SynthMaker Tutorial:

Triggers

About this tutorial

Synthmaker is an incredbly powerful piece of software, however, the documentation is,err... well, slimmer than a 'supermodel'!

Having been an SM forum regular for many years, I (and many of the other forum 'helpers') realised that SM's triggered data seems to be one of the most confusing aspects of SM – and not even just for beginners!

This series of tutorials should help to make the mysteries of triggers a little clearer.

As SM is one of those things that is best learned 'hands on', I've tried to illustrate the ideas using practical examples.

Table of Contents

What the heck are triggers, anyway?

SM can use many different types of data; numbers, text, arrays, MIDI etc.; but they all fit neatly into one of two categories: Streams and Triggered.

Streams

If you are using SM, then it's pretty likely that you want to make some *sounds*! Inside your PC, sounds are represented by streams – they are called that because they are a constant 'stream' of numbers 'flowing' through the computer at the sampling rate of your soundcard.

But a stream doesn't have to be an audio signal ('sound') – for example you might have a signal within SM that *controls* the sound ('modulation'). Many modulation signals are also streams, because they have to be processed at the sample rate in order for changes to be smooth and to keep things synchronised.

Streams in SM are either mono, mono4 (both shown in blue), or poly (shown in white). When your schematic is running, all of those parts are simply calculated once for every single audio sample. So if there are a lot of them, that can use up a lot of your CPU power – but it has to be that way, otherwise the streams would be interrupted, and your sounds would get very 'glitchy'!

Triggered

How much time do you spend twiddling the controls on your plugin front panels?

OK, quite a lot probably, but compared to a stream changing 44100 times or more every second, it's really only a tiny amount of data that you are creating. Once your synth patch is all set, chances are you won't be moving the controls much at all. Even the notes that you send in from your MIDI tracks is incredibly slow compared with audio data.

So, it would be really wasteful if the maths for working out knob positions, and which notes are playing, were calculated 44100 times every second – wouldn't it be better if they were worked out once, when you moved a knob or played a note, and then just left alone until they changed again?

This is the essence of triggered data – when you interact with your synth's panel, or send it a MIDI note, those parts are 'triggered' to wake them up for a moment; all the maths is done; and then they go back to sleep again – leaving yout CPU free for the heavy duty audio number crunching.

There are many types of data that use the triggering method (see left). For the most part they are shown in green (numbers and text data), red (MIDI) and yellow (Graphics).

Triggers are the individual events that are sent around your schematic giving trggered components a nudge to tell them when to wake up. They don't carry any data, they are just 'events' telling something to happen.

In the centre, on the top row of the screenshot above, you will see the icon for a trigger connection – that is, a connection which carries only 'do it now' messages. However it is very mportant to note, that <u>all</u> of the data types shown contain trigger events along with the numbers, text, MIDI etc. that you would expect. Whenever they are asked to send a piece of data, they also send a trigger along with it, so that the components to which they are connected know that they also have to 'wake up' and do something with the data.

Where do triggers come from?

So how does your schematic know when you want it to trigger something to happen?

GUI Interaction

Obviously, you need your plugin to react when you twiddle with the front panel controls! So the biggest category of trigger sources are the mouse interaction primitives.

If you are new to SM, you may well have not seen these primitives before – but they will be there buried away inside all of the standard knobs and switches.

Whenever you click on a control, or drag a value, these prmitives will create triggers telling your plugin to update the parameter values.

Note that it is not just the trigger type outputs that send triggers – all of the others will too. So, very often, the trigger outputs will not be used, only the outputs that contain data that we're interested in.

For the mouse dragging 'accumulators', a trigger is created each time your mouse moves by at least one pixel – so throwing that filter frequency knob around can generate quite a lot of triggers in a short space of time!

Automation

Many of the user controls on a front panel can be set to read and wrte automation data to a VST host program.

When reading automation moves, triggers are generated just as if you were moving the controls with the mouse.

MIDI

A synth wouldn't be a lot of use if it didn't react to notes being played! - so the MIDI In devices create a trigger for each MIDI event received. Just like automation and mouse twiddling, there is no 'thinning' of the data – if you go mad with your pitch bend wheel, you will get a trigger for every single little movement of the control.

Tickers, Timers and Loops

There are some SM primitives that can create new triggers all of their own – these are used if you have a section of schematc that needs to be refreshed on a regular basis (e.g. graphics animations), or if you need a whole series of things to happen all in one go.

Tickers are just that; the ticking of a clock. They send out new triggers at set intervals. Tick25 sends around 25 triggers per second, and Tick100 at around 100 triggers per second.

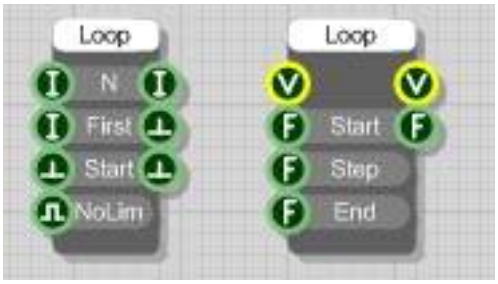
However, note that the time intervals are worked out using a

'Windows Timer' – these have a very low priority, so the timing will not be very accurate, and will vary depending how hard your CPU is working; e.g. a Tick100 will very rarely manage more than about 60 ticks per second in a 'real world' use.

The timer works very much like the tickers, but is a little more versatile. There is an input to control the interval between ticks, and Start and Stop inputs so that you can turn the 'clock' on and off.

Like the tickers, it also uses Windows timers, so it is also not very accurate.

Loops are used when you need to do something many times over, but treat is as if it were a single event. For example, you might want to do some maths on every number within a large array – the integer loop can automatically count through every item in the array in turn. So a single start trigger, generates many new triggers at its Loop Value and Step outputs, one for each step of the loop.



The Draw Loop works in a similar way, but is used when you need to draw many repeated copies of a graphic on screen all at once; for example, to draw a grid of lines over a graph.

Editing and Loading

Triggers are also created when you are within the SM editing workspace, so that you can instantly see the changes that your editing makes. This is not only when you edit 'data entry' type primitives (e.g. floats, integers, strings), but also when you create or move links between primtives and modules.

NB) One little quirk is that connecting links always makes <u>two</u> new triggers.

However, copying and pasting, and loading a schematc file don't generally cause any triggers.

This is done deliberately, as it's often handy for SM files and toolbox modules to have default values already loaded when you open them. In fact, when you save an SM schematic, almost all of the triggered types of data will remember their values – even the contents of arrays, and bitmap graphics! Making a new trigger every time you loaded or coped something might accidentally overwrite the default values, which could even result in a blank front panel!

Sometimes this can be a problem – you might prefer some values to always be reset when loading or pasting. To do this, you can generate triggers automatically using the 'After Load' and 'After Duplicate' primitives,; and then use these to 'wake up' your value-resetting components.

So What's the Catch?

This all seems really cool so far – big chunks of a schematic can use hardly any CPU power because they are 'asleep' until a trigger wakes them up. Too good to be true?

Well, somewhere deep in the bowels of SM, there has to be a system for keeping track of the triggers, and which modules are 'asleep' or 'awake'. It stands to reason, that the more triggers there are, the harder this 'behind-the-scenes' processing will have to work.

The triggers are designed to handle nice slow 'human paced' changes in data – that's when they're really useful – but if the green data starts changing too fast, then keeping track of all those triggers will soon start to eat a lot more CPU than we'd like.

Invisibility!

The little CPU readout in the bottom corner of the SM workspace doesn't measure the amount of CPU used by triggered events, only the usage of the stream components.

To see whether triggered events are eating CPU you will need to use an external CPU meter – the one in Windows Task Manager or Performance Monitor are useful for ths.

As an example, get the Task Manager CPU meter on screen (CTRL+ALT+Delete, then click the 'Performance' tab). Now create an SM schematic with some standard knobs from the toolbox. If you wiggle a knob like carzy, you should see the how the CPU load rises. This is mainly due to the drawing of the animation on screen...

Greedy GUI!

Another downside, is that drawing graphics on your front panel will eat a lot of CPU if you ask it to update the graphics too quickly. Imagine grabbing a control and going really mad with it; that might generate 100s of triggers every second, but it would be really wasteful to update the screen that often – you wouldn't even see many of the animation frames because your monitor doesn't update that fast! The bigger the area of screen that gets redrawn, the worse this effect will be.

Optimising

For these reasons, it is good to be able to optimise your triggered data to have only those triggers that are really needed. That will be the main purpose of the following tutorials – but first we need to understand exactly how triggers move through a schematic...

How Triggers Move

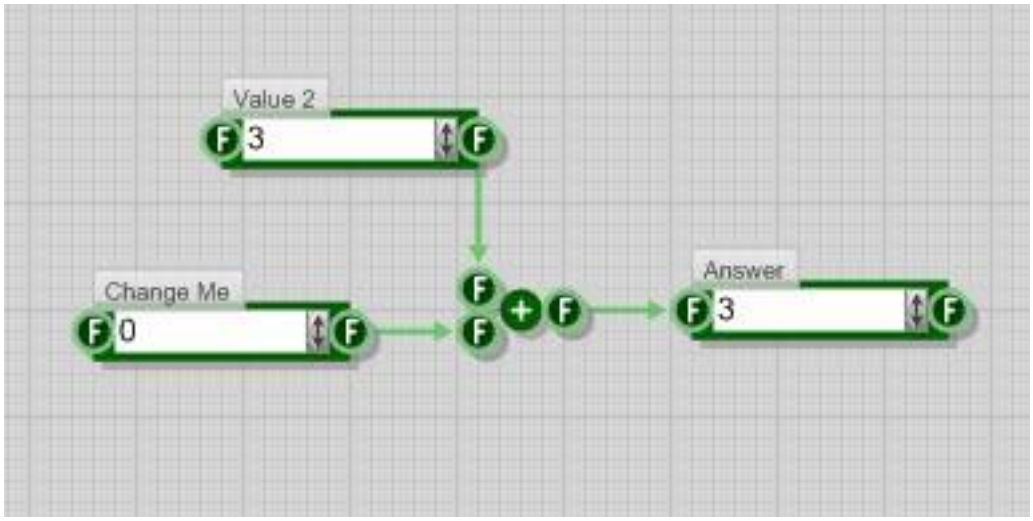
The earlier description of triggers seems simple enough – you move a control, a trigger is created, and gets passed along a chain of modules and primitives telling each one in turn to 'wake up' and do something. When everythings done, they all do back to 'sleep'.

Well, kind of; but in reality it's not quite that simple...

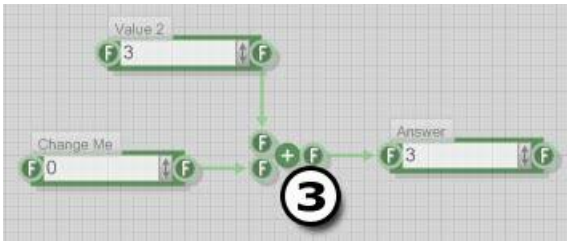
Backwards and forwards

SM actually has two kinds of triggers – the obvious kind, working down the line of modules telling them that there is new data available, and a second, hidden, type.

To get an idea of how this works, let's look a little closer at the example shown in the official SM user guide...

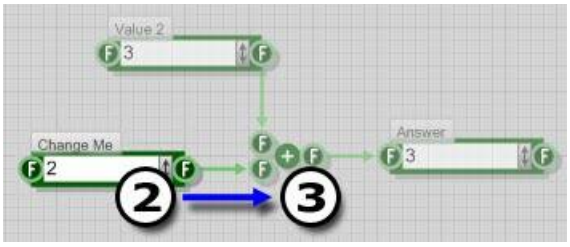


Let's see what happens when we type a new value into the 'Change Me' float primitive. To make things a little clearer, I'll show in the diagrams the current component values, and which modules are 'asleep' (greyed out). So everything starts out like this...



All peaceful, everything 'asleep', so no CPU being used at all.

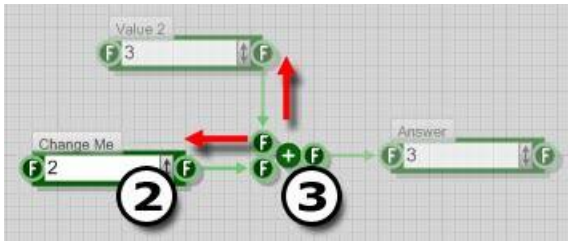
Time to wake it up with some user input...



So lets enter '2' into the 'Change Me' primitive.

Press enter, and the float value changes, and as we'd expect, a trigger is sent out towards the add.

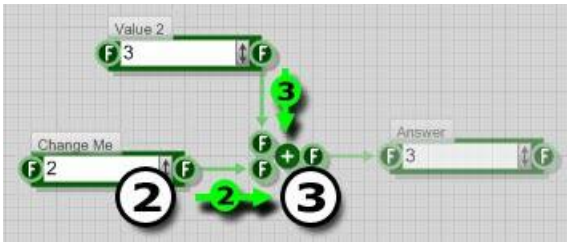
And now the add will calculate 2+3, and pass on the value.... surely?



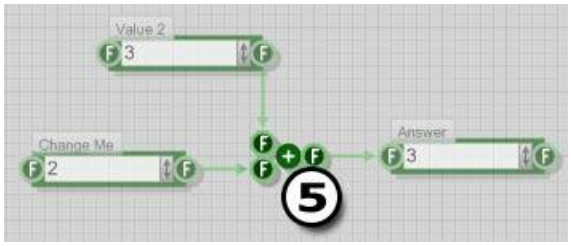
Not quite!

The blue trigger that the edit sent out, was only a 'do something' event, it didn't send the float number to the add at all.

What happens now, is that (just like me in the morning) the add 'wakes up' wondering what the heck is going on. So the add now sends out triggers 'backwards' from all of its inputs, asking to be updated with all the latest news.

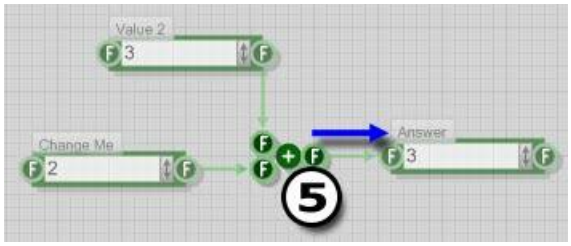


The Float primitives respond to the 'backwards' triggers by sending their values to the Add primtive.

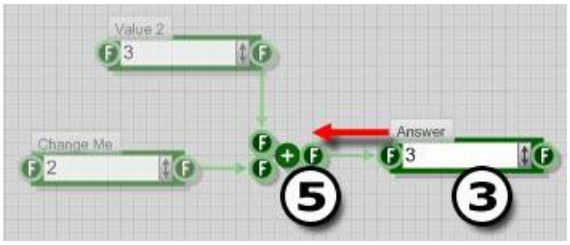


Finally, the Add primitive can do its maths.

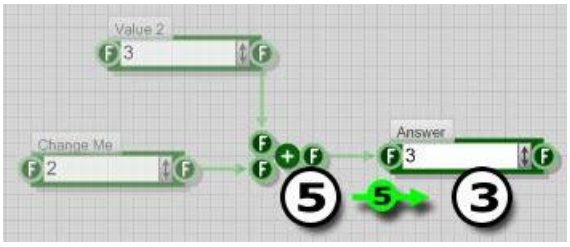
The two float inputs can now go back to sleep; their work is done



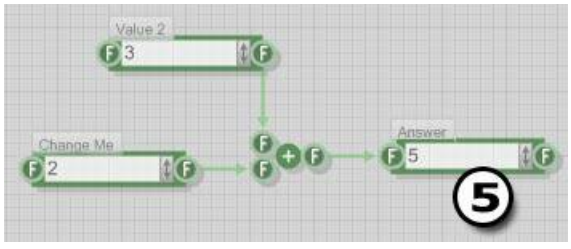
With the sum calculated, the Add primitive now sends a trigger (not the number value!) onwards to the float box at the output...



...which rubs the sleep out of its eyes, and sends a 'backwards' trigger to the Add, asking what it's supposed to be doing.



The Add then passes along the result of the calculation.



And finally, the result arrrives at its final destination.

All the prmitives now go back to sleep to await the next trigger.

Now, this was just about the simplest example possible – but look how many blue, red and green arrows there are on the diagrams. Our single change in value needed eight different pieces of information to be tracked before the answer finally popped out.

And this is really a huge simplification – each trigger also contains 'tagging' information that is used to detect when there are multiple branches, and to prevent feedback that could freeze the plugin in an infinite loop.

This is why it can pay to minimise the number of triggers in a schematic – imagine the scheme above if there were a longer chain of primitives. Imagine if instead of an add, we used something that had a dozen different inputs, all sending a reverse trigger when it got woken up!

The most important thing to remember from this, is that your triggers rarely take a straight line path from input to output, if any component used has more than one input, reverse triggers will be sent backwards along many paths – and the number gets big very fast if there are lots of 'branches' all ultimately feeding into the same final answer.

In the lessons that follow, I'll be looking at the techniques we can use to get triggers working the way we'd like, without too much CPU wastage.

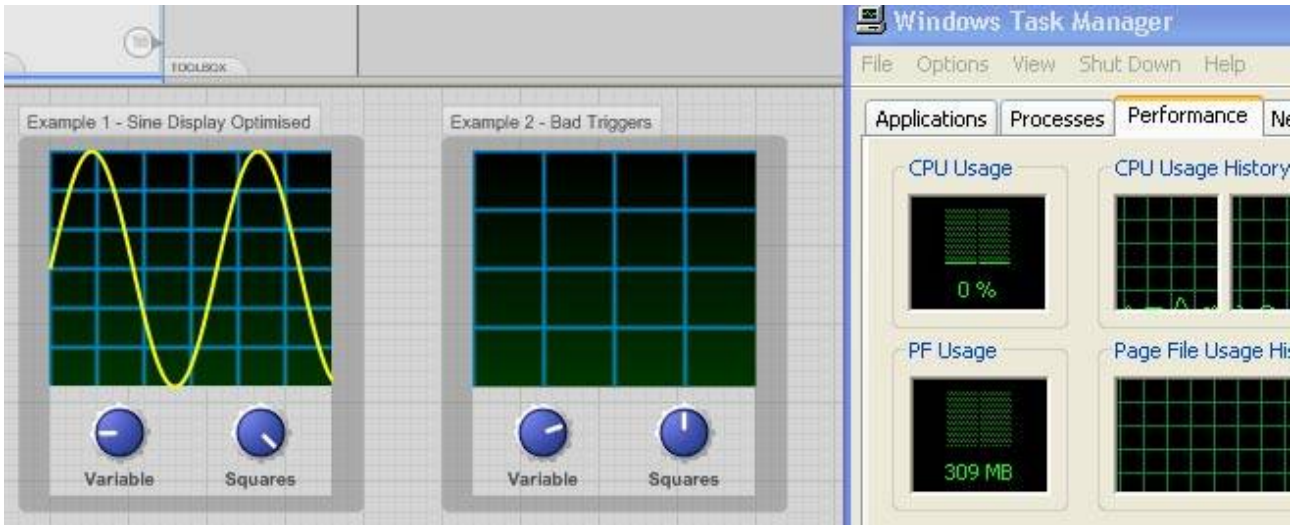
Introducing the Example Schematic

Getting triggers right is too hard to learn if it is all just theory, so at this point I'm going to introduce an example schematic – after a bit of experimenting, I have managed to construct something that has truly awful trigger behavour; in fact, to start with, it doesn't even work!!

We'll take the example and work through all of the steps required to get the bugs fixed, and streamline the triggers to reduce the CPU hit.

So, if you don't already have it, go back to the forum and download the file "**Trigger Tutorial Examples.osm**". It contains several modules, each of which shows the example schematic at a different stage in its development. There's also a 'bonus' module containing some useful bits and bobs for your toolbox.

So, to start with lets look at the first two modules...



Example 1 is the final optimised module, to give an idea of what we're aiming for. Example 2 is the initial flawed design.

Note that I have Windows Task Manager open, showing the CPU usage. As mentioned before, the SynthMaker built-in meter doesn't show the CPU load for triggers and graphics – so get this on screen before starting (Ctrl-Alt-Delete), it will be very instructive!

Have a play with the optimised example first, to see what it does – not much really, a sine wave display where you can change the frequency of the sine wave and the size of the grid squares. Even if you wiggle the controls like crazy, it should barely show any action on your CPU meter.

Now try the 'Bad Triggers' version – whoa, what's going on here? Even on a super hot-rodded PC, you will probably notice...

- The controls are so slow, they're almost impossible to use.
- Display doesn't show just your chosen wave, but a whole range in between....
- ...and then goes blank!
- And look at that CPU meter – nearly a whole CPU core, just for a little sine wave!?

This needs some serious fixing. First, the bugs. We need that display to show what we want, when we want.- why does the display go blank? - we can see that there is some sine wave data generated, but where does it go?...

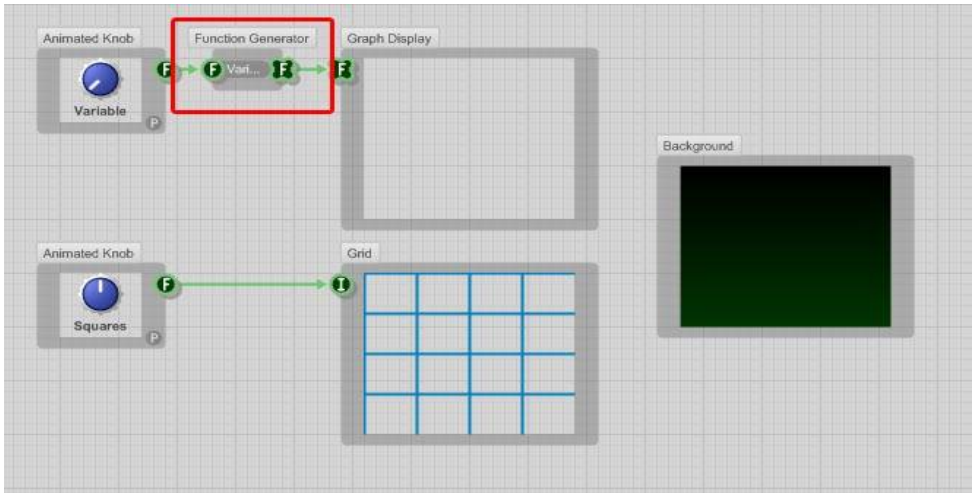
Getting the Order Right

Looking at the 'Bad Triggers' module again, let's examine more closely what it happening.

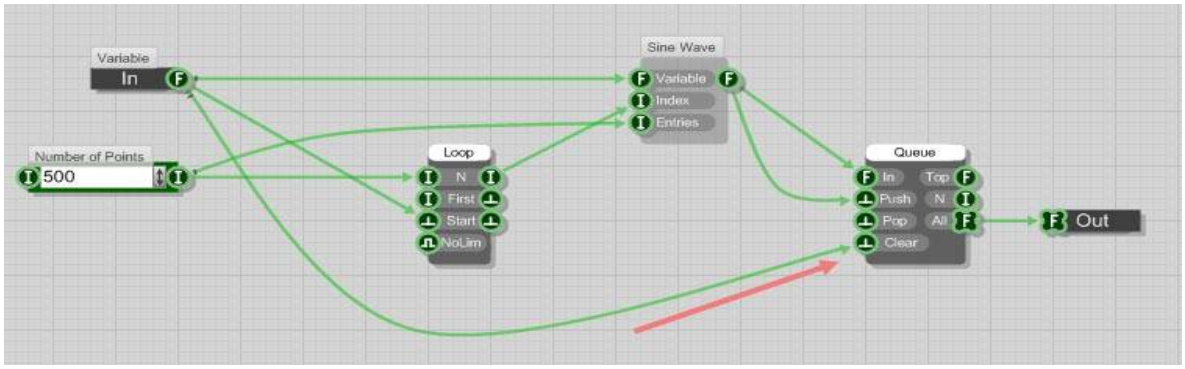
Turn the 'Variable' knob all the way anti-clockwise – the display is then supposed to show a single cycle of a sine wave.

Watch closely – right at the end of the weird, sweeping 'animation', you will see your perfect single sine wave cycle for a split second, just before the display goes blank. From this, we can deduce that the display is probably working OK, it must be the array data that is not right somehow.

So, looking inside the module, the most likely source of the bug is inside the module 'Function Generator'.



Inside 'Function Generator' we find this...



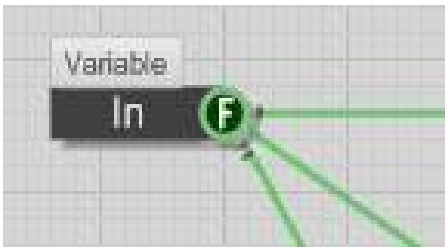
This is supposed to create an array of values that will be plotted by the graph. What should be happening is this...

1) Clear the Queue, so that the array begins empty.
2) Start a loop counting the number of points that we want to plot.
3) For each point, calculate the value.
4) Put the value into the array by 'pushing' it to the end of the queue.
5) Send the data to the display.

Notice step (1), 'Clear the Queue'; the array is supposed to be empty **at the start** - but the display shows that the array is empty at the end! The triggers at the 'Clear' input (arrowed) must be wrong.

So why is the Queue 'Clear' input being triggered at the wrong time. We can see that it comes from the main input to the module – i.e. from the knob. So it is being triggered every time the knob value changes – but so is step (2) 'Start the counter', **and** the value is also sent to the 'Sine wave' module that does the maths.

The order of the steps is very important here, the array must be cleared **before** the loop starts, so we really need some control over when the triggers get sent to each part of the module.



Look closer at the input connector...

...notice the little black 'stripes' or 'ticks' on the links where they leave the connector. They are the key to trigger ordering.

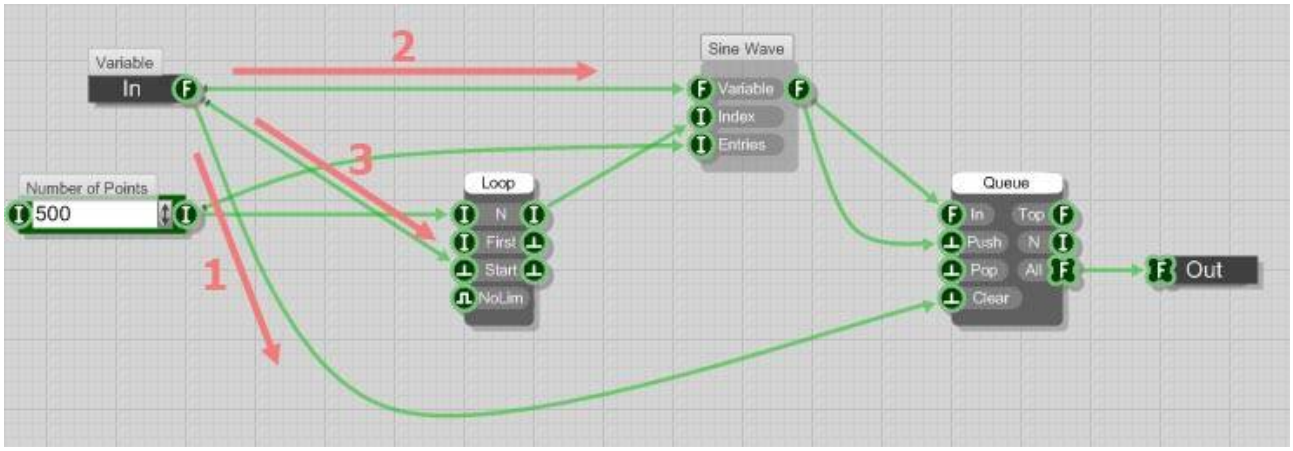
The link with no 'ticks' will always see the trigger first, then the one with one 'tick', then two 'ticks' etc. etc.

And there's something else that is important to understand...

When the first 'no ticks' link gets a trigger, it has total control over the schematic until **everything** connected to that link has finished working – only then will the next link see the trigger.

In this case, if you look at the trigger order 'ticks', you'll see that the loop is started **before** the array is cleared. The loop will do all of its 500 counts, **and** all the maths triggered by the counts, **and** all the array filling, before the 'Clear' link even knows that there has been a trigger at the input.

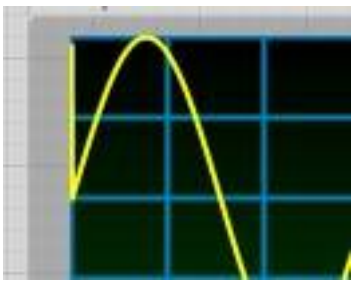
Changing the order is a bit fiddly as the trigger order is the same as the order that you made the links. To change the order, you will have to delete the links and then re-make them in the order you want...



So now, the array gets cleared first, the 'Sine Wave' part has all the info it needs, and finally the loop is triggered to fill the array.

This is much better, the display is still sluggish, and has the weird 'animation' effect, but it got rid of the blank display at the end. This is shown in the module 'Example 3 – BUGFIX Blank Display.'

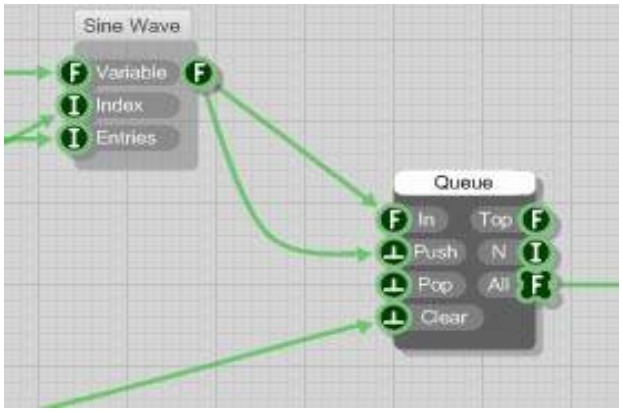
But what is this weirdness...



Every once in a while you might see this on the final display.

Surely, the sine of zero is zero, so what is that vertical line at the start of the graph plot?

Controlling the Flow



So why are we getting that odd first value?

A value is pushed into the queue every time that the float output of the 'Sine Wave' module changes.

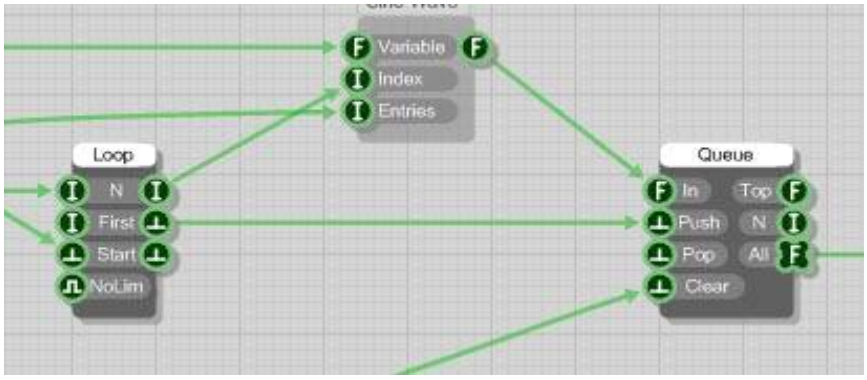
The 'Sine Wave' module will output a trigger whenever its 'Variable' input value changes (i.e. moving the knob), not just when the index changes.

So, we still got the order a little bit wrong - we sent the input to the 'Sine Wave' module after the array got cleared, and it's sending an extra value to the array before the counter starts!

Well now, I actually dd this on purpose so that I could show an alternative way to get this right. Sometimes it can be a real pain having to remake lots of links to change the trigger order when editing.

So some modules have extra trigger outputs to help you to get the right timing...

On the 'Integer Loop' module, you'll see two extra outputs – 'Step' and 'Finished'. The 'Step' output sends a trigger every single time that the index changes – and importantly, it always sends this trigger **after** the index value output has changed. Using this, you can be sure that the maths done by the index output is finished, and that the queue only gets updated by genuine loop counts...

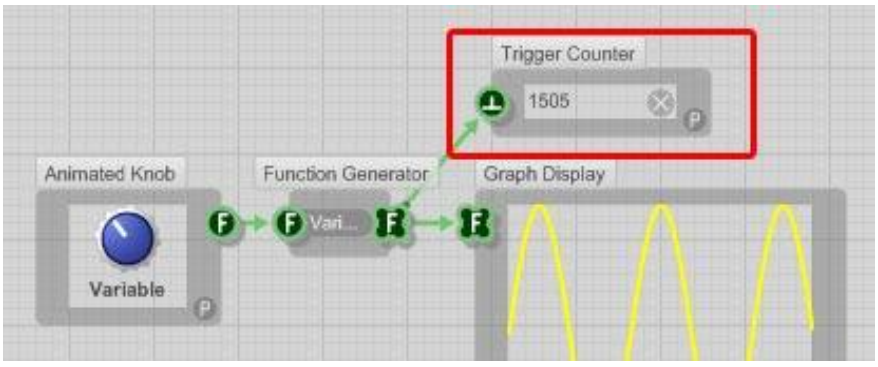


But what about that mysterious 'Finished' output?

Well, that outputs just one trigger, after the whole of the loop has finished counting.

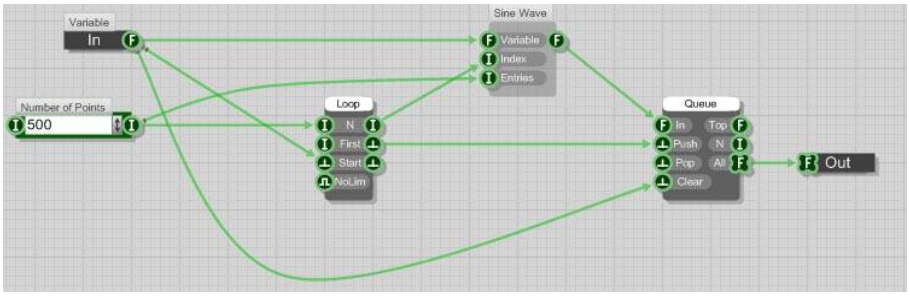
To see why that might be useful, we can use one of SM's most useful little modules...

The Trigger Counter



The trigger counter does just that, it counts the triggers coming into its input. You can clear the count any time by clicking the little X icon.

Hook one up to the output of the Function Generator module and then move the knob. Wow, hundreds of triggers – even if you only move the knob a little bit. We'll need to look again inside that module to see the problem...



What do we know about triggers and value changes?

Yes, every change also sends a trigger - and that includes arrays!

For every single one of the 'Integer Loop' counts, the array has a new value added to it, and will send a trigger from the output of the module. That's 500 triggers for every single change made by the knob – and every single one is making the whole display get re-drawn. No wonder our schematic is so slow!

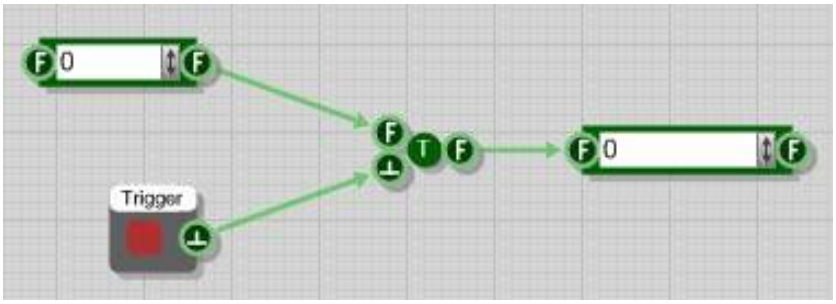
This also explains the 'animation' effect – we are seeing every stage of the array 'filling up' on screen instead of just the final values.

This is why we need that 'Finished' output from the loop module – if only there were a way to 'freeze' the module output every time the loop finished...

Sample and Hold

And of course, there is just such a thing – the sample and hold module. Module**s**, in fact, as there is a sample and hold module for almost every type of triggered data.

Here's how they work; I'll just use a simple float example...



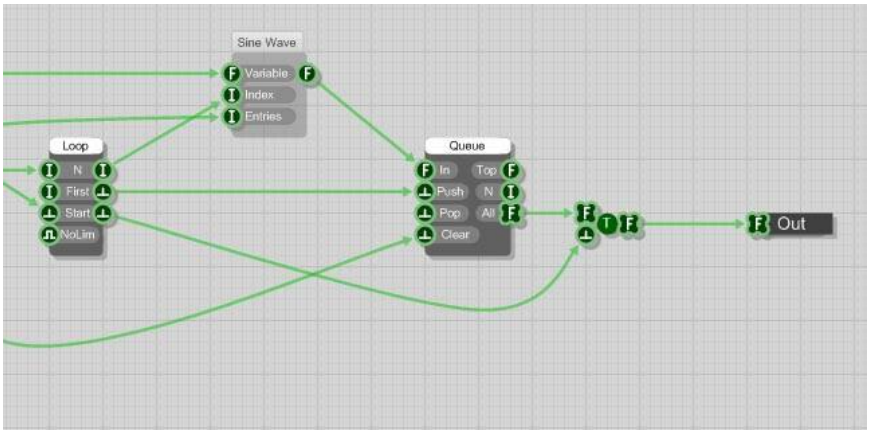
If you play around with this, you'll find that you can change the float input value as much as you like, and nothing will happen to the output.

The output **only** changes when you send a trigger to the other input.

This is one of the most useful modules for trigger control and optimising. Lets use one to control exactly when our array output gets updated...

Sampling a value to reduce forward triggers.

Here's what we do to our 'Sine Wave' module to serously reduce those triggers. Simply connect a float array sample and hold at the output, and trigger it from the 'Loop Finished' trigger.



The final output now only gets changed once, after the loop is complete, and the array is completely filled.

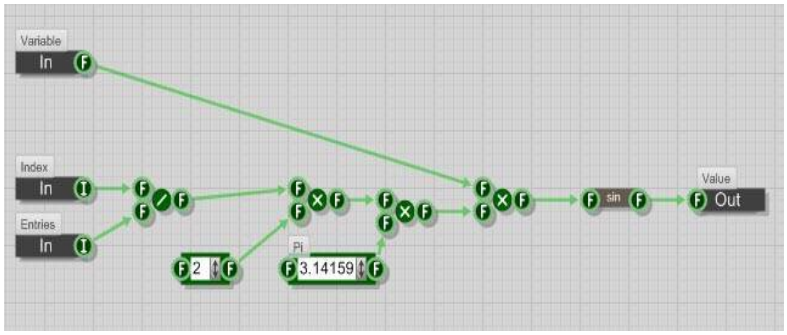
The improvements so far are in the 'Example 4' module.

It certainly looks a lot better, but do a quick check of the CPU load – there's a long way to go yet!

Sampling a value to reduce backwards triggers.

Before we're finished optimising the 'Function Generator' module, there are some other little tweaks that we can use to lower the CPU load further...

We've seen that a sample and hold primitive only updates when it receives a trigger at its trigger input. This means that sample and holds will not pass 'backwards' triggers to upstream components.



To see why this is useful, go inside the 'Sine Wave' part of the 'Function Generator' module...

You can see that for every count of the loop, there is a divide, three multiplys and a sine function to calculate.

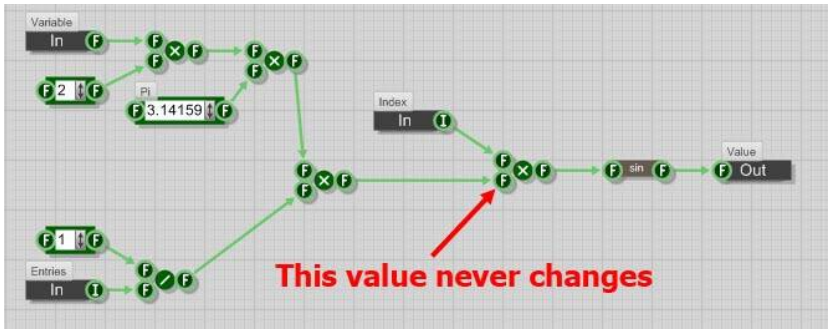
Looking at the maths more closely...

Value = sin (Index/Entries * 2 * Pi * Variable)

But '**2 * Pi * Variable**' is going to have the same value for every single count of the loop. It's also true of most modern computers that dividing uses much more CPU power than multiplying. So let's re-write that equation a little bit...

Value = sin (Index * 1/Entries * 2 * Pi * Variable).

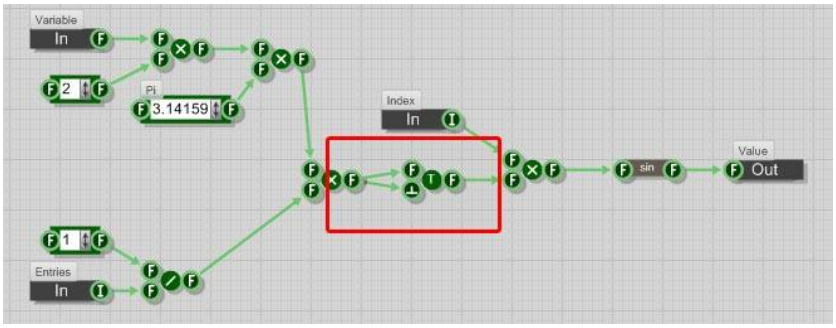
The part in red has the same value every time – it only changes when we alter the input values manually - so lets re-arrange our schematic a little bit to reflect this...



But what's the point of this?

We know from previous lessons that the final multiply will send out backwards triggers, telling all those other bits of maths to do their calculations over again every time! - and we added an extra multiply, so this surely can't be better?

Quite true – until we use a little sample and hold trick...



Notice how the sample and hold has both inputs connected to the same place.

So, if you move the knob or change the number of entries, the value will still get sampled and sent on to the following components.

However, when the loop index changes, things behave slightly differently to before...

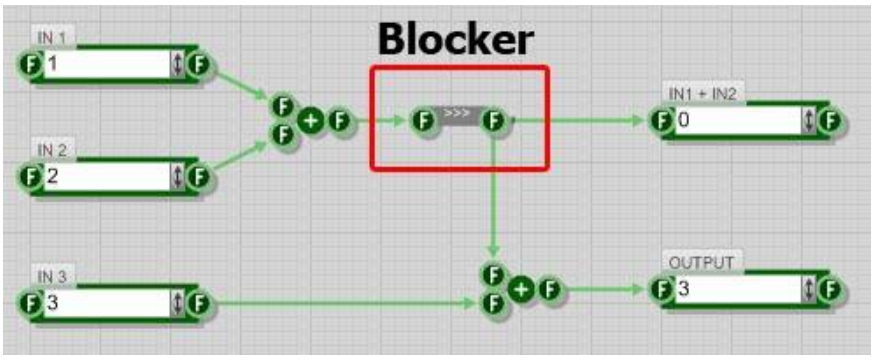
The multiply connected to the index value will still send out a reverse trigger to ask for the results of the other maths – but that trigger only gets as far as the sample and hold. It will read the value there, but it **won't** make all of the other maths get calculated again.

This saves SM from doing all that maths 500 times over just to calculate the same result, and also means a lot less triggers and value messages for SM to keep track of.

The Trigger Blocker

OK, so we've seen how blocking 'backwards' triggers can be useful – what about the regular forward triggers? Can we block those? Would we want to?

The component for this is called... ...a trigger blocker! And here's what one looks like...



Notice something odd about this picture?

Since when did 1 + 2 + 3 come to 3 ?

Or 1 + 2 = 0 ?

The module 'Example 5 - Trigger Blocker' has this little design in it for you to play with.

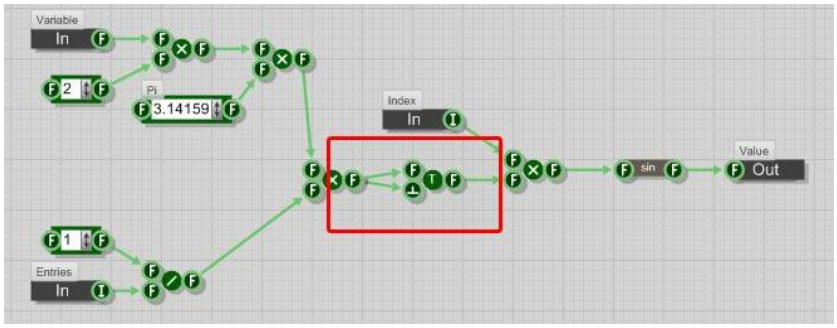
Changing IN1 or IN2 just has no effect on the outputs, the blocker stops them from sending triggers further into the schematic.

But try changing IN3, and miraculously, the correct answer appears at the output. That's because that final add primitive will send out a 'backwards' trigger asking for the result of the IN1 and IN2 maths, so that the final answer comes out right.

So, the trigger blocker is opposite to a sample and hold – it only blocks forwards triggers but does let backwards triggers through.

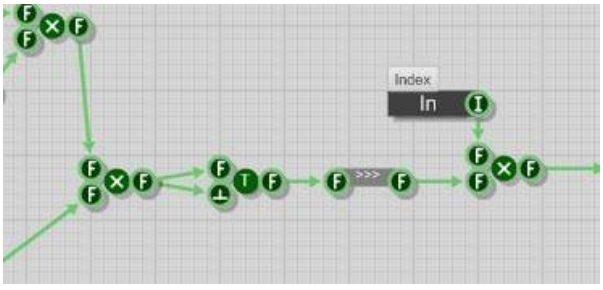
Notice that the IN1 + IN2 result doesn't change. That's because it wasn't the 'IN1 + IN2' float that requested the answer – the true value only gets sent to primitives that ask nicely with a proper backwards trigger – and no new forwards triggers are ever created. If you click on the input connector of the 'IN1 + IN2' float, it will then send its own request, and then gets updated properly.

So why might this be useful? Let's look again at our sine wave array maker...



We've seen that the sample and hold will still send values through in a forward direction when we connect it like this.

But really, we're not at all interested in the number there until the loop starts counting – until the loop starts, it will just end up sending extra values out to the array that we don't really need. So we can make this work even better by putting in a blocker as well...



So now our sampled value doesn't get sent further on when it isn't needed – but the schematic can still ask for it whenever it is requred by a change in the Index value.

This technique can be used whenever two parts of a schematic are designed to run at differemt speeds or at different times. The sample and hold stops a value from being re-calculated because of backwards triggers, and the blocker forces the value to be ignored until it is asked for.

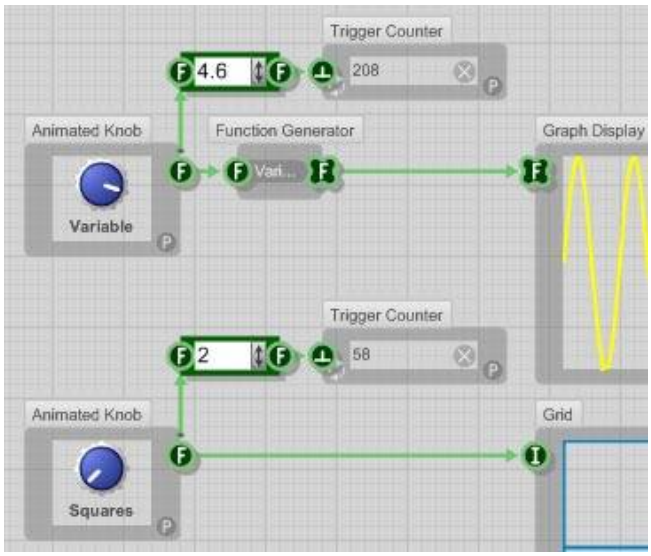
It's All Just Too Much!

OK, so the schematic is now working properly; all of the data for the display is correct, and we see all the right things on screen.

But, open the Task Manager CPU meter again, and have a play with the new working version (it's module 'Example 5 – All Working but Greedy' in the examples schematic).

There's still a huge CPU load when you go mad on the controls, much more still than the optimised 'Example 1' version.

To find out why, attach float boxes and Trigger Counters to the output of the knobs...



You can easily generate dozens of triggers every second by moving the controls quickly. Every single one of those triggers will re-fill the array and cause the screen to be re-drawn.

But how fast can you see changes on screen? As few as 15 frames per second is used for many animated cartoons – so re-drawing the display 100s of times a second will be wasting a lot of CPU power.

If you look more closely, there are also a couple of other really odd things going on...

Try pushing the 'Variable' knob up as far as it will go, then keep on moving the mouse upwards – or move it downwards past the minimum value.

Now try moving the 'Squares' knob very slowly – it is set to have click stops at whole numbers, so you have to move it quite a bit for it to 'click' onto the next step.

You should have spotted that the knobs are sending out triggers even if the numbers being sent out aren't changing!

This is because the triggers are created by the movements of the mouse – the stock knobs that you got when you installed SM are made to be ultra-reliable; they have to always do **something** when the mouse moves so that beginners will not accidentally create 'broken' schematics. But this means that they are not very optimised.

Now, we could start editing the knobs to make them work better – but they are a bit too complicated inside to cover in a few tutorial pages. So instead, let's see if there is any way to 'thin out' all of that surplus data that they are sending out...

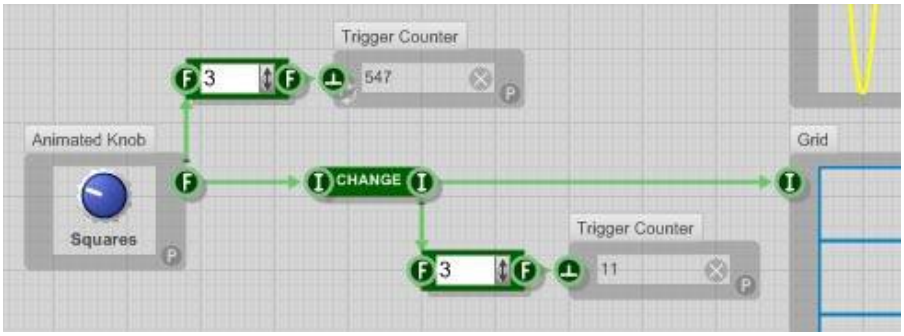
Only When Changed

The first thing to do is to make sure that we only get triggers when the knob values have really changed. Recent versions of SM have modules in the toolbox for doing just this – though if you are using an older version (before SM 2.0) you may not have them.

Try searching the toolbox for 'changed' – if you don't have them, no problem, I have included them in the 'Toolbox' module of the examples schematic; just drag them into the appropriate folder of your SM toolbox. You might want to have a look even if you do already have the modules – I have 're-skinned' my version to make it more compact and easier to spot.

So let's see if the module does what we want.

Insert an 'Integer Changed' module at the output of the 'Squares' knob, and add a second readout box and trigger counter...



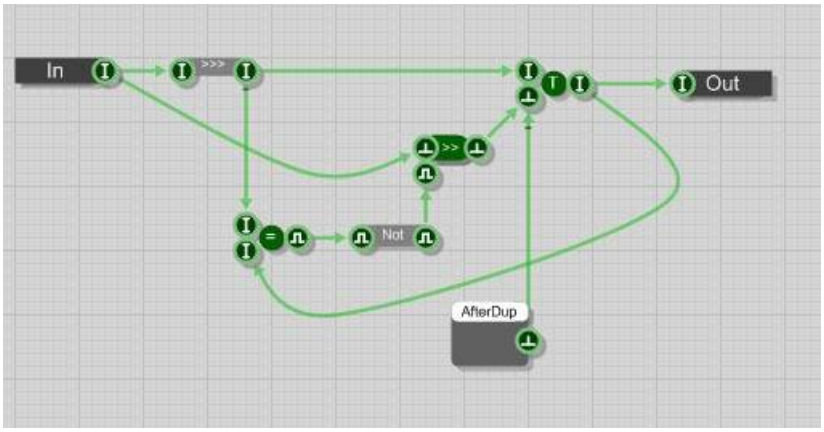
Ahh, that's much better – all of the surplus triggers are gone; there's just a single trigger each time the integer output changes.

When adding a 'changed' module to the other knob, remember that it is supposed to output float numbers; so you'll need to use the float version of the 'changed' module.

NB) When you're done studying the triggers, remember to delete all the extra float boxes and trigger counters. As well as using extra CPU, the counters might appear on your plugin's front panel, which would look a bit messy!

Now, I could just leave it there – we have a module that does the job, great!

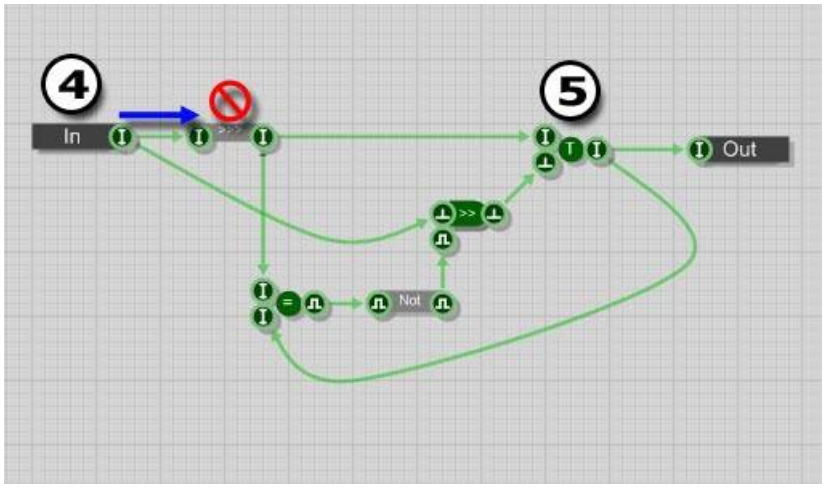
But, the way that the changed modules work is rather clever, and they make a good example for revising the things we've learned so far. So, let's have a peek inside one of them...



Here's the workings of the integer changed module.

The other kinds (float, boolean, string) are pretty much the same – the only difference is that they use the correct version of each primitive to match the data type.

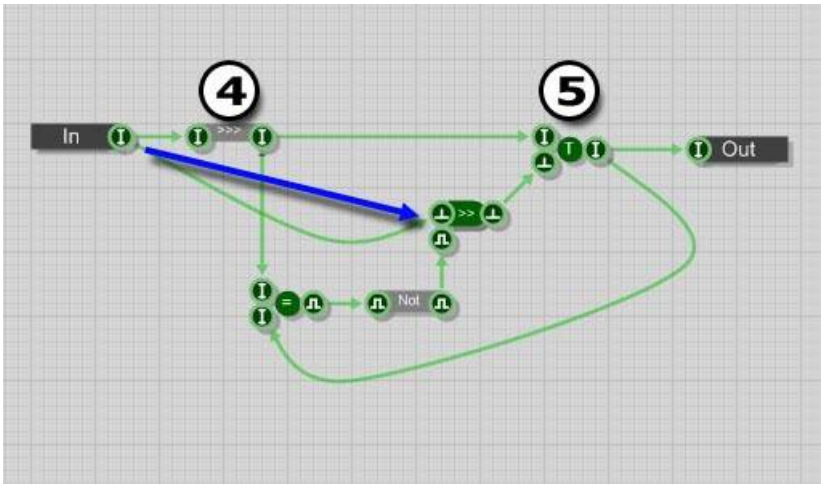
The 'AfterDup' is an After Duplicate primitive. This generates a trigger whenever you drag the module in from the toolbox, or do a copy and paste. All it does is to trigger the sample and hold when you add the module to your schematic – just to make sure that the output is 'initialised' with the current input value. It doesn't affect how the module works in normal usage, so I'll delete it from the following diagrams, just to make things a little clearer.



Let's assume that the previous knob value was 5, and we just moved the knob to the 4 position.

Looking at the link order, the first thing that happens, is that the integer is sent to the trigger blocker.

And that's it, the trigger won't carry on to the 'sample and hold', or the 'equals', because of the trigger blocker.

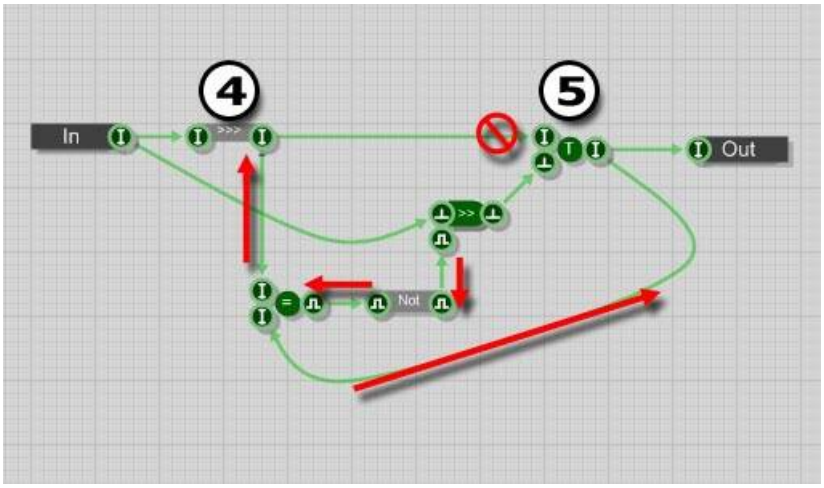


Next, the input trigger gets sent to the trigger switch.

So does it go through or not?

The input value didn't get sent to the 'equals' because of the trigger blocker.

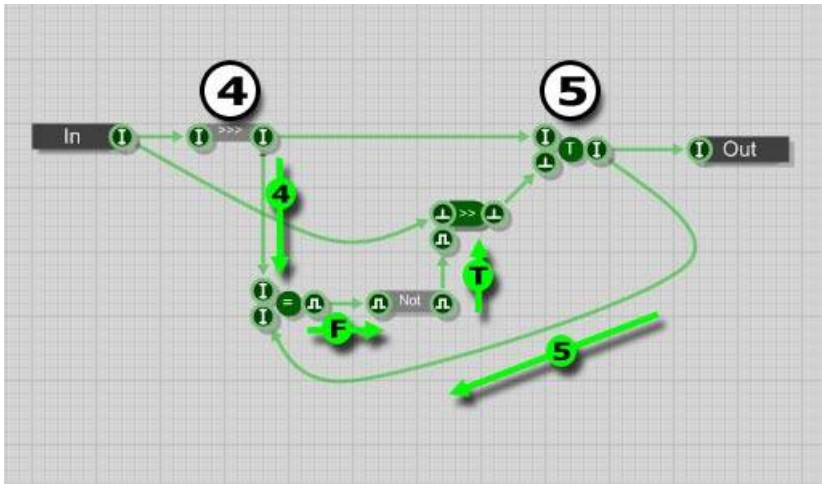
But it would be a bit useless if the 'equals' was still holding a 'true' value from a previous trigger – that would stop anything from happening!



So, the Trigger Switch sends out a 'backwards' trigger from its boolean input, asking for updated values.

These spread out along the various branches so that the 'Not' and the 'Equals' also request their newest input values.

However, remember that a 'sample and hold' will block backwards triggers, so it won't be asking to get updated.



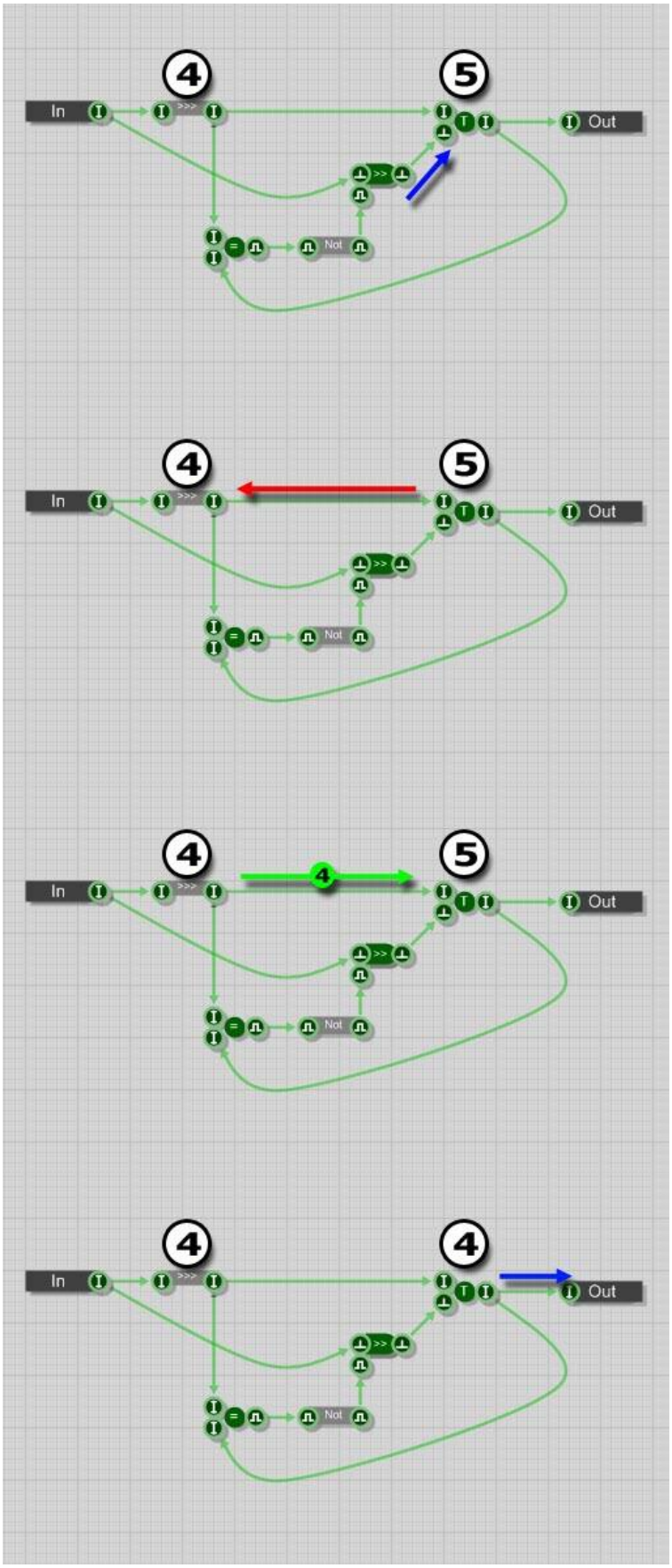
All the primitives that got a 'backwards' request trigger, now respond by passing on their current value.

The values then pass along the chain until they arrive back at the 'Trigger Switch' that made the original request.

Notice how this results in the 'Equals' primtive seeing the new value at its upper input, but the old value still appears at the lower input.

This is because... A) the sample and hold hasn't yet seen the new incoming trigger – the trigger switch hasn't even decided whether or not to pass it on yet! ...B) sample and hold primitives don't get updated by reverse triggers, so the equals can only ask to see the currently held value - the value at the sample and hold's integer input makes no difference.

Using a sample and hold in this way is a very powerful way of creating sequential logic – that is, any time that 'what happens now' depends upon 'what was the last thing that happened'. A good example of this is a toggle switch, where each click must take the previous value and invert it.



Anyway, our 'changed' module is now back on track; the Trigger Switch now has its answer – the numbers were not equal, so it should pass on the orginal trigger from its input.

Of course, had the two numbers been the same, the trigger would have been blocked – and that would have been the end of it, there's nowhere else left for the trigger to go!

The sample and hold now needs to know what value it should sample.

So it send a backwards trigger to the blocker asking for the latest input...

The Blocker responds with the input value...

And finally, the value is now sampled into the sample and hold, and a trigger is sent to the output to let the rest of the schematic know that there is a new value.

...well, not quite finally. There are actually two branches that the output trigger can take. As well as the final output, it will be sent back around to the equals. This can't change the output because the equals will now see two numbers that are the same – but maybe there is another tiny optimisation here by adding another trigger blocker?

The new schematic, with the 'changed' modules is shown in 'Example 7 – Only If Changed'.

Do the CPU meter test again. When you move the 'Squares' knob, the CPU load is now much lower than before.

But it hasn't made much difference to the 'Variable' knob. That's to be expected – the 'Square' knob has clicks to make it output only integers; even moving the knob all the way around will only generate five different values (and only 5 triggers, now that we have added the 'changed' modules.)

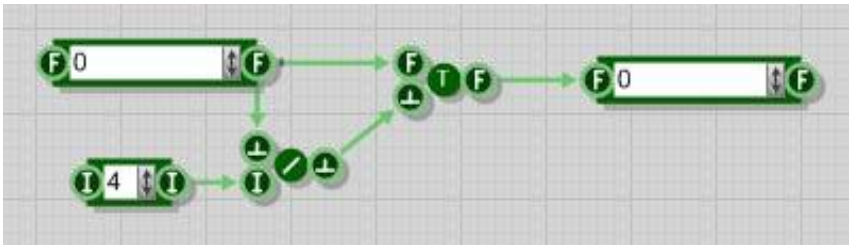
But the 'Variable' knob is sending out float values. We will have got rid of the surplus triggers from moving the knob past the ends of its travel, but when used normally, there will still be a new value for each tiny movement of the mouse – so still a lot of triggers if you move it fast enough.

Trigger Thinning

What we need is a way to 'thin out' those triggers so that there are less of them; we just don't need to see every single little increment on the display – though it would be nice to still see some sort of 'animation', as that helps to give a visual guide when moving the knob.

Trigger Divider

Looking through the toolbox, we notice that there is a primitive called a 'Trigger Divider' – now that sounds handy, let's see if that might work...



Here's a trigger divider in action. This is inside the 'Example 8' module of the examples schematic.

Try changing the float input value a few times to see what happens.

Well, it's certainly thinning the triggers out – the sample and hold now only updates the value on every fourth change. By changing the integer box value, you could make there be even fewer triggers!

But can you see the fatal flaw in using this method?

What if the input float was a knob, and you wanted to turn it up all the way to get the maximum value?

The final trigger when you get to the maximum might not be one of the one-in-four that the divider lets through – and your output value might get stuck at some value a little bit less than maximum. And that also means that the position of the knob would be a lie – what you see on screen wouldn't match the actual value being used by your plugin.

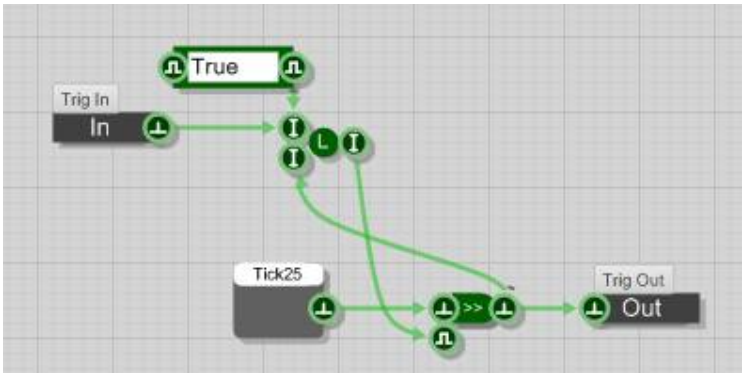
Or imagine that we were only moving the knob slowly – it would still block three out of four triggers, even though they were slow enough not to need thinning out any more.

Damn, it looked like such an easy solution, but it wouldn't be good to have that kind of unpredictable behaviour. We really do need for the **final** output value to always be the one that you set with the knob, and to only block triggers that are way too fast.

Trigger Limiter

In fact, a module to do what we want can be made wth only a handful of primitives, and better still, it will allow us to set a maximum trigger rate without ever blocking any slower triggers.

You can find this module in the 'Toolbox' module of the example schematic – it's called 'Trigger Limiter 25Hz'

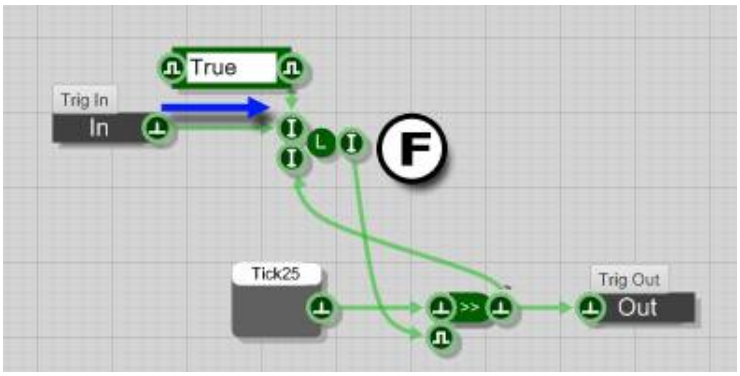


Hmm, this one confused the hell out of me the first time I saw it!

So, I think we need to look a bit closer at how it does its job!

Firstly, lets assume that when we load the schematic, the 'Float Switch' is initialised, so the value there is zero or 'false'.

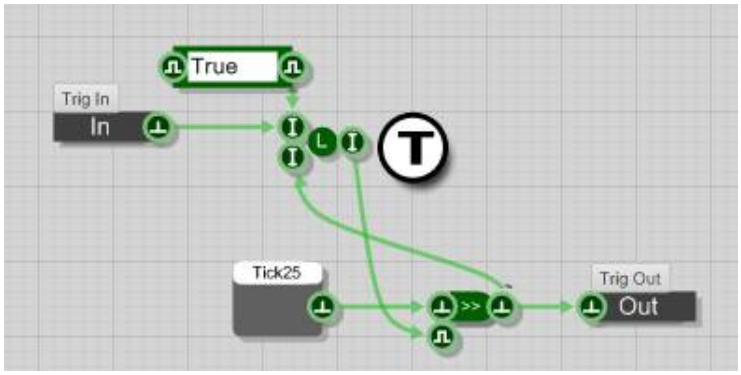
That makes it easy, the triggers from the Tick25 will be blocked by the trigger switch, so there will be no triggers at the output.



Now, here comes a trigger at the input to the Integer Switch.

But we are sending just a trigger to an input that is expecting to see a number, so what's that going to do?

All it does is to force the input to scan the input links to see if there is a number anywhere that it can use...

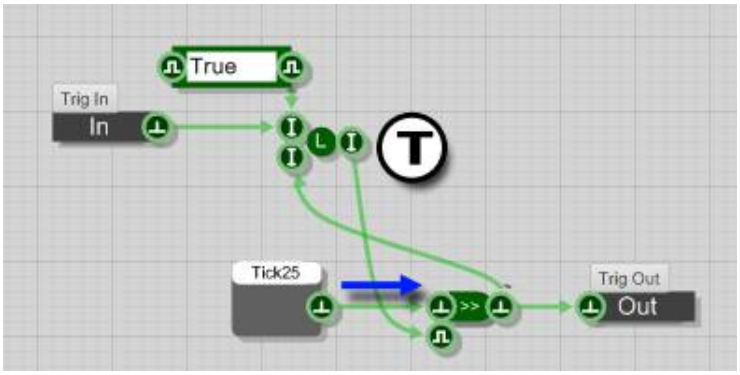


...and there is, because there is a boolean true also connected there – which is equivalent to one.

So the Integer Switch flips value to one (or 'true')

What if even more triggers come in at the input?

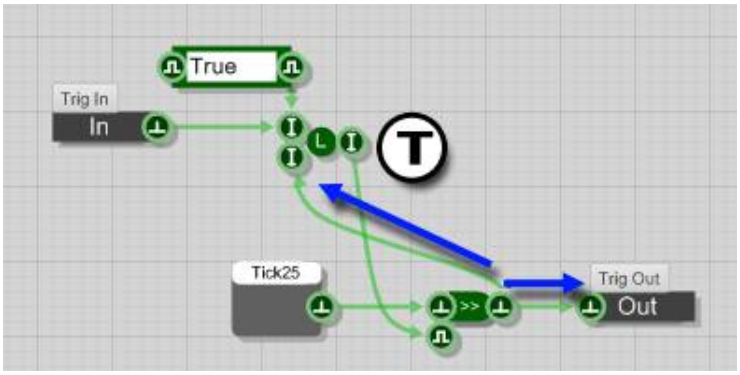
Well, nothing much will happen because the switch is already set to true - so there could be hundreds of input triggers and we just get a one out of the Integer Switch.



About 40 milliseconds later, it's time for the Tick25 to send out its next trigger.

The trigger switch will now ask the integer switch for its value, to see if the trigger should go through or not.

The switch is set to true, so it's OK for the trigger to pass through...

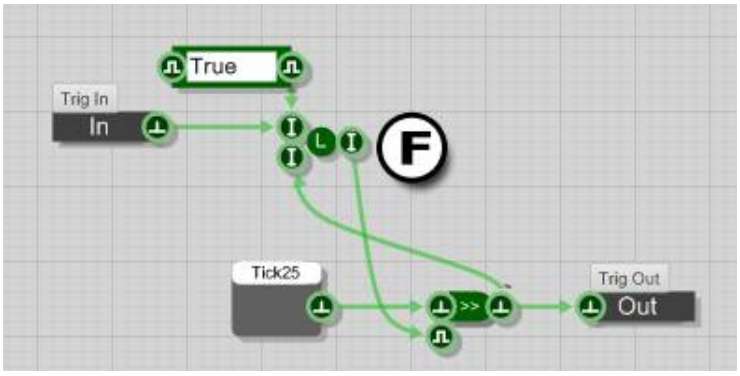


So a trigger gets sent to the output, to be seen by the rest of the schematic.

But note that it also gets sent to the bottom input of the Integer Switch.

But this input only has a trigger connection, there is no number for it to use!

But a number input with nothing connected is just taken to be zero, so the Integer Switch now flips back to zero ('false').



And that gets us back to where we started; the Tick25 triggers are now blocked again.

So, if there are no input triggers, there can be no output triggers.

There can be 100's of input triggers during the 40ms between 'Ticks', and that will still only generate **one** output trigger.

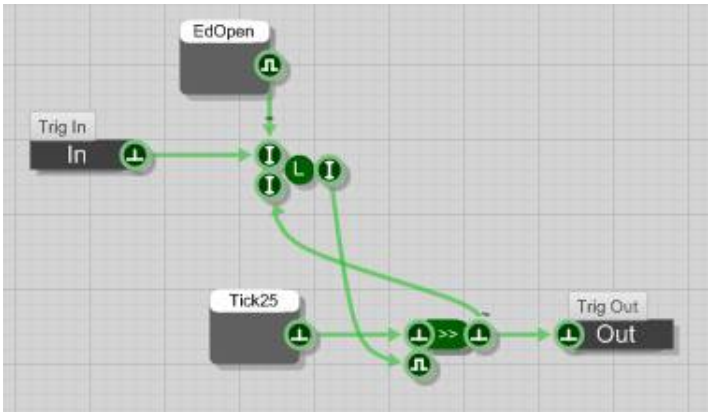
And each output trigger also resets the module, making it wait until there are more input triggers before it 'wakes up'.

Hook one up to a knob and a couple of Trigger Counters – the difference will be clear!

In module 'Example 9 – Trigger Limited' you'll find the Sine Generator with these new additions. This is now getting very close to the low CPU of the fully optimised version.

The only slight downside is that an input trigger could be delayed by up to 40ms, if it arrives just after the module has been reset. But, there's nothing to stop you from limiting the triggers that go to screen redraws (where you'd not notice the delay), but send them 'un-limited' to other parts of your design where the timing is more critical.

In fact, if it is just screen re-draws that you want to limit, there is a modification that makes the design even better...



'EdOpen' means 'VST Editing Window Open' – the output is True whenever you have your plugin GUI on screen inside a VST host.

If the editing window is not open, there's no reason to do all the calculations and re-draws for the GUI.

In this case EdOpen would output 'False', and the input triggers could never change the Integer Switch to the 'True' state.

So, as well as reducing the frequency of triggers, it can also prevent GUI re-draws when there is nothing to see.

SM's GUI parts can often be quite a big CPU drain, so, in the next tutorial, we'll look at other ways to lower the impact of drawing things on screen...