

THE ESSENTIAL GUIDE TO REAKTOR EVENTS

Chris List

Revision 1.0

Copyright 2009, Christopher List

clist@mindspring.com

Table of Contents

| | |
|--|----|
| Preface | 3 |
| Disclaimer..... | 3 |
| Prerequisites..... | 4 |
| 1. Introduction | 5 |
| 1.1 What is an “Event”? | 5 |
| 1.2 Why Do We Like Events? | 6 |
| 1.3 Tools of the trade – the EventWatcher | 9 |
| 2. Event Sources | 12 |
| 2.1 Defining “Event Source” | 12 |
| 2.2 When do Event Sources Create Events?..... | 14 |
| Event Initialization and the Global Reset | 14 |
| GUI and MIDI Input | 15 |
| The Control Rate Clock and “Steaming” Sources..... | 15 |
| 2.3 Non-Source Event Modules..... | 15 |
| 3. Event Life and Death | 16 |
| 3.1 Events as Values | 16 |
| 3.2 Events as Triggers..... | 16 |
| 3.3. Event Flow and Ordering | 16 |
| 3.4 Death of an Event..... | 18 |
| 4. A Brief History of (Event) Time..... | 19 |
| The Control Rate | 20 |
| 5. Event Module Details | 22 |
| 5.1 EventValue | 22 |
| 5.2 EventOrder | 22 |
| 5.3 Iteration | 22 |
| 5.4 EventTable..... | 22 |
| 5.5 AtoE..... | 22 |
| 5.6 Routers Selectors and Distributors..... | 22 |
| 5.7 EventHold..... | 22 |
| 6. Event Tricks and Common Mistakes | 23 |

Preface

I spend a lot of time answering a lot of similar questions on the forums related to event processing in Reaktor. FAQs are helpful in getting new users up to speed and for answering simple questions, but I've come to realize that people really need a reference that will allow them to get to a deep level of understanding on their own. Something that explains the nuts and bolts, the hows and why, is really essential if we ever hope to have a community in which people can quickly up to speed and building their own complex structures. The more quickly people understand how to program Reaktor, the more they'll be encouraged to use it to express their creativity. That creative expression is something we all benefit from; both musically and intellectually. That's what I hope to achieve in documenting this knowledge of mine.

I hope you find it helpful.

Happy building!
Chris List
New York, 2008

Disclaimer

This document represents one user's impressions of the operation of a very complex piece of software. The descriptions are based both on information in the manual as well as things I and other users have discovered during our years of working, experimenting, and analyzing. This is in no way an official document. The author has no financial relationship with Native Instruments. I've tried to make this document as accurate as possible as of Release 5 of the software. Please forgive me for any mistakes.

Prerequisites

You think reading a manual for Reaktor is tedious? Try writing one!

Seriously though, when people ask questions online about things that are spelled out clearly in the manual is what causes people to put the “F” in “RTFM”. It’s like saying; “I don’t have time to read the manual, but I expect you have time to answer my questions”.

With that in mind, this document is meant to be an addendum to the Reaktor manual, not a replacement. I assume that, at a minimum, you’ve read and understood chapters 9 through 19 of the Reaktor 5 Manual (in the Reaktor 5 Documentation folder). It would be helpful to read the entire manual and be familiar with the various primary modules – even if you don’t understand *exactly* how each one works. I assume that you’ve installed Reaktor and used it enough to begin building your own structures. Knowledge of core is not required; in fact a strong understanding of core without an understanding of primary-level event processing may end up confusing you as the logic for each has subtle but important differences. Lastly, I assume you know your multiplication tables and some basic algebra as these are things that everyone should learn in school before the age of eighteen.

Labs

This document comes with a set of folders that contain “lab” ensembles. I strongly encourage you to open the lab ensembles and follow along. They are there for you to play with, explore, and modify to get the sort of hands-on understanding that simply reading a manual cannot achieve. If you have any suggestion concerning the labs, please feel free to contact me.

<Labs have not yet been created!>

1. Introduction

1.1 What is an “Event”?

Ultimately, an “event” in Reaktor is an abstract concept used for manipulating data and tracking a series of procedural steps through an instrument’s structure. You can’t see, feel, touch, or even hear an event. Still, because Reaktor is graphical programming language, I find it helpful to have a mental picture of what an event looks like. When I visualize event processing, I think of the events like numbers stored inside little balls moving through tubes (wires). I imagine them speeding through the wires triggering different things to happen as they move along... almost like the balls that move through those big cartoon mouse-trap machines. The important thing about this analogy (or any that you come up with to suit your imagination) is that events have certain rules associated with their behavior and our visualization must follow those rules.

Event Characteristics

1. They are “created” or “instantiated” at some source, and from there they flow “downstream” through the structure causing things to happen along the way.
2. They have a value associated with them (the number inside the capsule), the number may change as they pass through other modules, but every event has some value.
3. Any time they arrive at an Event Module they trigger something to happen after which they may (depending on the module) continue on from an output port of the module. Event output ports with multiple wires connected to them will send a *different* event down each wire, but each event will have the same *value*.
4. At some point every event must stop moving, or “die”.ⁱ When they die, the “value” part of the message is left behind, but the event will not trigger any more actions. When an event “dies” it’s like the ball breaks and just the number inside is left.
5. There is only one “living” (or “active”) event at any given instant. A single event may cause many others to be created (“spawned”) as it flows through the structure, but they will be created and do their processing only after the previous event dies.
6. There is no *measurable* time between when an event is created and when it dies. Since #5 says that a single event may spawn other events, this means that 1000s of events may (and often do) spawn, process their actions, and die in an instant.

Characteristics 5 and 6 together seem to cause a lot of confusion for people. In fact each of these characteristics may be confusing for you. For now simply accept them as rules and don’t think too hard about them. The body of this document will be dedicated to addressing each characteristic in (hopefully) exhaustive detail.

ⁱ An event-loop can be created that causes an event to never die. In this case Reaktor may crash in a very unpleasant way. This is discussed in the section on Event-Loops.



A Note On Terminology

If you don't know what I mean when I say "one tick of the sample clock", you might want to skip ahead and do a quick read of **Chapter: 4. A Brief History of (Event) Time**, and then come back here.

1.2 Why Do We Like Events?

Reaktor sends values along "wires", but each wire is classified as either *audio* or *event*ⁱ. There are both similarities and differences between the two types of wires. Why do we want to use events? How are they different from audio? (at this point you might want to check out the Reaktor 5 Manual chapter 17.2 for a quick refresher)

CPU Savings

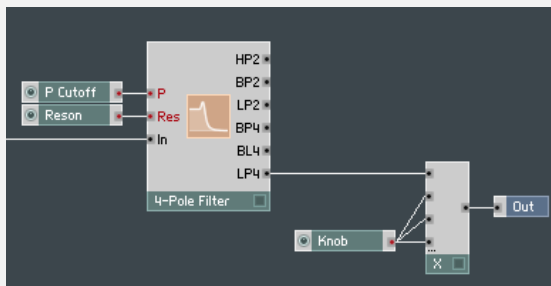
Every time the sample rate clock ticks, 44100 times per second (or more!), every *audio module*ⁱⁱ looks at its input ports, processes, and outputs a new sample value at its output port. That's a lot of work. If you have some processing to be done only when a midi note comes in or when a knob gets turned, it makes sense for the system to only do the work when that value changes and not on every tick of the sample clock. This is the primary role of events – to use processing power only when it's needed for things that don't change very often.

We can build our structures to take advantage of this by keeping as much processing as possible in the event realm and only sending fully "massaged" data to the audio modules. By doing this we save CPU.

Example: CPU Savings from Events

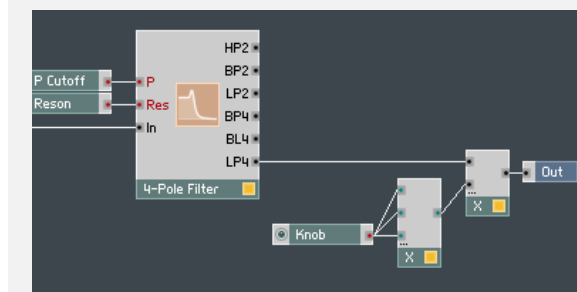
It's common to use a "power" function like x^3 to simulate an exponential (a.k.a. "reverse logarithmic") volume curve. I personally find this power curve more musically useful for volume knobs than using a reverse-log dB scale for reasons I once outlined here: http://www.native-instruments.com/forum_us/showthread.php?t=16794

Now consider the following two structures



ⁱ The Reaktor manual sometimes refers to event wires as "control signals"

ⁱⁱ Any module with an audio output port



...mathematically they are the same.

If "Knob" goes from 0...1, then
 $\text{Knob} * \text{Knob} * \text{Knob}$
will also go from 0...1.

In each case we multiply this by the output of the filter to scale its amplitude and we have a final result of:

$\text{Knob} * \text{Knob} * \text{Knob} * \text{FilterOutput}$

...and the volume of the filter output is scaled from 0 (off) to it's max value (when Knob is set 1), but the "gain curve" as we turn the knob is no longer linear, it's exponential (read the above linked post if you aren't clear about that).

While the result of the structures is the same, there is a very important difference in the CPU usage of these two structures!

The first structure will do 3 multiplication operations on every tick of the sample clock because the whole multiply module is audio. Audio modules do all their work for every sample – they don't discriminate between values that change and those that don't. The second structure will do two multiplies only when the knob outputs an event (when we turn it or when Reaktor decides to re-initialize the values of events in the system), and only one multiply on each tick of the sample clock. This example represents just a tiny savings in CPU, but if you have more complicated math operations, or if you have 5 or 10 cases like this in your structure, the CPU usage adds up quickly!

The CPU savings from events coming from panel controls applies in the same way to incoming MIDI events, or ticks of a sequencer clock. Like the movements of panel controls these are things that happen much less frequently than the usual sample rate of 44100 times per second, so it makes sense for them to be processed by modules that only use CPU when they occur.

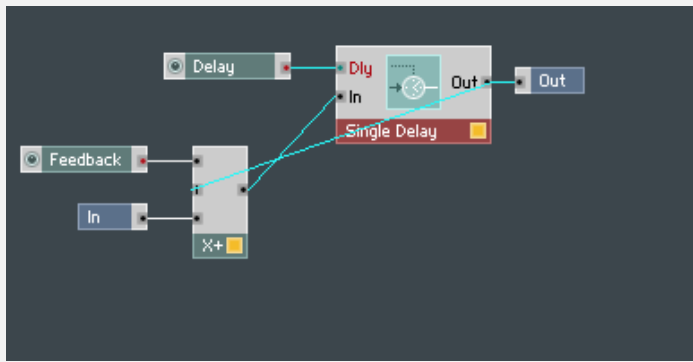
Iterations and Loops

The other reason why we use events instead of audio is that there are many cases where we need to do a lot of processing, or send a lot of information to one port instantly. A perfect example would be clicking a button that causes the values of an entire row of an event table to be copied to another table (or to a different row in the same table). Events and event-based modules allow us to process many different events arriving at the same event port all in the same tick of the sample clock. Audio signals don't work with this way. With audio, only the last value at the audio port when the sample clock ticks is recognized and used for processing.

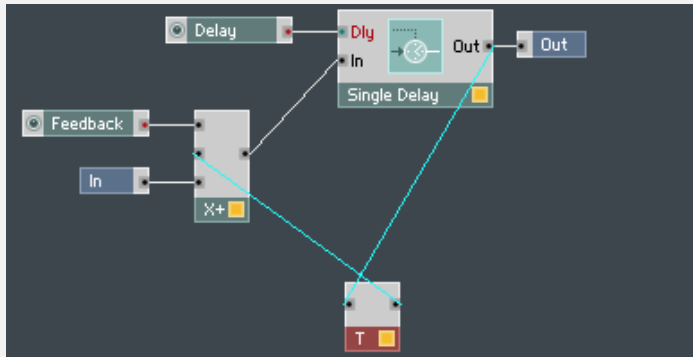
Similarly, audio does not allow us to have feedback loops that process *in the same tick of the sample clock*. If you send an audio output back to an audio input *upstream* of the module with the output, it creates feedback, but it puts in a hidden Unit-Delay into the structure somewhere in the feedback loop.

Example: Audio Loops

This...



...is the same as this...

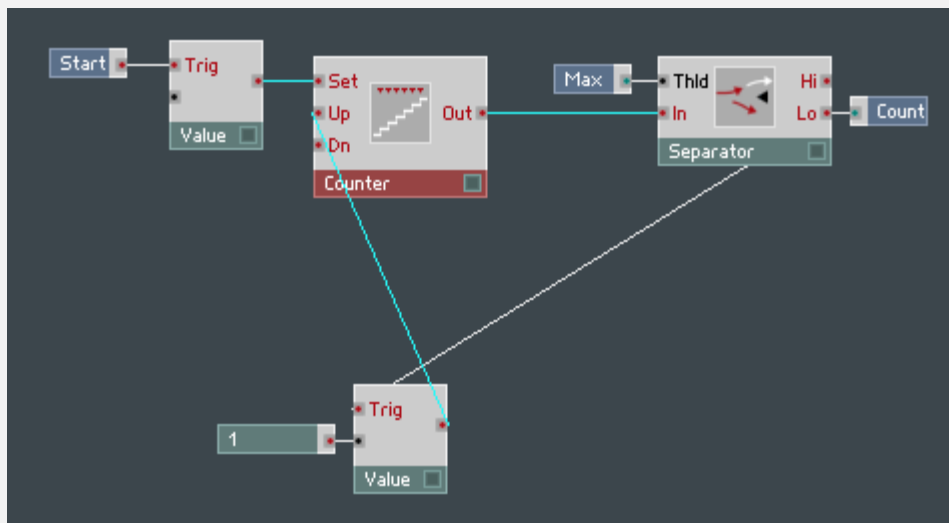


...but the unit-delay in the first structure is hidden by Reaktor and inserted automatically.

This means the value of the audio output will be fed back in the next tick of the sample clock, and not the current one (a “unit delay” delays by one sample).

Events, on the other hand, let you send an event around and around in a loop all in the same tick of the sample clock (provided you have Enable Event Loops activated for your ensemble). You must build your structure in such a way that this looping will stop at some point, for example a separator that says “only send the event back around if it’s less than 5”, otherwise you’ll crash Reaktor with an “infinite loop”. While event loops like this scare some people off (“If I have to do something special to enable event loops, aren’t they dangerous?”), loops represent a common programming idea (the “do...while” loop), and can often lead to more elegant and CPU-efficient structures than structures which try to avoid them. Just be careful of the “infinite loop” issue, it’s an awful feeling when you lose an hour’s work because Reaktor crashes from an infinite loop!

Example: Event Loop



In Release3 of Reaktor, there was no Iterator module. We had to make our own Iterator using counters and event loops. The structure shown is an example of a structure that will iterate from 0 to Max, incrementing by one each time.

Any event (regardless of its value) arriving at "Start" will reset the counter to 0 (the disconnected Value input on the EventValue module is the same as a constant of zero).

The setting of the counter to 0 will cause the Counter to pass the Set event to its output with the same value of 0.

If the event's value is \leq Max the Separator will pass the event to its Lo output. It will be sent to both the "Count" output, and back around to increment the counter.

The event's value will be set to 1. This is needed because the "Up" inputs of the Counter module will only increment the counter when they receive a positive value, and the first event being passed around will have a value of 0.

The event increments the counter and the counter passes it through to its output with a new value (the value of the current count).

The event goes into the separator again, and is checked against Max and the possibly gets sent around again.... And again...

This will all happen in one tick of the sample clock, "instantly" as far as we're concerned.

The above structure will *never* crash reaktor with an infinite loop because at some point the value of the counter will be higher than "Max" and the event will simply "die" in the separator and the counter will stop incrementing.

1.3 Tools of the trade

1.3.1 The Event Watcher

In both this document and on the forums I make frequent reference to the EventWatcher. It is an essential tool to understanding and debugging events. You can get my last version of the EventWatcher in the "Labs" folder of this document or here: <http://www.native-instruments.com/index.php?id=userlibrary&type=0&ulbr=1&plview=detail&patchid=2785>

...other community members have made enhancements to it (citations needed) that add various useful features.

Note that the current EventWatcher is little outdated now that R5 has come out (with its new event initialization scheme) and I'll probably release an update sometime soon to add some features and fix some initialization issues.

Why do you need it?

When I first started using R3 I used the mouse-hover over wires and sometimes Numeric Output panel modules to display the values of event lines. I quickly ran into two problems with this:

1. All you see is the *value* of the events (the pieces of paper inside the events). If 200 events pass by with the same value, or one event passes by you have no way of telling the difference if all you are doing is looking at the value.
2. Per Event Characteristic #6, two or more events can flow through the same wire (one after another) in the same tick of the sample clock. In this case a Numeric Output panel box will only show the value of the last event. You can't even tell that one (or possibly 100s) of previous events with other values went down the same wire. Sometimes you can ignore these other events, but many times you need to be able to see them to debug your work.

The EventWatcher addresses these issues by creating a log of each event that hits its input ports – even if they are all flying in during the same tick of the sample clock. The event watcher shows you events themselves, and not simply the value of the last event to have passed through a wire (see Chapter 3 for a more detailed discussion of Event-as-values vs. Events-as-triggers).

How do you use it?

In general you simply add the macro to a structure and connect any event output port to the input of the event watcher. Make sure the input to the EventWatcher is monophonic, and that the part of the structure where you're pasting the EventWatcher is visible. Make sure On switch on the EventWatcher is On. Hit the reset switch to clear the event watcher's "memory", then start watching events that arrive at the EventWatcher through the connected inputs!

The output ports on the EventWatcher macro, and why you might want to use them are described here:

http://www.native-instruments.com/forum_us/showthread.php?t=61654

Lab1: The EventWatcher

1. When you open the ensemble you'll see that the event watcher is wired to several panel controls and a constant.
2. Click the Reset button at the top of the EventWatcher (EW). Typically this will always be the first thing you'll do after wiring up the EW. The Reset button clears the "memory" of the EW and clears all of its counters, so that you don't need to wade through old events to find the ones you're now studying.
3. Try turning the knob, you see the watcher quickly fills with the events being sent from the knob as you move it. The "Event#" column counts the events as they arrive, the "Source" will always be "1" (since the knob is wired to port1 on the EW), and the "Value" will be the value coming from the knob.
4. Reset the EW again, to start with a clean slate.
5. Try the Toggle and Trigger buttons. Note that the trigger button will send events to ports 3 and 4. The value going to Port4 is the current value of the knob times itself. Note that multiple values will go to port 4, this is because there 2 wires coming out of the Event Value. An events will be sent down each wire, each with the same value. The [x] in this case must process twice, once for each event. The temporary value (when the first value comes in, the old value for the other input port on the [x] will be used) is sent as an event followed by the final value. This happens all the time in reaktor, but the events generated as part of an intermediate solution to the calculation are ignored by downstream structures. More on this (and on when you should try to avoid it) in Chapter 3.
6. Reset the EW again

7. Try pressing the Switch. This causes a Global Reset (see Chapter 2), and causes all event sources to fire events. Note especially that the value of the constant has arrived on port #5. Note also that the trigger has sent a value of 0, even though it doesn't send this value if you click it!

**I don't get it!?**

At this point we've only just begun our exploration of event processing, but I want to demonstrate some of the quirks and tricks that the EventWatcher reveals. That being the case, you may find some of the items in this lab a little confusing. Don't give up. Just go through the lab now as an exercise, and come back to it again after you've read Chapters 2 through 5.

Self Study Items

1. Try deleting and re-adding a wire to the structure, note the way the Global Reset happens.
2. Try connecting an EventTimer module between the knob and port 6 of the event watcher, then turn the knob to see the timing of the events coming from the knob.
3. Add an Iteration module to the structure. Connected as follows:
TriggerButton -> Iteration.In
Constant[1] -> Iteration.Inc
Constant[5] -> Iteration.N
Iteration.Out -> EventWatcher.In7

Reset the EW, and click the TriggerButton. See how the iteration fires multiple events to Port7 of the EW. Do the events at Port7 come before or after the events at Port4? To find out why see Chapter 3.
4. Connect the Iteration output to the input on the EventTimer from #2, and fire the Trigger button, note that there is no measurable time between the events!
5. Explore the structure of the EventWatcher. You might even try using a second EW on the structure of the first EW to see how it works!

1.3.2 The Oscilloscope

While the event watcher is useful for examining events that all happen in the same tick of sample clock (where time cannot be measured), the oscilloscope is useful for examining audio and event timing.

Lab2: The Oscilloscope

2. Event Sources

...and thus it was written that Events shall be created at event sources, and from thence shall they floweth downstream through paths and modules causing a great many wondrous changes in their wake, and thus it was...

Events do not appear out of the blue, they only come from **Event Sources**. Some modules are Event Sources, and some aren't. Understanding where, when, and why events are created is the first step to understanding the flow of events through your structure. Before continuing, you might want to brush up on the manual sections: 17.3 and 17.7 **however** I find some of the information in section 17.7 to be misleading and/or incomplete, so keep that in mind when you read it!

2.1 Defining "Event Source"

Any module that actually creates new events – those at the start of a chain of event processing - are Event Sources. Modules which simply pass along incoming events are not sources.

Which modules are Event Sources? How can you spot one when you see it?



Event Source Rule of Thumb

Any module with event output ports and no event input ports is an Event Source.

A quick look through the list of modules shows us that this rule defines the following modules as Event Sources:

- Panel controls (knobs, buttons, lists, mouse areas)
- Midi input modules
- System modules like TempoInfo, and SystemInfo
- AtoE modules, EventSmoother, and Slow Random; they have audio inputs but no event inputs
- The Constant module; has an event output with no input, but it's unique in that it only sends an event during a **Global Event Reset** (see below).
- The IC Receive; Acts like an internal MIDI Input module but it also resends it's last value on a global reset, MIDI Input modules do not do this

Exceptions to the rule

- The Receive module. It outputs events and has no input ports, but it can basically be considered to be exactly the same as a wire, so it simply passes events along and does not source them.
- Event sources that are simply ports and not modules. The Len and Pos outputs of the Sampler modules, for example; or the DX and DY ports of the Audio and Event Tables. These ports will fire events on their own without us having to send an event to one of their input ports.
- The LFO sends a constant stream of events
- Snap Value and Snap Value Array can both pass events and create new ones

- All Event Core Cells! (see below)



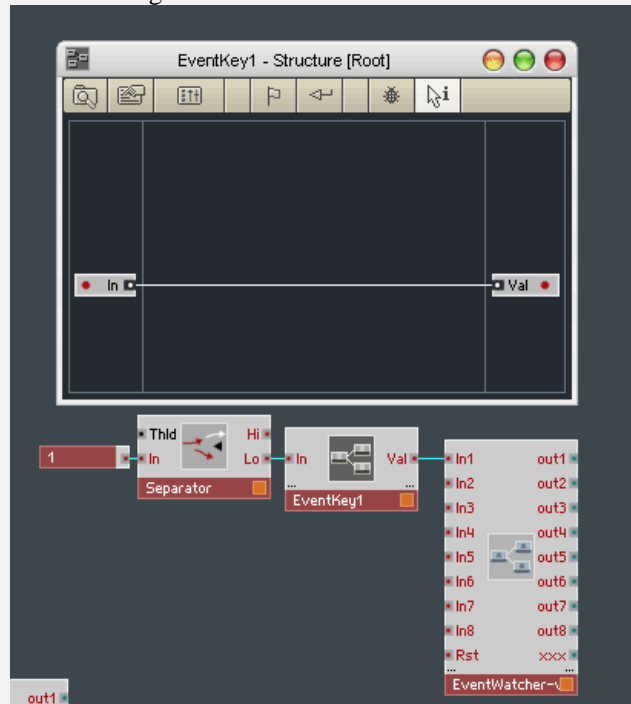
Core Cells Don't Play Nice!

Any Event core cell will treat disconnected inputs, or connected inputs that do not receive an event on initialization (due to the routing) as if they are receiving an event with a value of "0" when a Global Reset happens. This means they will internally process and will most likely fire an output event without ever receiving a "real" event at their input port.

This means all Event Core Cells are Event Sources!

Since there is no way to set a default for disconnected input ports, there's no way to avoid this. It's really a shame. In my opinion this is a big bug in the way core cells work. Its non-standard behavior and I hope NI fixes it as soon as possible!

The following structure demonstrates this:



A global reset will cause an event with a value "1" to come out of the constant, this will not get passed through the Separator, and so no event will reach the core cell. The core cell, however, will output an event with a value of "0"!

If anyone can come up with a trick to prevent this (using *only the structure inside the core cell*) please let me know!

What is not an event source?

All of the modules not mentioned above! Any module that only outputs events when they receive an event at their input is not an event source. This includes all of the math modules, all of the Event modules, etc. If you leave both of the inputs to an Add module [+] disconnected, and you connect the output to the EventWatcher, nothing you do will cause an event to come out of the module. You might see a value of zero if you hover on the wire, but this is the default value for all wires and is not the value that was set by an event.

2.2 When do Event Sources Create Events?

Event Initialization and the Global Reset

There are various times while Reaktor is running that it needs to do a “Global System Reset”ⁱ. The reasoning for this is simple: you need to have some initial values when you start up and when the structure changes.

Event Resets Happen When:

- Ensemble is opened
- Sample or control rate is changed
- Structure is edited
- Pressing any *switch* in any instrumentⁱⁱ
- Snapshot changes that cause a change in the state of a switch (I mention this only because you need to be clear that snapshot changes by themselves don’t cause a global reset)
- Audio stopped and started by clicking the power button in the tool bar
- Instrument/ensemble panel mute buttons unmuted
- Instrument/ensemble panel number of voices changed



Strange Bugs in My Ensemble!

By far the most common bugs people encounter in both user-built ensembles and the NI-built “factory” ensembles are those related to global resets. Very often they don’t even seem to be related to global resets because they may only be noticed when, for example, a snapshot change causes a switch to change its value. In this case, the end user may see it as a snapshot-change bug, when in fact it’s a global reset bug. Avoiding and/or fixing these kinds of bugs can only be done with a firm understanding of event processing and the use of the debugging tools like the EventWatcher.



Initialization Algorithm Document

NI has included a document called “Initialization Algorithm.rtf” with Reaktor. You won’t find a link to it, you must dig into the Reaktor install directory. You’ll find it buried down in the “Documentation / Technical detail information” folder.

It lists all of the Event Sources and the order their ports fire during a Global Reset.

It may be a little confusing at first, but once you become a Reaktor Event Processing Expert it’ll make a lot of sense to you. One of the most important things it describes is the late firing of events from outputs 2 & 3 of the EventOrder, see my description of that module in [chapter reference needed]

ⁱ The term “Global System Reset” or “GSR” was coined by the members of the Reaktor forum and does not appear in any official documentation.

ⁱⁱ Please note the very important difference between a *Switch*, a *Button*, and a *List*. The *Switch* module actually changes the structure of the ensemble by connecting one of its inputs to its output, and setting all of the other inputs to be disconnected, as if you had deleted the wires. It is this changing of the structure that causes the global reset. A *Button* and a *List* each simply output values.

GUI and MIDI Input

GUI inputs include everything from the “Panel” list of modules. These things will (obviously) fire events when they are manipulated by the mouse or when controlled by MIDI. Note that no matter how fast you turn a knob the rate at which they will send events out is based on the System Display Clock which typically runs at about 25Hz. MIDI control of a knob can make it send events at a faster rate.

Panel controls will always send values on global resets.

MIDI Input modules send events when they receive MIDI input. The rate at which they will send events is essentially unlimited as internally connected MIDI connections can send multiple events in one tick of the sample clock.

MIDI Input modules do NOT send events on global resets.

The Control Rate Clock and “Steaming” Sources

These are things that generate events at a constant rate. Streaming Event Sources (as far as I know) will **not** generate an event on a Global Reset, they will only fire events at their pre-defined times based on their clocks.

The following modules fire events at the Control Rate:

- LFO
- Slow Random
- AtoE
- The CR output on the SystemValue module (which spits out the control rate number **at** the control rate)
- EventSmootherers are sort of an exception - they pass all incoming events to the output and also fire events at the control while smoothing, once the output value is = to the input value, they stop streaming out events.

2.3 Non-Source Event Modules

So, what about all of the other event modules – the ones that are not “Event Sources”? All other event modules only output events when they receive an event at an input port. If they do not receive an incoming event on initialization, the *value* posted at all ports not receiving an event will be zero, but they will not actually process, and they will not send an event from their output ports (see the caveat about Core event cells). In fact, if you use the mouse to hover a wire on an output of an event module, you will always see a value there, so you may think it’s constantly “streaming” events, but this is not the case. This is just the value of the last event, the “number left behind when the ball breaks”. The value is always there, even when there may not be any events actually flowing from these output ports.

3. Event Life and Death

...and to each Event shall have Value that it carries like a chameleon's skin carries a color. The color of the chameleon may change during it's life, but it's body stays the same. When the chameleon dies, we can still see the skin but it will never change color again. Thus is the Event's does not exist without a value, but the value may exist after the event is gone...

3.1 Events as Values

As mentioned, all events have a value. The value is often all people think of when they think of an event, but it's important to remember that the event is triggering actions while it exists, but after it dies, only the value is left behind. The value is what we see when we hover the mouse over an event signal wire, and these numbers are only left behind by events that are now dead. So, for example, a knob might send an event to one of the two inputs of an Add module, and the value of that event stays on that input and is used by the add to compute new output values until a new event arrives with a different value.

Similarly, Audio ports only care about the value of the last event that they receive in a given tick of the sample clock. Audio ports are only concerned with the value attribute of the event.

3.2 Events as Triggers

In addition to the value associated with an event, when an event arrives at an Event port (as opposed to an Audio port), the event triggers something to happen within the module.

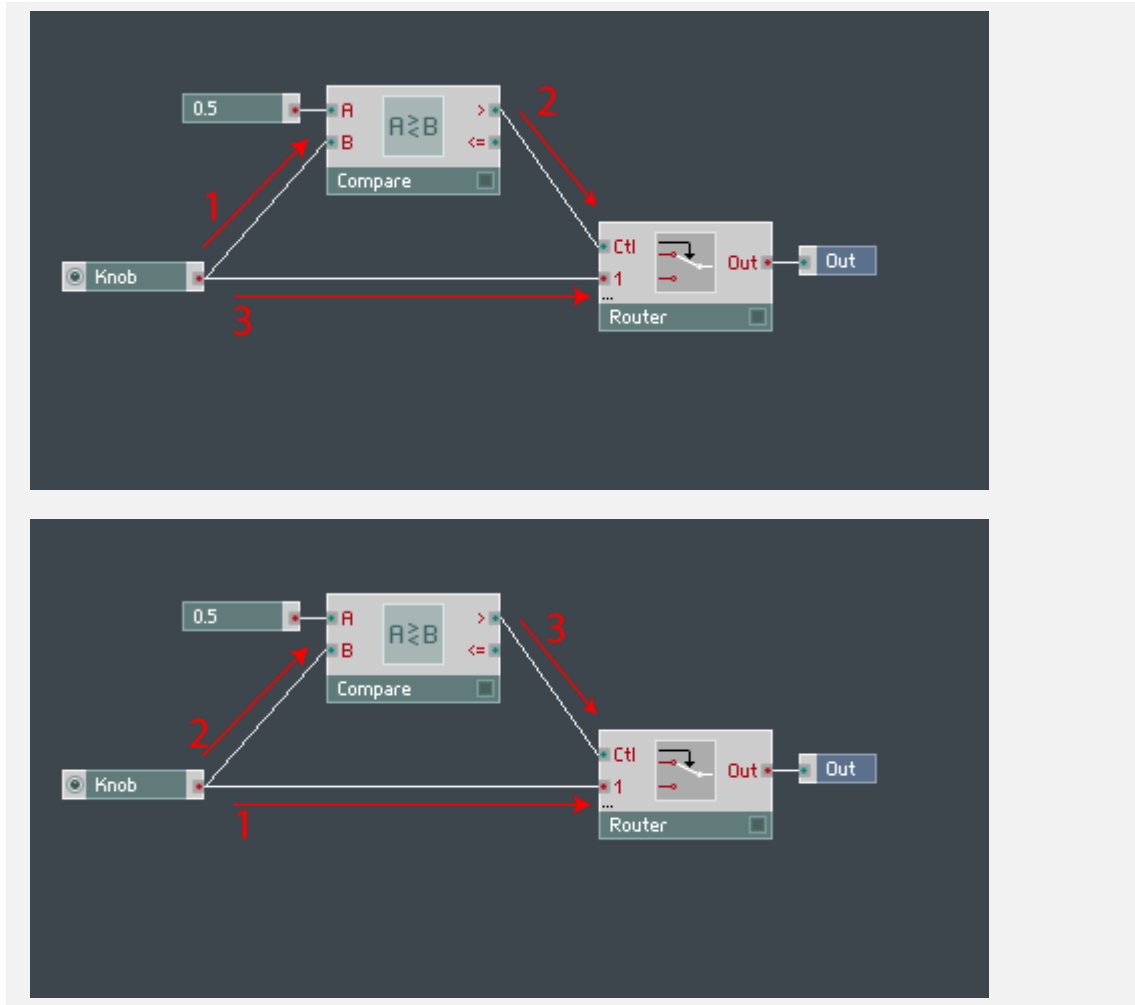
Most Event ports, like the inputs to event-based math modules, care about both the value of the event, and the fact that they are being triggered by the event. When an Add module with two inputs receives an event on either port, it “wakes up” to run it's calculation. The calculation takes the value of the arriving event, adds it to the saved value from the last event that arrived at the other port, stores the value in the arriving event, and passes the arriving event to the output.

Some other event ports, like the Trig input on EventValue or the R input on EventTable, do not care about the value of the event! They only need any event to arrive at their port for them to “wake up”, look at the value on their other ports and send out an appropriate value.

Triggers like this are very often used when we only care about *when* something should happen. The MIDI 96Clock module is a good example of this. It only sends out events with a value of “1”, but it sends them when a 1/96th note on the MIDI clock should occur, and we can use these events to trigger things to happen at a particular time.

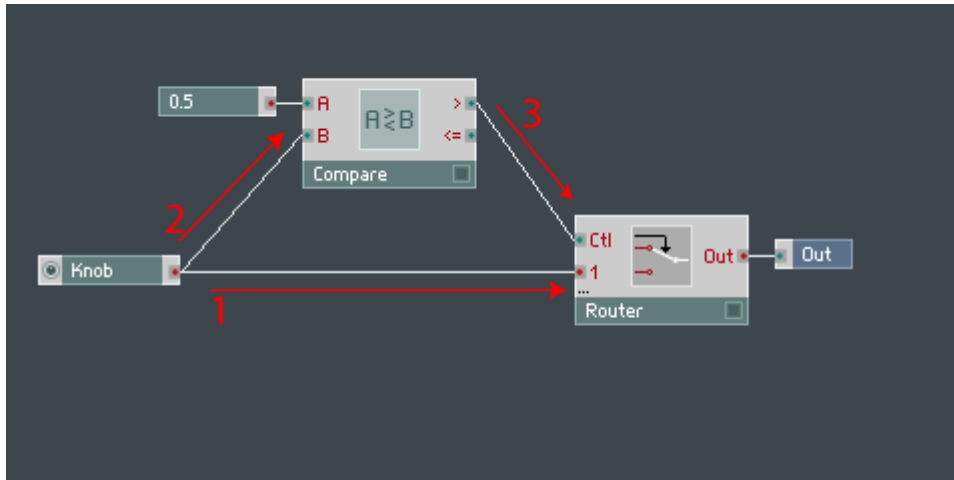
3.3. Event Flow and Ordering

Example: The Importance of Event Ordering



“Breadth before Depth” Processing

The picture below would be an example of “Breadth before Depth” processing. This is *NOT* the way reaktor works, but I wanted to show it just for comparison. In this method, anytime an event reached each branch it would follow each wire only to the next module before backing up and going to the next module. If you consider the “downstream” modules – those further to the right - to be “deeper” in the structure, perhaps the idea of “breadth” vs. “depth” priority will be more understandable.



3.4 Death of an Event

Events die in one of several ways

- They reach an audio input port
- They reach an event input port on a module with no event output ports (eg, Oscillators, MIDI Out, etc)
- They are routed to output port of a module that is not connected to anything
- They reach an event input port on a module that does not cause the module to output an event

Since every event dies, you can assume that these are pretty broad categories, and indeed they are! The first two are pretty straightforward, and should require no explanation. A common example of the third case would be an event separator or router where one of the outputs is disconnected. Some examples of the fourth case would be:

- Modules where the output is generated as a stream by the Control Rate Clock, like the LFO
- Step Filter when the event's value matches the previous event's value
- Clock Divider is an interesting case because it blocks some events and passes others
- If the event has a value ≤ 0 and it reaches a port that only processes when it receives events with a value > 0 (Up & Dn inputs on the Counter module, e.g.)
- The In, Xo, XR, YO, and XR inputs on the EventTable
- ...etc...

4. A Brief History of (Event) Time

*“Time is a basic component of the measuring system used to sequence events, to compare the durations of events and the intervals between them...
...defining time in a non-controversial manner applicable to all fields of study has consistently eluded the greatest scholars.”*
- Wikipedia.org

Understanding Event Timing

Imagine you are studying a running automobile engine remotely through a video camera. You can stop the video and study individual frames, but you can only see the state of the engine when the camera takes each picture for each frame. You know the engine is doing things between the frames, but you cannot see that activity. Let's say you want to know when a sparkplug is firing - but you can't tell because you don't see it happen in your video. From your perspective, the engine only really exists in those states that you see in the frames of video. Now imagine you have a stopwatch with a smooth-moving second hand. You might be able to rig some mechanical attachment to the engine that stopped the watch when the sparkplug fires. If you did that, then even if the sparkplug may fired between frames of the camera, you could find out exactly when it fired because your frame after it fired would show the time on the stop watch!

Sound good so far?

OK, now imagine that instead of a stopwatch with a smooth moving, high precision second hand, the only stopwatch you can get is one where the second hand ticks in time with the video frames. Now you're back where you started; you can no longer measure time between the frames. You can see the results of the actions happening between the frames, but because of the limits on the stop watch you have, you cannot possibly know how long they take because you have no way of seeing or measuring the time difference between them. You might be able to rig up something that says; when the sparkplug fires, increment a counter, so that you can find out if it's firing once, five times, or not at all between frames, but, without that high-precision stopwatch, you can't know exactly *when* it fires.

That's essentially the way things are in Reaktor with the sample rate and events; the sample rate is like our frame-locked stop watch and we can't measure time between frames. Your processor is doing thousands of calculations between each tick of the sample clock, and all the modules are being processed in a certain order in that time, but because the sample clock is the lowest level of time resolution for Reaktor's input (midi and GUI) and output (audio and GUI), we cannot measure the time for things that happen between those clock ticks. This means that even though those actions between ticks happen one after another, we often say that they happen “instantly”. We must keep in mind, however that a computer is a sequential-processing engine where no two things ever happen at the same time and that “instantly” for a Reaktor user really means; “in a single tick of the sample clock”. Events can do a LOT in one tick of the sample clock, just like your CPU, and anything that is tied to audio signals and the sample clock will ignore all that the events are doing and will only rely on the value the events leave behind after they finish processing and the audio clock ticks.

From all we have seen in our user experiments, a tick of the sample clock in Reaktor works as follows;

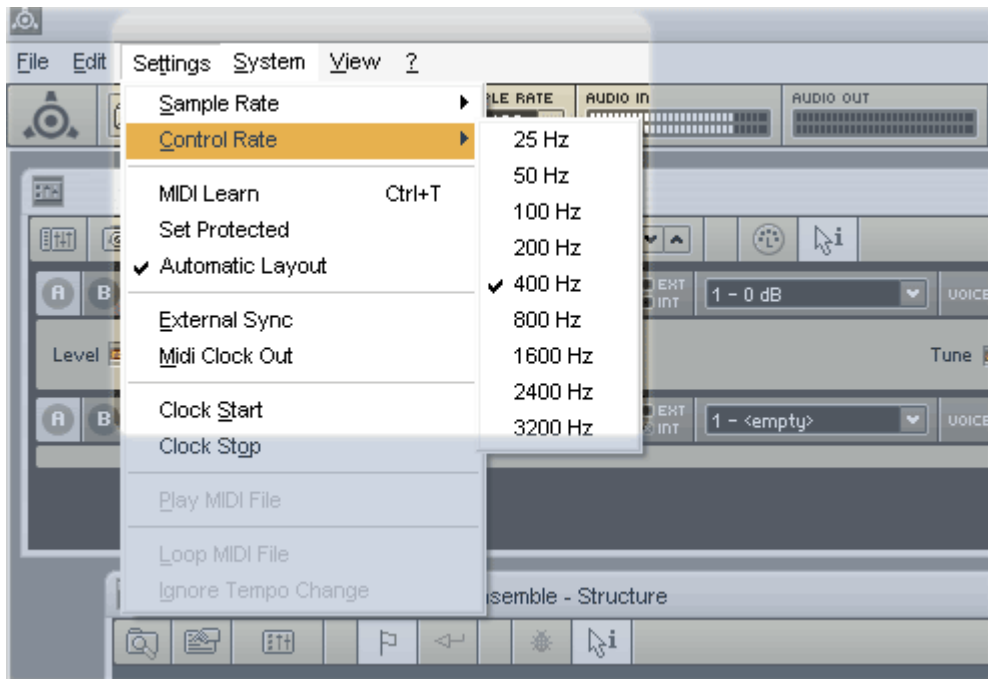
1. Any EventSources that need to fire their events fire them
2. The events are fully processed, propagated, and die
3. All modules with audio outputs process one sample by referencing the current values on any of their event inputs, and a sample is output. Audio modules, which cannot have feedback loops without a delay, are processed from left to right, inputs to outputs, across the structure.

From these rules we can infer several important things about events...

- Events can happen during any tick of the sample clock and have sample-accurate timing (regardless of the Event Control Rate!)
- Event values are all computed and stable for all audio modules in the system at the time those modules process a given sample.
- You cannot process Event -> Audio -> Event in one tick of the sample clock, because the all events are processed before the audio

The Control Rate

So if events can happen at any time, then what the heck is the “Control Rate”?



The Control Rate does nothing as far as when events *can* fire. I'll repeat that because it's one of the most common misunderstandings about reaktor...

The Control Rate has nothing to do with *when* events can fire.

The control rate clock is only used by certain event modules that need to generate events constantly, in a stream. Like the AtoE and the LFO. It also affects the timing of a couple of modules in quirky ways (like the GateHold module), but in general Events fire on sample-clock ticks not only on the control rate clock.

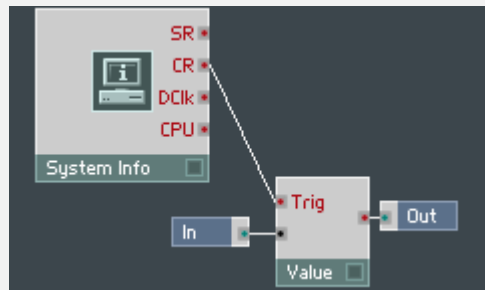
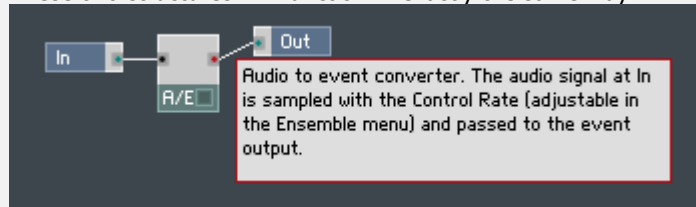
Part of the confusion about this, I think, are the terms that NI uses; "event rate" or "event control rate". I think a more accurate term for "Control Rate" would be something like; "Global Event Clock Rate". There's a global event clock that ticks away at a different (and much lower) rate from the sample rate. It's a very useful thing to have; it means you don't need to create a new oscillator any time you want to get a constant stream of events at a reasonably-high-but-much-lower-than-the-sample-rate rate. Still, those events (and all others) have their timing locked only to the Sample Rate. In fact, if you think about it, a control rate of 400 and a sample rate of 44100 would mean that the first tick of the event clock would happen **between** samples, but that's not the case, even events triggered by the "Global Event Clock" must have their timing quantized to the Sample Rate!

SystemInfo Control Rate Output

We can use the CR output of the SystemInfo module to tell a our structure what the current control rate is, which is useful if we are doing something like making a low-cpu usage oscillator or envelope out of event modules. An important thing to keep in mind when doing this, however, is that the SystemInfo CR port *streams* the values of the control rate *at the control rate!* This is important because if we're doing math on the value those math calculations will get repeated at the control rate, unless we put a StepFilter on the CR port to send the events out only if they change.

Example: SystemInfo CR Port

These two structures will function in exactly the same way!



5. Event Module Details

I thought it would be a good to go into some detail on a few of the most important (and often the most confusing) event modules. I will not cover *every* event module here, as some of them are either not used very often, or are pretty easy to understand from the NI documentation.

5.1 *EventValue*

5.2 *EventOrder*

5.3 *Iteration*

5.4 *EventTable*

5.5 *AtoE*

5.6 *Routers Selectors and Distributors*

5.7 *EventHold*

6. Event Tricks and Common Mistakes

“Trigger” and “gate” type buttons output a zero on global resets