Основы Reaktor Core (часть 1)

Введение в Reaktor Core (часть 1)

Введение
Интерфейс и ячейки ядра

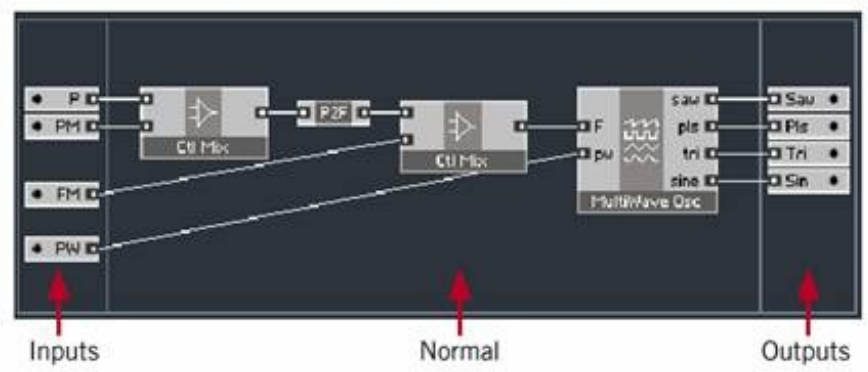
Reaktor Core дает новые возможности для моделирования устройств на низком уровне. Технология Reaktor Core не интегрирована в первичный уровень Reaktor, а реализуется посредством особых модулей core cells (ячеек ядра). Эти ячейки можно использовать при построении структур первичного уровня. На данный момент внутри модулей core cells невозможно создавать различные регуляторы для осуществления регуляции с панели инструмента.

Хотя главное назначение Reaktor Core – сознание новых низкоуровневых устройств цифровой обработки, в поставляемой с программой библиотеке можно найти достаточное количество уже готовых модулей, которые можно использовать в проектировании собственных структур. Функциональность ячеек эквивалентна функциональности макросов первичного уровня. Также некоторые макросы стандартной библиотеки созданы с использованием ячеек ядра.

Доступ к модулям core cells возможен при помощи контекстного меню окна структуры, а также используя встроенный браузер (клавиша F5).

Чтобы сохранить выбранную ячейку ядра в файле – выберите в ее контекстном меню *Save Core Cell As.*

Важное ограничение использования core cells – это то, что их нельзя использовать внутри петель событий. Каждый такой случай будет блокироваться Reaktor.



Итак, войдем внутрь структуры какой-нибудь ячейки ядра: появится структура Reaktor Core. Она состоит из трех областей, разделенных вертикальными линиями: **Inputs** (только порты входа), **Normal** и **Outputs** (только порты выхода). В области **Normal** модули можно перемещать во всех направлениях, тогда как в двух других – только вертикально. Причем, порядок портов входа и выхода ячейки ядра будет соответственно изменяться при перетаскивании модулей портов внутри этой ячейки ядра. Также можно заметить, что перемещение модулей внутри области **Normal** иногда ведет к изменению этой области (для того чтобы модули не выходили за ее пределы). Командой контекстного меню *Compact Board* можно оптимизировать расположение модулей в пространстве, уменьшив размеры структуры.

Существует два различных типа core cells: audio core cells (аудио-ячейки) и event core cells (ячейки обработки событий). Если ячейка ядра имеет выходной аудиопорт (то есть она является аудио-ячейкой), то можно переключить тип их входов между audio и event (подробнее об этом - далее). Это можно осуществить в окне свойств порта (параметр **Signal Mode**).

Черная точка, изображенная на модуле порта означает, что это аудио-порт, а если это точка имеет красный цвет – то это порт сигнала событий (event).

Иногда невозможно переключить тип порта с одного на другой. Например, это произойдет тогда, когда к входному порту уже подключен провод, несущий реальный аудиосигнал (но не сигнал огибающей). В этом случае переключить режим порта на event будет невозможно.

Если разрабатывается ячейка ядра, то порты с частотой дискретизации аудио обычно переключаются в режим event без проблем. Входные event-порты могут быть переключены на аудио только если не имеют функции переключателя (или другой чувствительной к событиям функции).

Если необходимо уменьшить нагрузку на процессор – можно отключить все соединения с модулями выходных портов, которые не будут использоваться.

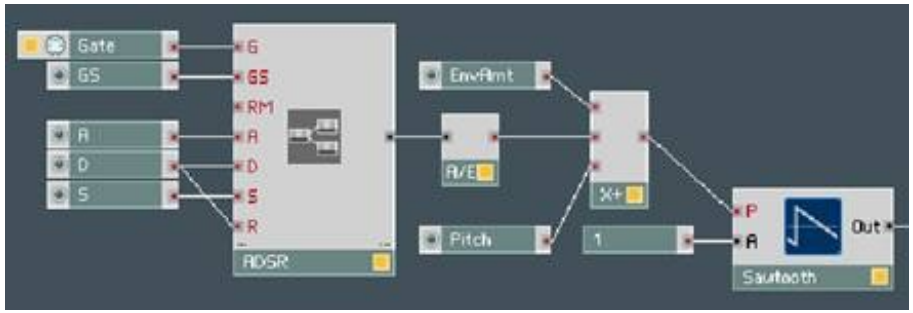
Рассмотрим подробнее ячейки. Как уже известно, их существует два типа – аудио и ячейки событий. Последние могут получать с

первичного уровня только event-сигналы на входе, и на выход выдавать также только event-сигналы. Аудиоячейка поддерживает входные порты обоих типов, но выходные порты – только типа audio.

Внутри аудиоячеек можно реализовывать различные осцилляторы, фильтры, огибающие и т.д. Event-ячейки служат только для реализации обработки событий.

Flavor	Inputs	Outputs	Clock Src
Event	Event	Event	Disabled
Audio	Event/Audio	Audio	Enabled

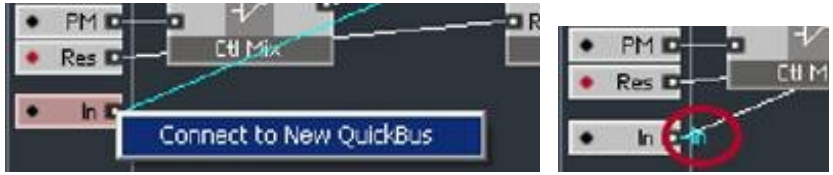
Другое важное различие этих двух типов ячеек это то, что в event-ячейках заблокированы все источники синхросигналов, поэтому в них нельзя реализовать, например, LFO с частотой дискретизации event (control rate). Если такой модуль необходим – нужно использовать аудиоячейку и конвертировать сигнал с ее аудиовыхода уже на первичном уровне (при помощи модуля A/E).



На рисунке изображена аудиоячейка ядра, реализующая огибающую, сигнал выходного порта которой конвертирован в event-сигнал, для управления высотой тона осциллятора.

Соединения по **QuickBus**

В больших структурах, где присутствует очень много проводов, более удобно использовать новую возможность в Reaktor Core – **QuickBus**. Такую связь можно создать, используя меню выходного порта командой *Connect to New QuickBus* .



В окне свойств можно задать другое имя для этого соединения, а также иные параметры (пока они нас не интересуют). Соединение по **QuickBus** – имеет локальное значение для структуры, таким образом, соединения с одинаковыми именами в разных структурах (например, в макросах) конфликтовать не будут.

Теперь источник сигнала определен, и следующее, что нужно сделать – это задать порты-получатели сигнала. Это можно осуществить также через контекстное меню, но уже порта-входа (см. меню *Connect to QuickBus >In* для присоединения порта к шине с именем **In**):

Теперь входной порт будет получать сигнал непосредственно с выбранной шины, то есть присоединенного к ней входного порта. Один порт-источник может иметь и несколько присоединенных к его шине портов-получателей, в этом смысле соединения по **QuickBus** не отличаются от соединений проводами.

Сигналы управления и аудиосигналы

В рамках библиотеки стандартных макросов Reaktor существует такое соглашение: все модули описываются в рамках четырех типов сигналов: аудио (audio), сигнал управления (control), event-сигнал и логический (logic) сигнал. Логические и event-сигналы рассмотрим чуть позже, а сейчас рассмотрим сигналы первых двух типов.

Аудиосигналы несут реальную аудиоинформацию (такие сигналы производят порты выхода осцилляторов, фильтров, усилителей, сатураторов-сглаживателей, модулей задержки и т.п.). Сигналы управления – не несут аудиоинформации, а используются для управления. Например, выход огибающей или LFO, а также сигналы pitch и velocity не издают звуков, но могут быть использованы для управления входными портами частоты среза фильтра, резонанса, времени задержки, частоты осциллятора.

Некоторые типы обработки, например микширование, могут быть чувствительными к обоим типам сигнала – аудио и управления. В этих случаях существует две версии макросов: для обработки аудио и для обработки сигналов контроля. Например, есть аудиомикшеры, и микшеры сигналов управления, аудиоусилители и усилители сигналов управления и т.д. Обычно не следует использовать эти модули не по назначению.

Часто возможно использовать аудиосигнал в качестве сигнала управления. Наиболее распространенный пример такого использования – это модуляция частоты осциллятора или частоты среза фильтра аудиосигналом. Обратная ситуация – использование сигнала управления в качестве аудиосигнала конечно возможна, но встречается довольно редко.

Различие между этими четырьмя типами сигналов не соответствует делению сигналов первичного уровня на аудио и сигналы событий. Классификация событий первичного уровня базируется на частоте обработки – аудиосигналы обрабатываются с большей частотой, нежели сигналы событий. Также, как вы уже вероятно знаете, event-сигналы первичного уровня имеют другие правила распространения в отличие от аудиосигналов.

Различия в типах сигналов Reaktor Core чисто семантические, характеризующие тип использования сигнала. Не существует какого-либо взаимно-однозначного отношения между сигналами первичного уровня и сигналами Reaktor Core, но общие правила все же существуют:

Аудиосигнал первичного уровня обычно соответствует в c_ сигналам аудио и управления.

Event-сигнал первичного уровня обычно – это сигнал управления в Reaktor Core. Например – это сигналы выводов LFO, регуляторов, источник MIDI pitch или velocity.

Иногда event-сигнал первичного уровня соответствует event-сигналу Reaktor Core. Одним из таких распространенных случаев является сигнал MIDI gate.

Иногда event-сигнал первичного уровня представляет собой логический сигнал Reaktor Core. Но эти сигналы совместимы не полностью и должны бить конвертированы один в другой (как это сделать рассматривается позже). Пример – сигнал от модуля AND первичного уровня.

Важно понимать, что, выбирая тип входного порта ячейки ядра вы выбираете между типами сигналов audio/event первичного уровня, но не уровня Reaktor Core. Порты ячеек ядра являются как бы интерфейсами между средой первичного уровня Reaktor и средой Reaktor Core.

Создание макроса

Пустой макрос можно создать при помощи контекстного меню *Build-In Module > Macro* .

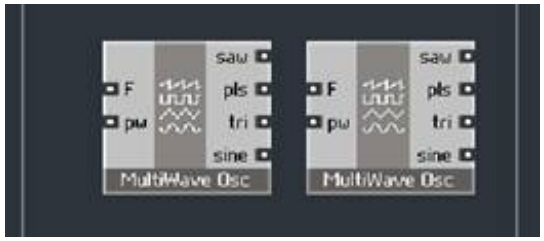
Существует несколько отличий в построении структур макроса ядра от ячейки ядра. Эти различия заключаются в том, что для макроса доступно несколько типов входных и выходных портов (**Out**, **Latch**, **Bool C**). **Latch** и **Bool C** используются в продвинутом программировании и будут рассматриваться позже. Порт **Out** (как и **In**) – общий порт, который применяется для передачи всех типах сигналов Reaktor Core (audio, control, event, logic). Фактически, порт не различает что за сигнал он передает, это различие важно только для пользователя, потому как описывает, в каком качестве этот сигнал используется. Для Reaktor Core эти сигналы не в коей мере не различны. Также нет различия между audio/event сигналами входных и выходных портов приходящих из структур вышележащего уровня, поэтому внутри ячейки ядра сигналов первичного уровня Reaktor нет.

Макрос ядра можно переименовать также как и макрос первичного уровня. В свойствах макроса настраиваются некоторые параметры (рассматриваются далее).

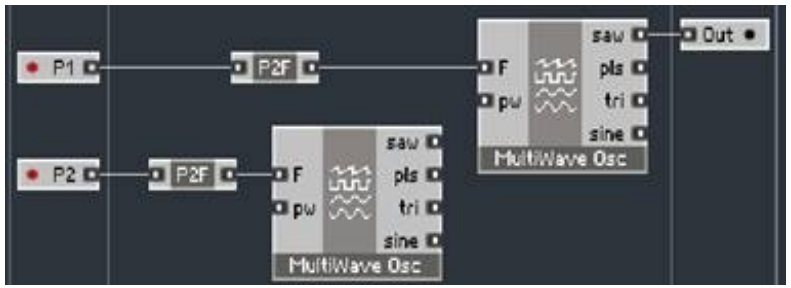
Для входного порта макроса существует возможность задать некоторое значение при помощи **QuickConst** (используя контекстное меню этого порта) – значение по умолчанию. Значение порта будет равно этой константе, если порт не активен, то есть, не подключен проводом на вышележащем уровне структуры. Значение **QuickConst** можно установить в окне свойств, которое открывается двойным щелчком левой кнопки мыши на константе.

Использование аудиосигнала как сигнала управления

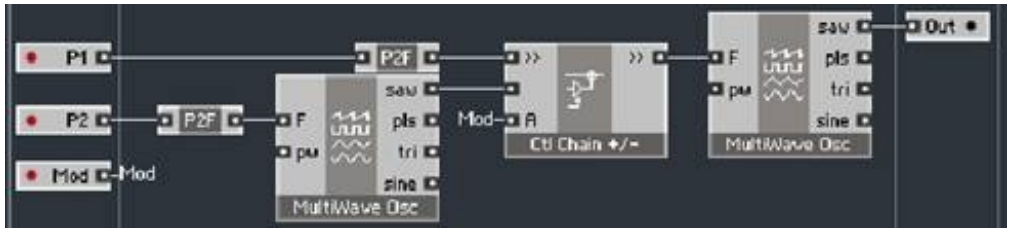
Давайте рассмотрим, как можно использовать аудиосигнал в качестве сигнала управления. Рассмотрим пример, когда есть два осциллятора **MultiWave Osc**, и сигнал перового будет модулировать второй осциллятор.



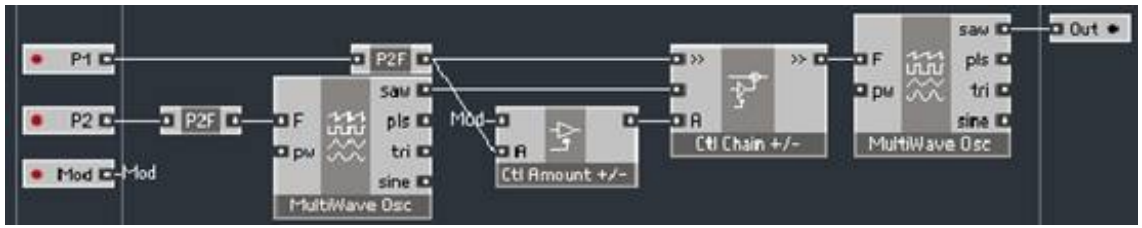
Итак, реализуем для каждого осциллятора управление высотой тона:



Теперь осуществим модуляцию левого осциллятора и используем для этого частоту правого осциллятора (входной порт **Mod** выдает значение глубины модуляции):



Заметим, что микширование сигнала модуляции с портом **P1** произведено уже после преобразования его модулем **P2F** (это преобразование из pitch в частотное), поэтому модуляция осуществляется в масштабе частоты. Также неплохая идея масштабировать количество модуляции в зависимости от соответствующей частоты осциллятора:



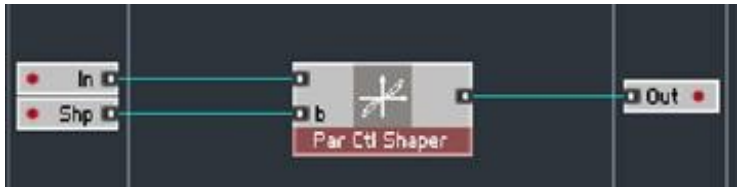
Если проанализировать получившуюся структуру с точки зрения управляющих и аудиосигналов можно заметить, что все сигналы структуры, за исключением выходных сигналов осцилляторов – это сигналы управления.

Заметим, что мы неправильно используем выходной порт левого осциллятора как сигнал управления в точке соединения с модулем **Ctrl Chain**.

Сигналы событий в Reaktor Core

Ранее говорилось о различных значениях термина «сигнал события». Есть несколько путей использования сигналов событий первичного уровня. Первый это – использовать его в качестве сигнала управления (например, сигналы LFO, регуляторов и т.д.), потому что такие сигналы меньше нагружают процессор, нежели аудиосигналы. В этом случае, можно достичь того же самого эффекта, что и с аудиосигналом.

Итак, event-сигнал первичного уровня можно использовать как сигнал управления. Рассмотрим пример управления макросом шейпера (в варианте Ctl):

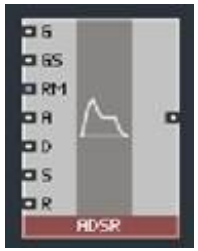


Макрос получает сигнал с частотой дискретизации control rate с первичного уровня (например, сигнал MIDI velocity или LFO), bends в соответствии с параметром **Shp** и возвращает результат в выходной порт.

Важное ограничение ячеек ядра события, которое мы упоминали ранее это то, что все источники синхронизации в них заблокированы.

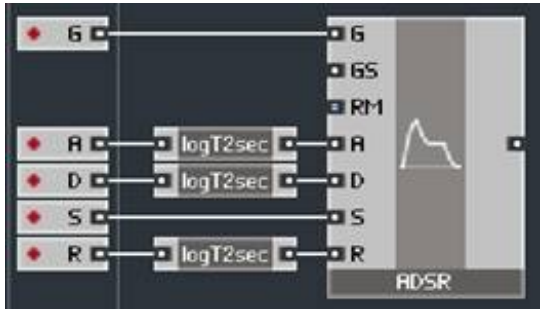
Это значит что не только осцилляторы и фильтры, но также и огибающие и LFO не будут работать внутри event-ячеек ядра. Такие модули служат для того, чтобы получить события с первичного уровня Reaktor, обработать их, и затем передать их обратно на первичный уровень, как в приведенном примере.

Также сигнал события, прибывающий с первичного уровня, может быть использован как сигнал события внутри структуры Reaktor Core. Первый случай - это использование огибающей (например, из контекстного меню *Standard Macro >Envelope >ADSR*).



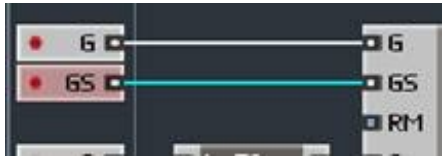
Верхний входной порт макроса работает подобно входному порту gate модуля огибающей первичного уровня – то есть он открывает и закрывает огибающую в ответ на приходящее событие. Поэтому для этого порта создается входной порт ячейки с event-режимом работы:

Этот входной порт будет транслировать входящие события первичного уровня в события Reaktor Core. Теперь рассмотрим входные порты **A**, **D**, **S**, **R**. Порт **S** (sustain) работает подобно соответствующему порту модуля огибающей первичного уровня. Он ожидает входящий сигнал в диапазоне 0…1. Значения портов **A**, **D**, **R**, в отличие от первичного уровня, задаются в секундах, значит необходимо преобразование сигнала. Оно осуществляется при помощи модуля **LogT2sec** (*Standard Macro >Convert >logT2sec*), который конвертирует время в секунды:

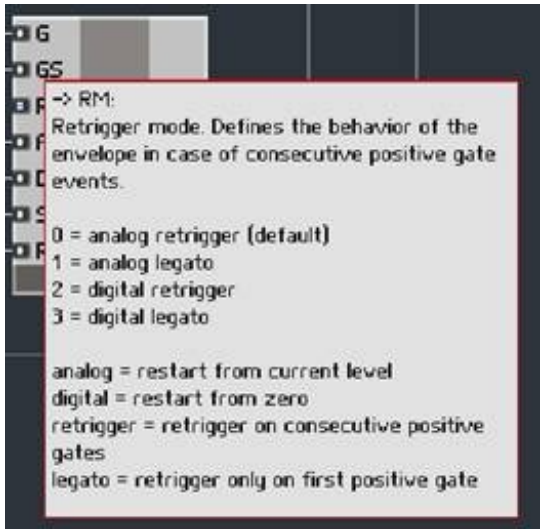


Хотя все порты в данной структуре в режиме event, первый входной порт выдает event-сигнал, тогда как остальные – сигнал управления.

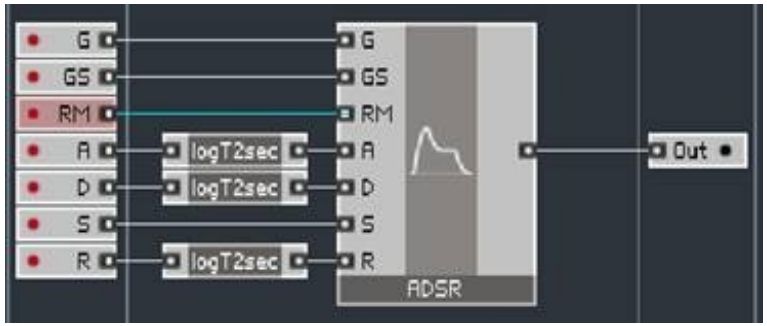
Два порта огибающей все еще не соединены – это порт **GS** – который устанавливает чувствительность сигнала gate. При нуле – огибающая полностью игнорирует уровень gate-сигнала и всегда работает в полную амплитуду. При значении равном 1, уровень gate-сигнала имеет максимальный эффект (такой, какой принят по умолчанию на первичном уровне). Можно этот порт также контролировать при помощи event-событий:



Входной порт **RM** осуществляет выбор режима работы огибающей:



Видно, что этот порт отличается от всех других (синий значок, похожий на знак равенства). Дело в том, что этот порт работает в режиме целочисленных значений (integer), но это не означает, что к нему нельзя подключить нецелочисленные сигналы. Если порт получает нецелочисленный сигнал – он округляет его к ближайшему целочисленному значению.



Рассмотрим еще один пример использования сигнала события:

Эта структура реализует тип модуляции по сдвигу тона. Эффект производит задержки сигнала в диапазоне 250+-100 миллисекунд. Частота определяется входным портом **Rate**, который контролирует частоту LFO (в герцах) – это сигнал управления. Входной порт **Rst** – сигнал события и может быть использован для рестарта LFO. Значение порта определяет фазу с которой производится рестарт (0 – с

начала цикла, 0.5 с середины, и 1 – с конца). Этот порт можно присоединить, например, к кнопке с определенным заданным значением.

Логические сигналы в Reaktor Core

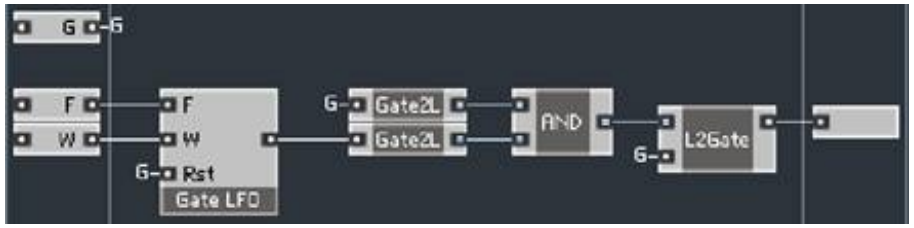
Пример модуля обрабатывающего логические сигналы.

Такой тип порта (похожий на знак равенства синего цвета) означает, что этот порт работает в целочисленных значениях (integer). В данном случае – только с значениями 0 и 1. для логических сигналов 1 означает true (истина), 0 – false (ложь).

Например, используя модуль **Gate2L** можно осуществить преобразование сигнала gate к логическому.

Таким образом, модуль будет выдавать 1 при нажатии на MIDI-клавишу и 0 – при отпускании ее (если конечно этот сигнал gate исходит от одноименного модуля MIDI In).

Рассмотрим еще один пример, изображенный на рисунке.



Макрос **Gate LFO** создает регулярные сигналы gate с заданной частотой и уровнем. Модуль **L2gate** – преобразует логический сигнал в сигнал gate.

G – это QuickBus (по сути заменитель проводов).

Структура формирует периодические gate-сигналы с заданной частотой и длительностью (а также с уровнем исходного gate-сигнала), как только присутствует внешний сигнал gate с порта **G**. Такая структура может использоваться, например, при построении секвенсора.

Структура макроса **Gate LFO** приведена ниже:

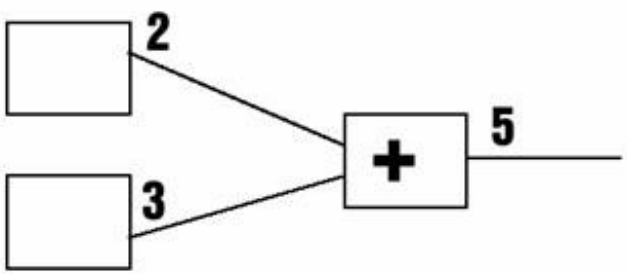
Входной порт **F** определяет частоту повторений сигнала, **W** – определяет длительность этого сигнала (0 – 50% периода, -1 – 0%, 1 – 100%). **Rst** – осуществляет рестарт LFO, как только по порту пришло событие. Модуль  **Value** замещает значение

пришедшего события на новое (в данном случае по умолчанию на ноль, потому как второй порт не активен).

Основы обработки сигналов в Reaktor Core Сигнальная модель Reaktor Core

Большинство выходных портов модулей Reaktor Core производят значения. Это означает, что в любой момент времени это значение ассоциируется с портом. Эти значения доступны всем портам входа модулей, которые соединены с выходными портами других модулей.

Например, модуль сложения получает значения 2 и 3 от двух модулей, присоединенных к его портам входа, и он производит значение на своем выходном порту равное 5.



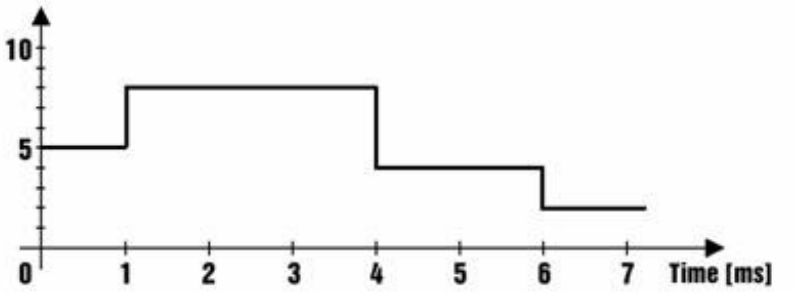
Аналогией могут служить значения уровней сигналов (вольт). Значения не ограничены типом обработки – это просто значения, которые могут быть использованы в определенном алгоритме.

События

В цифровом мире время не имеет продолжительности, оно дискретно. Цифровые записи не сохраняют полную информацию о сигнале, который непрерывно продолжается во времени, а сохраняют только информацию об уровне сигнала в некоторых определенных равноотстоящих временных отсчетах. Количество отсчетов в секунду называется частотой дискретизации.

Поскольку среда Reaktor – цифровая среда, то значения выходных модулей не могут изменяться непрерывно.

С другой стороны, мы не должны быть ограничены изменением значений только в регулярно заданных временных отсчетах. Нет необходимости поддерживать специфическую норму дискретизации на всем протяжении структур. В некоторых случаях, в определенных областях структур мы даже не должны поддерживать никакой нормы дискретизации вообще, то есть, наши изменения не должны происходить равномерно.

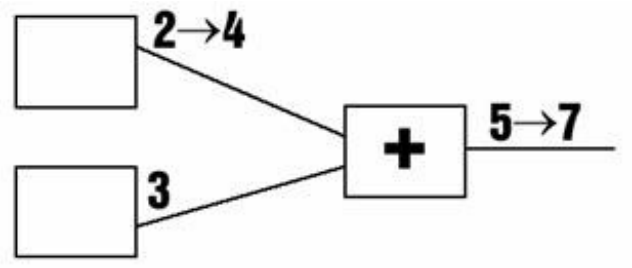


Например, в нулевое время выход модуля сложение принимал значение 5. первая смена значения произошла через 1 миллисекунду, второе – через 4 мс, а третье – через 6 мс.

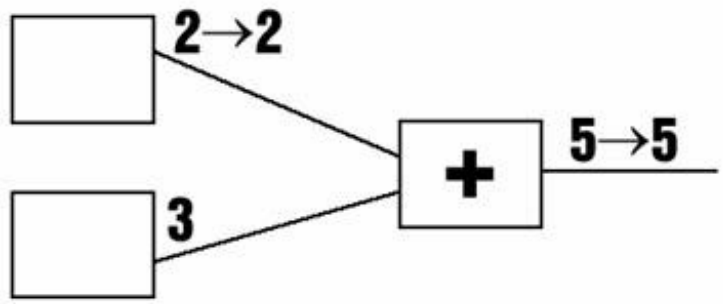
На рисунке выше мы видим изменения значений выходного порта модуля сложения, которые происходили в течение 7 мс. В момент времени, когда выходной порт меняет свое значение – он генерирует событие. Событие – означает, что выходной порт сообщает об

изменении состояния, что говорит об изменении значения этого порта.

На следующем примере верхний левый модуль меняет свое значение с 2 на 4, тем самым, генерируя событие. В ответ на это модуль сложения изменит свое значение и также сгенерирует новое событие.



Также верхний левый модуль может сгенерировать новое событие с тем же значением что у него и было. Модуль сложения также ответит генерацией нового события, не меняя своего значения. Новое значение, которое появляется на выходном порту не требует своего различия с прошлым значением.



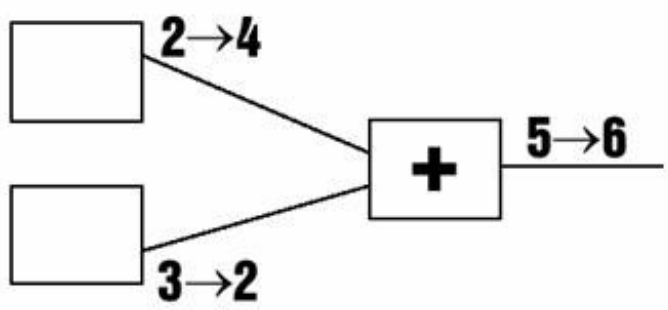
Таким образом, единственный путь изменить значение порта – это генерация события.

Как видно из предыдущих примеров, события, происходящие на выходных портах какого-либо модуля принимаются всеми нижеследующими (по пути прохождения сигнала) модулями, которые также должны произвести события. Заметим, что модуль производит на выходном порту событие только в ответ на входящее событие. Итак, новые события будут восприниматься всеми модулями, соединенными к соответствующему выходному порту модуля, и распространяться дальше по пути следования сигнала, пока это распространение не остановится по какой-либо причине (такие причины рассматриваются в дельнейшем).

События в Reaktor Core не то же самое что события в первичном уровне Reaktor. К событиям в Reaktor Core применяются другие правила.

Одновременные события

Рассмотрим ситуацию, когда два левых модуля одновременно генерируют события.

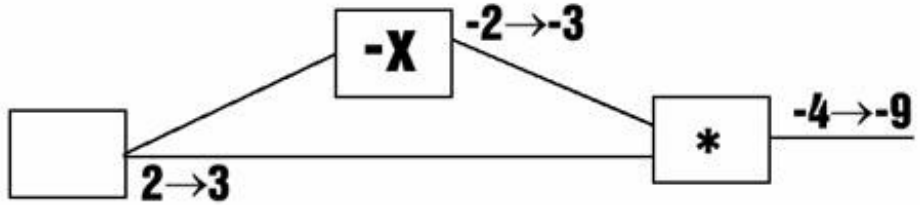


Это одна из ключевых особенностей модели Reaktor Core – события могут происходить одновременно в некоторых случаях. В такой ситуации оба события от двух левых модулей прибывают на входные порты модуля сложения одновременно, и важно, что модуль сложения производит в точности **одно** событие на своем выходном порту.

Это отличается от того, как обрабатываются события первичного уровня Reaktor – там события **не могут** случаться одновременно, и, например, модуль **Add** (в event-режиме) произведет в такой ситуации два события на выходном порту.

Конечно, в реальности события этих двух левых модулей не обрабатываются одновременно, потому что оба эти модуля обрабатываются реальным центральным процессором, и процессор может одновременно обрабатывать только один модуль. Но это важно для нас, что эти события **логически одновременны**, что трактуется как одновременность взаимодействий модулей.

Вот другой пример одновременного распространения событий:



В этом примере самый левый модуль посылает событие, изменив значение своего выходного порта с 2 на 3. Это событие послано **одновременно** двум следующим модулям – инвертору и модулю умножения. В ответ приходящему событию инвертор генерирует новое значение –3. Важно отметить, что хотя событие выходного порта инвертора сгенерировано в ответ на событие, пришедшее от левого модуля, как будто бы позже, но все события все еще логически одновременны. Это означает также и одновременное прибытие событий на входы модуля умножителя, таким образом, умножитель генерирует событие со значением –9.

На первичном уровне Reaktor мы бы имели два события на выходном порту модуля **Event Mult**. Так же там не определено, какое же из событий выходного порта левого модуля будет послано первым – на инвертор или же на умножитель (хотя это для данной структуры не так важно).

В общем можно использовать следующее правило одновременности событий: все события, посылаемые в ответ на некоторое событие одновременны. Все события, происходящие из произвольного числа одновременных событий (поданные происходящих на выходных портах, но известно, что они одновременны) также одновременны.

Последний пример показывает выгодность имеющихся одновременных событий. В этом случае мы устраняем избыточную обработку второго события умножителем, который берет особенно много времени процессора.

В длинных структурах, в отсутствие одновременных событий, число событий не сможет расти неконтролируемо, если разработчик структуры обращает особое внимание на происхождение дублированных событий.

Порядок обработки

Как вы видели в предыдущих примерах, если модуль посылает событие, то нижележащие модули отвечают на это событие.

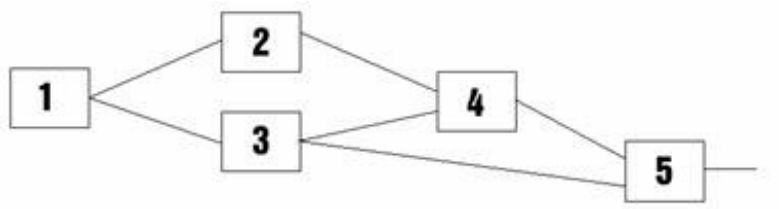
Из этого, можно заключить, что, несмотря на создание логически одновременных событий, модули не обрабатываются одновременно, и что будет разумно обработать сначала вышележащий модуль перед нижележащим. Все эти заключения, фактически, правильны.

Общее правило порядка обработки следующее: если два соединенных модуля обрабатывают логически одновременные события, то вышележащий модуль будет обработан первым. Если события не одновременны, то, конечно, порядок обработки модулей будет в порядке обработки событий.

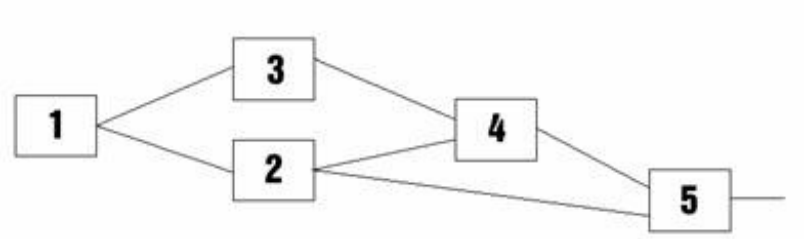
Из этого правила следует, что между двумя модулями существует однонаправленный путь соединений (всегда upstream или downstream), что и определяет порядок обработки: модуль upstream (выше по потоку) будет обработан первым.

Если не существует однонаправленного пути соединения между двумя модулями, их порядок обработки относительно друг друга не определен для логически одновременных событий. Это означает, что порядок может произвольно измениться в результате некоторых действий. Разработчик структуры должен заботиться, чтобы такие ситуации происходили только для тех модулей, порядок обработки которых неважен. Это условие обычно выполняется автоматически, кроме случаев с OBC-подключениями (см. ниже).

Рассмотрим пример, где числа показывают порядок обработки модулей:

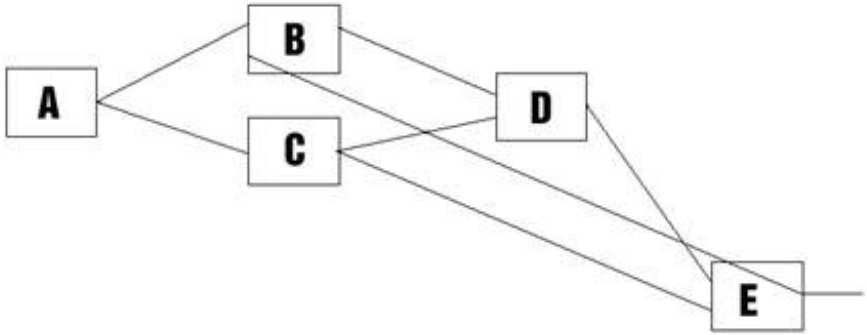


Для этой структуры также может быть задан альтернативный порядок обработки:



Невозможно точно сказать какой из вариантов последовательности обработки будет выбран программой. Но если только вы не используете в структуре OBC-связи, относительный порядок обработки модулей в этом случае действительно не важен.

Это правило определения последовательности обработки модулей не может быть применено, если в структуре есть обратная связь, потому как в этом случае для любой пары модулей в замкнутой цепи нельзя сказать что один лежит выше другого. Описание таких обратных связей (включая и порядок обработки) будет рассмотрено позже.



Для структуры изображенной на рисунке выше, невозможно определить, например, какой модуль лежит ниже или выше в пути обработки – B или D.

Обзор event-ячеек ядра

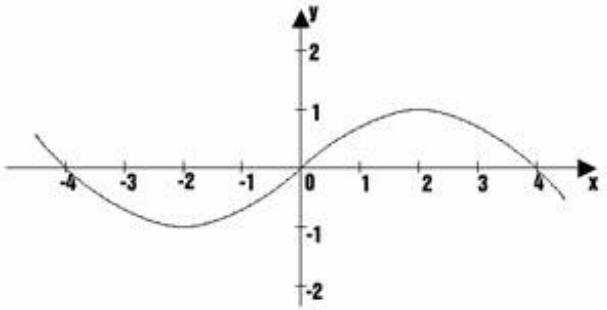
Давайте взглянем на event-ячейки ядра с точки зрения концепции событий Reaktor Core. Как известно, event-ячейки имеют входные и выходные порты в режиме event. Эти порты являются интерфейсами между первичным уровнем Reaktor и уровнем Reaktor Core, они осуществляют преобразование между событиями первичного уровня и событиями Reaktor Core. Правила таких преобразований следующие:

Входной порт **event input** посылает событие ядра внутрь структуры в ответ на событие, пришедшее с первичного уровня. Поскольку события первичного уровня не могут приходить на порт одновременно, генерация внутренних событий также не происходит одновременно.

Выходной порт **event output** генерирует событие в ответ на внутреннее событие в ячейке, пришедшее в порт. Хотя события в ячейке ядра могут происходить одновременно на некоторых выходных портах, события первичного уровня не могут быть посланы одновременно. В связи с этим события первичного уровня, соответствующие одновременным событиям ядра будут посланы одно за другим, причем с верхнего выходного порта посылается before нижнего.

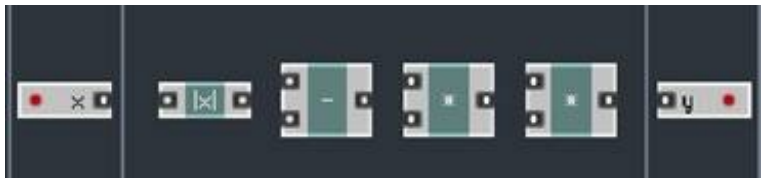
Давайте создадим модуль обработки event-событий, осуществляющий нелинейное преобразование сигнала по формуле $y = 0.25*x*(4-|x|)$.

График этой функции:

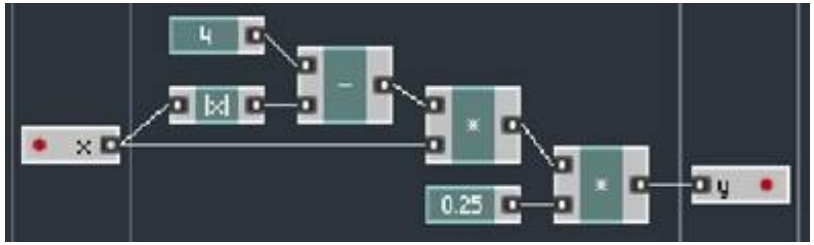


Создадим два порта:

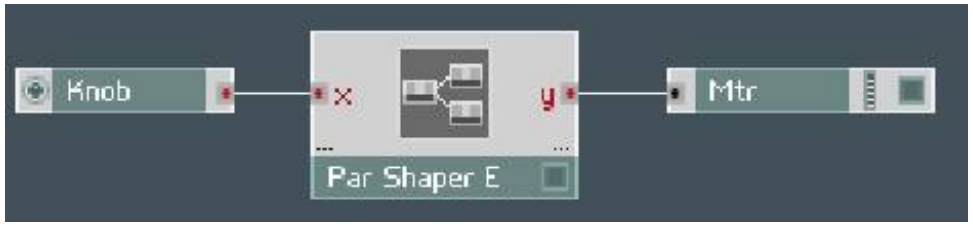
Теперь создадим структуру, которая произведет вычисление по заданной формуле. Нам нужен один модуль взятия абсолютного значения, один модуль разности и два умножителя (эти модули можно найти в меню *Built-In Module >Math*).



Также необходимы две константы: 0.25 и 4. Их можно создать при помощи меню *Built-In Module >Const* . Значение константы следует изменить в окне ее свойств.



Теперь протестируем работу созданной структуры.



Для начала выставим в свойствах инструмента, для того чтобы работал измеритель уровня, количество голосов (**voices**) равное 1. У модуля **Mtr** необходимо выставить опцию **Always Active** и отобразить поле со значением (**Value**) сигнала.

Структуры с внутренними состояниями Сигналы синхронизации

Как модуль Reaktor Core обрабатывает входящее событие – полностью зависит от этого модуля. Обычно модуль обрабатывает входящее событие определенным заданным способом, но в некоторых случаях может также просто его игнорировать. Типичный случай такой обработки – это входные порты синхронизации.

Примером модуля с входным портом синхронизации может служить модуль **Latch**. Этот модуль не встроенный, а является макросом, тем не менее, он отлично демонстрирует принцип синхронизации. Модуль **Latch** имеет два входа – один для значения, а другой для синхронизации.

Значение входного порта (верхний) запоминает входящее значение во внутренней памяти модуля в ответ на входящее событие, но на выходной порт не посылается никакого события. Выходной порт пошлет последнее сохраненное значение только то в ответ на сигнал, поступивший с входного порта синхронизации.

Входы синхронизации (за исключением специфических, описанных в руководстве) полностью игнорируют значение приходящего события, их интересует только тот факт, что некоторое событие на этот порт пришло.

(более подробно работу модуль **Latch**рассмотрим чуть позже).

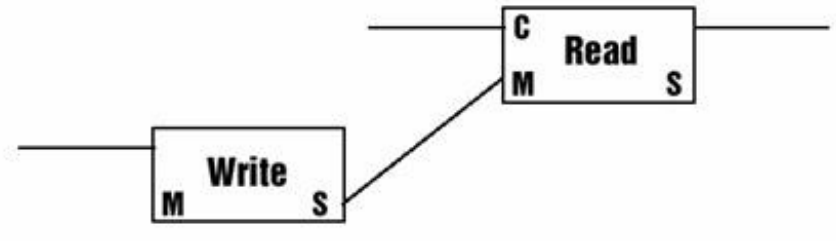
Потому как такие модули имеют входной порт синхронизации, это очищает структуру от лишних сигналов, значения которых не нужны. Некоторые сигналы могут часто быть сгенерированы для sole свойств и использованы в качестве сигналов синхронизации. Мы назовем их термином **clock signals**.

Частота дискретизации Reaktor (audio rate) – это также пример сигнала синхронизации. Он производит событие для каждого генерированного отсчета (например, частота дискретизации 44.1 килогерц дает синхросигналы 44 100 раз в секунду). Уровень сигнала синхронизации не имеет значения, оно нигде не используется, и в данной реализации программы – это значение всегда ноль.

OBC-связь

Object Bus Connection – специальный тип связи между модулями. Такая связь между двумя модулями **определяет**, что эти два модуля разделяют какой-либо один внешний объект. Типичными представителями таких модулей являются модули чтения и записи памяти – **Read** и **Write**, которые разделяют общую память, в случае если они соединены OBC-связью.

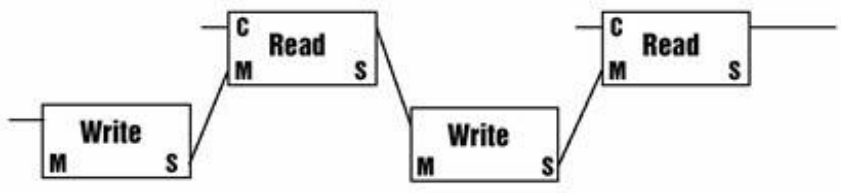
Задача модуля **Write**– записать значение входящего события в OBC-общую память. Задача модуля **Read** – прочитать значение из OBC-общей памяти в ответ на входящий синхросигнал (на порт **C**). Прочитанное значение посылается в выходной порт модуля **Read**.



На рисунке изображена структура макроса **Latch**(фактически это внутренняя структура этого макроса). **M** и **S** порты модуля **Read** и **Write** это порты типа OBC. **M** –обозначен главный вход (master), **S** (slave) – подчиненный выходной порт. Главный входной порт модуля **Read** соединен с подчиненным выходным портом модуля **Write** (другие два master/slave порта не задействованы). Таким образом, в

этой структуре модули **Write** и **Read** разделяют общую память.

В следующей структуре изображены две пары модулей **Write и Read**. Каждая пара имеет свою собственную разделяемую память.



Заметим, что связь в середине – это не OBC-связь.

Какие же отличия главного и ведомого порта? С точки зрения разделяемого объекта (в данном случае памяти) различий никаких нет. Однако, как вы помните из ранее сказанного о порядке обработки модулей – вышележащие в потоке обрабатываются раньше, в случае одновременных событий. Так, в последних двух примерах модуль **Write** будет обработан до подчиненного модуля **Read**, но никак не в обратном порядке.

Таким образом, относительный порядок обработки OBC-связанных модулей определяется использованием того же правила (первый в потоке – обрабатывается первым).

Давайте рассмотрим два различных случая. В обоих случаях начальное состояние памяти имеет значение 2, и некоторое событие со значением 5 послано обоим модулям **Write** и **Read** одновременно. В первом случае, модуль **Write** будет главным, а во втором – главным будет модуль **Read**.

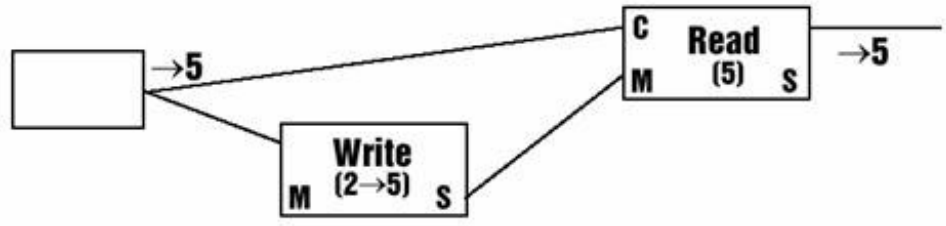
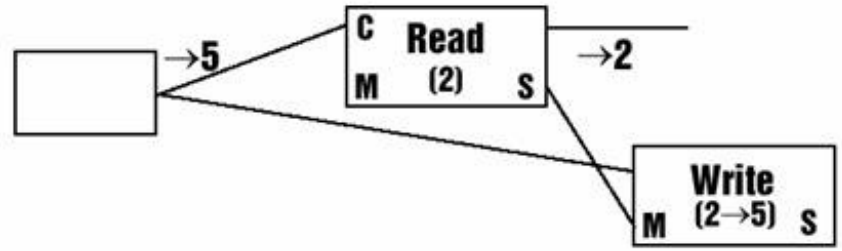


Рисунок выше изображает первый случай. Левый модуль посылает событие со значением 5, которое сначала достигает модуля **Write** потому как он главный. Значение 5 записывается в разделяемую память. Далее, событие прибывает на модуль **Read** в качестве сигнала синхронизации (clock), и заставляет модуль совершить операцию чтения из памяти. Таким образом, на выходной порт модуля **Read** посылается событие со значением 5. Именно так работает макрос **Latch** библиотеки Reaktor.

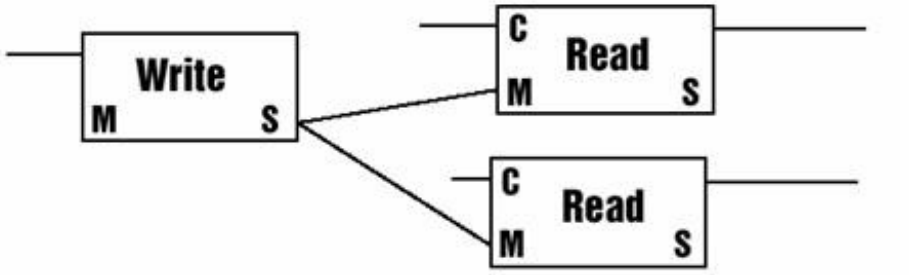


Теперь рассмотрим противоположный случай. Сначала, сигнал clock прибывает в модуль **Read**, который читает память и посылает в выходной порт событие со значением 2. Только после этого модуль **Write** изменяет значение памяти на 5. Эта структура реализует

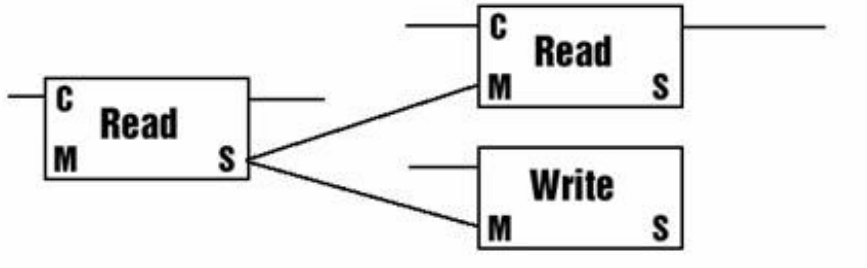
функциональность $\mathbf{Z^{-1}}$ блока (задержка на один семпл), который используется в теории цифровой обработки сигналов (DSP). Таким образом, значение на выходе этой структуры всегда на один шаг позади входного значения.

Иногда OBC-объект разделяют более двух модулей. Такое возможно. В этом случае, очень важно знать каков порядок операций чтения и записи, и какой порядок должен быть.

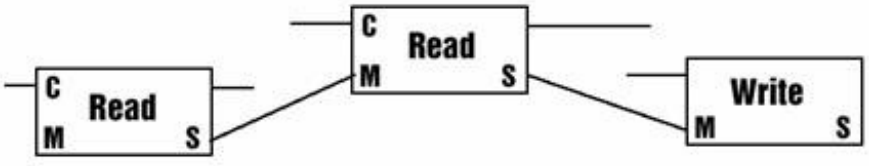
Например, в структуре, изображенной ниже относительный порядок двух операций чтения не определен, но обе эти операции случаются после операции записи, поэтом здесь проблем не возникает.



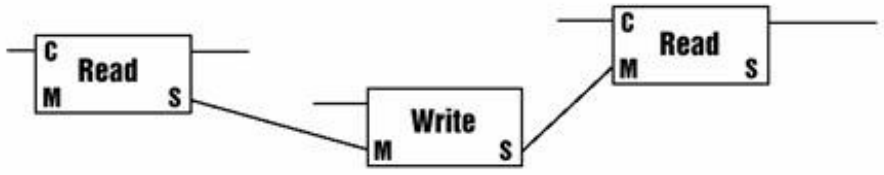
В следующей структуре относительный порядок первой и второй операции чтения также не определен. Эта структура может быть потенциально опасной (и таких структур нужно избегать):



Лучший путь реализации этой структуры такой:



или такой:



Относительный порядок операций записи важен. Относительный порядок операций чтения не так важен, пока их порядок относительно операций записи остается определенным.

OBC-связи не совместимы с нормальными сигнальными соединениями. OBC-связи, соответствующие различным типам объектов (например, различной точности представления числа в памяти) так же несовместимы друг с другом. Порты несовместимых типов не могут быть соединены, и, например, нельзя соединить обычный сигнал выходного порта к OBC-входу.

Инициализация

Поскольку мы начинаем работу с объектами, которые имеют внутреннее состояние (в случае модулей **Read** и **Write** внутренним состоянием является их общая память), важно знать, что такое **начальное состояние** проектируемой структуры. Например, если мы читаем значение из памяти (используя модуль **Read**) до того как туда что-то записали, какое же значение будет прочитано? И если нам необходимо другое начальное значение – как мы можем его изменить?

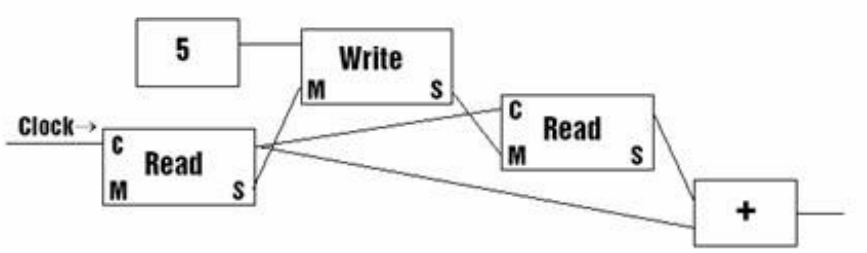
Эти вопросы адресованы механизму инициализации Reaktor Core.

Инициализация структур ядра выполняется следующим образом: вначале все элементы инициализируются некоторым значением по умолчанию, обычно нулем. Вся разделяемая память и все значения выходных портов будут установлены в ноль, за исключением тех, значения которых явно указаны в документации.

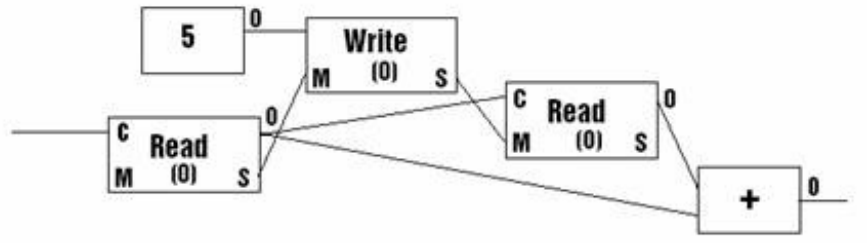
Во-вторых, **событие инициализации** посылается одновременно всем источникам инициализации. Источники инициализации это модули, которые не имеют входных портов: модули констант (включая и **QuickConsts**), входные порты ячеек ядра (обычно), и некоторые другие. Источники, которые обычно посылают их начальные значения в течение события инициализации, например, константы, пошлют свои собственные значения на входной порт ячейки ядра и этот порт будет инициализирован этим значением.

Если модуль не является источником события инициализации, то он обрабатывает событие инициализации также как и любое обычное входящее событие. В большинстве случаев источники событий инициализации это только те модули, которые не имеют входов.

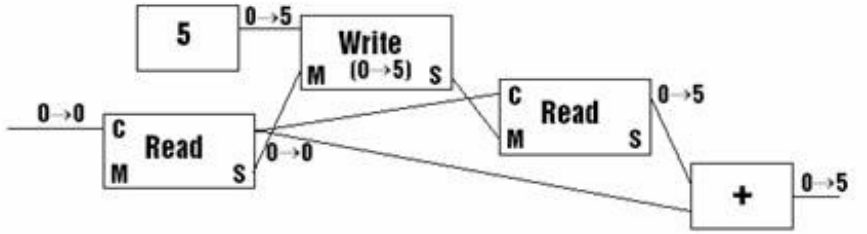
Рассмотрим, как работает инициализация:



Модуль **Read** слева соединен с некоторым источником синхросигнала, который также посылает событие инициализации (источники синхросигналов обычно так и делают). Первоначально все сигналы выходных портов и внутренние состояния цепочки **Read-Write-Read** установлены в ноль.

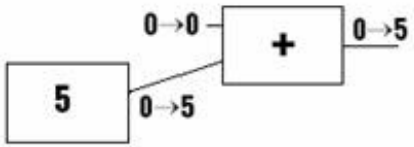


Событие инициализации посылается одновременно от источника синхросигнала и от константы 5.



Левый модуль **Read** обрабатывается до модуля **Write** и поэтому событие от модуля синхронизации прибывает до того как новое значение будет записано в память, поэтому выходной порт этого модуля **Read** выдаст ноль. Затем значение будет записано в память модулем **Write**. Теперь второй модуль **Read** произведет чтение и выведет в выходной порт уже другое значение – 5. И, наконец, модуль-сумматор выведет значение 5.

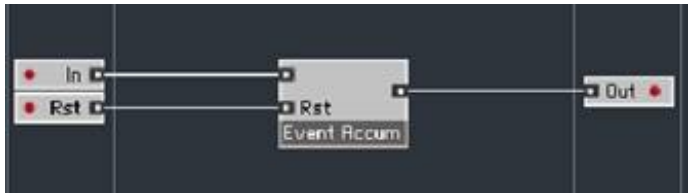
Как вы уже знаете, несоединенные входы трактуется в Reaktor Core, как правило, как нулевые значения, а более точно – как подсоединенные константы с нулевым значением. Это означает, что эти входные порты также получают событие инициализации, как будто бы к порту была нулевая константа.



На рисунке выше, модуль сумматора получает два одновременных события инициализации, одно с присоединенной константы, а другое с константой по умолчанию верхнего порта.

Создание аккумулятора (сумматора) событий

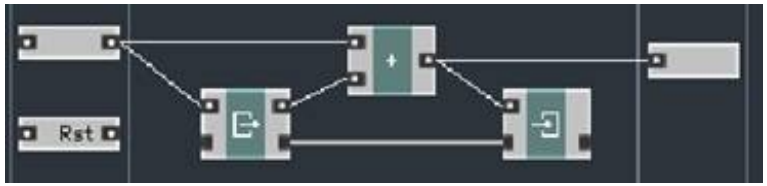
Модуль-аккумулятор будет иметь два входных порта – один для событий, которые будут обработаны, а другой – для сбрасывания аккумулятора в ноль. Выходной порт один – будет выдавать накопленную сумму значений пришедших событий. Начнем построение аккумулятора с пустого макроса внутри ячейки ядра.



Внутренность макроса:

Очевидно, что аккумулятор должен иметь внутреннее состояние (для хранения суммы значений пришедших событий), поэтому здесь необходимо использовать модули **Write** и **Read** (они расположены в контекстном меню *Built-In Module > Memory*).

Модуль слева – это **Read**, справа – **Write**. В ответ на входящее событие аккумулятор должен взять текущее значение из памяти и прибавить к нему значение пришедшего события (а также записать новое значение в память). Таким образом, мы используем модуль **Read** чтобы узнать текущее состояние счетчика, далее используем модуль сумматора чтобы добавить новое значение к нему, и в конце модулем **Write** запишем полученную сумму.



Заметим, что модуль **Read** синхронизируем по входящему событию, и, конечно, он расположен ниже по потоку (то есть в режиме ведомого) нежели ОВС-связанный с ним модуль **Write**, для того чтобы чтение памяти производилось до записи. На рисунке выше изображена структура реализующая это.

Теперь нам надо создать механизм обнуления аккумулятора. Давайте представим, что оба входных порта одновременно генерируют событие. Что тогда произойдет раньше – обнуление счетчика или же обработка входящего события для сложения? (Это похоже на различие между функциональностью **Latch** и $\mathbf{Z^{-1}}$, которые различаются только по относительному порядку обработки события и синхросигнала). Мы предлагаем использовать подход, реализованный в **Latch**, потому что этот модуль очень часто используется в структурах Reaktor Core, и таким образом, делать более интуитивно понятные и привычные структуры. В **Latch** сигнал синхронизации логически прибывает позже, чем сигнал значения события. В нашем случае сигнал сброса счетчика должен прибыть логически позже аккумулированного сигнала (вызывающий изменение и вывод нуля). Поэтому мы должны, так или иначе, отменить вывод сумматора с начальным значением. Чтобы достичь этого придется использовать новый принцип слияния событий.

Слияние (объединение) событий

Мы выдели различные способы комбинации двух разных сигналов в Reaktor Core, включая алгоритмические операции и т.д. То, что еще не известно – это просто объединение двух сигналов. Такое объединение не является сложением, оно означает, что результатом операции является последнее пришедшее значение события, а не сумма всех входящих значений. Чтобы объединить сигналы необходимо использовать модуль **Merge**. Рассмотрим, как он работает. Начальное значение выходного порта модуля (до события инициализации) – как и у всех модулей нулевое.

Когда событие прибывает на нижний порт со значением 4 – это же значение выводится на выходной порт:

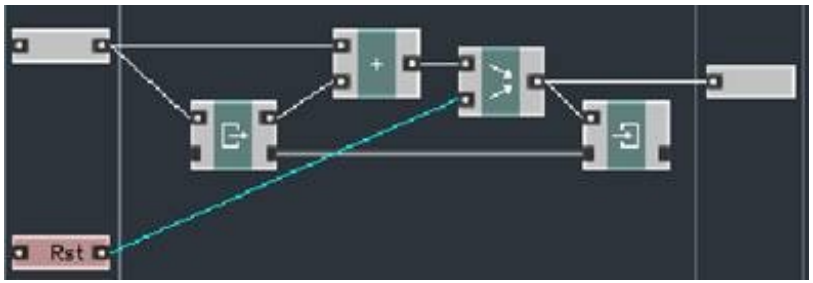
Следующее событие приходит по верхнему порту со значением 5:

Теперь сразу два события одновременно приходят на оба порта:

Здесь существует определенное правило для модуля **Merge**: события, прибывающие на два входных порта одновременно, обрабатываются в (обратном???) порядке нумерации входных портов (см. на рисунке порты обозначены 1 и 2). Выходным портом генерируется только одно событие, потому как порт модуля не может генерировать два события одновременно. В этом случае событие из второго входного порта будет произведено выходным портом после первого события, заменяя значение 8 новым значением 2.

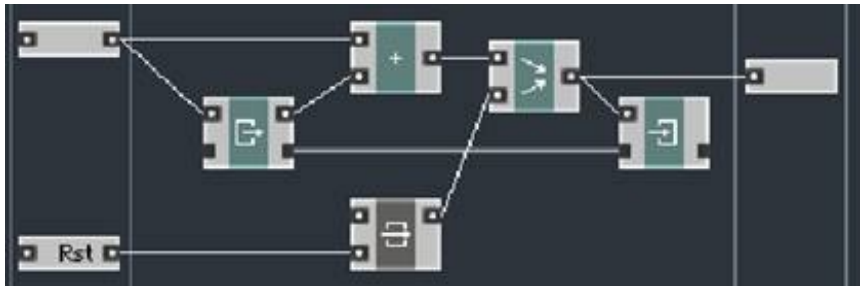
Аккумулятор событий с инициализацией и сбросом

Итак, чтобы создать функцию сброса счетчика необходимо заменить значение выходного порта сумматора другим начальным значением. Чтобы это сделать необходимо использовать модуль **Merge** (он расположен в подменю *Built-In Module > Flow*). Простейший путь – соединить второй входной порт модуля **Merge** с входным портом **Rst**.



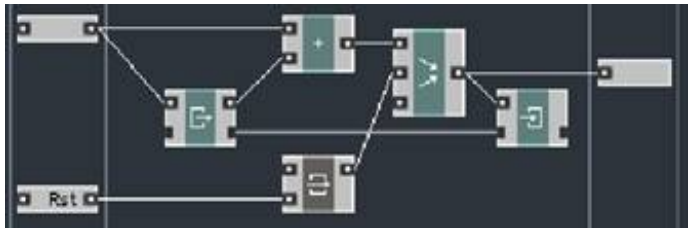
Теперь событие от порта сброса будет немедленно послано на модуль **Merge**, перекрывая значение выходного порта сумматора, событие от которого должно прибывать в тоже самое время. Оттуда сигнал записывается во внутреннее состояние аккумулятора. В этой структуре значение порта **Rst** будет использовано как новое значение для аккумулятора. Возможно это не такая уж и плохая идея, но это работает не точно как функция сброса, а как функция установки (по типу встроенного в библиотеку Reaktor Core модуля аккумулятора событий). Если мы хотим сделать настоящую функцию сброса мы должны записать только нулевое значение во внутреннее состояние, а не то, которое пришло из порта **Rst**. Поэтому в модуль **Merge** как-то нужно послать именно нулевое значение. Отправку на порт выхода события с заданным значением в ответ на любое входящее событие (а такое требуется довольно часто в Reaktor Core)

можно осуществить используя модуль **Latch**  (его можно найти в подменю *Expert Macro > Memory > Latch*). Как уже известно, этот модуль имеет два порта – порт значения (верхний) и порт синхронизации (нижний). Нам нужно соединить входной порт **Rst** с портом синхронизации модуля **Latch** чтобы модуль отправил событие, также нужно соединить верхний порт с нулевой константой для того, чтобы значение выходного порта было всегда нулевым (в данном случае можно и не соединять, потому что по умолчанию значение для несоединенного порта устанавливается также в ноль).



Теперь порт сброса работает так, как и было задумано.

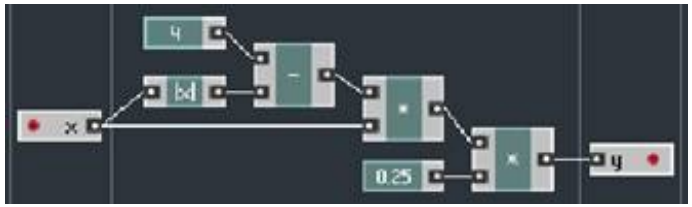
Последнее что необходимо проверить – это корректную инициализацию. Давайте рассмотрим, как эта структура будет инициализирована. Если событие инициализации посылается одновременно с портов **In** и **Rst** ячейки ядра из структуры верхнего уровня и также от неявной нулевой константы (которая у верхнего порта модуля **Latch**), то **Latch** пошлет ноль ко второму порту **Merge**, отменяя любое значение, прибывающее на первый порт. Поэтому ноль будет записан в начальное состояние аккумулятора и также выведен на выходной порт макроса. Но здесь есть небольшая проблема. Может произойти так, что событие инициализации не достигнет одного или обоих портов. Это может произойти, потому что событие инициализации не достигнет соответствующего входного порта ячейки ядра, или потому что макрос используется в более сложной структуре Reaktor Core, которая также не получает событие инициализации на всех ее проводах (об этом будет разговор позже). Итак, нам нужно сделать еще небольшую модификацию структуры, чтобы сделать ее более универсальной. Нужно отобразить окно свойств модуля **Merge** и сменить количество входов модуля на 3.



Теперь, если событие не прибудет с входа **Rst**, неявная нулевая константа на третьем входном порту модуля **Merge** произведет событие инициализации и пошлет событие инициализации на выходной порт.

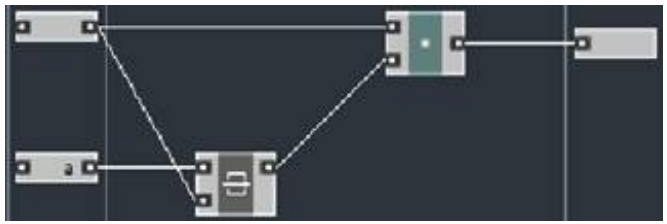
Модернизация шейпера событий

Теперь мы можем обсудить более подробно, что сделано не совсем верно в структуре шейпера, рассмотренной ранее:



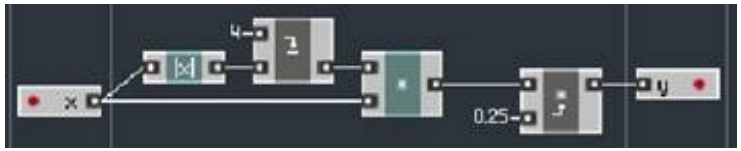
В этой структуре есть проблема с событием инициализации. Если рассмотреть как происходит инициализация этой структуры, можно заметить следующее: во вход **x** попадает или не попадает событие инициализации в зависимости от того, получает он событие инициализации снаружи (из первичного уровня) или нет. Константы 4 и 0,25 всегда генерируют событие инициализации.

Таким образом, в случае если на входном порту макроса инициализация не произойдет, тем не менее, выходной порт макроса получит событие инициализации от последнего умножителя (что ускорит инициализацию всех модулей расположенных после макроса шейпера). Хотя для сигналов управления это может быть и нормально (в случае потери входной инициализации входной порт считается нулевым, и событие инициализации также выводится в выходной порт макроса), это не то, что можно было бы ожидать от модуля обработки событий. Более правильным (и интуитивным) поведением для такого модуля было бы то, чтобы он посылал бы в выходной порт событие только в ответ на входящее событие. Итак, проблема состоит в том, что наши две константы могут посылать события, когда это нам не нужно (это происходит потому что они не принимают никаких событий). Решением этой проблемы может служить замена некоторых модулей (вычитания и умножения – то есть тех, которые соединены с константами) на другие им подобные из подменю *Expert Macro > Modulation* . Название раздела меню «модуляция» хотя и не совсем верное, но все-таки отражает свойство содержащихся в нем модулей использовать один сигнал для модуляции другого (логичность такого названия будет видна позже, когда будут использоваться сигналы управления для модуляции аудиосигнала). Большинство макросов из этого раздела комбинируют два сигнала – несущую и модулятор. В отличие от встроенных в Reaktor Core арифметических модулей макрос модуляции генерирует событие выходного порта только в ответ на входящее событие несущей. Событие модулятора не вызывает процесс пересчета. Внутренняя структура макроса модуляции очень проста: он запирает сигнал модулятора (модулем **Latch**), который синхронизируется сигналом несущей. Вот пример макроса-модуляции умножителя:



Замок на входном порту гарантирует, что значение модулятора будет послано только тогда, когда прибудет событие с порта несущей.

Итак, после замены модуля вычитания на **a-x**, и последнего модуля умножения на **x mul a** структура выглядит так (также модули констант заменены на **QuickConst**, что эквивалентно):



Обычно по рисунку на модуле макроса модуляции можно определить, какой порт является модулятором (отображается стрелкой), а какой несущей. Также можно причитать всплывающие подсказки, задержав курсор мыши над портом.

Итак, вышеприведенной структурой события посылаются только в ответ на входящее в структуру событие.

Аудиообработка в ядре Аудиосигналы

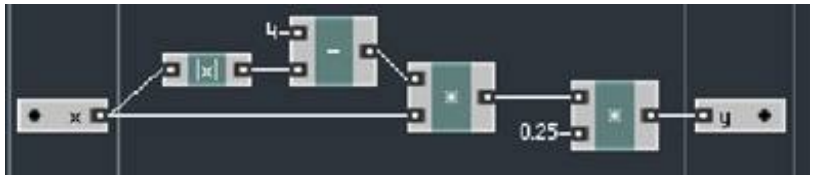
В Reaktor Core не существует специального типа аудиосигнала. Аудиосигналы представляются событиями так, что с точки зрения структуры они ничем не отличаются от любых других событий. Что отличает аудиосигналы от прочих событий это то, что они обычно равноотстоящие друг от друга и генерируются регулярно с частотой заданной в sample rate. Чтобы сгенерировать такие события необходим какой-нибудь источник событий. Как и в event-ячейках ядра, где входные порты модуля – это источники сигналов событий, в аудиоячейке ядра выходные порты также могут быть источниками событий. В дополнение к этому в аудиоячейке может быть входной аудиопорт:

Audio input периодически генерирует события ядра внутри структуры на частоте, определенной в sample rate первичной структуры. События посылаются одновременно со всех входных аудиопортов ячейки. Аудиопорты также посылают события инициализации в структуру ячейки ядра. Эти события посылаются независимо от того, что происходит в структуре первичного уровня. Тем не менее, значение, посылаемое этими входными портами в течение инициализации зависит от процесса инициализации снаружи.

Также в аудиоячейке есть новый выходной аудиопорт:

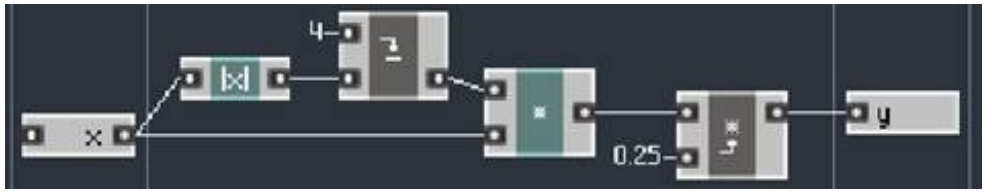
Audio output доставляет последнее полученное значение изнутри структуры на первичный уровень Reaktor.

Теперь мы реализуем аудиорежим того же самого шейпера, который мы делали ранее для обработки событий. Для этого создадим новую аудиоячейку. Структура модуля повторяет ранее реализованную, за исключением типа входного и выходного порта.



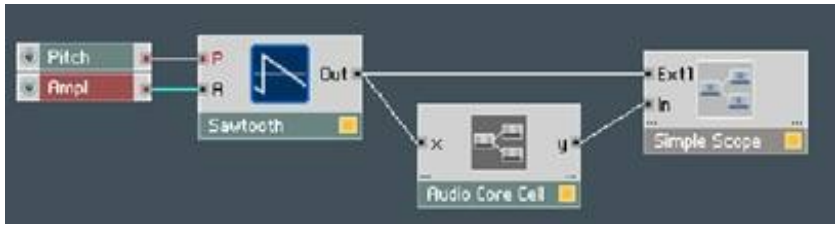
Здесь использованы обычные арифметические модули, а не модули-модуляторы, потому как здесь ведется обработка аудиосигнала, и аудиосигнал всегда посылает событие инициализации (т.о. разницы между макросами модуляции и арифметическими модулями здесь нет – можно использовать любые).

Реализуем нашу структуру в форме макроса:



Внутри ячейки он будет выглядеть так:

Чтобы протестировать нашу ячейку создадим простую структуру с пилообразным осциллятором и стандартным осциллографом (который расположен *Insert Macro >Classic Modular >00 Classic Modular – Display >Simple Scope*):



Также необходимо установить количество голосов инструмента в 1.

Для того чтоб осциллограф был автоматически синхронизирован с сигналом, на порт **Ext** подается сигнал от осциллятора (чтобы это работало нужно еще дополнительно подсветить одноименную кнопку на панели осциллографа). Вращая регулятор **Ampl** можно наблюдать как шейпер модифицирует сигнал.

Шина синхронизации и частота семплирования

Многим DSP-алгоритмам необходимо знать какова текущая частота дискретизации. Reaktor Core обеспечивает возможность создавать в каждой структуре ядра подключение к особой шине синхронизации **sampling rate clock bus**. Эта шина несет два сигнала: сигнал синхронизации и частоты.

Источник сигнала синхронизации (**clock**) посылает регулярные события с частой дискретизации аудиосигнала (sampling rate). Он также посылает события инициализации, как и все стандартные аудиосигналы. Значения всех событий нулевые, но обычно эти значения не используются в структуре, поэтому разработчики оставляют за собой право изменять в будущем их значения.

Rate – это источник сигнала, значение которого всегда равно текущей частоте дискретизации (в герц). Событие посылается из этого источника в течение процесса инициализации, а также в случае изменения частоты дискретизации аудио.

Можно подключить входной порт к нужной шине используя контекстное меню. **Connect to SR.C** - подключение к синхросигналу, **Connect to SR.R** для сигнала текущей частоты семплирования.

Эта шина не работает в event-ячейках ядра.

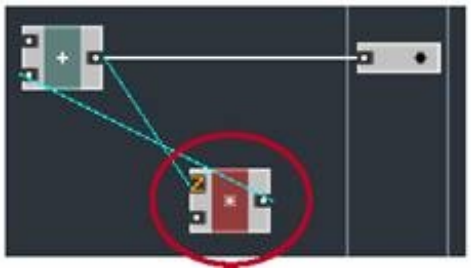
Структуры с обратной связью

Как известно, правило порядка обработки модулей не может быть применено, если в структуре имеются обратные связи. Поэтому необходимы дополнительные правила, определяющие как же работать с обратной связью.

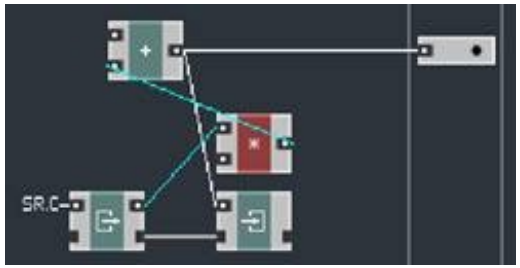
Технически, структуры Reaktor Core не могут обрабатывать обратную связь. Но это не совсем так. Можно создавать обратную связь в Reaktor Core, но потому как движок Reaktor Core не может обработать такие структуры с обратной связью, он их некоторым образом преобразует. Избавление от обратной связи означает, что структура будет внутренне модифицирована (но вы не увидите это на экране) так, что в ней не будет обратных связей.

Причина необходимости такого решения – это то, что без обратной связи не осуществить эффект задержки. Обычно минимальная задержка на один семпл в цифровом пути обратной связи, решает проблему обратной связи. Это и делает Reaktor Core в процессе преобразования структуры – он вводит модуль задержки в один семпл (**Z^-1**) в путь обратной связи.

Как вы уже, наверное, знаете, размещение такой встроенной задержки отображается большой оранжевой буквой **Z** в том месте, где обычно расположена иконка порта:



Мы уже видели структуру, реализующую эту **Z^-1** задержку, в которой были использованы модули **Read** и **Write**. Давайте попытаемся вставить эту конструкцию в нашу структуру. Мы вставим ее на провод в то место, на котором это разрешение имело место:



Итак, сначала мы пишем, и только затем читаем (заметим, что модуль **Read** синхронизирован по шине SR.C, то есть чтение будет совершаться один раз на отсчет аудио). Это делает прочитанное значение всегда на один семпл позади значения записи. Сейчас в

структуре обратной связи нет.

Итак, вставка скрытого модуля односемплерной задержки обычно удаляет обратную связь из структуры, но сохраняет ее логически (с задержкой на один семпл).

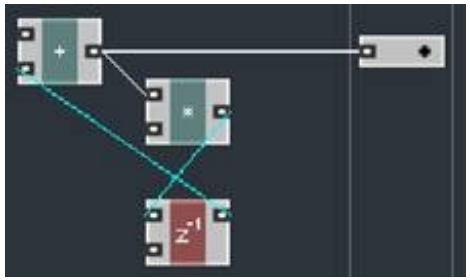
Фактически, внутренняя структура макроса **Z ^-1** чуть более сложна, чем пара модулей чтения и записи. Об этом - в следующем разделе .

Нельзя управлять тем местом, где произойдет автоматическое разрешение обратной связи. Оно происходит на произвольном сигнальном проводе в цикле обратной связи. Это не гарантирует, что разрешение будет всегда происходить на каком-то определенном проводе – положение его может измениться в следующей версии программы, оно может измениться при некоторой модификации структуры, или даже в случае повторной загрузки структуры с диска.

Следовательно, автоматическое разрешение обратной связи для самой структуры означает, что для нее не важно где в точности происходит это разрешение. Например, такие структуры обычно могут быть созданы тем пользователем, который еще не очень глубоко понимает работу DSP-устройств. Тем не менее, автоматическая обратная связь позволяет делать работоспособные структуры.

Если вы нуждаетесь в точном контроле над точкой разрешения обратной связи, можно явно вставить модуль **Z^-1** в структуру. Такой модуль явно устраняет обратную связь, и механизм разрешения обратной связи уже не потребуется.

Вот версия нашей структуры, в которой явно используется модуль **Z^-1** (доступен из подменю *Expert Macro >Memory*):



Видно, что большая оранжевая буква Z исчезла. Теперь заметим, чем односемплерная задержка отличается от такой же задержки вставленной автоматически: автоматическая задержка была установлена на проводе идущего от сумматора до входного порта умножителя, а теперь задержка происходит после выходного порта умножителя. Значение второго порта модуля **Z^-1** будет объяснено позже (здесь его можно оставить неподключенным).

Обратная связь на OBC-подключениях и других типах несигнальных подключений (которые будут рассмотрены позже), не имеет никакого смысла и, поэтому, запрещена.

Если такие петли обратной связи все же происходят, то одно из подключений будет отмечено как недопустимое. Такая связь отмечена красной буквой X на порту.

С другой стороны, петли обратной связи со смешанными типами подключений допускаются, если они содержат некоторые сигнальные провода; в этом случае они будут разрешены обычным способом, с разрешением, происходящем на одном из проводов сигнала:

Несигнальные подключения никогда не затрагиваются механизмом разрешения обратной связи.

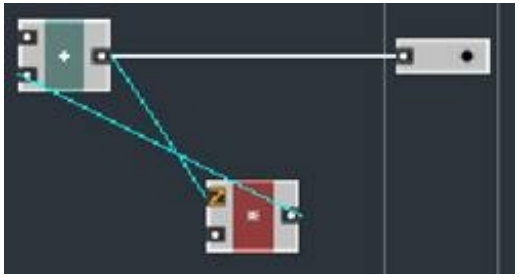
Обратная связь через макрос

Таким же образом, что и модули, механизмом разрешения обратных связей обрабатываются и макросы. Рассмотрим макрос, только лишь транслирует сигнал через себя:

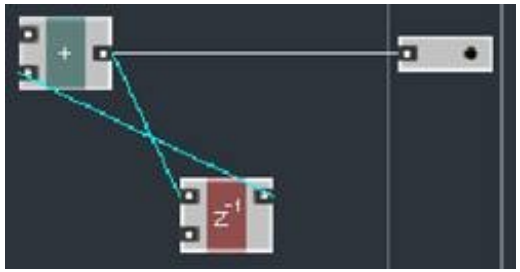
и создадим обратную связь:

Петля обратной связи состоит из двух проводов и проходит через внутренность макроса. На рисунке выше видно, где произошло разрешение. Но где же оно может происходить в принципе? Представим на мгновение, что макрос **Thru** – это встроенный модуль. В этом случае разрешение не могло бы произойти где-то внутри модуля, оно произошло бы только снаружи. Разработчиками это так и было реализовано: по умолчанию цепь разрешается вне макроса. Как правило, разрешение от петли обратной связи совершается на самом высоком уровне структуры. Однако, это можно изменить, и позволить разрешать петли внутри макроса.

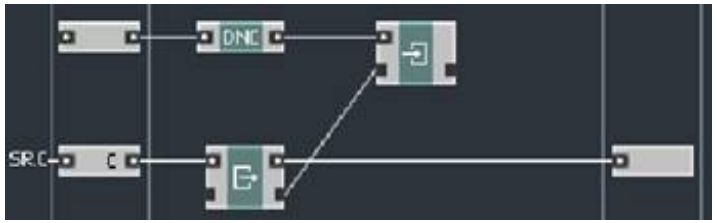
Фактически, нужно задаться вопросом: если макрос обрабатывается подобно встроенному модулю, как же макрос **Z^-1** может разрешать обратную связь? Рассмотрим структуру:



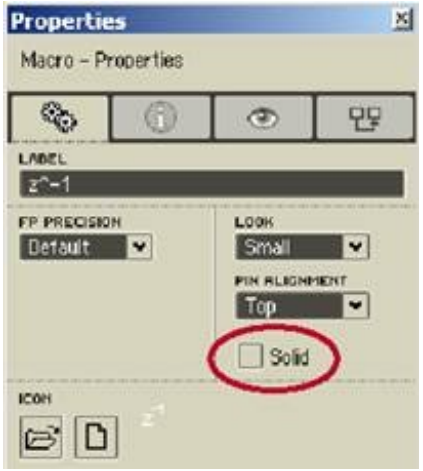
Если макросы и встроенные модули эквивалентны, то ничего не должно измениться, если вместо умножителя вставить модуль односемплерной задержки:



Но эти структуры уже различны, потому как неявной обратной связи уже нет. Значит в макросе **Z^-1** должна быть какая-то дополнительная функциональность, и это так и есть. Рассмотрим структуру этого макроса:



Модуль чтения синхронизирован по шине SR.C (т.е. с частотой обработки аудио). Значение порта **C** по умолчанию – не ноль, а сигнал синхронизации аудио (модуль **DNC** рассматривается позже). Каким же образом механизм разрешения петель знает, что в данной структуре нужно осуществить разрешение обратной связи? Этим управляет установка **Solid** в свойствах макроса.



Solid (переводится как – твердый) говорит движку Reaktor Core нужно ли рассматривать макрос как встроенный модуль в случаях разрешения обратных связей. В 99% случаях нет необходимости снимать здесь галочку, и разрешать петли обратной связи внутри макроса. Хорошая практика – не разрешать цепи в макросах (такое разрешение довольно неудобно: вы не увидите, что в структуре есть глобальная обратная связь, пока не отобразите структуру макроса).

Нестандартные уровни сигнала

Значения сигнала в структурах, которые мы создавали ранее, представимы внутри компьютера бинарными данными в форме чисел с плавающей точкой. Такие числа эффективно представляют широкий диапазон численных значений. Термин «число с плавающей

точкой» не говорит нам, какое же внутреннее представление они имеют. Центральные процессоры персональных компьютеров использую сейчас стандарт представления чисел с плавающей точкой созданный IEEE. Этот стандарт точно определяет, как числа должны быть представлены, все операции над ними и т.д. В частности, стандарт говорит, что группа чисел с особо малыми значениями из-за ограничений представления чисел не может быть выражена обычным путем, и должна использоваться дополнительная специальная форма их представления. Эта форма называется «нестандартным» представлением.

Например, для 32-float сигнала диапазон «нестандартных» значений простирается от 10^{-38} до 10^{-45} и от -10^{-38} до -10^{-45}. Значения меньшие, чем 10^{-45} не могут быть представлены и считаются эквивалентными нулю.

По той причине, что представление таких чисел несколько отличается от нормального, некоторые центральные процессоры имеют проблемы с обработкой таких чисел. Они, в частности, могут выполнять операции над такими числами гораздо медленнее (в 10 раз и более).

Типичная ситуация где появляются нестандартные числа на длительных периодах времени – это вычисление экспоненциально убывающих значений (возможных в фильтрах, некоторых огибающих, и цепях обратных связей). В таких структурах, после того как входной сигнал достигает нулевого значения – сигнал выхода затухает в ноль асимптотически. То есть, этот сигнал становится все ближе и ближе к нулю, но его не достигает. В этой ситуации нестандартные числа могут появляться и оставаться в структуре относительно долгое время (до тех пор, пока их значение не упадет ниже 10^{-45}), и это может существенно повысить вычислительную нагрузку на процессор.

Другая ситуация, где могут происходить нестандартные числа, это ситуация, в которой изменяется точность представления значений сигнала с 64-битного на 32-битный, потому как значение, например 10^{-41}, не является нестандартным в 64-битном представлении.

Давайте рассмотрим моделирование аналогового фильтра низких частот первого порядка с частотой среза 20 герц. Значения нашего цифрового сигнала соответствуют аналоговому напряжению (в вольт). Представим, что на вход фильтра подается сигнал в 1 вольт достаточно долгий период времени. Тогда напряжение на выходе фильтра также равно единице. Теперь резко изменим входное напряжение в ноль. Выходное напряжение будет затухать по такому закону: $V_{out} = V_0 e^{-2\pi f_c t}$

f_c – частота среза фильтра в герц, t – время в секундах, V_0=1 начальное напряжение.

Выходное напряжение будет изменяться так:

после 0.5 сек Vout около 10^{-29} вольт

после 0.6 сек Vout около 10^{-33} вольт

после 0.7 сек Vout около 10^{-38} вольт

после 0.8 сек Vout около 10^{-44} вольт

Т.о. в периоде между 0.7 и 0.8 секунд наше напряжение представимо нестандартными числами. И такое происходит не только внутри фильтра. Сигнал выходного порта фильтра обрабатывается далее по потоку, и заставит, по крайней мере, несколько нижележащих модулей обрабатывать такие нестандартные числа. На частоте семплирования 44.1 килогерц и временному интервалу в 0.1 секунд соответствует 4 410 отсчетов. Буфер ASIO имеет обычно несколько сотен отсчетов и при высокой загрузке центрального процессора его придется увеличить (иначе будут выпадать отсчеты).

Из всего вышесказанного можно заключить, что нестандартные числа – плохи в аудиопроцессинге реального времени.

Модули первичного уровня запрограммированы так, что в них обычно исключается возможность появления нестандартных чисел. Обычно можно модифицировать DSP-алгоритмы так, чтобы они не производили нестандартных чисел. Если вы разрабатываете собственную низкоуровневую структуру в Reaktor Core, то вы также должны заботиться о том чтобы избавиться от нестандартных значений сигнала. Помочь в этом может модуль **Denormal Cancel** (**DNC**), который можно найти в меню *In Module> Math* . Этот модуль

имеет один вход и один выход, и он пытается немного модифицировать входящее значение таким образом чтобы на его выходе не появлялись нестандартные значения.

Способ, которым изменяется сигнал, может различаться от версии программы, и даже от местоположения этого модуля в структуре! Обычно он добавляет к входному сигналу константу с очень маленьким значением. Чтобы не потерять в точности представления числа, этого добавления не происходит на относительно больших значениях (порядка 10^{-10}).

Если по какой-то причине модуль **DNC** не используется в вашей структуре, вы, конечно, можете использовать свои собственные методики по избавлению от нестандартных чисел. Но проблема может состоять в том, что на одной вычислительной системе тот же самый механизм может и не работать, а алгоритм с использованием встроенного модуля **DNC** будет работать везде.

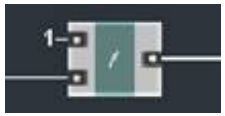
Некоторые процессоры предлагают опцию, нарушающую стандарт IEEE, блокируя работу с нестандартными числами, и устанавливают все нестандартные числа в ноль. Поскольку структуры Reaktor Core разрабатываются в качестве платформонезависимых, разработчики настоятельно рекомендуют всегда избавляться от нестандартных чисел в разрабатываемых структурах, даже если система, в которой вы разрабатываете структуры, нормально обрабатывает нестандартные числа.

Также распространенная ситуация возникновения нестандартных чисел – это экспоненциально затухающие петли обратной связи аудиосигнала (например, сюда включаются фильтры, обратные связи с задержкой и т.д.). Именно в связи с этим в стандартный макрос **Z^-1** встроен модуль **DNC** (см. структуру макроса). Поскольку макрос часто используется в цепях обратной связи, нестандартные значения он будет получать часто.

Также существует версия этого макроса без встроенного модуля **DNC** (этот модуль называется **Z^-1 ndc**, ndc = no denormal chancel). Если вы уверены что в структуре не возникает нестандартных чисел, то можно использовать этот модуль.

К плохим для обработки числам относятся числа, возникшие как результат недопустимых операций (самый простой пример – деление на ноль). Некоторые другие случаи вычислений приводят к слишком большим численным значениям, а также выходят за разумные пределы осуществления вычислительной операции. Такие числа часто «застревают» в структурах и от них избавиться гораздо сложнее, чем от нестандартных. Если нестандартное значение складывается со стандартным – то в результате получается стандартное значение. Если же стандартное значение складывается с бесконечным (INF) то в результате сложения получится та же бесконечность (для обработки сигналов бесконечность – бессмысленна). Помимо блокирования нормальной работы структуры, такие числа часто приводят к более сильной вычислительной загрузке процессора. Возникновению таких чисел нужно активно препятствовать. Это означает, что тогда, когда, например, вы используете модуль деления – нужно следить чтобы знаменатель не обращался в ноль. События инициализации в этом случае также требуют пристального внимания.

Например, рассмотрим следующий фрагмент структуры:



Если по какой-либо причине событие инициализации не придет на нижний порт модуля деления, то в процессе инициализации произойдет деление на ноль. В этом случае лучше использовать аналогичный по функциональности модуль модуляции.