Architect Beta Quick-Start guide

Welcome to Architect, and thank you for being part of the Architect beta test. A beta software product is not quite a finished software product, and as such does not necessarily represent the quality of the final release: features may be missing, and bugs may be present.

Architect is a modular MIDI toolkit and music production environment for macOS, Windows, and Linux. Similar to the classic modular synths, you build patches from small modules that work together to generate pieces impossible to conceive of using conventional compositional techniques.

Architect is a large product, and in many ways very different from conventional music software. Reading this quick-start guide should hold your hand enough for you to feel confident experimenting in Architect on your own. But if you're still stuck, help is available:

- Try posting to the support forum at https://www.kvraudio.com/forum/viewforum.php?f=141
- Try contacting support directly at support@loomer.co.uk

Evaluation limitations

In evaluation mode, Architect can not load old projects, nor restore state when loaded in a host as part of a larger set. However, this state is still saved, and when a licence is purchased your previously saved projects will be available.

A licence for Architect can be bought from
http://sites.fastspring.com/loomer/product/architect

The latest beta versions can be found at https://www.loomer.co.uk/architect.htm

The most recent version of this document can be found at
https://www.loomer.co.uk/downloads/architect-quick-start.pdf

Finding your way around Architect

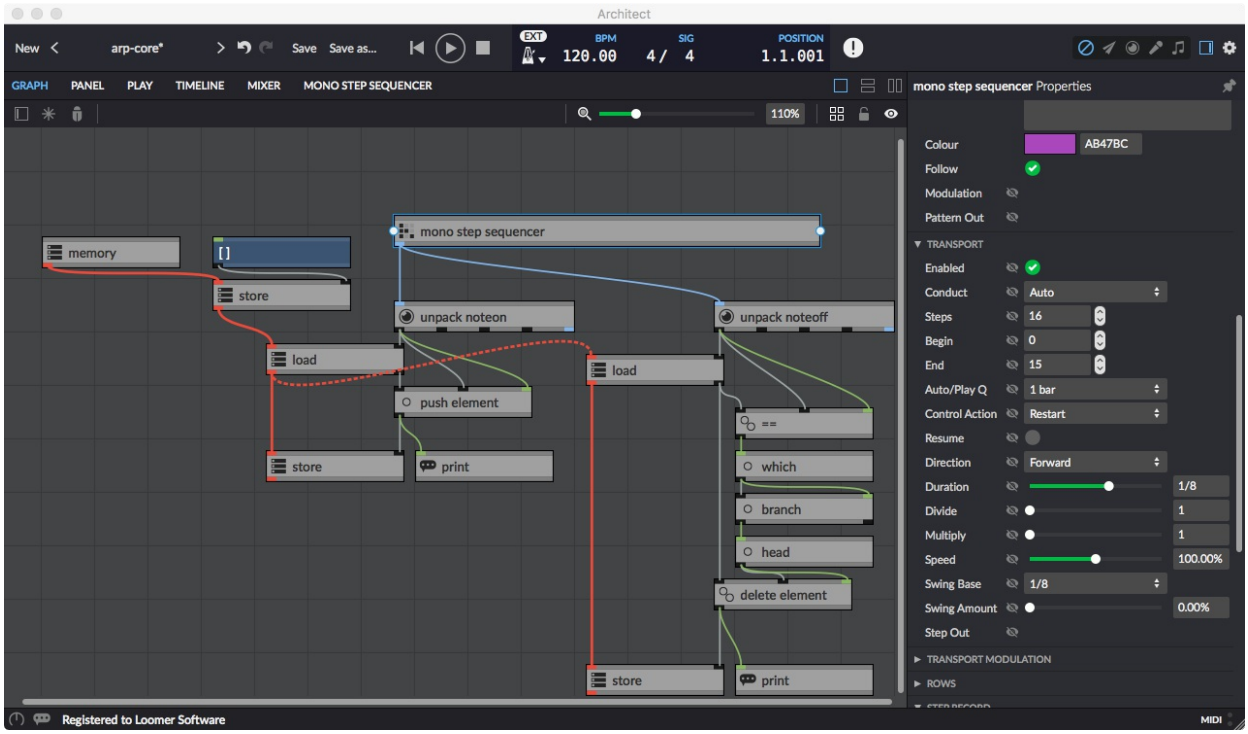
Architect mainly uses a single window interface. This window can be split in various ways, and many optional elements can be hidden when they are not being used. Two of these warrant special mention, as you'll generally want to always keep them open.

The properties

The properties window is shown on the right-hand side of the interface. If can be shown and hidden with the properties icon next to the cog on the top-right toolbar.

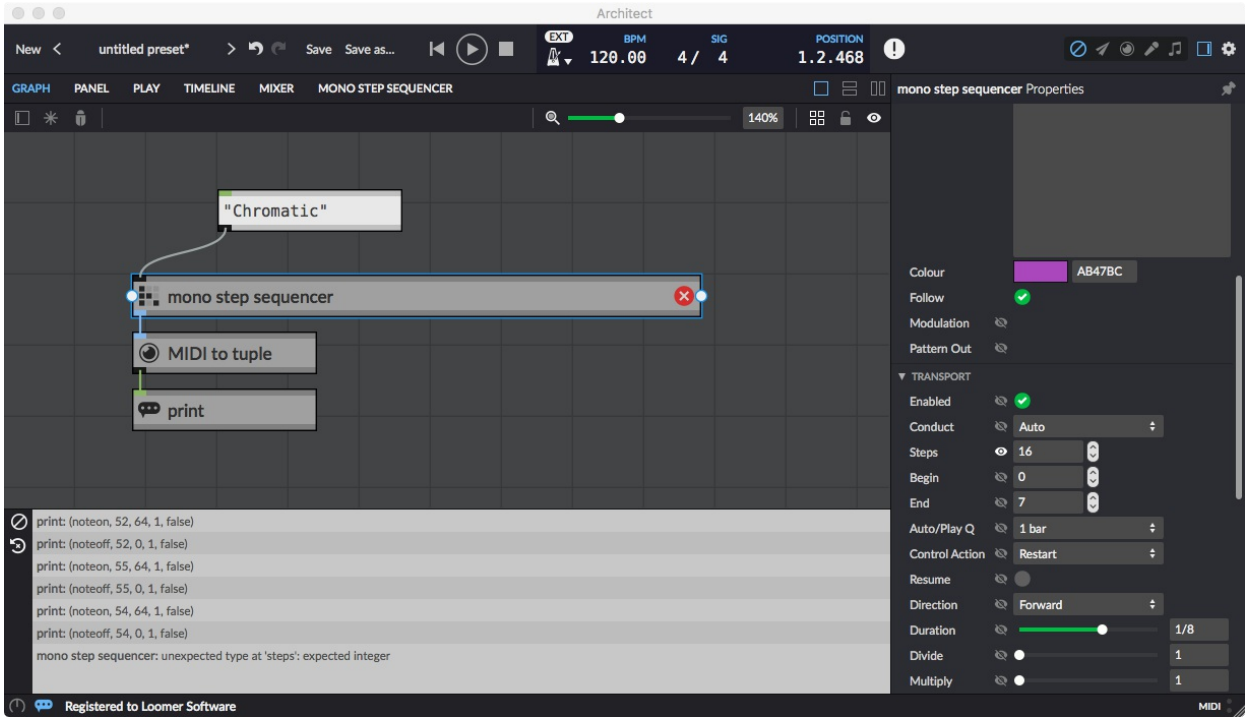
> Want to know what a button does? Hover your mouse over it for a second and you will be told.

The properties window shows the properties of the currently selected object. Here, we see (some) of the (many) properties of a mono step sequencer.



The console

The console, shown at the bottom of the interface, displays errors, warnings, and general feedback from the Architect engine.



Show and hide the console window with the button in the status bar at the bottom of the interface.

> If a new message is added to the console whilst the console is not visible, the button will display a notification icon to let you know.

You'll likely make many mistakes when first building your Architect programs. The console should be your first place to check when your creations don't work, as very often Architect will inform you where you've went wrong.

> Clear the console history with the clear button on the top left of the console window.

> Remove any error indicators with the clear error buttons below this.

The main windows

Architect's interface is divided into five main windows: graph, panel, play, timeline, and mixer. Additionally, contextual windows may be added to this in some circumstances. Selecting a sequencer, for example, adds a specific window to view the sequencer interface.

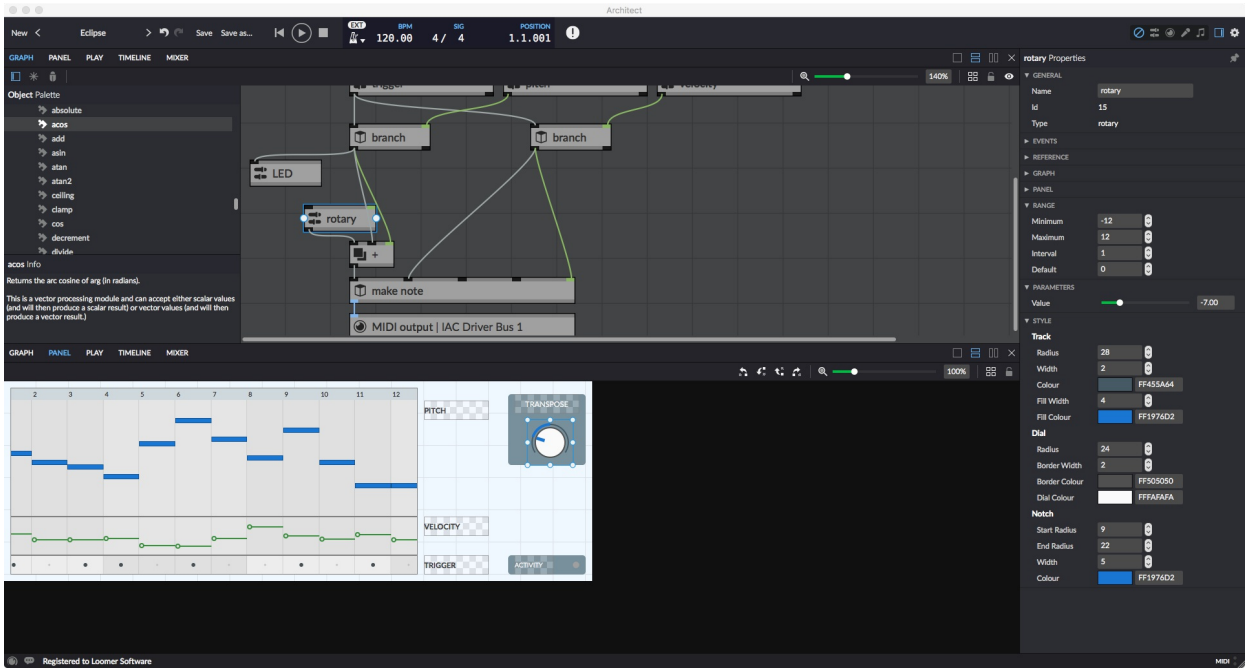
> Multiple windows can be display by splitting the window using the horizontal and vertical split icons on the toolbars.

Graph

This is where you build your Architect patches by adding modules and wiring them together. If you've used a modular synth before, you'll be at home here.

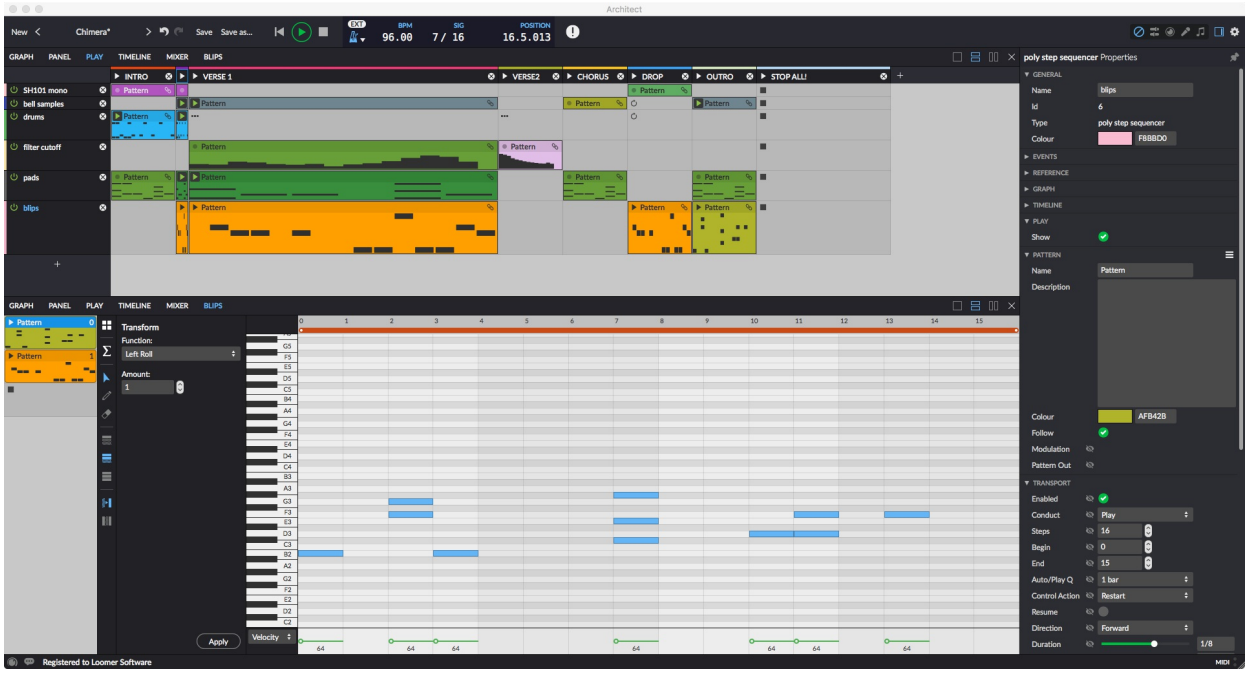
Panel

You can create your own custom interface for your patch by adding various UI components, such as buttons, labels and sliders. The components can be styled to give your patch its own look.



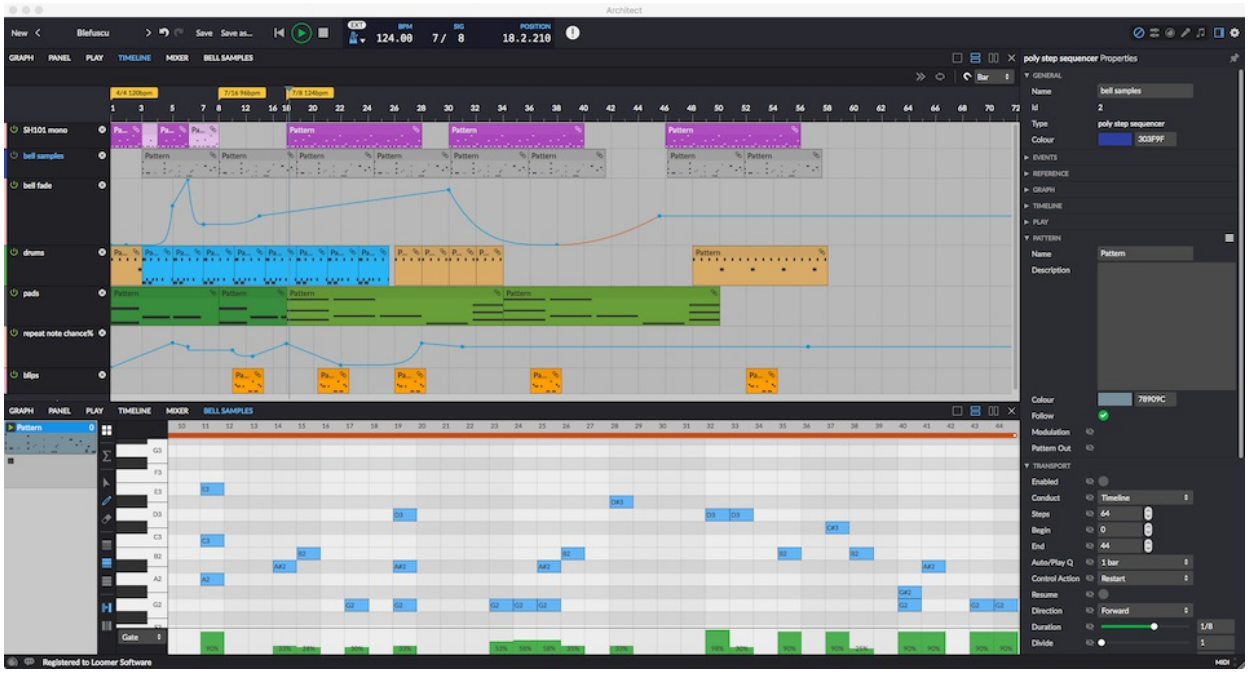
Play

The play window is for non-linear sequencing. Scenes are added horizontally, and sequencing devices vertically. Playing a scene will play all the clips in the same column. You can also set a cell to stop, restart the current sequence, or continue from where it currently is.



Timeline

The timeline is for linear sequencing. If you've use any other sequencing application before, you'll likely be familiar with this type of arrangement.



Mixer

The mixer if where your Architect MIDI output is routed to be turned into audio. Like a traditional software DAW, you can add tracks and plug-ins to these tracks.

Performance Parameters

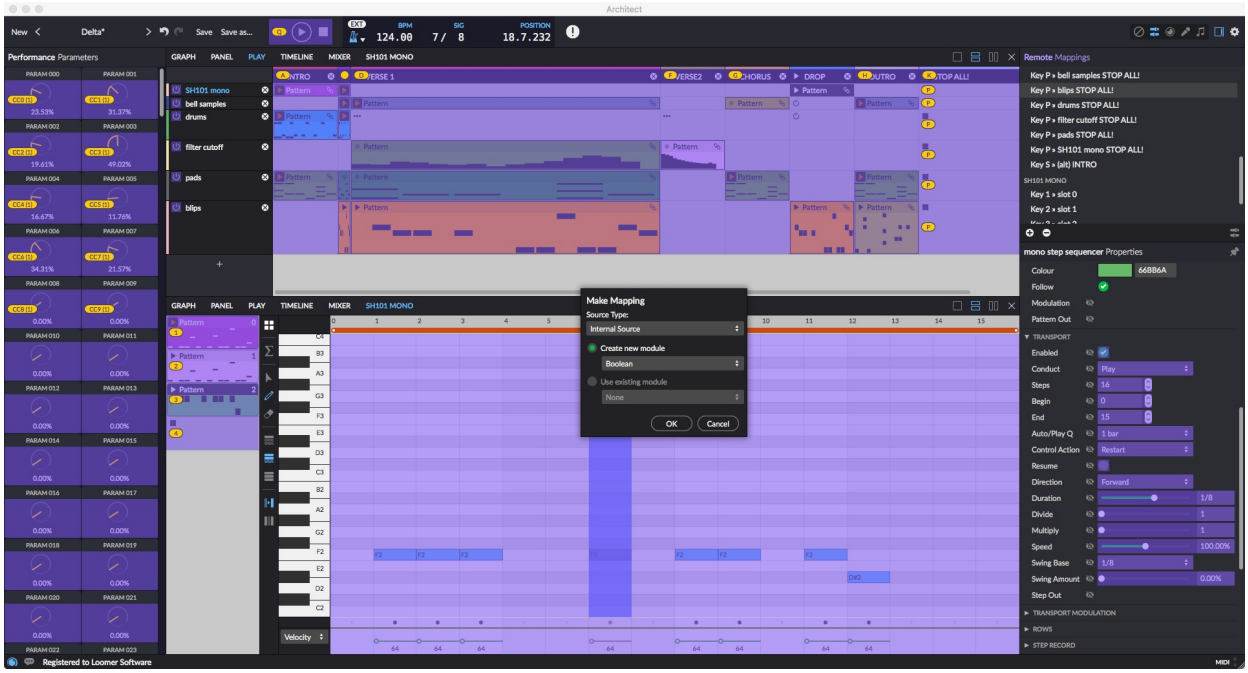
These parameters can be shown and hidden with the button the status bar. Performance parameters can be mapped to other components in your Architect patch. The parameters are also exposed to any compatible host, so when using Architect as a plug-in they provide an easy way to automate things externally.

The secondary windows

These smaller windows sit above the properties windows. Only one (or none) can be shown at a time.

Remote mappings

This window displays the current remote mappings. When the window is visible, all potential mapping targets will be highlighted.



MIDI files

MIDI files can be imported here to be played with the `MIDI player` module. You can also capture your Architect MIDI output here with a `write to MIDI Pool` module.

Audio files

Captured audio output from Architect is displayed here. Audio files can be exported or dragged into other applications.

Scales

Architect comes with over 30 build-in scales, but here you can also define your own custom scales.

The file browser

Clicking the preset name (which may be "untitled preset" if you've not yet saved your preset) will open the integrated file browser.

Your first patch

When trying a new programming language, it's tradition to write a program called "hello, world", which simply prints those very words. Let's do that in Architect.

- Start with an empty preset. If you've already made some changes, then either save or discard them and start a new preset by clicking "New".
- We'll need the following windows visible: the graph, the properties, and the console.
- First off all, let's add a `print` module to the graph. There are many ways to add modules. You can:
    - Show the Object Palette window (using the button on the top left of the graph) and drag it in from the selection. The `print` module is found under the Built-in > Output branch.
    - Right-click on the graph and select `print` from the Built-in > Output menu. Hmmm, both of these methods seem a little slow. Is there a faster way?
    - Yes! Use the quick dialog! With the graph focused, type the word print, and then press return. This method is by far the fastest way to add modules and so should be the preferred way.
- The green slit in the top of the print module is called an inlet. Inlets are how data is passed into a module. Outlets appear on the bottom of modules, and this is how modules pass data out.
- So let's create some data. Type the word data (or choose the `data` modules from Built-in > Data Source). A `data` module allows you to create an object of any type understandable by Architect.

Types

Objects passed between modules have a specific type, which tells you what information they represent. Architect supports several simple data types:

string - Some text. These are represented by "things in quotes".

integer - Whole numbers, such as 0, -23, 42.

float - These are numbers with decimal points, such at 0.5, 23.7, and 1.0.

boolean - These can either be true or false.

> All data types can be implicitly converted to boolean values. 0, 0.0, undefined, the empty string, empty array, empty map, and empty tuple are all false. Every other value

> is true.

undefined - This type represents the absence of a value. When a `get key` module (which outputs the MIDI key that the given MIDI message has) is passed a sysex message, its answer is: undefined.

signal - This is a generic type that contains no further information. Signals are commonly used to indicate events. For example, a `metronome` module will periodically send out signal objects.

Architect also supports several composite data objects, which are built up from these simple data types.

array - Zero or more items in a list. They are denoted with square brackets, for example [0, 2, 3, 5, 7]. Arrays are homogenous, i.e., all elements must be of the same type.

> Be warned that when it comes to creating arrays, integers and floating-point numbers are treated as discrete types.

tuple - A tuple consists of zero or more elements. They are denoted with parenthesis, for example ("noteon", 60, 10). Unlike arrays, they elements need not be of the same type, but the tuple must be of a fixed length. In tuples, each element position represents something specific, so reordering a tuple's elements makes no sense.

map - A map consists of a number of key and value pairs, They are denoted with curly braces, for example {"noteons": [60, 62, 64], "noteoffs": [60]}.

> There are also types for each MIDI message. We'll deal with them later.

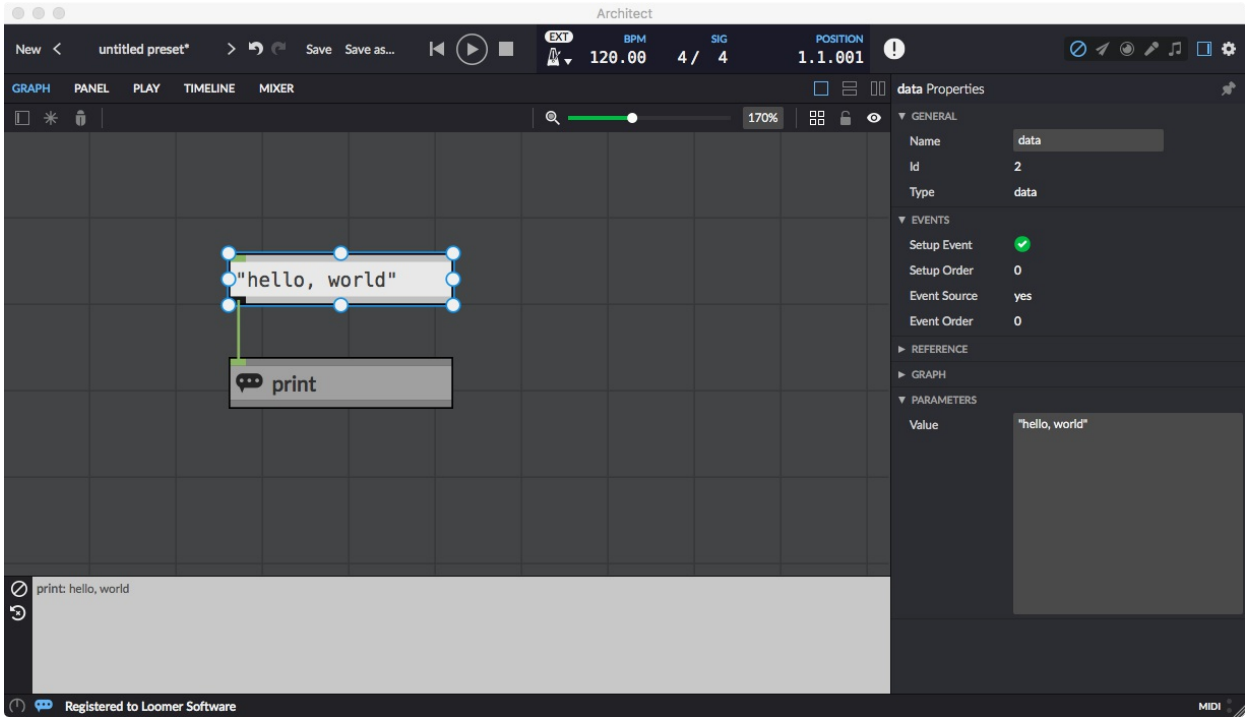
- Back to "hello, world". With your `data` module selected, enter the phrase "hello, world" (including quotes) in the parameters > value field in the data properties window.

- Now we need to route the data from the `data` object to the `print` object. Click on either the `data` outlet or `print` inlet and drag to the other corresponding pin. A connection will appear between the two objects.

- If you find it difficult to make these fine-tuned connections, you can zoom in the graph with the percentage slider in the graph toolbar.

> Connections can be removed by right-clicking on them.

- If everything has went according to plan, the words "hello, world" should appear in your console window.



- Double-clicking on a `data` module will send the string again. Try it! Notice that Architect will collect together consecutive identical console entries with a (x2) icon, rather than flooding the console.

- Try changing the value of the string in the `data` module. Try sending some of the other simple types. Try sending some composite types.

Module order

Everything in Architect happens in a well-defined and predictable order. Architect's engine runs at 960 PPQ, and every tick the following two things occur:

- **setup events** - Any modules that have pending setup events will have an opportunity to process. You can check which modules do have setup events in the properties window, under Events. Some modules, such as `data` can have their setup events disabled if you'd prefer for them not to output anything at setup time. This is handy is they are controlled by other modules before them. Setup events for a module occur only if the modules have not output anything yet, and when they are first connected. If a module causes another module to process, the second module will not use its setup event.

> The reset / panic button on the top of the interface causes setup events to be fired again. This can be a handy way of testing how well your patch would behave when first loaded. It can also be handy to get things back to a well-known state.

- **event sources** - Any modules that are event sources will have an opportunity to process.

Every event in Architect will start from either a setup event or event source.

In general, the order that modules process is decided by their position. A module above another will process first; for modules on the same vertical position, the left-most module will process first.

For setup events, modules inside macros will process before modules outside them.

For event sources, modules inside macros will process at a time decided by the macro's position.

> You can view the order of events using the Visual Options (top right of the graph toolbar) > Show Order.

Event order

When a module sends an object to more than one other module, the order that the downstream modules receive the event is decided by the same top-to-bottom, left-to-right order. If an event goes to multiple inlets on the same module, they will arrive left-to-right.

Sequencers and MIDI types

- Start with a new preset, and add a `mono note sequencer` from Built-ins > MIDI Source.
- Double-clicking on the `mono note sequencer` will open the sequencer's interface in a split window. Using the mouse, compose a melody on the piano roll display by clicking on a cell.
- There are several more unusual parameters available by selecting a different row from the dropdown that currently says "Velocity":
    - **multiply** multiplies the length of a step by the given integer amount.
    - **divide** divide the length of a step by the given integer amount. With these two parameters, you can have exotic step-lengths such as 13/9ths of a beat.
    - **speed** allows a step to be stretched or slowed down by a fractional percentage.
    - **count** indicates how many times a step is played. 0 skips the step altogether.
- Start the transport by clicking play, either in Architect or in your host. You should see a position indicator moving, but nothing seems to be playing yet. We need to route the MIDI output from this module to somewhere else.

- If you've got external MIDI devices, you can add a `MIDI Output` module from Built-in > Output and choose the device destination. Connect the outlet from the sequencer to the `MIDI Output` inlet. Notice that MIDI cables are blue.

- If you're running Architect as a plug-in, you can route MIDI to your host by selecting "host" as the destination in the `MIDI Output.`

- If you'd prefer to route the MIDI to a hosted plug-in, you'll first need to ensure that Architect knows about your other plug-ins. Scan your plug-ins from the Preferences > Plug-Ins window (the cog on the top right of the interface). Now, under the Mixer tab, add a track with the add button, and drag a synth plug-in to the new track. If you can't see your plug-ins, ensure that the Plug-in Palette window is visible by clicking the button on the top left of the mixer. Choose the track as the destination in the `MIDI Output.`

Sequencers can be synced in a variety of ways. By default, a sequencer created in the graph is in *auto* mode. This can be changed in its properties, under Transport > Conduct.

auto - automatically starts playing a sequencer in time with the Auto/Play Q settings.

play - allows a sequencer to be played live using either the pattern window on its left, or by adding it to the play window. A sequencer created in the play window will automatically have its conduct set to play. Other sequencers can be added to the play window by enabling show under the sequencer's play property group.

timeline - allows a sequencer to be controlled by an arrangement in the timeline. A sequencer created in the timeline window will automatically have its conduct set to timeline. Other sequencers can be added to the timeline window by enabling show under the sequencer's timeline property group.

modulation - allows a sequencer to be controlled by its modulation inlets.

Modulating sequencers

Many of the sequencer parameters can be modulated by other sources. Enable such a modulation inlet by clicking the eye icon in the sequencer properties, next to Note > Transpose. Notice that an inlet has appeared on the graph module.

> Hovering the mouse over an inlet or outlet pin or module will show the names of the pins.

Create a `data` module and set it's value to 3. Now, connect this to the sequencer's transpose inlet. Notice that the output sequence has been transposed.

Transforms

Architect provides a number of transforms for manipulating sequencer rows. They can be shown by clicking the transform icon on the toolbar to the left of a sequencer.

> Select which steps you want to process with the select tool, underneath the transform button.

Which rows are affected by transforms depends upon which grouping mode is selected.

> Choose the grouping mode from the sequencer toolbar. Select either "selected only", "selected plus dependencies", or "all".

Step sequencers consist of a number of discrete rows, such as pitch, velocity, and count. Sometimes, you wish to manipulate these rows together, and sometimes individually.

> "Selected only" mode will only apply a transform to selected rows.

> "Selected plus dependencies" will apply a transform to the selected rows and other closely dependent rows (generally, ones in the same colour.)

> "All" will process all rows.

Choose "Shuffle" from the transform functions, and ensure "all" rows are to be transformed. Apply the transform to rearrange your sequence.

> "Coherent shuffle" will keep all the data for particular steps together when shuffled.

Data processing

- Start with a new preset, add a `print` module, and add two `data` modules.

> Or create one data module, and duplicate it with ctrl/cmd + D, or by right-clicking the module and choosing "Duplicate".

- Add an `add` module from Built-ins > Maths. The `add` module sums two other objects together. Why, then, does it have three inlets?

> Green inlets are called "active" inlets. Modules will generate events in response to events received to these inlets. Dark, inactive inlets simply set a module's properties or values.

properties - are visible in the properties window and are persisted when the patch is reloaded.

values - are only temporary, are not visible in the properties window, and are lost when the preset is reloaded.

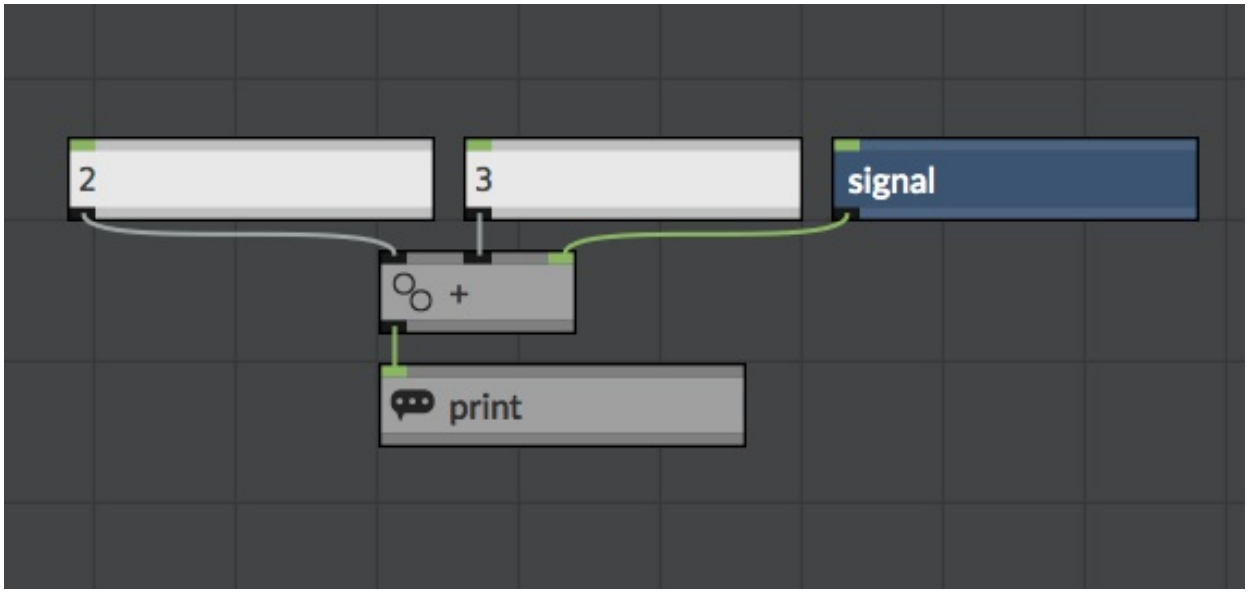
The `add` module, then, has two values, and an active `call` inlet. Sending *any* object to the `call` inlet will cause the module to sum the previously received values. Let's do that now.

- Set one of the `data` modules values to 2, and the other to 3. Connect the first `data` to the left inlet of `add`, and the second `data` to the middle of `add`. Connect the `add` outlet to the `print` inlet.
- Create a `signal` module from Built-ins > Constant, and connect this to the right-most `call` inlet of `add.`

> A `signal` module simply sends a signal object during its setup or when double-clicked.

> If you send an event to the `call` inlet before the add arguments have been specified, a "missing argument" error will occur.

Changing the value of the `data` modules connected only to the others inlets will not cause the `add` to process; only when it receives the `call` (such as by double-clicking the `signal`) will it produce an output.



Vector processing

Single values, such as what we have been using, are called **scalar** values. **Vectors** are an array of such values. Many modules in Architect support vector processing, simplifying the patch by removing the need for looping structures.

Change one of the `data` values to an array of values, such as [0, 2, 3, 5]. Double-click the `signal` to run the `add` module. Notice in the output how Architect has added the scalar and vector values to produce a vector value.

Recycling vectors

If a vector processing module receives arrays of different lengths, it will *recycle* the array elements to produce an array the length of the longest input array. With your current preset, set the `data` values to [1, 2, 3, 4, 5] and [-100, 100], respectively. Notice the output: [-99, 102, -97, 104, -95]. Because the second array is shorter, its elements will be reused, giving an array of [1 + -100, 2 + 100, 3 + -100, 4 + 100, 5 + -100].

Creating an interface

- Start with a new preset, and go to the panel view. Right-click and add a Built-in > Component > Rotary. This will create both the panel component, and a corresponding graph module.

> If you create a component module in the graph it will not appear in the panel unless you tick Show from its Panel properties.

- Notice how the rotary can be moved on the panel, resized, or styled with the Style properties.

- In order to actually manipulate the rotary component, you will need to leave edit mode by either pressing ctrl/cmd + E, or clicking on the padlock on the toolbar.

> When locked, items can't be moved, added, or removed from the view.

- In the graph, create a `print` and connect the `rotary` to the `print` module.

- Notice how when the rotary is moved, the value is printed in the console.

Mappings

Many components within Architect can be mapped to internal or external controllers, such as

MIDI keyboards or even your computer keyboard.

- Enter mapping mode by pressing ctrl/cmd + M, or click the mappings button at the top right of the interface.
- Anything that can be mapped is now highlighted in purple. You can either manually map to a device by right-clicking and choosing "Add Mapping", or by using auto-mapping.
- Enable auto-mapping by clicking on the button on the bottom right of the Remote Mappings window. Now, when a mapping target is selected, any events received will be automatically mapped to this target.
- Click on the rotary component and either press a key on your computer keyboard, or play an attached MIDI keyboard or controller.

> Ensure your MIDI devices are selected as input sources in the Preferences > Devices window.

> If you are using the computer keyboard as a mapping source and the graph is focused, be sure that the graph is locked otherwise key presses will just open the quick dialog.

> Remember to turn off auto-mapping after you're done!

- Mappings sources can be:
  - **boolean sources** that generate on and off states, such as a MIDI keyboard or computer keyboard.
  - **numeric sources** that generate a range of values, such as a MIDI CC rotary controller.
  - **trigger sources** that generate a single event.

> Mappings can be routed into and out of the graph using the modules in the Built-in > Mapping category.

Building macros

Macros are reusable groups of modules that can be treated as a single module. Here, we will build a transpose macro for transposing MIDI pitches.

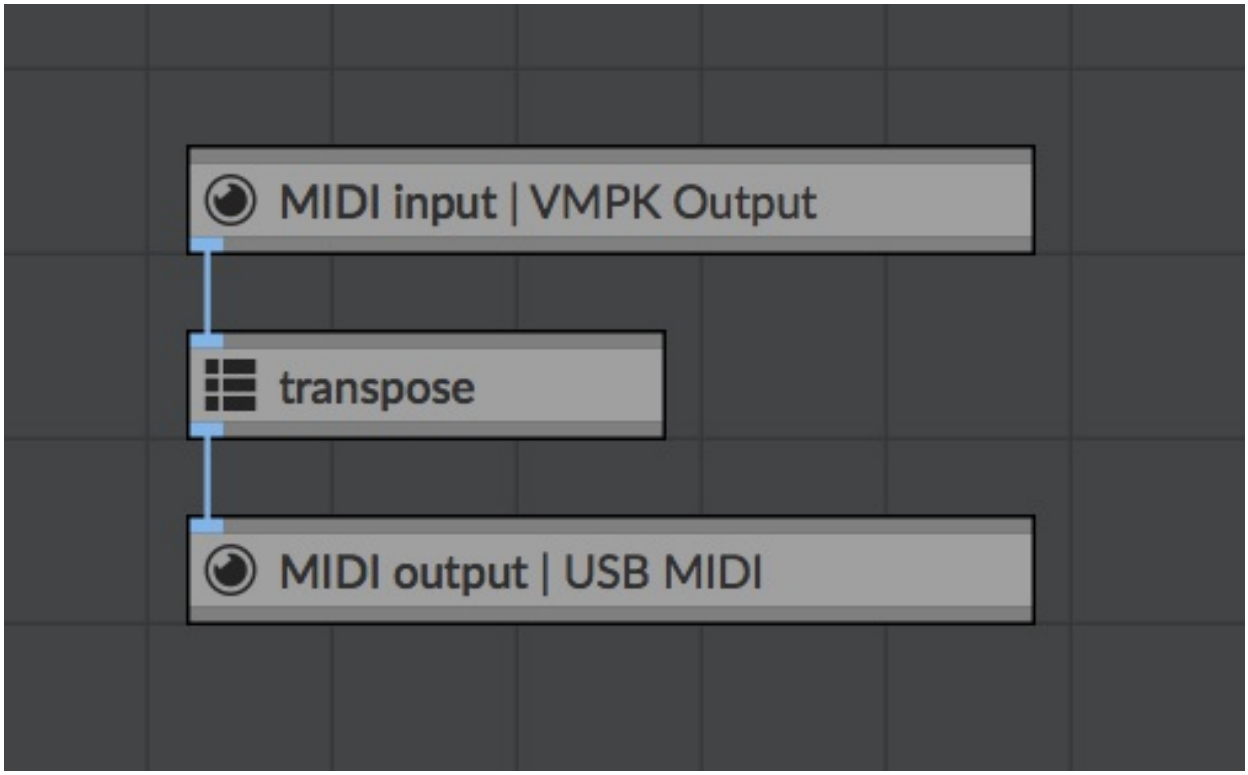
> This macro is already available in Built-in > MIDI process.

- On the graph, create a `MIDI input` module (from Built-ins > Input), and a `MIDI output`. Choose appropriate source and destination devices for each.
- Create a new `macro` from Built-in > Macro.
- Rename the `macro` to transpose in the properties.
- Double-click the macro to view its (currently empty) contents.

> When inside macros, you can see where you are in the graph using the breadcrumb trail at the top of the graph.

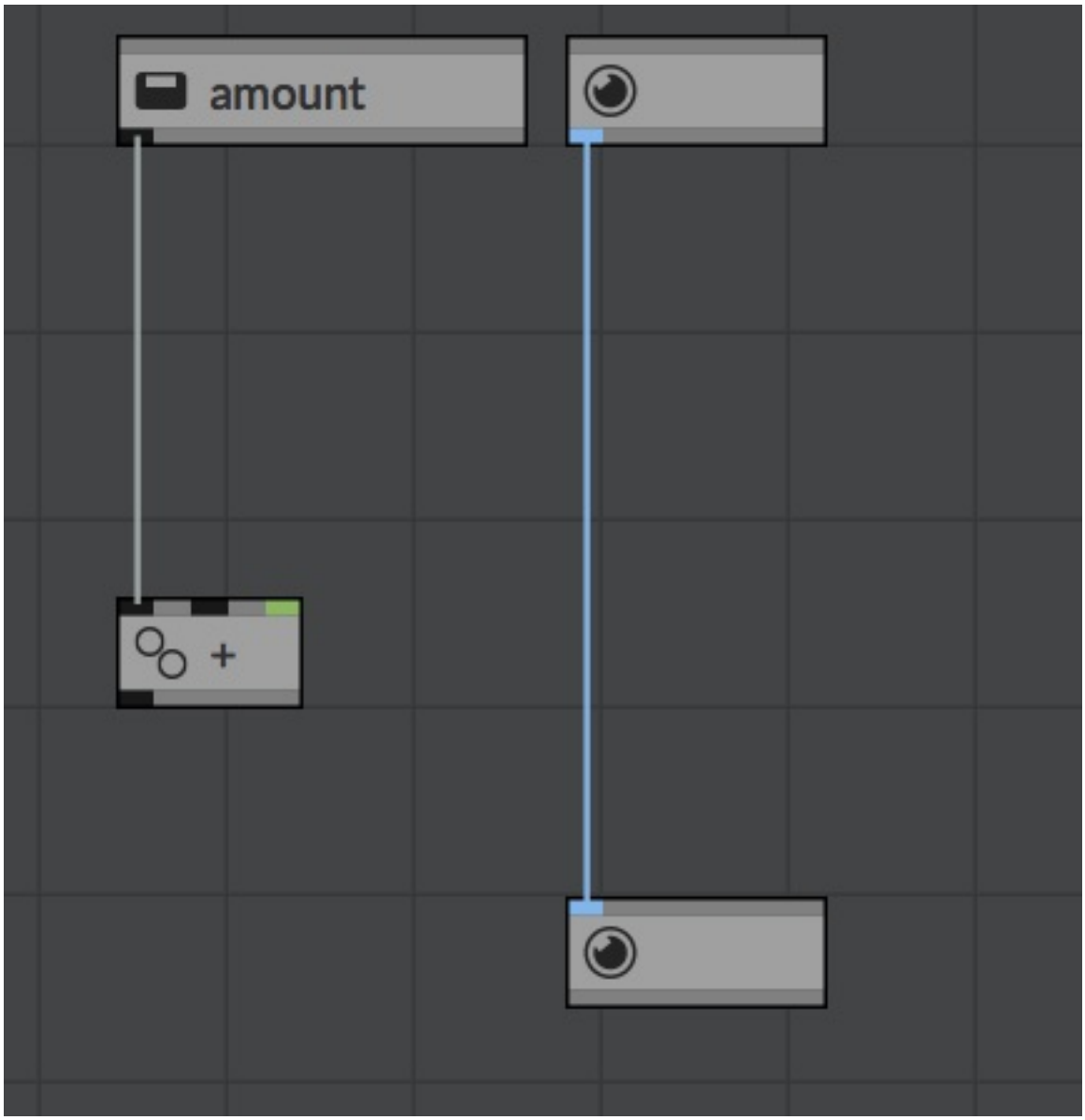
- Macros need inlets and outlets to communicate with the outside world, so add `MIDI inlet` and `MIDI outlet` modules. For now, let's simply connect them together.
- Return to the graph root by either clicking on root in the breadcrumb trail, or by double-clicking an empty spot in the macro. Connect your `MIDI input` to the `transpose` macro, and then the `transpose` macro to the `MIDI output`.

Confirm that MIDI messages are routed from input to output via the macro.



- We need to specify an amount by which to transpose the MIDI notes. Inside the macro, add a `data inlet`, and call it `amount`. Notice that the name of the inlet module is used to name the inlet in the macro.

- Transposing means adding something to the MIDI note key, so create an `add` module, and connect its left-hand inlet to the `amount` inlet.

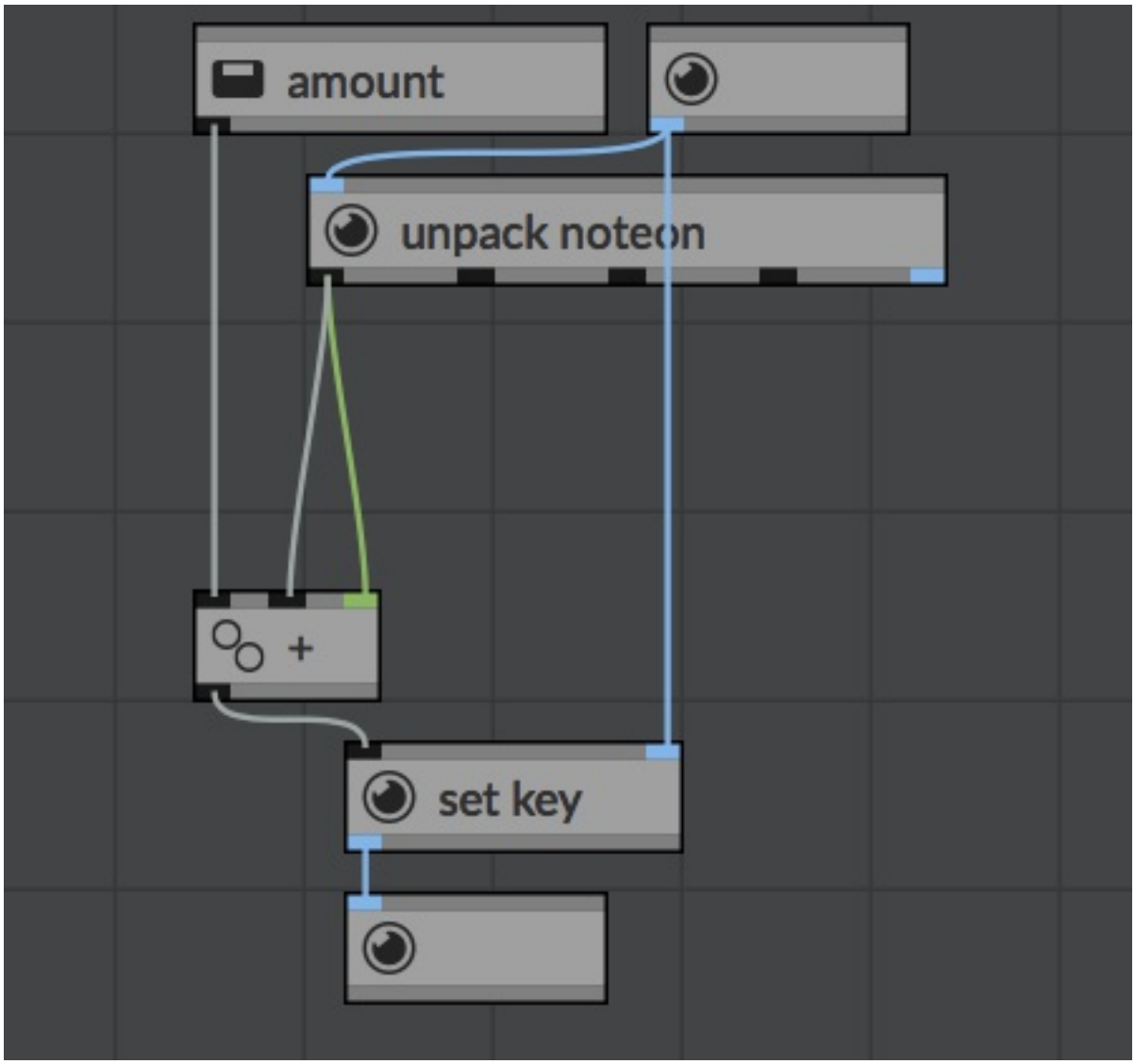


- Setting the key of a MIDI note is done with a `set key` module (from Built-ins > MIDI process). Connect the `add` outlet to the left inlet of `set key`. Remove the cable between the MIDI inlet and outlet, and go from inlet to `set key` to outlet. (See screenshot below.) Now, how do we know to what key we are adding the transpose amount? We need some way of extracting this from MIDI notes.

> `set key` will pass through unaltered any MIDI messages that don't have keys.

- Add an `unpack noteon` module from Built-ins > Data Source. This module, on receiving a noteon MIDI message, will send values containing the note's key, velocity, and channel (and uncoupled status: more of that later.) Any other type of MIDI message will be passed to the surplus outlet.

- Connect the macro MIDI inlet to the `unpack noteon` inlet. Connect the `unpack noteon` key outlet to both the middle and right inlet's of `add`. This will both set the second argument, and call the `add` module.



- Now, if you play a MIDI note, you will get… a "missing argument" error.

> Right-click on an error in the console and choose "Find Source" to locate an erroring module.

- We've not yet specified the amount to transpose by. Let's give it a default value so it still works even if no amount is specified. Add a `0` from Built-in > Constant. This module simply sends a constant 0 value. You can do the same with a `data` module, but these constants are a little more efficient. Connect the `0` to the left-hand inlet.

> This inlet should now be connected to two things, the `0` module and `amount` inlet.

Because the `0` module has a setup event, when you first instantiate a transpose macro, it will

set the first argument of `add` to 0.

- Now, in the root of the graph, create a `data` and set its value to 12. Connect this to the `transpose` amount. MIDI notes should now be transposed by an octave. Result!

- You may be wondering, what happens if the transpose amount changes between a noteon and noteoff. Won't we get a hanging note? The answer is no, and that is because `set key` has **note matching**.

Note matching

Note matching means that some modules (such as `set key`) will remember their state when they receive a noteon, and then apply the same state to a noteoff, even if things have changed in the meantime. Try it: play a note, change the transpose amount, and then release the note. Notice that the noteoff ignored the current transpose amount and instead used the one its coupled note had.

> Note matching can be disabled, per module, in the properties.

Note coupling

> This is an advanced topic, and one you don't need to worry about if you're just starting with Architect. Be aware of it, but don't panic if you don't fully understand it.

A closely related concept to note matching is **note coupling**. A coupled noteon is matched with a coupled noteoff. By default, notes are coupled, which means that a noteon message will eventually be followed by a noteoff message. Architect uses this fact to try to avoid hanging notes by automatically inserting noteoffs if the source of the noteon is disconnected. In the vast majority of cases, you needn't worry about this concept and can feel secure that Architect won't leave notes hanging.

But sometimes, you want uncoupled notes, such as passing a noteon to a drone that you don't ever want to stop. Or maybe you're routing noteon and noteoffs to different places. In these cases (and to be honest, probably in only these two cases) you will want to uncouple your notes to stop Architect automatically generating noteoffs for you. You can convert notes with `uncouple note` and `couple note` modules from Built-in > MIDI process.

Finishing the transpose macros

So what happens if someone sends something odd to the `transpose` amount inlet, like a

string. Wouldn't it be good if Architect can specify that only a subset of types are accepted? Well it can, with **type matching**.

Type matching

Data inlets (and the standalone `type check` module) can raise an error if the type passed to them don't match the expected types. This can be set in the Type Check property.

You can specify the following simple types: `any, boolean, string, integer, float, signal, undefined`.

The `number` matcher will accept both integers and floats.

All numeric types can be prefixed with `positive`, `nonpositive`, `negative`, or `nonnegative`, for example, `nonnegative integer`.

You can also specify `map`, `array`, or `tuple`. Arrays can prefixed with `nonempty`, and for arrays and tuples the type of expected elements can be specified with `array[integer]` or `tuple(string, integer)`.

Finally, these can be combined with `or` to build complex matches, such as `integer or array[integer] or tuple(integer,integer) or tuple(integer,array[integer]) or array[tuple(integer,integer)]`

> This last example is what a mono sequencer pitch row inlet accepts.

So what type should we have for the amount inlet? `integer` would fit the bill, but why limit ourselves: the vector processing modules can also work on arrays, so maybe `integer or array[integer]` is a better fit. By specifying an array for the transpose amount, say [-12, 12], each note will produce two harmonies.

Memory

Memory modules allow a group of modules to all share access to the same variable. Memory cables are thick and red, and rather than representing data flow they simply show that all connected modules are accessing the same variable.

To show memory in action, lets build a counter macro.

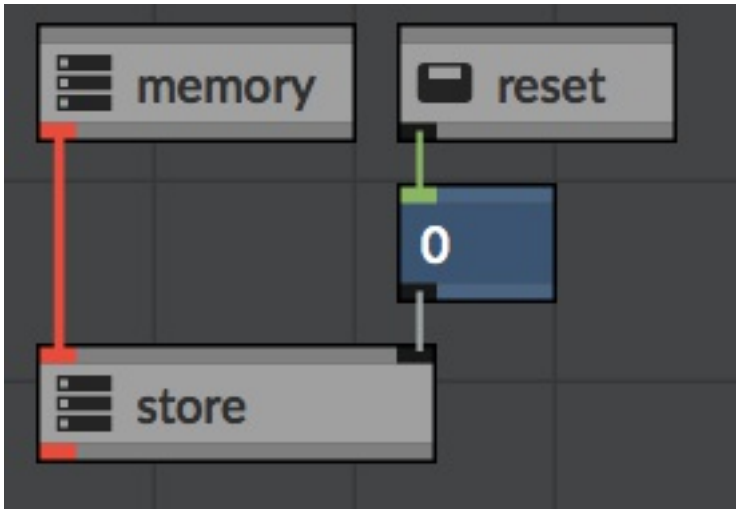
- From a new preset, add a `macro`, rename it `counter` and give it a data inlet called `reset`, an active data inlet called `read`, and a data outlet.

- Inside the macro, add a `memory` module from Built-in > Memory, and `store` from the same location.

- Connect the `memory` outlet to the `store` inlet, remembering that this doesn't specify data flow, but simply says that the `store` uses the connected `memory` variable.

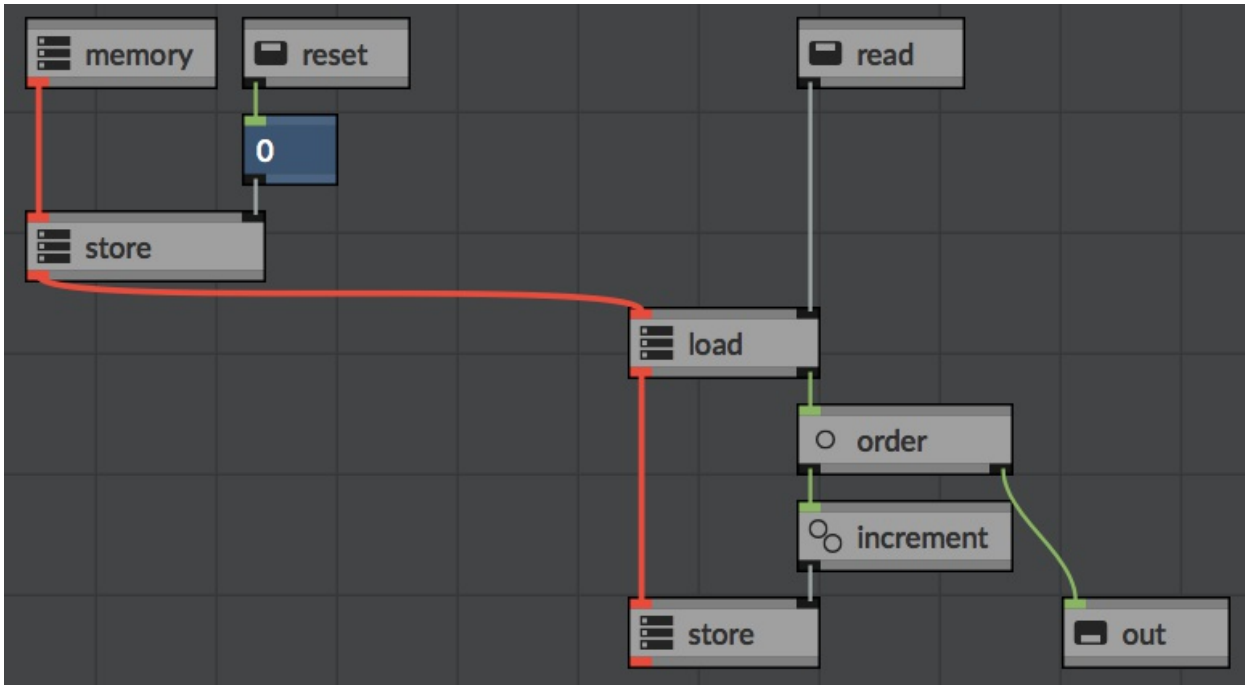
- Add a `0` module and connect its inlet to `reset`, and outlet to `store`.

> When a constant module such as `0` receives an event, it will send its value.



- When we receive an event to the `read` inlet, we want to (a) read the current value from the `memory`, (b) increment it, (c) write the value back to the `memory` and (d) output the original value. Easy!

- Creates another `store`, a `load` (from Built-ins > Memory), a `data order` from Built-ins > Data Process, and an `increment` from Built-ins > Maths. Wire them up as follows, and we'll discuss each in turn:



- The `load` module, when it receives an event, outputs the current value from the downstream `memory` module.
- `order` sends this value to its outlets in left-to-right order. You can rely on module positioning to dictate the order events are received, but often the `order` module makes this clearer.
- `increment` accepts a numeric value (or array) and returns the value + 1.
- Try the macro. Connect the `reset` and `read` inlets to `signal`s and the outlet to `print` and see how the output value is increased each time `read` gets an event.

Transactions

Often you make wish to make multiple changes to the graph, but without committing these changes until all are finished. The transaction button on the graph toolbar enables you add, delete, connect and disconnect modules without these changes being applied until the button is pressed again.

> Only graph-level changes are transacted. Changes to specific modules, such as a sequencer row, are not transacted.

The debugger

Debugging can be enabled with the button the graph toolbar. Any macros beginning with

`debug` or `assert` will only process when the debugger is on. These means that you can leave debug checks in your patch during development, and easily enable or disable them.

Building your own sequencer

Rather than relying on the built-in sequencers, Architect comes with all the components needed to built custom sequencers of your own designs. In this tutorial, we'll build a simple step sequencer.

- Create a `numeric table` and `boolean table` on the panel. Arrange them as you see fit.

> Any tables with the same parent module and who have the same View Group parameter will zoom and scroll together. If you lock the panel and scroll either table by dragging the ruler, you will see this in action.

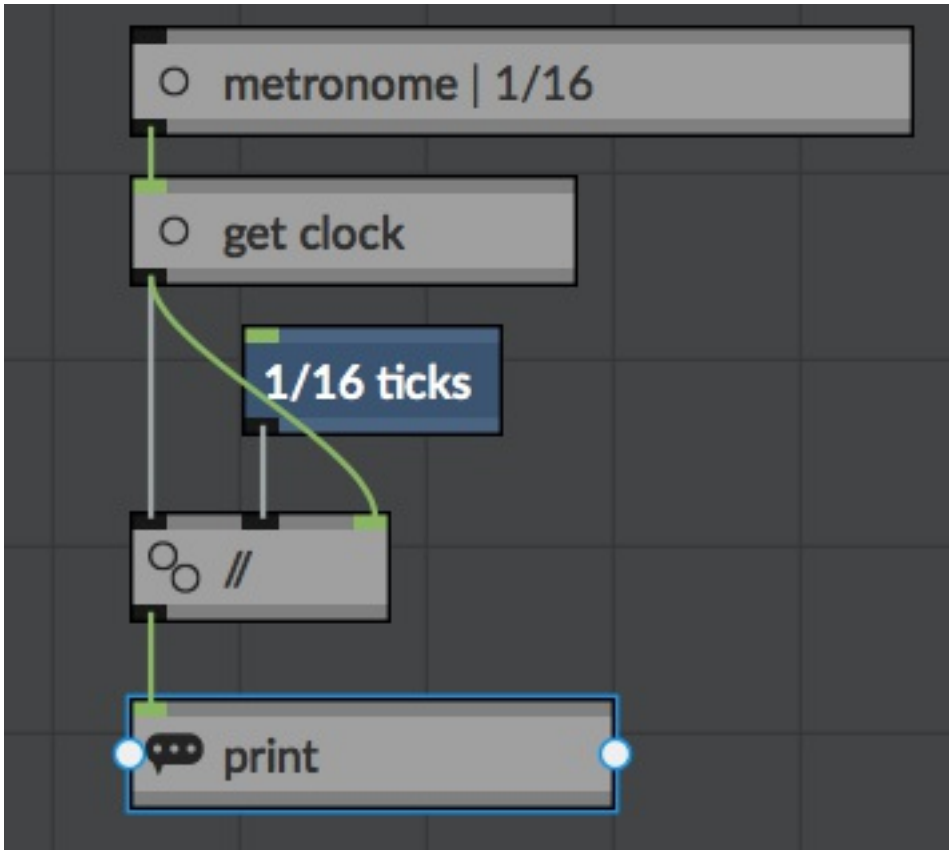
- For the `numeric table`, set the type to Cell, the range minimum to 48, maximum to 70, and leave the interval as 1.

- For both tables, turn off the Setup Event property. We only want these tables to output data when they are read from, not at setup time.

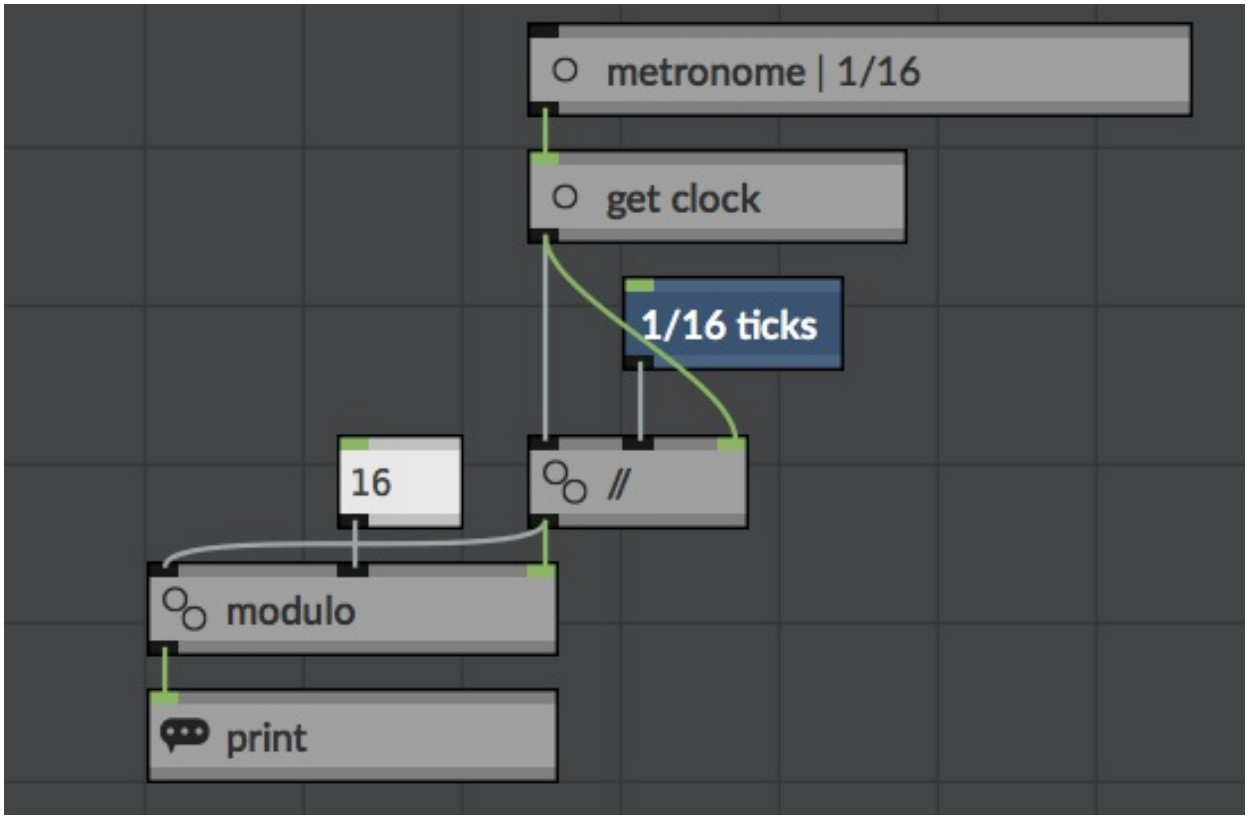
- Next we need to build a clocking source. Add a `metronome` from Built-in > Data Source, and set the interval to `1/16`. When the transport is playing, the metronome will output a signal object on every 1/16th of a bar.

- Now we work out what step should be played. A `get clock` (from Built-in > Arrangement) returns the current clock tick as an integer value.

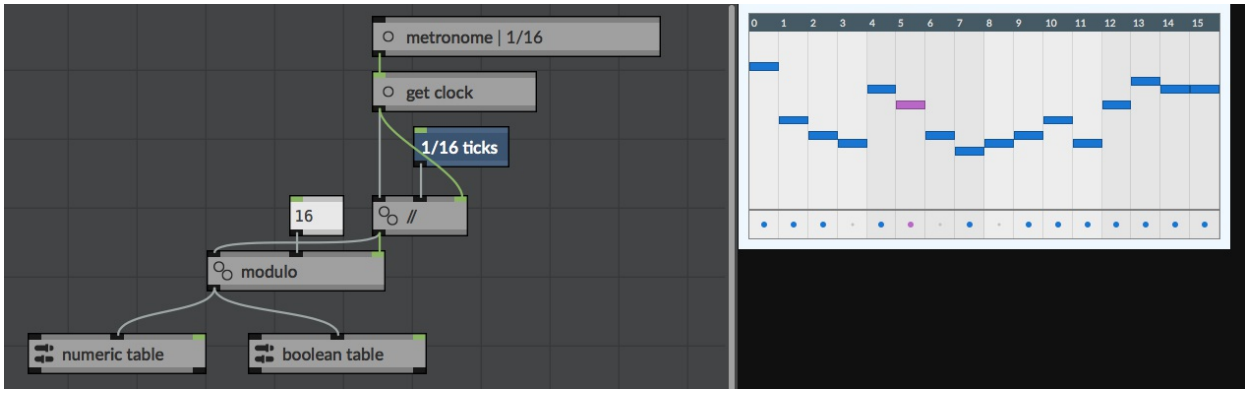
- Unless we want a note to be playing every 960 PPQ, this raw clock must be divided down to a step value. Add a `1/16 ticks` module from Built-in > Constant. This module outputs the number of ticks in a 16th note. Connect up, as illustrated below, a `floor divide` (Built-in > Maths) module. A `floor divide` divides two numbers, but ignores any fractional remainder. If you wire this into a `print` module, you should see the count of 16th notes being played.



- Our table only has 16 steps though, so (removing the `print` module first) wire up a `modulo` (Built-in > Maths) and a `data` with a value of 16. `modulo` returns the remainder when the first argument is divided by the second. In this case, it will always returns a value in the range of 0 to 15, which perfectly matches our table indices.



- Connect the `modulo` output to both table's highlighted step inlets. This will enable us to visually see the steps advancing. Start your transport and see for yourself.

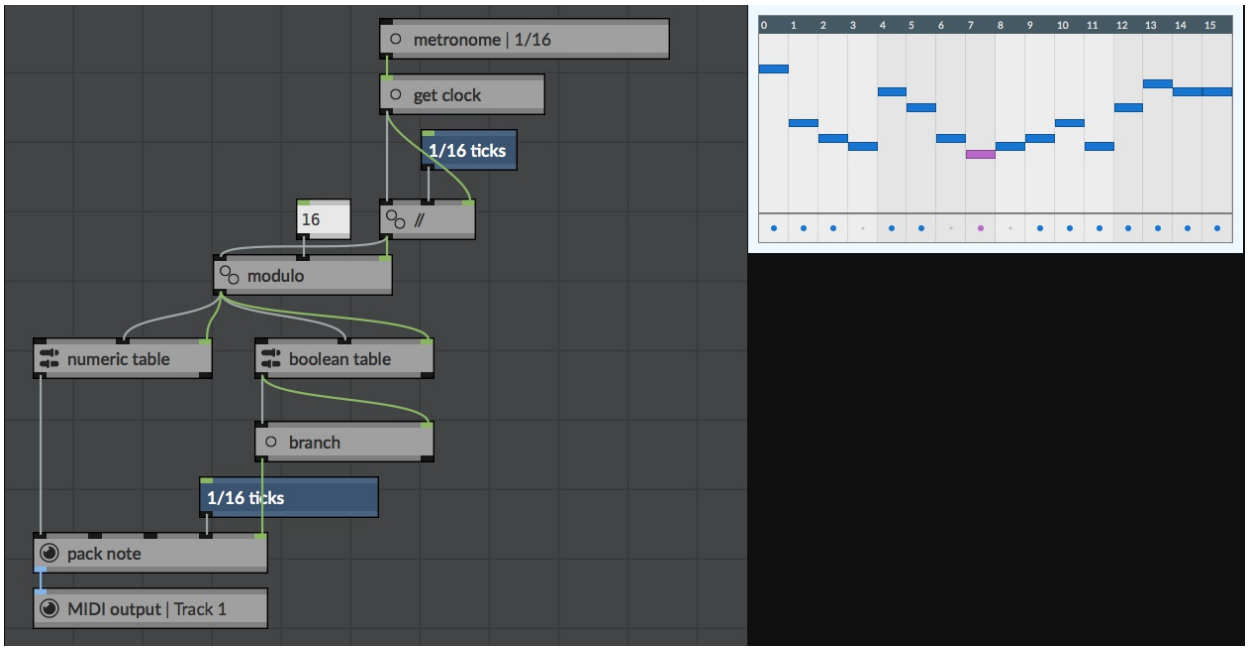


- We need to generate some MIDI notes, so create a `pack note` (Built-in > MIDI Source.), and connect this to a `MIDI output.`

- Each note will be 1/16th of a bar long, so create another `1/16 ticks` module and connect to the `pack note` duration inlet.

- The `numeric table` will provide the note keys. The `boolean table` will decide if a note plays or not, using a `data branch` module (Built-in > Data Process). `branch` modules have two inlets. If the control inlet receives a true value, all succeeding events will be routed to the left outlet. If the control inlet receives a false value, all succeeding events will be routed to the right outlet. In the finished sequencer below, this means that each false

cell in the `boolean table` is routed to nowhere, whilst true cells call the `pack note.`



Lua scripting

The `Lua script` module (Built-in > Script) allows you to run a Lua script within Architect. For a complete guide to programming in Lua, I recommend the book "Programming in Lua". Architect currently uses Lua 5.3.

Lua API

`print(x)` Writes x to the Lua output window or console.

`arc.VERSION` returns the version of the Architect application.

`arc.transport`

This contains methods for interacting with the main transport.

`arc.transport.TICKS & arc.transport.BARS` are constants for the `transport.isAtSync` function.

`arc.transport.isPlaying()` returns true if the transport is currently playing.

`arc.transport.isAtSync(type, nth)` returns true if we are currently exactly on a nth division of the given type.

```
-- returns true on every third bar.
arc.transport.isAtSync(lua.arc.transport.BARS, 3)
```

`arc.transport.getBeatTicks(numerator, denominator)` returns the number of ticks in the given number of beats.

```
-- returns the number of beats in 7 quarter notes
arc.transport.getBeatTicks(7, 4)
```

`arc.transport.getClock()` returns the current clock position in ticks.

`arc.transport.getTempo()` returns the current temporary

`arc.transport.getTimeSignature()` returns two integers containing the current time signature.

`arc.transport.getMetreChangeClock()` returns the historic clock tick of the most recent time-signature or tempo change.

`arc.module`

This contains methods for interacting with the script's housing module.

`arc.module.getName()` returns the name of the Modules

`arc.module.inlets` is a 1-based collections of inlet objects

`arc.module.outlets` is a 1-based collections of inlet objects

`arc.module.load()` is called after the script is first compiled, but before it is handy to the graph. This is the place to do CPU intensive activites such as generating look-up tables. You may implement this function in your script body.

`arc.module.reset()` is called when the graph is reset. You may implement this function in your script body.

`arc.module.setup()` is called when the module is first setup on the graph. You may implement this function in your script body.

`arc.module.tick()` is called one per tick during the event source phase. You may implement

this function in your script body.

`arc.module.receive(inlet, object)` is called when the script receives an event. `inlet` is a integer representing the inlet number, and `object` a Lua object representing the event data. You may implement this function in your script body.

`arc.inlet`

`arc.inlet:getType()` return "inlet"

`arc.inlet:getName()` return the name of the inlet.

`arc.inlet:getInletType()` returns the type of the inlet, either "data" or "MIDI".

`arc.outlet`

`arc.outlet:getType()` return "outlet"

`arc.outlet:getName()` return the name of the inlet.

`arc.outlet:getOutletType()` returns the type of the inlet, either "data" or "MIDI".

`arc.outlet:isConnected()` return true if the outlet has anything connected to it.

`arc.outlet:send(object)` sends `object` to all modules connect to this inlet.

`arc.array`

`arc.array.isTypeOf(x)` returns true if x is an arc.array.

`arc.array.new(values)` creates a new array object populated with values.

`arc.array:getType()` returns "array".

`arc.array:clone()` returns a new copy of this array.

`arc.array:length()` returns the number of elements in this array.

`arc.array:pack()` returns a lua table containing this array's elements.

`arc.array:get(nth)` returns the nth element. Arrays indices are 0-based. Braces notation, eg `arr[3]`, is also supported.

`arc.array:set(nth, val)` set the nth element to val. Array indices are 0-based. Braces notation, eg `arr[3] = 23`, is also supported.

`arc.array:insert(val)` inserts an new element on the end of the array.

`arc.array:insert(position, val)` inserts an new element at position.

`arc.array:remove(nth)` removes the nth element.

`arc.array:clear()` removes all elements.

Arrays can also be iterated using `pairs`

```
local arr = arc.array.new(2, 3, 5)
for k, v in pairs(arr) do print(k, v) end
```

`arc.tuple`

`arc.tuple.isTypeOf(x)` returns true if x is an arc.tuple.

`arc.tuple.new(values)` creates a new tuple object populated with values.

`arc.tuple:getType()` returns "tuple".

`arc.tuple:clone()` returns a new copy of this tuple.

`arc.tuple:length()` returns the number of elements in this tuple.

`arc.tuple:pack()` returns a lua table containing this tuple's elements.

`arc.tuple:get(nth)` returns the nth element. Tuple indices are 0-based. Braces notation, eg `tuple[3]`, is also supported.

`arc.tuple:set(nth, val)` set the nth element to val. Tuple indices are 0-based. Braces notation, eg `arr[3] = 23`, is also supported.

Tuples can also be iterated using `pairs`

```
local t = arc.tuple.new("noteon", 60, 5, 10)
for k, v in pairs(t) do print(k, v) end
```

`arc.map`

`arc.map.isTypeOf(x)` returns true if x is an arc.map.

`arc.map.new()` creates a new empty map object.

`arc.map:getType()` returns "map".

`arc.map:clone()` returns a new copy of this map.

`arc.map:length()` returns the number of elements in this tuple.

`arc.map:pack()` returns a lua table containing this map's elements.

`arc.map:get(key)` returns the element with this key.Braces notation, eg `map[3]`, is also supported.

`arc.map:set(key, val)` set the key element to val. Braces notation, eg `map[3] = 23`, is also supported.

`arc.map:member(key)` returns true if the map has a value with the given key.

`arc.map:delete(key)` removes the element with the given key.

`arc.map:clear()` removes all elements.

Maps can also be iterated using `pairs`

```
local m = arc.map.new()
m:set(2, "alfa")
m:set(3, "bravo")
m:set(5, "charlie")
for k, v in pairs(m) do print(k, v) end
```

`arc.signal`

`arc.signal.isTypeOf(x)` returns true if x is an arc.signal.

`arc.signal.new()` creates a new signal object

`arc.signal:getType()` returns "signal".

`arc.signal:clone()` returns a new copy of this object.

`arc.undefined`

`arc.undefined.isTypeOf(x)` returns true if x is an arc.undefined.

`arc.undefined.new()` creates a new undefined object

`arc.undefined:getType()` returns "undefined".

`arc.undefined:clone()` returns a new copy of this object.

`arc.noteon`

`arc.noteon.isTypeOf(x)` returns true if x is an arc.noteon.

`arc.noteon.new(key = 60, velocity = 64, channel = 1, isUncoupled = false)` creates a new noteon object

`arc.noteon:getType()` returns "noteon".

`arc.noteon:clone()` returns a new copy of this object.

`arc.noteon:getKey()` returns the key

`arc.noteon:setKey(val)` sets the key

`arc.noteon:getVelocity()` returns the velocity

`arc.noteon:setVelocity(val)` sets the velocity

`arc.noteon:getChannel()` returns the channel

`arc.noteon:setChannel(val)` sets the channel

`arc.noteon:isUncoupled()` returns true if the note is uncoupled

`arc.noteon:setUncoupled(val)` sets the uncoupled status

`arc.noteoff`

`arc.noteoff.isTypeOf(x)` returns true if x is an arc.noteff.

`arc.noteff.new(key = 60, velocity = 64, channel = 1, isUncoupled = false)` creates a new noteoff object

`arc.noteoff:getType()` returns "noteoff".

`arc.noteoff:clone()` returns a new copy of this object.

`arc.noteoff:getKey()` returns the key

`arc.noteoff:setKey(val)` sets the key

`arc.noteoff:getVelocity()` returns the velocity

`arc.noteoff:setVelocity(val)` sets the velocity

`arc.noteoff:getChannel()` returns the channel

`arc.noteoff:setChannel(val)` sets the channel

`arc.noteoff:isUncoupled()` returns true if the note is uncoupled

`arc.noteoff:setUncoupled(val)` sets the uncoupled status

`arc.program`

`arc.program.isTypeOf(x)` returns true if x is an arc.program.

`arc.program.new(program = 0, channel = 1)` creates a new program object

`arc.program:getType()` returns "program".

`arc.program:clone()` returns a new copy of this object.

`arc.program:getProgram()` returns the program

`arc.program:setProgram(val)` sets the program

`arc.program:getChannel()` returns the channel

`arc.program:setChannel(val)` sets the channel

`arc.channelpressure`

`arc.channelpressure.isTypeOf(x)` returns true if x is an arc.channelpressure.

`arc.channelpressure.new(value = 64, channel = 1)` creates a new channelpressure object

`arc.channelpressure:getType()` returns "channelpressure".

`arc.channelpressure:clone()` returns a new copy of this object.

`arc.channelpressure:getValue()` returns the value

`arc.channelpressure:setValue(val)` sets the value

`arc.channelpressure:getChannel()` returns the channel

`arc.channelpressure:setChannel(val)` sets the channel

`arc.pitchbend`

`arc.pitchbend.isTypeOf(x)` returns true if x is an arc.pitchbend.

`arc.pitchbend.new(value = 8192, channel = 1)` creates a new pitchbend object

`arc.pitchbend:getType()` returns "pitchbend".

`arc.pitchbend:clone()` returns a new copy of this object.

`arc.pitchbend:getValue()` returns the value

`arc.pitchbend:setValue(val)` sets the value

`arc.pitchbend:getChannel()` returns the channel

`arc.pitchbend:setChannel(val)` sets the channel

`arc.controller`

`arc.controller.isTypeOf(x)` returns true if x is an arc.controller.

`arc.controller.new(controller = 0, value = 64, channel = 1)` creates a new controller object

`arc.controller:getType()` returns "controller".

`arc.controller:clone()` returns a new copy of this object.

`arc.controller:getController()` returns the controller

`arc.controller:setController(val)` gets the controller

`arc.controller:getValue()` returns the value

`arc.controller:setValue(val)` sets the value

`arc.controller:getChannel()` returns the channel

`arc.controller:setChannel(val)` sets the channel

`arc.polypressure`

`arc.polypressure.isTypeOf(x)` returns true if x is an arc.polypressure.

`arc.polypressure.new(key = 60, value = 64, channel = 1)` creates a new polypressure object

`arc.polypressure:getType()` returns "polypressure".

`arc.polypressure:clone()` returns a new copy of this object.

`arc.polypressure:getKey()` returns the key

`arc.polypressure:setKey(val)` sets the key

`arc.polypressure:getValue()` returns the value

`arc.polypressure:setValue(val)` sets the value

`arc.polypressure:getChannel()` returns the channel

`arc.polypressure:setChannel(val)` sets the channel

`arc.start`

`arc.start.isTypeOf(x)` returns true if x is an arc.start.

`arc.start.new()` creates a new start object

`arc.start:getType()` returns "start".

`arc.start:clone()` returns a new copy of this object.

`arc.stop`

`arc.stop.isTypeOf(x)` returns true if x is an arc.stop.

`arc.stop.new()` creates a new stop object

`arc.stop:getType()` returns "stop".

`arc.stop:clone()` returns a new copy of this object.

`arc.continue`

`arc.continue.isTypeOf(x)` returns true if x is an arc.continue.

`arc.continue.new()` creates a new continue object

`arc.continue:getType()` returns "continue".

`arc.continue:clone()` returns a new copy of this object.

`arc.clock`

`arc.clock.isTypeOf(x)` returns true if x is an arc.clock.

`arc.clock.new(position = 0)` creates a new continue object

`arc.clock:getType()` returns "clock".

`arc.clock:clone()` returns a new copy of this object.

`arc.songposition`

`arc.songposition.isTypeOf(x)` returns true if x is an arc.songposition.

`arc.songposition.new()` creates a new songposition object

`arc.songposition:getType()` returns "songposition".

`arc.songposition:clone()` returns a new copy of this object.

`arc.songposition:getPosition()` returns the position

`arc.songposition:setPosition(val)` sets the position

`arc.sysex`

`arc.sysex.isTypeOf(x)` returns true if x is an arc.sysex.

`arc.sysex.new(values)` creates a new sysex object

`arc.sysex:getType()` returns "sysex".

`arc.sysex:clone()` returns a new copy of this object.

`arc.sysex:length()` returns length of this object.

`arc.sysex:get(nth)` returns the nth element. Arrays indices are 0-based. Braces notation, eg `sy[3]`, is also supported.

`arc.sysex:set(nth, val)` set the nth element to val. Array indices are 0-based. Braces notation, eg `sy[3] = 23`, is also supported.

`arc.sysex:resize(length)` resizes the sysex array to the given length.

Sysex objects can also be iterated using `pairs`

```
local sy = arc.sysex.new(0x20, 0x32, 0x11, 0x44)
for k, v in pairs(sy) do print(k, v) end
```

A Lua example

Create a `Lua script` module and add 1 MIDI inlet and 1 MIDI outlet.

This script passes through MIDI objects, but noteon and noteoffs have an additional harmony added.

```
function arc.module.receive(inlet, object)
  -- pass the original object through
```

```
    arc.module.outlets[1]:send(object)

    -- if it is a noteon or noteoff, send a perfect-5th harmony
    if (arc.noteon.isTypeOf(object) or arc.noteoff.isTypeOf(object)) then
      object:setKey(object:getKey() + 7)
      arc.module.outlets[1]:send(object)
    end
  end
```

Writing to Step Sequencer Patterns

Step sequencer patterns consist of a number of individual rows, such as pitch, velocity, or speed. To write to a row programatically using graph modules, the inlet for that row must first be made visible by toggling the "show" icon in the Rows property list of the current sequencer. Data can then be sent to the inlet in one of the following formats:

- `number: value` will write the given number to the first step of that row.
- `[number: value]` will write the array of numbers to each step in turn. For example, `[2, 3, 5]` will write 2 to the first step, 3 to the second step, and 5 to the third step.
- `(number: step, number: value)` will write the given value to the specified step. For example `(12, 60)` will write value 60 to step 12.
- `(number: step, [number: value])` will write the array of values to the steps, starting from the given step. For example, `(10, [60, 64, 68])` will write the value 60 to step 10, 64 to step 11, and 68 to step 12.
- `[tuple(number: step, number: value)]` will write the array of tuples, using the same `(number: step, number: value)` format as above. For example, `[(12, 60), (13, 72), (20, 58)]` will write value 60 to step 12, value 72 to step 13, and value 58 to step 20.

These same data formats are used when writing to table modules.

Where now?

Thank you for taking the time to read this quick-start guide. As I'm sure you'll now appreciate, Architect is a large software application capable of many wonderful things. But having so many options can be overwhelming, so here are a few pointers:

- A `MIDI to tuple` (Built-in > MIDI Process) module connected to a `print` is very handing for debugging which MIDI messages are being transmitted.

- Most things in the sequencer modules can be modulated, either by the inlets or by remote mappings. Try combining two sequencers, one altering the other, to turn two simple patterns into an ever changing one. The aux outputs from the sequencers are ideal for

this.

- There are lots of random sources in Built-in > Data Source, ideal for generative pieces.