine ADSR-Hüllkurve.

A, D, R bestimmen die Zeiten für Attack, Decay und Release in Sekunden

S bestimmt den Sustain-Pegel (im Bereich von 0 bis 1; bei 1 ist der Sustain-Pegel gleich dem Spitzenpegel)

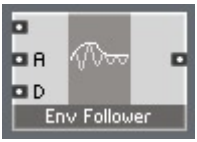
G Gate-Eingang. Positive ankommende Events starten die Hüllkurve (neu). Events mit dem Wert 0 oder mit negativen Werten schließen die Hüllkurve.

GS Gate-Empfindlichkeit. Bei einer Empfindlichkeit von 0 hat der Spitzenpegel der Hüllkurve immer eine Amplitude von 1. Bei einer Empfindlichkeit von 1ist der Spitzenpegel gleich dem positiven Gate-Pegel.

RM Retrigger-Modus. Wählt zwischen analogem und digitalem Modus und zwischen Retrigger- und Legato-Modus. Im “digitalen” Modus startet die Hüllkurve immer bei Null neu, während sie im “analogen” Modus immer bei ihrem aktuellen Ausgangspegel startet. Im “Retrigger”-Modus starten nachfolgende positive Gate-Events die Hüllkurve neu, während die Hüllkurve im “Legato”-Modus nur dann neu startet, wenn das Gate seinen Wert von “negativ” oder “Null” in “positiv” ändert. Die erlaubten Werte für den Parameter “RM” sind:

RM = 0 Analog-Retrigger (Default)
RM = 1 Analog-Legato
RM = 2 Digital-Retrigger
RM = 3 Digital-Legato

H.45. Envelope > Env Follower



Erzeugt ein Kontroll-Signal, das der Hüllkurve des ankommenden Audio-Signals “folgt”. Die Eingänge “A” und “D” bestimmen die Attack- und Decay-Zeiten des “Verfolger-Signals” in Sekunden.

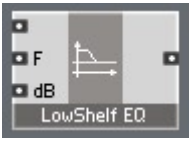
H.46. Envelope > Peak Detector

Gibt den letzten Spitzenpegel (Peak) des ankommenden Audio-Signals als Kontroll-Signal aus. Der Eingang “D” bestimmt die Decay-Zeit für den Ausgangspegel in Sekunden.

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

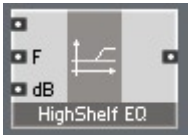
1-Pol-Tiefpass-/ Hochpass-EQ (6 dB/Oktave). Der Eingang “F” bestimmt die Cutoff-Frequenz (in Hz) sowohl für den Ausgang “LP” als auch für den Ausgang “HP”.

H.48. EQ > 6dB LowShelf EQ



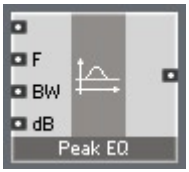
1-Pol-Low-Shelving-EQ. Der Eingang “dB” bestimmt die Stärke der Anhebung tiefer Frequenzen in dB (negative Werte beschneiden die tiefen Frequenzen), der Eingang “F” bestimmt die Mitten-Übergangsfrequenz in Hz.

H.49. EQ > 6dB HighShelf EQ



1-Pol-High-Shelving -EQ. Der Eingang “dB” bestimmt die Stärke der Anhebung hoher Frequenzen in dB (negative Werte beschneiden die hohen Frequenzen), der Eingang “F” bestimmt die Mitten-Übergangsfrequenz in Hz.

H.50. EQ > Peak EQ

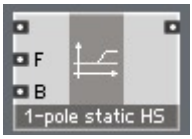


2-Pol-Peak- bzw. Notch-EQ. Der Eingang “F” bestimmt die Mittenfrequenz in Hz, der Eingang “BW” bestimmt die Bandbreite des Equalizers in Oktaven und der Eingang “dB” bestimmt die Höhe des Spitzenpegels (Peak). Negative Werte am Eingang “dB” erzeugen eine Kerbe (Notch).

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

1-poliges statisches Hochpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz.

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

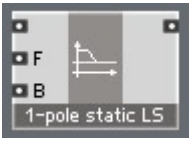


1-poliges statisches High-Shelving-Filter. Der Eingang “F” bestimmt die Cut-off-Frequenz in Hz, der Eingang “B” bestimmt die Stärke der Anhebung der hohen Frequenzen in dB.

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

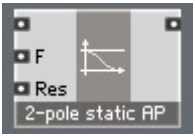
1-poliges statisches Tiefpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz.

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



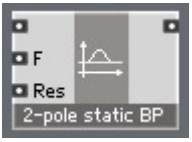
1-poliges statisches Low-Shelving-Filter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “B” bestimmt die Stärke der Anhebung der tiefen Frequenzen in dB.

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



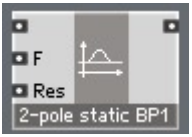
2-poliges statisches Allpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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



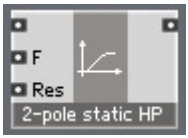
2-poliges statisches Bandpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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



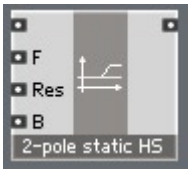
2-poliges statisches Bandpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1). Die Verstärkung an der Cutoff-Frequenz ist immer 1, ungeachtet der Resonanz.

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



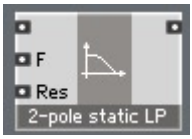
2-poliges statisches Hochpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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



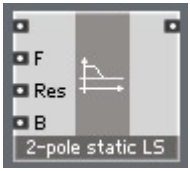
2-poliges statisches Hochpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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



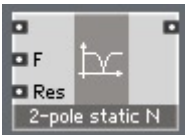
2-poliges statisches Tiefpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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



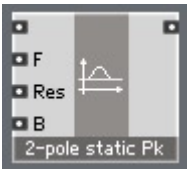
22-poliges statisches Low-Shelving-Filter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1). Der Eingang “B” bestimmt die Stärke der Anhebung der tiefen Frequenzen in dB.

H.62. EQ > Static Filter > 2-pole static N



2-poliges statisches Notch-Filter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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



2-poliges statisches Peak-Filter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1). Der Eingang “B” bestimmt die Stärke der Anhebung an der Mittenfrequenz in dB.

H.64. EQ > Static Filter > Integrator

Integriert das ankommende Audio-Signal unter Verwendung der Rechtecksummen-Methode. Ein Event am Eingang “Rst” setzt den Ausgang des Moduls auf den Wert dieses Events zurück.

H.65. Event Processing > Accumulator

Berechnet die Summe der Werte am oberen Eingang. Ein Event am Eingang "Rst" setzt den Ausgang des Moduls auf den Wert dieses Events zurück. Der untere Ausgangs-Wert ist die Summe aller vorangegangenen Events, der obere Ausgangs-Wert ist die Summe aller vorangegangenen Events mit Ausnahme des letzten.

H.66. Event Processing > Clk Div

Clock-Frequenzteiler. Die am oberen Eingang ankommenden Clock-Events werden so gefiltert, dass nur die Events (1), N+(1), 2N+(1) und so weiter durchgelassen werden. N ist der Wert des unteren Eingangs und bestimmt das Teilungsverhältnis.

H.67. Event Processing > Clk Gen

Erzeugt Clock-Events mit der vom Eingang (in Hz) festgelegten Rate. Dieses Modul arbeitet nur innerhalb von Audio-Core-Cells.

H.68. Event Processing > Clk Rate

Schätzt die Rate und die Dauer ankommender Clock-Events. Der Ausgang "F" sendet die Rate in Hz, der Ausgang "T" die Dauer in Sekunden. Dieses Modul arbeitet nur innerhalb von Audio-Core-Cells.
Der anfängliche Wert für die Dauer ist Null, die Rate hat einen extrem großen Wert. Ein brauchbares Ergebnis erhalten Sie erst nach dem zweiten Clock-Event.

H.69. Event Processing > Counter

Zählt die Anzahl der Events am oberen Eingang. Ein Event am Eingang "Rst"

setzt den Ausgang auf den Wert dieses Events zurück. Der untere Ausgangs-Wert ist das Zählungsergebnis aller vorangegangenen Events, der untere Ausgangs-Wert ist das Zählungsergebnis aller vorangegangenen Events mit Ausnahme des letzten.

H.70. Event Processing > Ctl2Gate

Konvertiert ein Kontroll-Signal (oder Audio-Signal) am oberen Eingang in ein Gate-Signal mit der Amplitude, die der untere Eingang bestimmt. Positive Nulldurchgänge öffnen das Gate, negative Nulldurchgänge schließen das Gate.

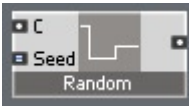
H.71. Event Processing > Dup Flt / IDup Flt

Filtert Events mit doppelten Werten (Duplikate) aus (nur Events mit Werten, die sich von denen ihrer Vorgänger unterscheiden, werden durchgelassen).

H.72. Event Processing > Impulse

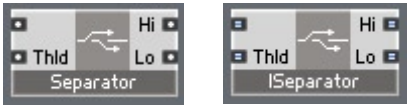
Erzeugt einen ein Sample langen Impuls mit der Amplitude 1 als Reaktion auf ein ankommendes Event. Dieses Modul arbeitet nur innerhalb von Audio-Core-Cells.

H.73. Event Processing > Random



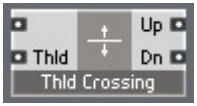
Erzeugt zufällige Zahlen als Reaktion auf ankommende Clock-Events. Der Wertebereich der Ausgabe reicht von –1 bis 1. Ein Event am Eingang “Seed” wird den Zufallsgenerator mit dem Wert dieses Events neu “besäen”.

H.74. Event Processing > Separator / ISeparator



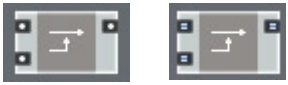
Am oberen Eingang ankommende Events mit Werten oberhalb des über den Eingang “Thld” bestimmten Werts werden an den Ausgang “Hi” geleitet. Alle anderen Events werden an den Ausgang “Lo” geleitet.

H.75. Event Processing > Thld Crossing



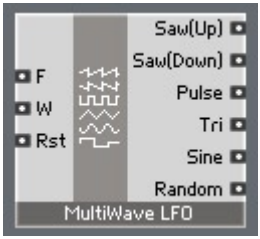
Jedes Mal, wenn ein ansteigendes Signal am oberen Eingang die vom Eingang “Thld” definierte Schwelle überschreitet, wird ein Event vom Ausgang “Up” gesendet. Wenn ein fallendes Signal die Schwelle überschreitet, wird ein Event vom Ausgang “Dn” gesendet.

H.76. Event Processing > Value / IValue



Ändert den Wert eines am oberen Eingang ankommenden Events auf den zu diesem Zeitpunkt am unteren Eingang anliegenden Wert.

H.77. LFO > MultiWave LFO



Erzeugt verschiedene phasenstarre Niederfrequenz-Wellenformen gleichzeitig. Der Eingang “F” bestimmt die Rate in Hz, der Eingang “W” kontrolliert die Pulsweite (im Bereich von –1 bis 1, betrifft nur den Ausgang “Pulse”). Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

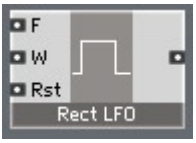
H.78. LFO > Par LFO

Erzeug ein parabolisches Niederfrequenz-Kontroll-Signal. Der Eingang “F” bestimmt die Rate in Hz. Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

H.79. LFO > Random LFO

Erzeugt ein gestuftes zufälliges Niederfrequenz-Kontroll-Signal. (“zufälliges Sample-and-Hold”). Der Eingang “F” bestimmt die Rate in Hz. Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

H.80. LFO > Rect LFO



Erzeugt ein rechteckiges Niederfrequenz-Kontroll-Signal. Der Eingang “F” bestimmt die Rate in Hz, der Eingang “W” die Pulsweite (im Bereich von –1 bis 1). Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

H.81. LFO > Saw(down) LFO

Erzeugt ein Niederfrequenz-Kontroll-Signal in Form einer fallenden Sägezahn-Welle. Der Eingang “F” bestimmt die Rate in Hz. Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

H.82. LFO > Saw(up) LFO

Erzeugt ein Niederfrequenz-Kontroll-Signal in Form einer steigenden Sägezahn-Welle. Der Eingang “F” bestimmt die Rate in Hz. Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

H.83. LFO > Sine LFO

Erzeugt ein sinusförmiges Niederfrequenz-Kontroll-Signal. Der Eingang “F” bestimmt die Rate in Hz. Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

H.84. LFO > Tri LFO

Erzeugt ein Niederfrequenz-Kontroll-Signal in Form einer Dreieck-Welle. Der Eingang “F” bestimmt die Rate in Hz. Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

H.85. Logic > AND

Verknüpft zwei logische Signale durch eine Konjunktion. Die Ausgabe ist nur dann 1, wenn beide Eingänge den Wert 1 haben; für andere Eingangs-Werte als 0 und 1 ist das Ergebnis undefiniert.

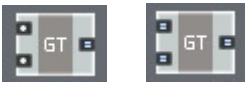
H.86. Logic > Flip Flop

Der Ausgangs-Wert springt bei jedem am Clock-Eingang empfangenen Event zwischen den Werten 0 und 1 hin und her (auf den jeweils anderen Wert).

H.87. Logic > Gate2L

Konvertiert ein Gate-Signal in ein logisches Signal. Ein offenes Gate erzeugt den Ausgangs-Wert 1, ein geschlossenes Gate erzeugt den Ausgangs-Wert 0.

H.88. Logic > GT / IGT



Vergleicht die beiden ankommenden Werte (Version GT: Fließkomma; Version IGT: Integer) und gibt den Wert 1 aus, wenn der Wert am oberen Eingang größer als der Wert am unteren Eingang ist; andernfalls wird 0 ausgegeben.

H.89. Logic > EQ

Vergleicht die beiden an den Eingängen ankommenden Integer-Werte und gibt den Wert 1 aus, wenn beide Eingangs-Werte gleich sind; andernfalls lautet die Ausgabe 0.

H.90. Logic > GE

Vergleicht die beiden an den Eingängen ankommenden Integer-Werte und gibt den Wert 1 aus, wenn der obere Wert im Verhältnis zum unteren Wert größer oder gleich ist; andernfalls lautet die Ausgabe 0.

H.91. Logic > L2Clock

Konvertiert ein Logik-Signal in ein Clock-Signal. Das Umschalten des Eingangssignals von 0 auf 1 sendet das Clock-Event. Für andere Eingangs-Werte als 0 und 1 ist die Funktion undefiniert.

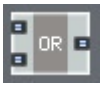
H.92. Logic > L2Gate

Konvertiert ein Logik-Signal in ein Gate-Signal. Das Umschalten des Eingangssignals von 0 auf 1 öffnet das Gate, bim Umschalten von 1 auf 0 wird das Gate geschlossen. Der Pegel für das offene Gate wird von dem Wert am unteren Eingang bestimmt (Default = 1). Für andere Eingangs-Werte als 0 und 1 ist die Funktion undefiniert.

H.93. Logic > NOT

Konvertiert 1 in 0 und vice versa. Für andere Eingangs-Werte als 0 und 1 ist das Ergebnis undefiniert.

H.94. Logic > OR

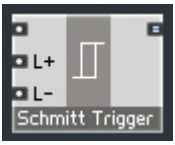


Führt eine Trennung (Disjunktion) zweier logischer Signale durch. Der Ausgang nimmt den Wert 1 an, wenn mindestens einer der Eingänge den Wert 1 hat. Für andere Eingangs-Werte als 0 und 1 ist das Ergebnis undefiniert.

H.95. Logic > XOR

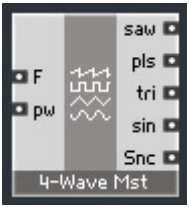
Führt eine exklusive Trennung (exklusive Disjunktion) zweier logischer Signale durch. Der Ausgang nimmt den Wert 1 an, wenn eins der beiden Eingangs -Signale den Wert 1 und das andere den Wert 0 hat. Für andere Eingangs-Werte als 0 und 1 ist das Ergebnis undefiniert.

H.96. Logic > Schmitt Trigger



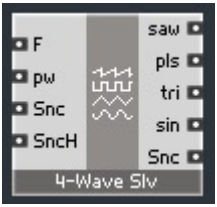
Schaltet den Ausgang auf 1, wenn der Eingangs-Wert größer wird als L+ (Default = 0.67); schaltet den Ausgang auf 0, wenn der Eingangs-Wert weniger als L- (Default = 0.33) wird.

H.97. Oscillators > 4-Wave Mst



Erzeugt 4 phasenstarre Audio-Wellenformen. Der Eingang "F" bestimmt die Frequenz in Hz. Der Eingang "PW" bestimmt die Pulsweite (im Bereich von –1 bis 1; betrifft nur den Ausgang "pls"). Dieser Oszillator kann mit negativen Frequenzen schwingen und bietet zusätzlich einen Synchronisations-Ausgang für das Oszillator-Modul 4 Wave Slv.

H.98. Oscillators > 4-Wave Slv



Erzeugt 4 phasenstarre Audio-Wellenformen. Der Eingang "F" bestimmt die Frequenz in Hz. Der Eingang "PW" bestimmt die Pulsweite (im Bereich von –1 bis 1; betrifft nur den Ausgang "pls").

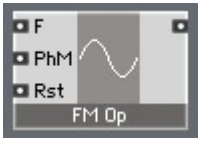
H.99. Oscillators > Binary Noise

Binärer Rauschgenerator, der Weißes Rauschen erzeugt. Gibt zufällig alter-nierend die Werte 1 und –1 aus. Ein am Eingang “Seed” ankommendes Event initialisiert den internen Zufallsgenerator mit einem bestimmten Seed-Wert (neu).

H.100. Oscillators > Digital Noise

Digitaler Rauschgenerator, der Weißes Rauschen erzeugt. Gibt zufällige Werte im Bereich zwischen –1 und 1 aus. Ein am Eingang “Seed” ankommendes Event initialisiert den internen Zufallsgenerator mit einem bestimmten Seed-Wert (neu).

H.101. Oscillators > FM Op

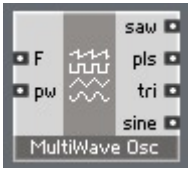


Klassischer FM-Operator. Gibt eine Sinuswelle aus, deren Frequenz (in Hz) vom Eingang “F” bestimmt wird. Der Sinus kann über den Eingang “PhM” phasenmoduliert werden (in Radianten).Ein am Eingang “Rst” ankommendes Event startet den Oszillator an der durch den Wert des Events (im Bereich von 0 bis 1) spezifizierten Stelle neu.

H.102. Oscillators > Formant Osc

Erzeugt eine Wellenform mit einer Grundfrequenz, die über den Eingang “F” festgelegt wird (in Hz); die Frequenz der Oberschwingungen (Formanten) wird über den Eingang “Fmt” bestimmt.

H.103. Oscillators > MultiWave Osc



Erzeugt 4 phasenstarre Audio-Wellenformen. Der Eingang “F” bestimmt die Frequenz in Hz. Der Eingang “PW” bestimmt die Pulsweite (im Bereich von –1 bis 1; betrifft nur den Ausgang “pls”).
Dieser Oszillator kann nicht mit negativen Frequenzen schwingen.

H.104. Oscillators > Par Osc

Erzeugt eine parabolische Audio-Wellenform. Der Eingang “F” bestimmt die Frequenz in Hz.

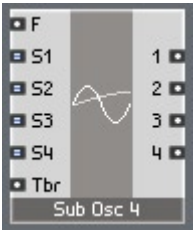
H.105. Oscillators > Quad Osc

Erzeugt ein Paar phasenstarrer Sinus-Wellenformen mit einem Phasenversatz von 90 Grad. Der Eingang “F” bestimmt die Frequenz in Hz.

H.106. Oscillators > Sin Osc

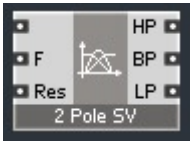
Erzeugt eine Sinuswelle. Der Eingang “F” bestimmt die Frequenz in Hz.

H.107. Oscillators > Sub Osc 4



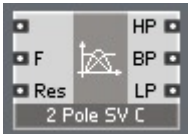
Erzeugt 4 phasenstarre Subharmonische. Die “grundlegende” Frequenz wird vom Eingang “F” bestimmt (in Hz). Die Nummern der Subharmonischen werden von den Eingängen S1 bis S4 spezifiziert (im Bereich zwischen 1 und 120). Der Eingang “Tbr” kontrolliert den harmonischen Gehalt der Ausgangs-Wellenform (im Bereich zwischen 0 und 1).

H.108. VCF > 2 Pole SV



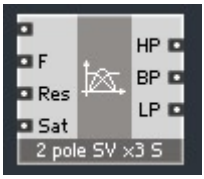
2-poliges Zustandsvariablenfilter (State Variable Filter). Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 0,98).
Die Ausgänge “HP”, “BP” und “LP” erzeugen Hochpass-, Bandpass- respektive Tiefpass-Signale.

H.109. VCF > 2 Pole SV C



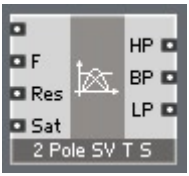
2-poliges Zustandsvariablenfilter (kompensierte Version). Bietet ein verbessertes Verhalten bei hohen Cutoff-Einstellungen. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 0,98). Sie können auch negative Resonanz-Werte verwenden, die das Abfallen des Signals weiter verwischen.
Die Ausgänge “HP”, “BP” und “LP” erzeugen Hochpass-, Bandpass- respektive Tiefpass-Signale.

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



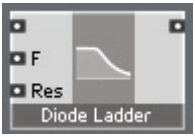
2-poliges Zustandsvariablenfilter mit optionalem Oversampling (Version x3) und Sättigung. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 1). Der Eingang “Sat” bestimmt den Grad der Sättigung (typischer Bereich: 8 bis 32).
Die Ausgänge “HP”, “BP” und “LP” erzeugen Hochpass-, Bandpass- respektive Tiefpass-Signale.

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



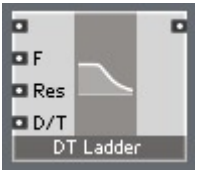
2-poliges Zustandsvariablenfilter mit Tabellen-Kompensation und optionaler Sättigung (Version S). Bietet ebenfalls ein verbessertes Verhalten bei hohen Cutoff-Einstellungen, klingt aber etwas anders als das Modul *2 Pole SV C.* Der Eingang "F" bestimmt die Cutoff-Frequenz in Hz, der Eingang "Res" legt die Resonanz fest (im Bereich von 0 bis 1). Der Eingang "Sat" bestimmt den Grad der Sättigung (typischer Bereich: 8 bis 32).
Die Ausgänge "HP", "BP" und "LP" erzeugen Hochpass-, Bandpass- respektive Tiefpass-Signale.

H.112. VCF > Diode Ladder



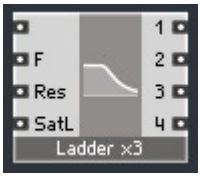
Lineare Nachbildung eines Dioden-Leiter-Filters. Der Eingang "F" bestimmt die Cutoff-Frequenz in Hz, der Eingang "Res" legt die Resonanz fest (im Bereich von 0 bis 0,98).

H.113. VCF > D/T Ladder



Lineare Leiter-Filter-Nachbildung, die stufenlos zwischen Dioden- und Transistor-Leiter-Verhalten überblendet werden kann. Der Eingang "F" bestimmt die Cutoff-Frequenz in Hz, der Eingang "Res" legt die Resonanz fest (im Bereich von 0 bis 0,98). Der Eingang "D/T" blendet zwischen Diode und Transistor über (0 = Diode, 1 = Transistor)

H.114. VCF > Ladder x3



Nachbildung eines Transistor-Leiter-Filters mit dreifachem Oversampling und Sättigung. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 1). Der Eingang “SatL” bestimmt den Grad der Sättigung (typischer Bereich: 1 bis 32).
Die Ausgänge 1 bis 4 führen die Signale der entsprechenden Stufen der emulierten “Leiter”. Verwenden Sie die 4. Stufe für den klassischern Leiter-Filter-Sound.

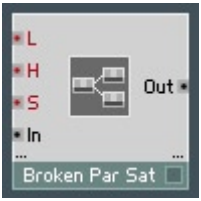
Appendix I. Core cell library

I.1. Audio Shaper > 3-1-2 Shaper

Audio-Signal-Shaper mit veränderlichem Anteilen von Verzerrungen 2. und 3. Ordnung. Der Anteil und die Art der Verzerrung wird über den Eingang “Shp” kontrolliert:

Shp = 0	kein Shaping
Shp > 0	Shaping 3. Ordnung
Shp < 0	Shaping 2. Ordnung

I.2. Audio Shaper > Broken Par Sat



Kaputter parabolischer Sättiger (Saturator). Hat ein lineares Segment um den Nullpegel herum.
Der Eingang “L” bestimmt den Ausgangspegel für die “volle Sättigung” (Default = 1).
Der Eingang ”H” bestimmt die Härte (Bereich 0…1). Größere Werte bedingen ein größeres lineares Segment um den Nullpegel herum.
Der Eingang ”S” kontrolliert die Symmetrie der Shaping-Kurve (Bereich –1…1). Am Wert 0 ist die Kurve symmetrisch.

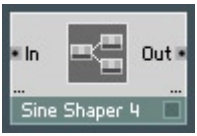
I.3. Audio Shaper > Hyperbol Sat

Einfacher hyperbolischer Sättiger. Der Eingang “L” bestimmt den Ausgangspegel für die “volle Sättigung” (Default = 1). Allerdings erreicht dieser Typ Sättiger niemals die “volle Sättigung”.

I.4. Audio Shaper > Parabol Sat

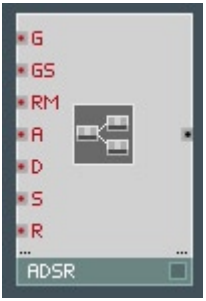
Einfacher parabolischer Sättiger. Der Eingang “L” bestimmt den Ausgangspegel für die “volle Sättigung” (Default = 1).Anmerkung: Die volle Sättigung wird bei einem Eingangspegel erreicht, der doppelt so hoch ist wie der Wert am Eingang L.

I.5. Audio Shaper > Sine Shaper 4/8



Sinus-Shaper 4./8. Ordnung. Der Shaper 8. Ordnung hat eine bessere Sinus-Approximation, braucht aber mehr CPU-Leistung.

I.6. Control > ADSR



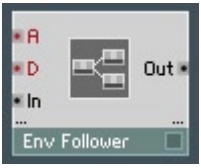
Erzeugt eine ADSR-Hüllkurve.

- A, D, R bestimmen die Zeiten für Attack, Decay und Release in Sekunden
- S bestimmt den Sustain-Pegel (im Bereich von 0 bis 1; bei 1 ist der Sustain-Pegel gleich dem Spitzenpegel)
- G Gate-Eingang. Positive ankommende Events starten die Hüllkurve (neu). Events mit dem Wert 0 oder mit negativen Wertenschließen die Hüllkurve.
- GS Gate-Empfindlichkeit. Bei einer Empfindlichkeit von 0 hat der Spitzenpegel der Hüllkurve immer eine Amplitude von 1. Bei einer Empfindlichkeit von 1ist der Spitzenpegel gleich dem positiven Gate-Pegel.

RM Retrigger-Modus. Wählt zwischen analogem und digitalem Modus und zwischen Retrigger- und Legato-Modus. Im “digitalen” Modus startet die Hüllkurve immer bei Null neu, während sie im “analogen” Modus immer bei ihrem aktuellen Ausgangspegel startet. Im “Retrigger”-Modus starten nachfolgende positive Gate-Events die Hüllkurve neu, während die Hüllkurve im “Legato”-Modus nur dann neu startet, wenn das Gate seinen Wert von “negativ” oder “Null” in “positiv” ändert. Die erlaubten Werte für den Parameter “RM” sind:

RM = 0 Analog-Retrigger (Default)
RM = 1 Analog-Legato
RM = 2 Digital-Retrigger
RM = 3 Digital-Legato

I.7. Control > Env Follower

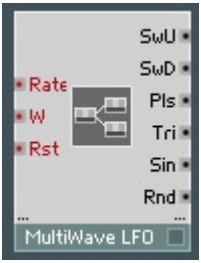


Erzeugt ein Kontroll-Signal, das der Hüllkurve des ankommenden Audio-Signals “folgt”. Die Eingänge “A” und “D” bestimmen die Attack- und Decay-Zeiten des “Verfolger-Signals” in Sekunden..

I.8. Control > Flip Flop

Der Ausgangs-Wert springt bei jedem am Clock-Eingang empfangenen Event zwischen den Werten 0 und 1 hin und her (auf den jeweils anderen Wert).

I.9. Control > MultiWave LFO



Erzeugt verschiedene phasenstarre Niederfrequenz-Wellenformen gleichzeitig. Der Eingang “F” bestimmt die Rate in Hz, der Eingang “W” kontrolliert die Pulsweite (im Bereich von –1 bis 1, betrifft nur den Ausgang “Pulse”). Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

I.10. Control > Par Ctl Shaper

Wendet eine doppelte parabolische Kurve auf ein Kontroll-Signal an. Das Eingangssignal muss im Bereich zwischen –1 und 1 liegen. Das Ausgangssignal liegt ebenfalls im Bereich zwischen –1 und 1. Der Grad der Beugung des Signals wird über den Eingang “b” kontrolliert (dessen Bereich ebenfalls von –1bis 1 reicht).

b = 0	keine Beugung (linearer Kurvenverlauf)
b = -1	maximal mögliche Beugung auf die x-Achse
b = 1	maximal mögliche Beugung auf die y-Achse zu

Sie können diesen Shaper auch für Signale verwenden, deren Wertebereich von 0 bis 1 reicht; in diesem Fall wird nur eine Hälfte der Kurve genutzt.
Typische Verwendung: Shaping von Velocity-Werten und anderen Kontroll-Signalen

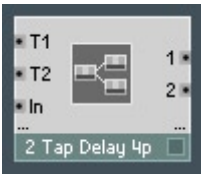
I.11. Control > Schmitt Trigger

Schaltet den Ausgang auf 1, wenn der Eingangs-Wert größer wird als L+ (Default = 0.67); schaltet den Ausgang auf 0, wenn der Eingangs-Wert weniger als L- (Default = 0.33) wird.

I.12. Control > Sine LFO

Erzeugt ein sinusförmiges Niederfrequenz-Kontroll-Signal. Der Eingang “F” bestimmt die Rate in Hz. Am Eingang “Rst” ankommende Events starten den LFO in der durch den Event-Wert (im Bereich von 0 bis 1) bestimmten Phase neu.

I.13. Delay > 2/4 Tap Delay 4p



2/4-stufiges Delay mit 4-Punkt-Interpolation. Die Eingänge T1 bis T4 bestimmen die Delay-Zeit in Sekunden für jede der Stufen.
Die maximale Delay-Zeit liegt standardmäßig bei 44100 Samples, beträgt also 1s bei einer Sampling-Rate von 44,1kHz. Um die Delay-Zeit einzustellen, ändern Sie die Größe des Arrays im Delay-Macro.

I.14. Delay > Delay 4p

Interpoliertes 4-Punkt-Delay. Der Eingang “T” bestimmt die Delay-Zeit in Millisekunden.
Die maximale Delay-Zeit liegt standardmäßig bei 44100 Samples, beträgt also 1s bei einer Sampling-Rate von 44,1kHz. Um die Delay-Zeit einzustellen, ändern Sie die Größe des Arrays im Delay-Macro.

I.15. Delay > Diff Delay 4p

Interpoliertes 4-Punkt-Diffusions-Delay. Der Eingang “T” bestimmt die Delay-Zeit in Millisekunden. Der Eingang “Dffs” bestimmt den Diffusionsfaktor.
Die maximale Delay-Zeit liegt standardmäßig bei 44100 Samples, beträgt also 1s bei einer Sampling-Rate von 44,1kHz. Um die Delay-Zeit einzustellen, ändern Sie die Größe des Arrays im Delay-Macro.

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

1-Pol-Tiefpass-/ Hochpass-EQ (6 dB/Oktave). Der Eingang “F” bestimmt die Cutoff-Frequenz (in Hz).

I.17. EQ > HighShelf EQ

1-Pol-High-Shelving -EQ. Der Eingang “dB” bestimmt die Stärke der Anhebung hoher Frequenzen in dB (negative Werte beschneiden die hohen Frequenzen), der Eingang “F” bestimmt die Mitten-Übergangsfrequenz in Hz.

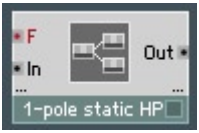
I.18. EQ > LowShelf EQ

1-Pol-Low-Shelving-EQ. Der Eingang “dB” bestimmt die Stärke der Anhebung tiefer Frequenzen in dB (negative Werte beschneiden die tiefen Frequenzen), der Eingang “F” bestimmt die Mitten-Übergangsfrequenz in Hz.

I.19. EQ > Peak EQ

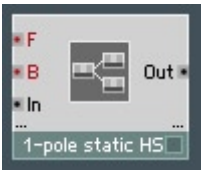
2-Pol-Peak- bzw. Notch-EQ. Der Eingang “F” bestimmt die Mittenfrequenz in Hz, der Eingang “BW” bestimmt die Bandbreite des Equalizers in Oktaven und der Eingang “dB” bestimmt die Höhe des Spitzenpegels (Peak). Negative Werte am Eingang “dB” erzeugen eine Kerbe (Notch).

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



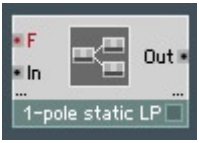
1-poliges statisches Hochpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz.

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



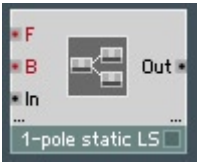
1-poliges statisches High-Shelving-Filter. Der Eingang “F” bestimmt die Cut-off-Frequenz in Hz, der Eingang “B” bestimmt die Stärke der Anhebung der hohen Frequenzen in dB.

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



1-poliges statisches Tiefpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz.

I.23. EQ > Static Filter > 1-pole static LS



1-poliges statisches Low-Shelving-Filter. Der Eingang “F” bestimmt die Cut-off-Frequenz in Hz, der Eingang “B” bestimmt die Stärke der Anhebung der tiefen Frequenzen in dB.

I.24. EQ > Static Filter > 2-pole static AP

2-poliges statisches Allpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

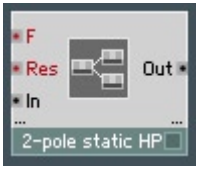
I.25. EQ > Static Filter > 2-pole static BP

2-poliges statisches Bandpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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

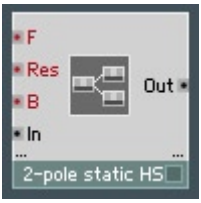
2-poliges statisches Bandpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1). Die Verstärkung an der Cutoff-Frequenz ist immer 1, ungeachtet der Resonanz.

I.27. EQ > Static Filter > 2-pole static HP



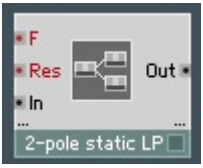
2-poliges statisches Hochpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

I.28. EQ > Static Filter > 2-pole static HS



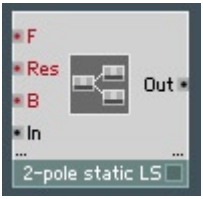
2-poliges statisches High-Shelving-Filter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1). Der Eingang “B” bestimmt die Stärke der Anhebung der hohen Frequenzen in dB.

I.29. EQ > Static Filter > 2-pole static LP



2-poliges statisches Tiefpassfilter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

I.30. EQ > Static Filter > 2-pole static LS

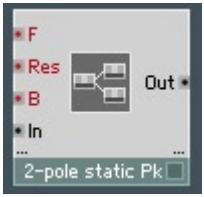


2-poliges statisches Low-Shelving-Filter. Der Eingang “F” bestimmt die Cut-off-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1). Der Eingang “B” bestimmt die Stärke der Anhebung der tiefen Frequenzen in dB.

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

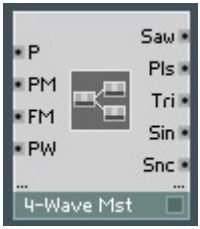
2-poliges statisches Notch-Filter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1).

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



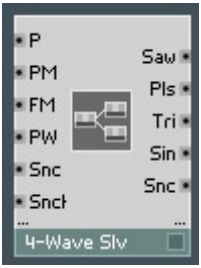
2-poliges statisches Peak-Filter. Der Eingang “F” bestimmt die Cutoff-Frequenz in Hz, der Eingang “Res” bestimmt die Resonanz (im Bereich von 0 bis 1). Der Eingang “B” bestimmt die Stärke der Anhebung an der Mittenfrequenz in dB.

I.33. Oscillator > 4-Wave Mst



Erzeugt 4 phasenstarre Audio-Wellenformen. Die Tonhöhe des Oszillators wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert. Der Eingang “F” bestimmt die Frequenz in Hz. Der Eingang “PW” bestimmt die Pulsweite (im Bereich von –1 bis 1; betrifft nur den Ausgang “pls”). Kann über den Eingang “PM” (in Halbtonschritten, exponentiell) und über den Eingang “FM” (in Hz, linear) moduliert werden. Der Eingang “pw” bestimmt die Pulsweite der Puls-Wellenform (im Bereich zwischen – 1 und 1).
Dieser Oszillator kann mit negativen Frequenzen schwingen und bietet zusätzlich einen Synchronisations-Ausgang für das Oszillator-Modul 4 Wave Slv.

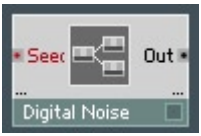
I.34. Oscillator > 4-Wave Slv



Erzeugt 4 phasenstarre Audio-Wellenformen. Die Tonhöhe des Oszillators wird über den Eingang "P" durch eine MIDI-Noten-Nummer spezifiziert. Der Eingang "PW" bestimmt die Pulsweite (im Bereich von –1 bis 1; betrifft nur den Ausgang "pls"). Kann über den Eingang "PM" (in Halbtonschritten, exponentiell) und über den Eingang "FM" (in Hz, linear) moduliert werden. Der Eingang "pw" bestimmt die Pulsweite der Puls-Wellenform (im Bereich zwischen – 1 und 1).

Dieser Oszillator kann mit negativen Frequenzen schwingen und kann mit einem anderen Oszillator-Modul der Typen 4 Wave Mst oder 4 Wave Slv synchronisiert werden. Der Eingang "SncH" kontrolliert die Strenge der Synchronisation (0 = kein Sync, 1 = Hard Sync, 0…1 = verschiedene Abstufungen der Synchronisations-Strenge). Zum Synchronisieren eines weiteren Moduls des Typs 4 Wave Slv steht der Ausgang "Snc" zur Verfügung.

I.35. Oscillator > Digital Noise



Digitaler Rauschgenerator, der Weißes Rauschen erzeugt. Gibt zufällige Werte im Bereich zwischen –1 und 1 aus. Ein am Eingang "Seed" ankommendes Event initialisiert den internen Zufallsgenerator mit einem bestimmten Seed-Wert (neu).

I.36. Oscillator > FM Op

Klassischer FM-Operator. Gibt eine Sinuswelle aus, deren Tonhöhe über den Eingang “P” bestimmt wird (als MIDI-Noten-Nummer). Der Sinus kann über den Eingang “PhM” phasenmoduliert werden (in Radianten).Ein am Eingang “Rst” ankommendes Event startet den Oszillator in der durch den Wert des Events (im Bereich von 0 bis 1) spezifizierten Phase neu.

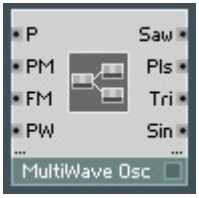
I.37. Oscillator > Formant Osc

Erzeugt eine Wellenform mit einer Grundfrequenz, die über den Eingang “P” bestimmt wird (als MIDI-Noten-Nummer); die Frequenz der Oberschwingungen (Formanten) wird über den Eingang “Fmt” bestimmt.

I.38. Oscillator > Impulse

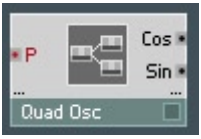
Erzeugt einen ein Sample langen Impuls mit der Amplitude 1 als Reaktion auf ein ankommendes Event.

I.39. Oscillator > MultiWave Osc



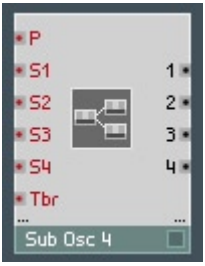
Erzeugt 4 phasenstarre Audio-Wellenformen. Die Tonhöhe des Oszillators wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert. Kann über den Eingang “PM” (in Halbtonschritten, exponentiell) und über den Eingang “FM” (in Hz, linear) moduliert werden. Der Eingang “pw” bestimmt die Pulsweite der Puls-Wellenform (im Bereich zwischen – 1 und 1).
Dieser Oszillator kann nicht mit negativen Frequenzen schwingen.

I.40. Oscillator > Quad Osc



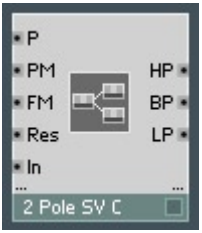
Erzeugt ein Paar phasenstarrer Sinus-Wellenformen mit einem Phasenversatz von 90 Grad. Die Tonhöhe des Oszillators wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert.

I.41. Oscillator > Sub Osc



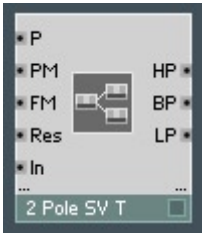
Erzeugt 4 phasenstarre Subharmonische. Die “grundlegende” Frequenz wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert. Die Nummern der Subharmonischen werden von den Eingängen S1 bis S4 spezifiziert (im Bereich zwischen 1 und 120). Der Eingang “Tbr” kontrolliert den harmonischen Gehalt der Ausgangs-Wellenform (im Bereich zwischen 0 und 1).

I.42. VCF > 2 Pole SV C



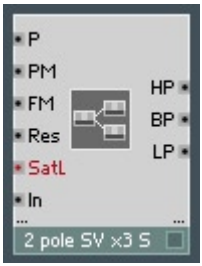
2-poliges Zustandsvariablenfilter (kompensierte Version). Bietet ein verbessertes Verhalten bei hohen Cutoff-Einstellungen. Die Cutoff-Frequenz wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert und kann über den Eingang “PM” (in Halbtonschritten, exponentiell) und über den Eingang “FM” (in Hz, linear) moduliert werden. Der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 0,98). Sie können auch negative Resonanz-Werte verwenden, die das Abfallen des Signals weiter verwischen.
Die Ausgänge “HP”, “BP” und “LP” erzeugen Hochpass-, Bandpass- respektive Tiefpass-Signale.

I.43. VCF > 2 Pole SV T



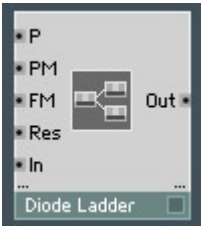
2-poliges Zustandsvariablenfilter mit Tabellen-Kompensation und optionaler Sättigung (Version S). Bietet ebenfalls ein verbessertes Verhalten bei hohen Cutoff-Einstellungen, klingt aber etwas anders als das Modul 2 Pole SV C. Die Cutoff-Frequenz wird über den Eingang "P" durch eine MIDI-Noten-Nummer spezifiziert und kann über den Eingang "PM" (in Halbtonschritten, exponentiell) und über den Eingang "FM" (in Hz, linear) moduliert werden. Der Eingang "Res" legt die Resonanz fest (im Bereich von 0 bis 1).
Die Ausgänge "HP", "BP" und "LP" erzeugen Hochpass-, Bandpass- respektive Tiefpass-Signale.

I.44. VCF > 2 Pole SV x3 S



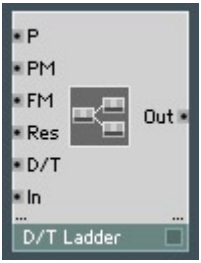
2-poliges Zustandsvariablenfilter mit optionalem Oversampling (Version x3) und Sättigung. Die Cutoff-Frequenz wird über den Eingang "P" durch eine MIDI-Noten-Nummer spezifiziert und kann über den Eingang "PM" (in Halbtonschritten, exponentiell) und über den Eingang "FM" (in Hz, linear) moduliert werden. Der Eingang "Res" legt die Resonanz fest (im Bereich von 0 bis 1). Der Eingang "Sat" bestimmt den Grad der Sättigung (typischer Bereich: 8 bis 32).
Die Ausgänge "HP", "BP" und "LP" erzeugen Hochpass-, Bandpass- respektive Tiefpass-Signale.

I.45. VCF > Diode Ladder



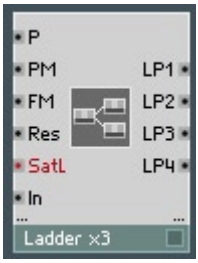
Nachbildung eines Dioden-Leiter-Filters. Die Cutoff-Frequenz wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert und kann über den Eingang “PM” (in Halbtonschritten, exponentiell) und über den Eingang “FM” (in Hz, linear) moduliert werden. Der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 0,98).

I.46. VCF > D/T Ladder



Lineare Leiter-Filter-Nachbildung, die stufenlos zwischen Dioden- und Transistor-Leiter-Verhalten überblendet werden kann. Die Cutoff-Frequenz wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert und kann über den Eingang “PM” (in Halbtonschritten, exponentiell) und über den Eingang “FM” (in Hz, linear) moduliert werden. Der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 0,98). Der Eingang “D/T” blendet zwischen Diode und Transistor über (0 = Diode, 1 = Transistor).

I.47. VCF > Ladder x3



Nachbildung eines Transistor-Leiter-Filters mit dreifachem Oversampling und Sättigung. Die Cutoff-Frequenz wird über den Eingang “P” durch eine MIDI-Noten-Nummer spezifiziert und kann über den Eingang “PM” (in Halbtonschritten, exponentiell) und über den Eingang “FM” (in Hz, linear) moduliert werden. Der Eingang “Res” legt die Resonanz fest (im Bereich von 0 bis 1). Der Eingang “SatL” bestimmt den Grad der Sättigung (typischer Bereich: 1 bis 32).

Die Ausgänge 1 bis 4 führen die Signale der entsprechenden Stufen der emulierten “Leiter”. Verwenden Sie die 4. Stufe für den klassischern Leiter-Filter-Sound.

Index