MidiFire Manual

Version 2.0, October 2019

MidiFire - powerful, extendable modular MIDI processing environment for iOS and macOS.

Quick Start

Want to get going fast? Here are the basics:

- Press the '+' button on top left to expose the scrollable block menu.
- Touch MIDI inputs, outputs and modules to drop them on to the canvas.
- Arrange the blocks however you like by touching and dragging them. Use two-fingers to scroll/pinch zoom to adjust and pan the canvas.
- Connect up the blocks by dragging a line between the arrowed connection points.
- Configure each module's parameters by touching the 'cog' icon on each block.

Tip - direct links to: Scenes, Module Summary, Stream Byter, Getting Help

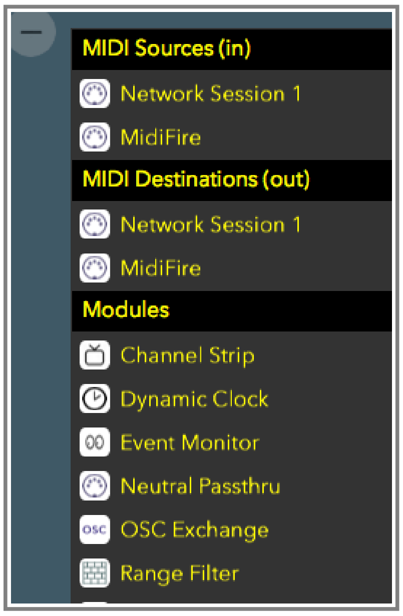
1. Organising Blocks

top

In MidiFire, a 'block' is a rectangular object on the canvas that represents a MIDI source (input to MidiFire), MIDI destination (output from MidiFire) or a processing module. Blocks are added to the canvas via the 'Block Menu'.

The Block Menu (+)

The Block Menu is expanded by touching/clicking on the circular '+' button at the top left of the canvas.



Block Menu

You add blocks to the canvas by touching their entry in the Block Menu. Scroll the Block Menu to access all the available block types. Each newly added block will be added to the canvas to the right of the Block Menu and will briefly flash in a coloured fashion to alert you to its existence.

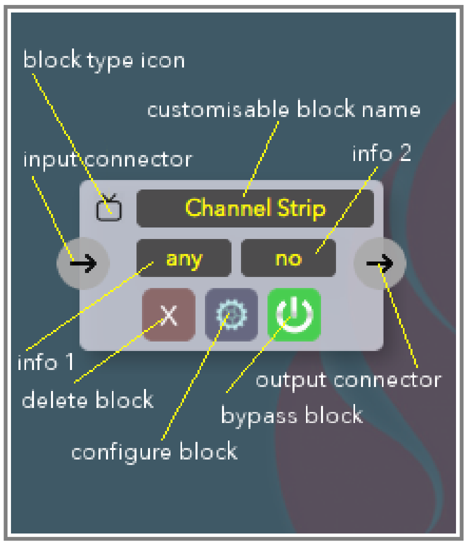
Tip - you can touch anywhere on the canvas or the top left '-' to collapse the Block Menu.

Arranging Blocks

Once you have 'dropped' your blocks, you can now arrange them however you like. Simply touch hold (anywhere in the block except on the block's buttons/arrow connectors) and drag the block into position.

Managing Blocks

Each block has a number of visible widgets used for managing the block:



Block

Connecting Blocks

In order for MIDI data to move from block to block, they need to be connected. Typically you would connect MIDI source blocks into modules and then on to MIDI output blocks to create a flow of MIDI in and out of the MidiFire application.

There are two ways to interconnect blocks:

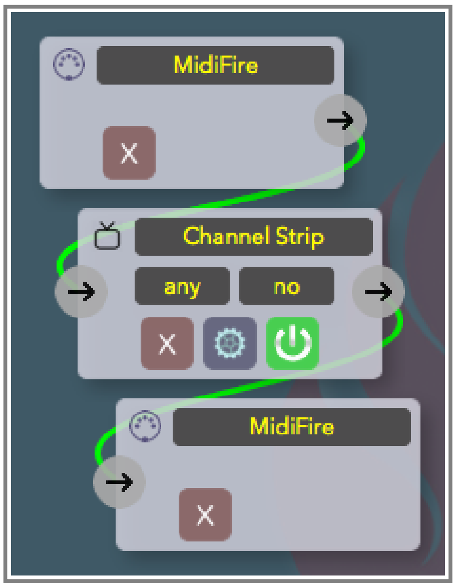
- Simple drag/connect

  To connect one block to another, drag your finger/mouse from the input or output connector of the block to a corresponding output or input connector on another block.

- Multiple touch/connect

  This is the 'MidiBridge' method, where you touch any input or output connector to 'select' it (it will start flashing yellow). All other blocks' corresponding output or input connectors will change to a cyan colour to indicate they can be connected to the selected block. Simply touch each cyan connector in turn to connect to the selected. Finally, touch the flashing connector again to deselect.

When blocks are connected together, a curved green line will be visible between their connectors:

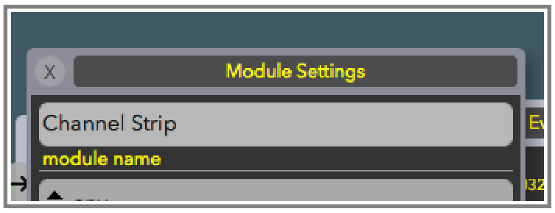


Connected Blocks

Tip - to remove a connection just 'remake' the connection on already connected blocks.

Configuring Blocks

Modules can be configured by pressing the configure block 'cog' button on the block. This will bring up a moveable panel in which you can set the name of the block to suit. Each module has a different set of configurable parameters and these are described in the relevant module sections further on in this manual.

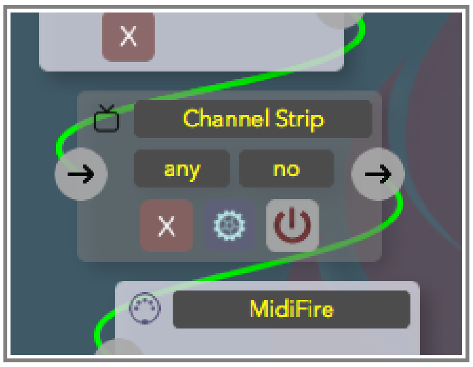


Set Module Name

The Module Settings panel is dismissed using the 'X' button at top left of the panel or by touching elsewhere on the canvas to dismiss.

Bypassing Blocks

The bypass 'power' button bypasses a module. This means all MIDI data passing through is not altered. A bypassed module is indicated by a red coloured bypass button and the block itself becomes more transparent:



Bypassed Block

To re-activate a bypassed module, just touch the bypass button again.

Removing Blocks

To remove a block from the canvas use the 'X' remove button. You will be prompted to confirm removal before the block is removed. All connections to the removed block are cleared also.

Event Visualisation

As MIDI data is passed through the blocks, they will flash briefly or in some cases take on a solid background colour. The meanings of each of these are as follows:

- Green flash - MIDI event was accepted and processed by the module
- Orange flash - MIDI event was received but blocked by the module
- Red flash - an error was detected in the module
- Cyan solid - (input ports only), the port is currently marked as held over. See the section on holdover mode.
- Yellow solid - (dynamic clock and output ports only), the module is sending a clock signal.

You can turn off the flashing behaviour via the 'Setup' panel covered later.

2. Navigating the Canvas

top

The canvas is the area of the screen in which you place blocks but it has some more features:

Pan, Zoom and Scroll

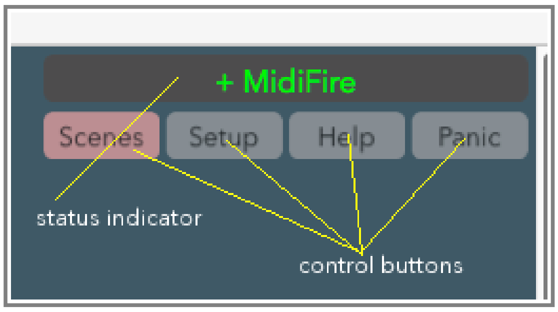
The canvas can be panned/scrolled by dragging your two fingers on a touch device or with a mouse or trackpad on a desktop computer. Use two fingers anywhere on the canvas background to pan up/down and left/right. If using a scroll mouse, the canvas can be scrolled up/down with a scroll wheel, or left/right if your mouse scroll wheel allows that. A trackpad can also be used to pan the canvas.

To zoom the canvas in/out you can use a two finger pinch-zoom gesture on a touch device or trackpad. For desktop computers without a trackpad, you can use the two zoom in/out buttons at the bottom right or the menu zoom command.

To clear the canvas, use the button at bottom right (blank square icon). Use the zoom to fit button next to it (square+arrows icon) to zoom the canvas to fit all modules on the screen automatically.

The Button Bar

At the top right of the canvas is a status indicator with a set of button controls beneath. (Note: Scenes and Setup are covered in their own separate sections below)



Button Bar

Status Indicator/Activity Log

The status indicator shows the last action performed by the user in the application. The colour of the text in the label will be:

- Green - Positive things
- Yellow - Less positive things
- Red - Errors

Touching the status indicator will open up the Activity Log panel which shows all actions performed in the app since it was started along with a timestamp. When blocks are added to the canvas, the name of the block is shown prefixed by '+' in green. Removals are shown prefixed as '-' and in yellow.

The log is currently just a read-only list for informational purposes.

Dismiss the Activity Log using the 'X' or touching the canvas background.

Help

Pressing the 'Help' button will bring up this manual.

Panic

Sometimes with MIDI things can go haywire. This is where the 'Panic' button comes in. When pressed, it will issue a standard set of MIDI panic messages (all notes off on all channels) to every MIDI port in the system (whether present on the canvas or not).

When you press the 'Panic' button, MidiFire will suspend all event routing so as not to exascerbate the problem. This is shown by making the whole canvas visibly faded and unable to receive user interaction. The 'Panic' button will also turn a yellow colour.

When you have resolved the problem, press the yellowed 'Panic' button once again to resume event processing and user interaction.

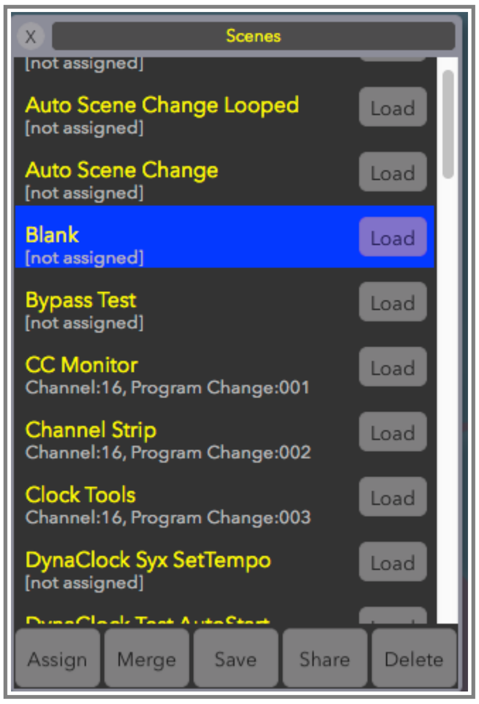
Dismissing Panels/Selections

The canvas can also be used to instantly dismiss all open panels or cancel any connection selections. Just tap anywhere on the canvas background.

3. Managing Scenes

top

Pressing the 'Scenes' button on the Button Bar will open up the Scenes management panel. A Scene is a snapshot of a canvas that can be loaded, merged, shared or deleted. Internally it is a text file in 'property list' XML format.



[not assigned]

Auto Scene Change Looped
[not assigned]

Auto Scene Change
[not assigned]

Blank
[not assigned]

Bypass Test
[not assigned]

CC Monitor
Channel:16, Program Change:001

Channel Strip
Channel:16, Program Change:002

Clock Tools
Channel:16, Program Change:003

DynaClock Syx SetTempo
[not assigned]

Scenes Panel

Each scene in your library is presented as a scrollable list showing the name of the scene and any MIDI program change assignment for that scene.

Saving your Work

As you work on your canvas and block configuration you can save a snapshot at any time to a name of your choice that you can use later:

- Press 'Scenes' button to open panel
- Press 'Save' button
- Enter a name (or modify) for the scene
- Confirm (optionally) if scene would be overwritten to update

Dirty Scenes

As you make changes to the canvas following a save (or load) the Scenes button will assume a reddish hue to remind you that unsaved changes have been made.

Following a save or load action the Scenes button will revert to its normal colour.

Loading and Merging

Scenes can be re-used in two possible ways:

- Load

  Next to each scene name is a separate 'Load' button. Pressing this will load in that scene **immediately and entirely**; ie. it will clear the canvas first and replace with the contents of that scene. Use the individual Load buttons to switch between your scenes quickly.

  Stored with each scene is the current zoom scale when saved. On 'Load' this zoom scale is honoured.

- Merge

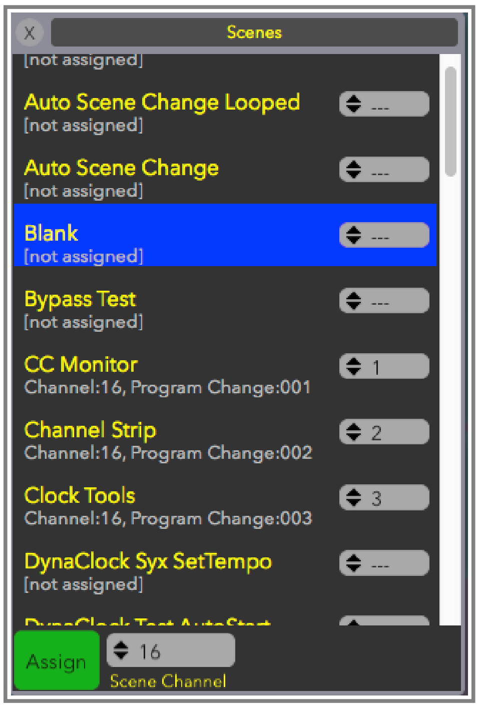
  Merging **adds** the contents of a scene to the canvas **without** clearing it first. Before you merge a scene, you need to select it in the list first, by touching it - it will highlight in a blue colour. Once selected, pressing the 'Merge' button will add it into the canvas.

MidiFire will intelligently place your merged scene into the current canvas so as not to cover any existing blocks on the canvas. The scene's zoom scale is not honoured during a merge.

Tip - you can save oft-used snippets of work into separate scenes and then merge them as needed.

Assigning Remote MIDI Control

Scenes can be loaded (not merged) remotely via MIDI Program Change message. Pressing the 'Assign' button on the Scenes Panel will switch to Program Change assign mode. This is indicated with the 'Assign' button taking on a green colour and the individual 'Load' buttons being replaced by program change number selectors:



[not assigned]

Auto Scene Change Looped
[not assigned] ---

Auto Scene Change
[not assigned] ---

Blank
[not assigned] ---

Bypass Test
[not assigned] ---

CC Monitor
Channel:16, Program Change:001 1

Channel Strip
Channel:16, Program Change:002 2

Clock Tools
Channel:16, Program Change:003 3

DynaClock Syx SetTempo
[not assigned] ---

Assigning Scenes

The first thing you will need to do in order to activate remote MIDI control is to set the 'Scene Channel' value using the dropdown selector next to the (now green) Assign button. By default, Scene Channel is OFF which means that remote control is disabled.

Scene change program change messages can be set to be recognised from a specific channel only (recommended) or (if you really must) on all (OMNI) channels. To prepare for remote control, select a specific channel or 'OMNI' for Scene Channel.

Once the Scene Channel is configured, you then assign a program change number to each of the scenes that you wish to be able to load remotely by setting the program change number using the dropdown in each scene cell.

Note that each scene must have a unique program change number. If you select a number that has already been assigned to a different scene, then the new scene will take on the requested assignment and the previous assigned scene will be marked as unassigned.

Once you have finished assigning program change numbers to scenes, press the 'Assign' button to leave assign mode. At this stage validly assigned program change messages that are received by MidiFire will trigger a scene change, as if the scene's individual 'Load' button had been pressed.

Tip - MidiFire will honour assigned program change messages appearing on any MIDI source whether it is currently displayed on the canvas or not.

Sharing

You can share your saved scenes with other devices/computers or other MidiFire users using the 'Share' button.

On iOS, the share action invokes the device's inbuilt sharing options that will vary depending on the device. This may include Airdrop, email applications or sharing applications like Dropbox.

Also, on iOS, iTunes sharing is enabled, so you can drag/drop scenes in/out of your device via the Apps > File Sharing area in iTunes. Please note that the app is not informed when you modify the directory remotely. If your newly inserted scenes are not shoing up, close and re-open the Scenes panel to refresh the list.

On macOS pressing the 'Share' button will open the raw library files in the Finder with the currently selected scene file highlighted for you.

To import scenes on an iOS device, MidiFire has registered '.mfr' files as belonging to it and you can download and install these files into MidiFire using the usual iOS mechanisms. Imported scenes will show up in your Scenes panel; they are not loaded automatically.

On macOS to import a scene externally you can use the 'Share' button to open the directory in the Finder and copy your scene into the directory.

Tip - because Scene files are plain text files, they can be shared via websites and imported using your web browser.

Cleaning Up

Finally, if you wish to delete a scene from your library, select it and press the 'Delete' button. You will be asked to confirm deletion. Please note that when you delete a scene the file is removed from your device/computer.

4. Setup

top

Use the 'Setup' panel to configure application wide settings. These are not stored in scenes.

Scenes Club

Because MidiFire is eminently extendable, we have a whole set of pre-made scenes available directly which may be of interest to you.

Pressing the 'Scenes Club' button will open a new panel showing a list of available scenes, description and author. To download any scene into your library, press the download icon to the right of each scene.

The Scenes Club is hosted on the Internet, so to access and download scenes you do need to be online.

Tip - if you have any scenes which you feel would be useful to others, email them to us and we can distribute them (with attribution) via the Scenes Club.

Virtual MIDI Ports

By default, MidiFire creates one pair of virtual MIDI ports that can be used to send/receive MIDI data to other apps running on your device. For some, this is not enough, so if you would like more separate virtual ports to assist in your workflow then you can have these created for you (up to 64 virtual ports).

When MidiFire has multiple virtual port pairs configured, the additional ports are numbered individually from 1 upwards. so that you can distinguish them easily. You still get to keep the

'MidiFire' default/main port pair which will always be present.

To change the number of virtual port pairs advertisied by MidiFire just make a selection for the 'midifire virtual ports' dropdown.

Event Visualisation

As MIDI data is processed by MidiFire, the ports flash briefly. If you find this annoying or want to improve performance when in the foreground (nothing flashes in the background) turn the flashing off by setting 'event visualisation' to 'no'.

Idle Timeout (iOS)

On iOS to prevent battery drain, MidiFire will automatically suspend itself after a period of inactivity.

You can adjust this timeout from the default 15 minutes to something larger, or if you are brave to 'never' which will disable auto-suspension entirely.

Note that MidiFire will never suspend if the device is powered.

Ignore Active Sense

By default, MidiFire will ignore all Active Sensing MIDI messages on input ports as they are rarely desired in non-hardware environments. If you need to Active Sensing messages to be processed, switch this option to 'no'

Holdover Processing

'Holdover' is a MidiFire feature that keeps track of held notes (or use of the hold pedal) on each external MIDI input and 'freezes' the MidiFire configuration for that specific input while the hold is in effect.

This means that as you make changes to MidiFire on the canvas, (or load a new scene) events from the held MIDI input are not affected by these changes until you let go of all notes and release the hold pedal.

Therefore, you won't get stuck notes or lost pitchbend messages and the like when a new scene is loaded or changes are made to the canvas that might affect these.

Although holdover has been designed to be very efficient, there could be some slight performance loss especially with complex scenes. For those who don't need the holdover processing and wish to save some extra CPU cycles, setting this to 'no' will disable it entirely.

Note, changes made to Stream Byter rules or AU Plugins are **not** frozen, so any changes made to rules will take effect whether a hold is active or not.

Tip - each MIDI input is frozen separately so you can use multiple external inputs (controllers, sequencers) and each will have its own frozen configuration while a hold is active.

Stern™ Method

This option is a specialised method for doing holdover:

1. The hold pedal is classed as 'continuous' which means that any value of the hold CC (64) other than zero means a hold and a zero means release.
2. When changes to the canvas are made while the hold pedal is depressed or notes are being held, new notes played will use the configuration of the current canvas. When all the originally held notes (and pedal) are released, the note off and pedal release MIDI events are sent via the frozen configuration.

Remote Control

You can control some on-screen actions of MidiFire via MIDI command. The actions (buttons, switches or dropdowns) that can be remote controlled are:

- Panic switch (application)
- Bypass switch of any module (scene)
- All 'Channel Strip' dropdowns, except note remap (scene)
- 'Note/Velocity Split' split point, lower channel and upper channel dropdowns (scene)
- 'Install Rules' button on Stream Byter (scene)
- 'Tap Tempo' button and 'Auto Start' dropdown on 'Dynamic Clock' (scene)
- Any writeable parameter of an AU plugin (scene)

Assignments of actions marked as 'scene' above are scene specific and thus saved and restored with a scene. Non scene actions, marked as 'application' above are application wide and saved with application preferences.

MIDI messages (on any specific channel) that can be utilised are:

- Continuous Controller (except bank select)

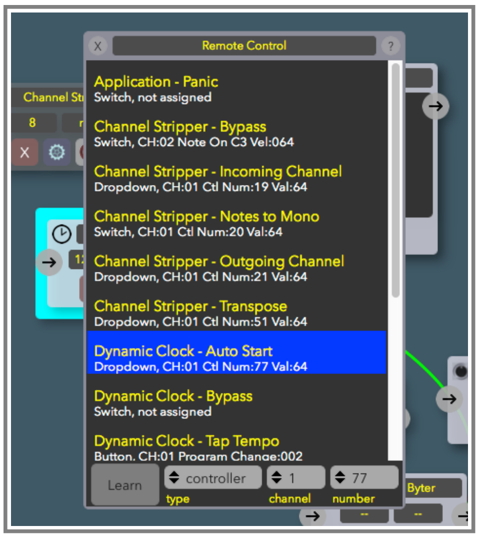
  can control buttons, switches and dropdowns
- Program Change

  can control buttons and switches
- Note On/Off

  can control buttons and switches

Tip - if you want to trigger controls with MIDI messages other than those listed above (eg. sysex) you can always use a Stream Byter to inject them using an SND +I rule

Remote Control messages can be assigned to actions or learnt via the 'Remote Control' panel found on the Setup panel.



Application - Panic
Switch, not assigned

Channel Stripper - Bypass
Switch, CH:02 Note On C3 Vel:064

Channel Stripper - Incoming Channel
Dropdown, CH:01 Ctl Num:19 Val:64

Channel Stripper - Notes to Mono
Switch, CH:01 Ctl Num:20 Val:64

Channel Stripper - Outgoing Channel
Dropdown, CH:01 Ctl Num:21 Val:64

Channel Stripper - Transpose
Dropdown, CH:01 Ctl Num:51 Val:64

Dynamic Clock - Auto Start
Dropdown, CH:01 Ctl Num:77 Val:64

Dynamic Clock - Bypass
Switch, not assigned

Dynamic Clock - Tap Tempo
Button. CH:01 Program Change:002

type: controller
channel: 1
number: 77

Remote Control Panel

To assign a MIDI message to an action, first select it in the list on the Remote Panel. Note, if the action you wish to assign belongs to a canvas block, then this block will be given a cyan 'ring' highlight so you can identify which module you are working with (handy when you have multiple modules with the same name).

You can either use the dropdowns at the bottom of the panel to assign the MIDI message to the selected action, or you can turn on 'Learn' using the button and MidiFire will assign the most recently received valid MIDI message to the action. Note that while you are 'Learning', normal remote control is disabled.

If you wish to remove an assignment, select '- none -' as the value for the 'type' dropdown at the bottom.

The following table explains how each remote MIDI message is interpreted for each type of action:

	Program Change	**Note On/Off**	**Controller**
Button	momentary push	momentary push on note on	momentary push if value >= 64
Switch	toggles switch	note on turns switch on, note off turns switch off	value 64-127 turns switch on, 0-63 turns switch off
Dropdown	- n/a -	- n/a -	value selects option in a scaled fashion. value 0 is first option, value 127 is last option

Tip - you may assign the same MIDI message to more than one action, so you could bypass a group of modules with the one remote message.

Bluetooth MIDI (some iOS devices)

From within MidiFire you can setup Bluetooth MIDI connections if your iOS device is recent enough to support it.

There are two methods of connectivity accessed by their respective buttons:

- Connect Device

  Use this button if you wish to connect your iOS device to a remote Bluetooth MIDI device such as another iOS device or a quicco or yamaha BT01.

- Host Service

  If you want to be able to accept incoming Bluetooth MIDI requests from remote devices then you use this button to start being a host.

If Bluetooth MIDI is not supported on your device, then these buttons will be missing. Obviously Bluetooth needs to be turned on in your device. If it's off and you press any of the Bluetooth buttons you will be reminded that you need to turn it on.

Tip - at the bottom of the Setup panel you will find the current version and build number of MidiFire that you have installed. Please pass on those details if you contact us for support.

CoreMIDI Networking (iOS)

MidiFire can be used to initiate CoreMIDI (rtpMIDI) WiFi connections from within the app. Pressing the CoreMIDI Networking button will open up a separate panel for making these connections:

- remote host/address

Enter the hostname or IP address of the remote device/computer that you wish to initiate the connection to. By default this is set to 'localhost' which creates a looped back connection where all events sent to the Network output port are reflected back into the Network input.

  The label for this field will also display the IP address of the device. You can use this address in another device if you wish to connect to this device from MidiFire on that other device.

- remote port number

  This is the IP port number that the remote is accepting rtpMIDI connections on. By default this is 5004, which is the standard port number for the first network connection. Generally this value does not need to be changed.

- auto connect

  Setting this option to 'yes' will instruct MidiFire to automatically initiate the connection when it starts up the first time.

- Connect/Disconnect

  Press this button to manually initiate a conection if not currently connected (button is grey and labelled 'Connect') or disconnect all connections (button is green and labelled 'Disconnect')'.

5. Modules

top

While you can do very sophisticated routing with just MIDI sources and destinations in MidiFire, the application comes into its own when you introduce modules into the mix. There is a variety of modules included ranging from simple/common, to more niche/purpose built modules to the most complex but powerful Stream Byter that lets you create your own customised modules. Here is a summary of the included modules:

- AU MIDI Plugin - Audio Unit MIDI FX plugin
- Channel Strip - commonly used MIDI channel management/transpose/monofy
- Note/Velocity Split - split an incoming signal over 2 MIDI channels
- Pressure Curve - remap note velocity or aftertouch
- Event Monitor - MIDI monitor to examine events at any canvas point
- Comment Block - for documenting your work
- Protocol Filter - block/allow events within a MIDI protocol range
- Dynamic Clock - intensely accurate and remotely controllable clock source
- Stream Byter - write your own processing modules
- Tracking Clamp - reduce aberrations from MIDI guitars
- Robotic Knob - generate complementary CC messages based on performance
- OSC Exchange - pipe OSC data across a MIDI interface

Each module is documented in detail below.

AU MIDI Plugin

modules

As of version 2.0, MidiFire supports hosting of Audio Unit MIDI FX plugins presented as integrated modules. Simply drop an AU MIDI Plugin module onto the canvas, select from your installed plugins and connect up to whatever you like just as a built-in module.

Uniquely, plugins can be hosted with zero latency. See further for details. on this.

Parameters:

- AU MIDI Plugin (default: -no selection-)

  Use this drop down list to choose from installed MIDI FX available. Once you select a plugin, its GUI will automatically open for you to configure it.

  Please note: only validated MIDI FX are supported and iOS 11 is required at a minimum.

- process with zero latency (default: depends)

  In other hosts, MIDI events are processed along with an audio engine which deals with small chunks of audio at a time. This results in live MIDI events being delayed variably anywhere up to 23ms (or greater) depending upon this chunk size (audio render buffer).

  As MidiFire is a MIDI only application, the option exists to process incoming live events independently from the render cycle and immediately. ie. zero latency processing.

  By default, most plugins will be configured not to operate at zero latency, so for these plugins you will need to switch on zero latency explicitly. Plugins that identify themselves as zero latency compatible (all Audeonic plugins) will default to zero latency switched on.

  Not all MIDI FX are suited to this; FX that monitor the timeline may get confused. If you find that a MIDI effect behaves erratically, switch this option off.

- Open Plugin

  Pressing this button will open/close the plugin's GUI panel. A shortcut for this is available by touching the block on the canvas.

Where an AU plugin exposes writeable AU parameters, these are available to be remote MIDI controlled using MidiFire's 'Remote Control' facility in the same way as built-in modules.

Plugins that respond to beat positions (eg. sequencers) can be driven and synced either using MidiFire's Dynamic Clock module or via external MIDI clock.

Tip - a plugin's configuration is saved/restored with scenes.

Channel Strip

modules

The Channel Strip module packs the most commonly used MIDI channelisation functions into one handy module. Place this module before MIDI outputs to limit incoming channels, remap outgoing channel, remap and/or transpose notes and convert poly to mono.

Parameters:

- allowed incoming channel (default: any)

  Set the specific channel that you want the module to accept. If set to anything other than 'any' the module will block all MIDI events that are not on the selected channel.

- transpose notes (default: 0 semitones)

  Transpose all incoming note events by the number of semitones selected in the dropdown.

- remap note from/to (default: no mapping)

Remap incoming notes to different output notes. First select the incoming note you wish to map **from** on the left. Next select the note you wish to remap **to** on the right. Repeat for as many notes as you wish to remap.

Be aware that any remapping happens after any transpose.

- convert notes to mono (default: no)

  Switching this on will convert a polyphonic source to mono intelligently and will operate like a mono keyboard from the good old days.

- outgoing channel remap (default: no)

  You can remap the outgoing channel of channelised MIDI events to match the channel of a receiver. Handy for remapping the default channel 1 of most controllers to a specific channel.

- block when bypassed (default: no)

  When this option is set to 'yes', then when the module is bypassed, rather than passing the events on as normal, all events are instead blocked. This is handy when you wish to use the bypass button to dynamically control event flows like a tap.

Tip - on the canvas block, there are two labels that show you the configured channel mapping of the module. The left (info 1) shows the allowed incoming channel number (if any) and the right label (info 2) shows the outgoing channel being remapped to (if configured).

Note/Velocity Strip

modules

The Note/Velocity Split module allows the splitting of an incoming event stream over two user defined MIDI channels. A split can be done on keyboard position (note) or note velocity. Incoming note and aftertouch events are remapped regardless of the channel that these events are on when entering the module.

Parameters:

- split type (default: note)

  If this parameter is set to 'note' then the split will operate based on the pitch of the note you select as the split point. All note ons, offs and key aftertouch will be sent to either the lower channel depending upon the pitch of the note.

  Setting this parameter to 'velocity' will send the note (and aftertouch for that note) to either the lower or upper channel depending upon the note's original strike velocity.

- split point (default: middle C or 64 velocity)

  This is the value that determines whether the note events will be directed to the lower or upper channel. The split point is the first note or velocity value in the upper range.

- lower channel (default: 1)

  Notes whose pitch or velocity (depending upon type parameter) are lower than the split point will be remapped to the channel specified in this parameter.

- upper channel (default: 2)

  Notes whose pitch or velocity (depending on type parameter) are greater than or equal to the split point will be remapped to this channel.

- non-note/aftertouch handling (default: pass unchanged)

Other channelised events passing through the splitter (pitchbend, channel pressure, controllers and program changes) can be handled in several different ways depending on the situation.

If these types of events should not be altered in any way, then set this to 'pass unchanged'. If you wish to block these other events entirely then set this to 'block'

Alternatively, you can remap these other events to either the lower split channel only, the upper split channel only or have them sent to both upper and lower split channels by selecting these in the dropdown menu.

All of these other events are affected as a group. If you need finer control (say sending CC A to lower and CC B to upper) then use a Stream Byter beforehand to do the controller remapping and leave this option at the default.

The graphic below the parameters displays the current split point. You can also touch/mouse the graphic to set the split point.

Tip - the split point and upper/lower channel parameters are all remote controllable via MIDI (configured in Remote Control)

Pressure Curve

modules

The Pressure Curve module allows the definition of a velocity curve map that can be applied to incoming notes, key aftertouch or channel pressure. The map can be adjusted point by point or graphically using touch/mouse control.

Parameters:

- curve type (default: note velocity)

  This parameter determines what type of events the map curve will be applied to. Either notes, aftertouch (polyphonic per note), channel pressure or controller value can be affected.

- remap value from/to (default: no (linear) mapping)

  Remap incoming velocity/aftertouch/controller values to different values. First select the incoming value you wish to map **from** on the left. Next select the value you wish to remap **to** on the right. Repeat for as many values as you wish to remap.

Tip - the graphic below the parameters displays the current curve. You can also touch/mouse the graphic to draw or modify the curve.

Event Monitor

modules

The Event Monitor module allows you to see at a glance the MIDI events coming from a port or a module. The canvas block shows the last few events in a compact form. Expanding the module shows all events received in a more detailed form.

Tip - use the 'Clear' button to clear events in the detailed view.

Comment Block

modules

The Comment Block is a non-functional module that you can use to record free-form notes about the canvas. You can add text to the block itself (summary) and more detailed notes in the module's settings.

Tip - Comment Blocks will be saved into scenes.

modules

Protocol Filter

The Protocol Filter module allows you to block or allow a range of MIDI protocol events passing through the module. It does this by marking a range of MIDI events (according to the MIDI data specification) and then allows you to block or only allow events that fall into that range.

Parameters:

- FROM (hex message)

  This value shows (and allows you to change) the starting event to which the filter (block/allow) will be applied. If you know the hex of your starting (and ending message) you can just enter it in this field. Alternatively, you can make a selection using the other parameters and the hex field will be updated too.
- event type, channel, number, value
- You can use these parameters to specify the MIDI message in a more user friendly way if you prefer. Note that as you change any of these values, the corresponding hex value above changes accordingly. Also, some of the parameter labels will change depending upon the event type selected to make sense MIDI-wise
- TO (hex message)

  You mark the end of the range in the same way as the start by adjusting the hex value (or other parameters).
- Block/Allow (default: Allow)

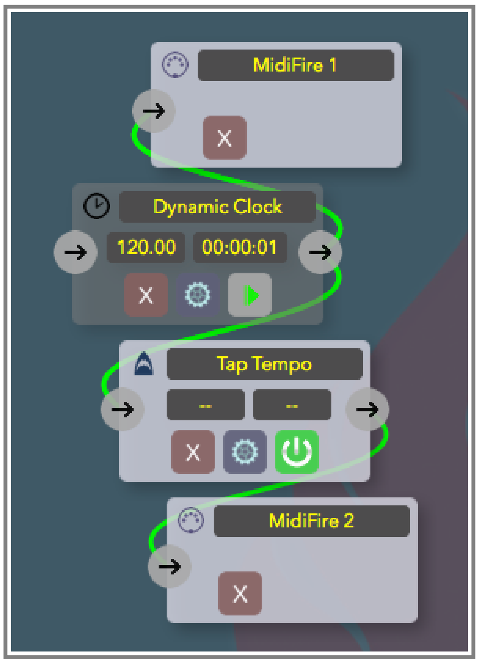
  Set whether the filter blocks or only allows your configured range by toggling these two buttons.

Tip - because this works on a horizontal range of hex values, the Protocol Filter is probably not the tool for blocking/allowing specific MIDI channels. Use the Channel Strip or Stream Byter for this.

Dynamic Clock

MidiFire implements the much lauded MidiBus clock via a specialised module. You can control many aspects of the internal clock dynamically via remote control MIDI messages; see further for details of this.

The output of the clock signal varies slightly depending upon how the module is connected:



Clock Connections

1. The virtual destination/output 'MidiFire Clock' which is presented to other apps by MidiFire will **always** be an unmodified copy of the clock signal. You do not need to add the 'MidiFire Clock' port to the canvas.
2. If the Dynamic Clock module is **directly** connected to a MIDI destination/output (like 'MidiFire 1' in the above diagram), then that MIDI destination will receive the internal clock output directly, bypassing any internal routing in MidiFire. This is the most efficient way of distributing the clock signal to a destination.
3. If the Dynamic Clock module is **indirectly** connected to a MIDI destination/output (like MidiFire 2' in the above diagram) then the internal clock signal is first of all fed into the interconnecting module(s) and then routed to the output. This type of connection is made when you want to do something with the clock signal beforehand.

Parameters:

- tempo

  Use this field to set a specific tempo manually. You can use decimal fractions like 121.34 if you like and the value can range from very slow to very, very high (how high will depend upon the processing power available on your device)

- Tap Tempo

  Tap this button a few times to set the tempo according to how quickly you tap it.

- Learn Tap

  You can set up a remote MIDI event to control tap tempo. Press the 'Learn Tap' to arm and the next MIDI event received will be set as the tap tempo trigger. Subsequent MIDI events that match the learnt event are processed as if you had tapped the 'Tap Tempo' button above. Learnt events can (only) be a note on, non zero continuous controller or program change types.

- learnt tap event

  This will either show the learnt tap tempo MIDI event, or if you know the hex message of the event you wish to use, you can simply enter that.

- Auto Start Clock

  You can tell MidiFire to automatically start the clock as the tempo is being tapped. The number of taps required to start is set using the field menu and the clock will start exactly on the beat after that number of taps. For example, if you set the value to '4' and start tapping, the clock will automatically start on the beat after the 4th tap, ie. on tap 5 (even if you don't tap 5 times).

  Setting this to 'no' (the default) will disable the feature and you will need to start the clock yourself.

The 'bypass' block button is repurposed to play/stop for this module (and the 'power' icon replaced by more transport friendly icons). Tap the start/stop button to start/stop the clock. Note, when the clock is running, the module block becomes yellow.

The left hand info label on the block shows the currently configured (or if running, calculated) BPM. The right label shows where the clock is up to in real world time in hh:mm:ss format.

Tip - the Dynamic Clock module is a clock generator. However, you can still feed (and manipulate and send out) an external clock source from another app or physical port.

Remote Control

The Dynamic Clock responds to a set of MIDI messages to control the transport and tempo. These are described as follows:

- Tap Tempo (default CC 63 on channel 1)
- Control the tempo dynamically via tap events. This MIDI event can be set on the configuration panel or learnt (via same panel)

- Start, Stop and Continue (fixed System Real Time)

  You can start, continue or stop the clock by sending the standard MIDI start/stop/continue messages (FA, FC and FB respectively in hex)

- Coarse/Fine tempo adjustment (fixed CC 19 and 51, channel 1)

  Sending CC 19 on channel 1 will adjust the tempo to an absolute value between 20 and 200 BPM depending on the value byte of the CC (0 = 20, 127 = 200)

  Sending CC 51 on channel 1 will adjust the current coarse tempo from -20 to +20 in tenths of a BPM (0 = -20.0, 127 = +20.0).

- Tempo increment/decrement (fixed CC 18 and 50, channel 1)

  Sending CC 18 on channel 1 will increment the current tempo by tenths of a BPM. (ie. value byte 1 = 1/10BPM, 2 = 2/10BPM, 10 = 1BPM, 100 = 10BPM)

  Sending CC 50 on channel 1 will decrement the current tempo by tenths of a BPM. (ie. value byte 1 = 1/10BPM, 2 = 2/10BPM, 10 = 1BPM, 100 = 10BPM)

- Absolute tempo value (fixed sysex)

  You can remotely set the clock tempo to an exact BPM value using a sysex message:

```
F0 5A <ThousandsHundreds> <TensUnits> <TenthsHundredths> F7

example: F0 5A 01 25 37 F7 = 125.37 bpm
```

Remote MIDI control messages can be fed into the Dynamic Clock module directly from other apps by sending the control messages to the 'MidiFire Clock' virtual port which is always listening to these messages. Alternatively, you can feed remote MIDI control messages into the Dynamic Clock by connecting MIDI inputs (or module outputs) to the Dynamic Clock's input connector in the usual fashion. Remote control events are processed whether the clock is running or not.

Tip - you can only ever have one instance of Dynamic Clock on the canvas at one time.

Tip - if you save a scene with the clock currently running, then when you load that scene later, the clock will autostart.

Stream Byter

modules

Check out StreamByter University on our forum for detailed articles on getting started and tutorials.

The Stream Byter started out as a way for us to extend MidiBridge 'out in the field' for customers, but gradually moved to be one of the most used (and infamous) aspects of MidiBridge. Following on from our experiences and suggestions, MidiFire implements a new backwards compatible but much more advanced Stream Byter module.

You use the Stream Byter to program your own custom MIDI processing modules which you can then go on to re-use again and again.

With power comes complexity, so the Stream Byter has a bit of a learning curve and you also need to understand the MIDI protocol. Many customers have mastered the Stream Byter but we do recognise it is somewhat esoteric. Therefore we offer to provide assistance in writing Stream Byter rules via email or via our soapbox forum for those that have better things to do with their time.

If however, you are not daunted by a bit of complexity, then here is 'the gory detail' (that's a nod to the PERL manual) for reference purposes.

Tip - if you're looking for the syntax for specific rules, then here is a handy set of links for you:
Stream Byter I, Stream Byter II, Variables/Values, Conditionals, Assign, Send, Maths, Labels, Logging, QWERTY keystrokes (mac)

Stream Byter (MidiBridge version)

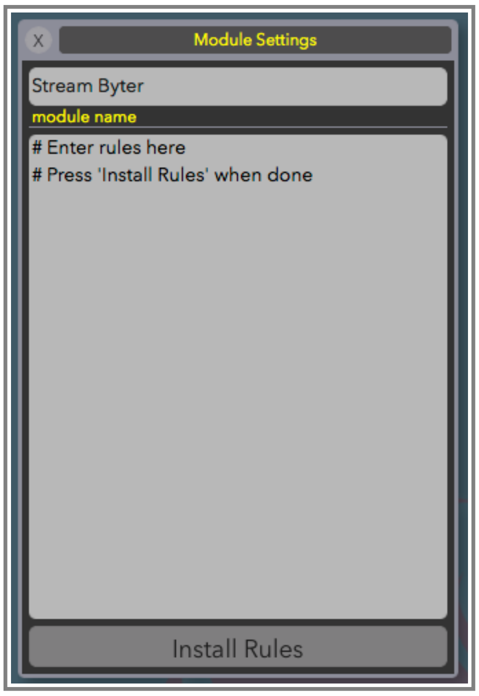
This first section of the Stream Byter reference covers the MidiBridge version of the Stream Byter to which MidiFire is (almost completely) backwards compatible. This section was lifted (with slight modification) from the MidiBridge manual and we've kept it mostly intact for posterity reasons.

Some of the things you can do with the MidiBridge Stream Byter are:

- Map any MIDI event to any other MIDI event including type, channel and value.
- Create up to 128 non-contiguous zones per channel.
- Create overlapping zones.
- Split controller messages into channelised zones.
- Use note events to change scenes.
- More precise blocking of events than the event filter.

This is hardly an exhaustive list, but with flexibility comes some complexity, and to use the Stream Byter you do need an understanding of the MIDI protocol, but fear not, as because it is possible to paste rules from an email, we can help by designing rulesets for what you are trying to achieve and email them to you.

We have also produced a detailed tutorial for creating Stream Byter rulesets (on our website) and you can also post queries about this (and of course anything MidiBridge related) on our support forum, again, see our website.



Stream Byter

Enter rules here
Press 'Install Rules' when done

Stream Byter

The Stream Byter panel contains an editable text window for you to define rules to match and act on MIDI events. One rule per line is permitted and you can comment your rules by preceding your commentary with a '#' symbol.

Once you have entered your rules, you press the 'Install Rules' button which checks your rules for validity. If any rules are incorrect, these are marked with 'ERR' and are commented out. To fix a rule with an error, simply edit the line (no need to remove the '#ERR' part!) and press the 'Install Rules' button to try again.

Each rule consists of two clauses, separated by one '=' sign.

The clause to the left of the '=' is the input clause where you specify which MIDI events are to be considered.

The clause to the right of the '=' is the output clause where you specify what happens to an event when it matches the input clause.

You can also specify flags at the end of the output clause:

- +C - clone the incoming message and apply the output clause to the clone.
- +B - block the incoming message if it matches the input clause
- +Dnnn - delay the event by nnn milliseconds

Both input and output clauses are constructed by 1,2 or 3 separate hex bytes depending upon the nature of the rule. Here are some simple examples:

```
# remap all controller events coming in on channel 1 to channel 2
B0 = B1

# clone all controller events coming in on channel 1 to channel 2
B0 = B1 +C

# remap controller 7 on channel 0 to controller 6 on channel 1
B0 07 = B1 06
```

```
# remap note C-2 to program change 0 (on channel 1)
90 00 = C0 00
```

You can also specify wildcards and ranges in the incoming clause:

- The value 'N' in the first nibble of the first byte represents note on and note offs (ie. 8 or 9)
- The value 'X' in the first nibble of the first byte represents all event types.
- The first nibble of the first byte (type) can be set with a range of types to match. (0-F)
- The value 'X' in the second nibble of the first byte represents any channel.
- The value 'XX' for the second or third bytes represent any value. (00-7F)
- The second nibble of the first byte (channel) can be set with a range of channels to match. (0-F)

Here are some examples of wildcards:

```
# rewrite all events on channel 1 to channel 2
X0 = X1

# rewrite all note on/off messages on channel 1 to channel 2
N0 = X1

# collapse all notes on all channels to channel 1
NX = X0

# block active sense messages
FE = XX +B

# control controllers 6 and 7 with controller 6
BX 06 = XX 07 +C

# rewrite all program changes to program change 1 on same channel
CX XX = XX 01
```

You'll note that you can use 'X' and 'XX' wildcards in the output clause. This signifies that the incoming corresponding value of the event is to be preserved.

You can also replace byte 2 with byte 3 (and vice versa) by specifying 'X2' and 'X3' for the values of byte 2 or 3 in the output clause.

You can specify ranges of values using the '-' sign inbetween low and high values for type (8-F), channel (0-F), number (00-7F) and value (00-7F). Examples:

```
# remap all events on channels 1-8 to channel 10
X0-7 = X9

# limit the max velocity on all notes on channel 2
N1 XX 40-7F = XX XX 40
```

These examples are quite simple and are to provide a foundation for writing more useful real-world rules. Again, please do look at our tutorial, post to our forum or email us if you would like help in creating custom rules for your requirements. There is no doubt that this module is not for the beginner.

Finally, some caveats to be aware of when writing rules:

- Rules are evaluated top to bottom and the results of each rule are fed into the next (unless the clone flag is set).

Stream Byter II (MidiFire extensions)

MidiFire extends the MidiBridge Stream Byter with many oft-requested features and incorporates a slightly different way of specifying rules. You can mix and match Stream Byter I and II rules in most circumstances.

Also, in the new freeform module paradigm, you can create multiple Stream Byter modules to operate in series or parallel.

Let's start with some basics:

Variables

Each StreamByter has its own set of four local variable arrays (prefixed by I, J, K and L) and there is one global variable array (prefixed by G) shared among all StreamByters per host app. These arrays each have 256 slots of unsigned 16 bit integers. A 'wide' array, prefixed by W, has 2048 slots of unsigned 16 bit integers. A 'precision' array, prefixed by P, has 256 slots of 32 bit signed integers.

The current MIDI message being processed is also addressed as an unsigned 8 bit array (max size 65536 bytes) prefixed by the letter M. Each variable letter is followed by a number (hex or decimal) that marks the position in the array starting from 0. Some examples:

```
L00         - local array L, 1st index
I2A         - local array I, 43rd index (hex)
I$43        - local array I, 43rd index (decimal)
G72         - global array G, 115th index (hex)
M03         - message byte number 4 (counting from 1)
M1234       - message byte number 4661 (sysex message!)
M$1234      - message byte number 1235 (using decmial)
```

A special variable ML contains the length of the current MIDI message, Special variable MC contains the MIDI channel (0-F) of the current message (although this returns F0 if the message is not a channelised message). Special variable MT contains the MIDI type/status ([8-F]0) of the current message (ie. the first nibble of the first byte, with any channel removed). None of these variables can be assigned to; treat as read-only.

An astute reader may realise that 'MC' should be the 13th byte of a message. This was an oversight and rather than break existing scripts, to get the 13th element of a message use M$12.

Another special variable 'R' returns a random number from 0 to (nnn is hex number or variable)). This variable may not be assigned directly.

The special variable 'BP' (or 'BPM' if you like) will contain the current tempo of the MidiFire Dynamic Clock module if present. The BPM value is in 100's of the BPM, so for example a BPM of 127.32 will have a value in the variable of 12732.

The special variable 'PO' (or 'POS' if you like) will contain the current MidiFire clock position in milliseconds (if running). This is a 32 bit signed value.

Variables can be indirect, much like an indirect cell in excel or a pointer in C. An indirect variable is a variable (as above, except ML/MC/MT) prefixed by G, M or I-L. Again, this might be better shown by example:

```
GL0  - global array G, index is that of the value stored in variable L0
       (local array L, 1st index)
```

```
MG03 - MIDI message array, index is that of the value stored in the variable G03
       (global array G, 4th index)
```

Variables can be used in conditional blocks, send commands, assign directives and maths directives (all explained below). Variables **cannot** be used in Stream Byter 1 rules.

Timer Variables

A special set of 8 'timer' variables, T00 to T07 are also available. These let you do timing calculations inside the Stream Byter.

Each time you refer to a timer variable, the value returned will be the number of milliseconds elapsed since that timer variable was last referenced.

The first time you refer to a timer variable after a scene load, it will return 0 milliseconds.

As the timer variables are 16 bit, the maximum time interval is 65.535 seconds.

Tip - the values of variables are not reset when you press the 'Install Rules' button, but they are reset during a scene load.'

Values

A value is either a variable (as above) or a literal hex value.

Don't like using hex numbers? You can prefix literal values with a '$' symbol to mark them as decimal values:

```
    MAT M0 = $20 + $40
    ASS L0 = $127
```

Tip - You can specify negative decimal number like: $-32

You can also use note literals (according to yamaha note numbering convention which numbers the notes from C-2 to G8) by prefixing the note with a '^' character.

```
    assign i0 = ^C-2 ^G8 ^Bb0 ^3C ^F#6

    if m1 == 90 ^3c
      # found a middle C
    end
```

Aliases

You can give any value an alias (single word) of your choosing to make your code easier to read. Aliases are set using the ALIAS keyword:

```
ALIAS <value> <name>
```

Whenever the 'name' parameter of the ALIAS rule is seen in place of a value, then that value will be referenced instead. Here is an example:

```
IF LOAD
  # setup some aliases
  ALIAS Q0 CHANNEL
  ALIAS $127 CC_MAX
  ALIAS I0 TEMP_VARIABLE
END

IF MC == CHANNEL
```

```
    MAT TEMP_VARIABLE = B0 + CHANNEL
    SND TEMP_VARIABLE 07 CC_MAX
  END
```

Conditionals (IF/ELSE/END)

A conditional block is a set of rules that will only be evaluated/executed if the condition is true. An IF must be terminated by an END to mark the end of the conditional, but an ELSE is optional. For example:

```
  # see if the current MIDI message
  # is a program change on MIDI channel 1
  IF M0 == C0 # compare 1st msg byte with 'C0' literal
    # do something
  ELSE
    # do something different
  END
```

The conditional expression (after the IF) is defined as

```
  <value> <operator> <value> [<value> [<value>] [<value>]] [+L[OOP]]
```

An operator can be one of:

```
  ==, !=, <, <=, >, >=
```

If more than one value is specified after the operator (max 4) then the left hand value should be a variable and each right hand variable corresponds to an incremental index. Example:

```
  IF M12 == 64 32 G01
    # do something
  END
```

This compares M12 against 64, M13 against 32 and M14 against G01 and only if all 3 are equal is the condition true. This is useful for identifying specific sysex messages that you wish to modify as they pass through.

Here is an example using an indirect reference to demonstrate that in a multi RHS comparison, against an indirect LHS value enumerates the primary array and not the second (!):

```
  ASS K0 = 1 2 3 4 5 6 7
  ASS I0 = 3

  IF KI0 == 4 5 6
    # this will be true
  END

  # the conditional above works out to be
  # the same as
  IF K3 == 4
    IF K4 == 5
      IF K5 == 6
      END
    END
  END
```

Conditionals can be nested to make an 'and':

```
IF M00 == B0
  IF M01 < 20 30
  END
END
```

or in series as an 'or'

```
IF G0 > 01
END
IF G0 <= 01
END
```

The '+L' flag indicates that the condition is to execute in a loop. When control reaches the matching END, instead of proceeding to the following rule, control is instead passed back to the original IF line where the condition is evaluated again. If the condition is (still) true, then the IF clause is executed again. If the condition is false, control jumps to the next rule after the matching END. Note, to avoid nasty hangs if an infinite loop is accidentally programmed, a looped conditional will not loop more than 128 times.

Tip - You can use the keyword 'WHILE' instead of the 'IF' which implies the +L flag is set.

```
ASS I0 = 0
IF I0 < 10 +L
  MAT I0 = I0 + 1
END
```

is the same as:

```
ASS I0 = 0
WHILE I0 < 10
  MAT I0 = I0 + 1
END
```

Finally, there is a special condition called 'LOAD' which is always true when a module is loaded (either when the app starts or is in a scene that is recalled). This lets you initialise or do stuff before any messages are processed. Inside a LOAD block, the M variable array is not available, since there is no MIDI message:

```
# send program change 2/ch1 with 2s delay on load
IF LOAD
  SEND C0 01 +D2000
END
```

Extra rules (BLOCK/EXIT)

In version 2.0, two new rules to make logic a little easier are available:

BLOCK - block the current event (same as XX = XX +B)

EXIT - stop any further processing and exit the script immediately

```
# block and exit script when we see an
# active sense message
IF M0 == FE
```

```
    BLOCK
    EXIT
  END
```

Tip - BLOCK and EXIT have no effect inside an IF LOAD section.

Send (SND)

You use the SND command to issue an arbitrary MIDI message:

```
  S[E]ND <value> [<value> {<value> ...}] [+F[ORCE]] [+I[NJECT]] [+Dnnnn]
```

Here are some examples

```
  SND C0 01      # send PC 1 to module's output
  SND M0 M1 7F   # send current message with 3rd byte
                 # fixed to 127
  SEND G0 L0 L1 # send a MIDI message constructed from
                 # global/local variables

  # inject message back into MidiFire in 500ms
  SND B0 07 72 +I +D500
```

The maximum number of bytes that can be sent in the one SND rule is 16.

Normally, StreamByter will check that what you are trying to send is a valid MIDI message. Sometimes, you might wish to make up a long sysex message and need to split over multiple SND lines. You can disable to validation checking using the +F (force) flag.

In addition, you can specify an '+I' flag at the end of a SND. Normally a SND just sends the message out of the module into the next. If you specify the +I (inject) flag, then the message is injected 'at the top' as if it was received from the current message's MIDI port (or the MidiFire virtual port in a LOAD block). Use the +I flag to auto-select a scene change (for example).

Finally you can delay the SND by using the +Dnnn flag (where nnn is a decimal value in ms), so +D2000 means with a 2 second delay. The maximum delay is 65535 milliseconds.

You can also use the keyword SEND instead of SND.

New in version 2.0 is the ability to send a UDP message using the SND command:

```
  S[E]ND <start> <size> +U<hostname>:<port>
```

The data to be sent is taken (only) from the 'W' array, so it is possible to send arbitrary binary UDP messages (for example OSC) of up to 2048 bytes. The 'start' parameter is the index number of byte 0 in the W array and the 'size' parameter is the number of bytes to send. 'hostname' and 'port' are the UDP destination credentials.

```
  IF LOAD
    # setup two OSC messages in W array
    ASS W00 = 4F 53 43 00 # 'OSC'
    ASS W10 = 41 42 43 00 # 'ABC'

    # you can use variables for start, size and port
    # this example for second 'ABC' message
    ASS I0 = 10 4
    ALIAS I2 PORT
    ASSIGN PORT = $2468
  END
  # send our OSC messages when we see sustain pedal
  IF M0 = B0 40 7F
    # send 'OSC' to host1 on port 1234
    SND 0 4 +Uhost1:$1234

    # send 'ABC' to local IP on port 2468
    # using variables I0-2 for credentials
    SND I0 I1 +U192.168.1.1:PORT
  END
```

Tip - The SND +U implies the +F flag is set.

Assign (ASS[IGN])

Assign is used to set the value(s) of array variables:

```
  ASS[IGN] <value> = <value> [<value> ...] [+P[RESERVE]]
```

Like conditionals, by specifying multiple values to the right of the '=' is a fast way of setting multiple values in an array:

```
  ASS L0 = 12                # assign '12' to L0
  ASS L0 = 01 02 03 04       # assign 01 to L0, 02 to L1,
                             # 03 to L2, 04 to L3
  ASS G0 = M0                # assign value in M0 to G0
  ASS GL0 = 12               # assign '12' to array G indexed
                             # by value of L0
  ASSIGN K0 = 00 +P          # assign 0 to K0, the value of K0
                             # is preserved
```

The '+P' flag indicates that the array values being assigned to should be preserved in the scene or between app invocations. When the scene is loaded or the app is restarted, the preserved value will be restored into the array value. You may only specify the +P flag where you are assigning to the global or local arrays directly.

Maths (MAT[H]|CAL[C])

MAT commands are a single assign but with two operands and a mathematical operator:

```
  MAT[H] <variable> = <value> <operator> <value>
```

operators are:

```
  +, -, *, /, &, |, ^ and %
```

```
  MAT L0 = L0 + 1  # increment L0 by 1
  MAT G0 = L3 % L4 # assign value of L3 modulo value of L4
                   # to variable G0
```

Tip - You can use MATH or CALC keywords; they are the same as MAT

Debugging Log (LOG)

When writing StreamByter code, it can be helpful to log what is happening in your script using the LOG rule:

```
LOG <string> [<value> [+D|+N]]
```

Where the mandatory one word string is a string of your choice (use underscores in the string to get spaces in the log entry).

Value is optional and can be a literal or variable.

Use the +D (decimal) or +N (note) flag to display the value as decimal or note.

```
# log when script is loaded
if load
  log Script_Loaded
end

# log each note on received
if MT == 90
  LOG Note_on M1 +Note
  LOG Note_Velocity M2 +Decimal
end
```

So, how do you actually **see** the log results? Connect an Event Monitor immediately after the Stream Byter and the log entries will show up in that monitor. Each log entry is timestamped with that of the MIDI event that was being processed when the LOG rule was reached (or the current time if inside an IF/LOAD).

Set label (SET)

You can set the value of the two 'info' labels on the Stream Byter block using a SET rule:

```
SET [LB0|LB1] <value|S<string>> [+D[ECIMAL]] [+N[OTE]]
```

Where LB0 is the left label and LB1 is the right label.

Value is as defined above (literal or variable)

You can set the label to an arbitrary string by prefixing with 'S'

The optional +D flag means display the value in decimal (default is hex)

The optional +N flag means display the value as a note name

```
# set left block label to the hex 32
SET LB0 32

# set right block label to the value stored in byte 3 of the current message
SET LB1 M02 +D

# set left block label to the string 'on'
SET LB0 Son

# set right block label to name of incoming note
IF M00 >= 90
  IF M00 <= 9F
    IF M02 > 00
      SET LB1 M01 +N
    END
  END
END
```

Introduced in version 2.0 are some extra things that can be set from a script:

NVR - with version 2.0 onwards, note events with a velocity value of 0 are automatically converted to note off events before being passed the the script. This saves you from the dreaded 9X XX 00 = 8X incantation. However, if you do not like that behaviour you can set the NVR flag to 0 and keep it the way it was.

```
SET NVR <0|1>
```

NAME - you can give your script a name and that name will be shown as the block name. This can be set dynamically.

```
SET NAM[E] <yourname>
```

FLUSH - setting this to 1 will flush all pending (advanced scheduled) MIDI events.

```
SET FLU[SH] 1
```

Macros and Subroutines

Version 2.0 introduces two new code features; DEFINE and SUBROUTINE

The DEFINE rule lets you give any sequence of code tokens a name and then use that name further in the code instead of typing in the original text. This is similar to a #define in C but no parameters. Like the C version, you can have DEFINES that refer to previous DEFINES.

```
DEF[INE] <name> <some code>
```

It is essentially a find/replace. Best explained by example:

```
IF LOAD
  DEFINE MSG0 M0
  DEFINE CHAN_VOLUME B0 07
  DEFINE CHAN_VOLUME_MIN MSG0 == CHAN_VOLUME 00
END

IF CHAN_VOLUME_MIN
END
```

The SUBROUTINE rule allows you to create code subroutines (with arguments) that you can call from elsewhere in your code.

```
SUB[ROUTINE] <name> [<args>]
   ... lines of code ...
END


# courtesy of Millie Jackson
IF LOAD
  SUBROUTINE MUFFLE ARG1 ARG2
    IF ARG1 >= 40
      SEND B0 ARG1 ARG2
    END
  END

  ALIAS L0 THAT
  ALIAS L1 FART
END
```

```
IF MT == 90
  MUFFLE THAT FART
END
```

Arguments can be any string you like. Note that arguments are passed to the subroutine via reference, so for example:

```
SUBROUTINE ZAP VARIABLE
  ASS VARIABLE = 0
END

ZAP I0 # will set I0 to 0
```

It is possible for a subroutine to call itself (recursion), but to prevent infinite recusion, the maximum recursion level is 256 calls.

When a subroutine is called and any of its arguments were not passed by the caller, those missing arguments will have a literal value of 0.

As well as the argument strings set in the SUB rule, the arguments are also available using the 'Z' array and the number of arguments passed is in the special variable 'ZN' (number of args). This means a subroutine can determine the number of arguments and cycle through them like this:

```
IF LOAD
  SUB FOO ARG1 ARG2
    # send a CC with each passed arg as
    # CC value
    ASS I0 = 0
    WHILE I0 < ZN
      SND B0 01 ZI0
      MAT I0 = I0 + 1
    END

    # note, Z0 will be the same as ARG1,
    # and Z1 is the same as ARG2
    SET LB0 ARG1 +D
    SET LB1 ARG2 +D
  END
END
```

Tip - It is suggested for efficiency that all ALIAS, DEFINE and SUBROUTINE rules be inside an IF/LOAD, but that it not mandatory.

Send Keystroke (KEY) (mac only)

The KEY rule allows you to send a single keystroke to the currently focussed application from your script. Think converting MIDI events to qwerty keystrokes.

```
KEY <N|S|C|A|O> <keysym>
```

Where the first parameter denotes the key modifier and can be NORMAL, SHIFT, CONTROL, ALT or OPTION.

The second parameter is the mac 'keysym' to be sent. This is a number and depends upon your keyboard layout. Search the web to find tables of keysyms for your keyboard layout.

```
IF M0 == B0 12 42
  # send 'q' key to current application
  KEY NORMAL $12
END
```

Advanced Examples

Finally, here are some examples of more advanced ways of using the Stream Byter

```
# controller value remap table
IF LOAD
  ASS L00 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
  ASS L10 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
  ASS L20 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
  ASS L30 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
  ASS L40 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
  ASS L50 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
  ASS L60 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
  ASS L70 = 10 10 10 10 10 10 10 10 12 12 12 12 12 12 12 12
END
# remap value of CH1/CC7
IF M0 == B0 07
  ASS M2 = LM2
END
```

```
# when loaded play C3 3 times (looping)
IF LOAD
  ASS L0 = 2
  SND 90 3C 40 +I
  SND 80 3C 00 +D500 +I
END
IF M0 == 90
  IF L0 > 0
    SND 90 3C 40 +I +D1000
    SND 80 3C 00 +I +D1500
  END
  IF L0 != 0
    MAT L0 = L0 - 1
  END
END
```

5a. 'Niche' Modules

top

We have included a few specific purpose modules based on our own ideas and work we have done for other customers. These may be useful to others.

Tracking Clamp

modules

The Tracking Clamp module is designed to reduce the artefacts of MIDI guitar signals (ie. miss-hits). It does this in two ways:

- Velocity Compression
- The clamp monitors the incoming MIDI signal and compresses outlying velocity values based on previous notes. This makes your note volumes smoother and miss-hit notes will not stand out so greatly.

- Artefact Removal

  The clamp monitors the incoming MIDI signal and blocks notes that are a long way from the previous notes that were played. This reduces the incidence of 'harmonic' miss-hits.

You can set the clamp to perform both velocity compression and artefact removal.

Adjusting the 'degree' parameter changes how many notes are examined to establish the 'norm' and for how long the norm is held in place. Best way to adjust this is with trial and error as playing styles vary.

Tip - when using a MIDI guitar and the Robotic Knob put a clamp before the Robotic Knob to tame the signal.

Robotic Knob

The Robotic Knob module monitors your playing and generates complementary CC messages that can be fed forward to apps or outboard FX to control sound parameters. The Knob can react to note velocity, keyboard position, MIDI guitar fretboard position (eg. Fishman Triple Play) and pitchbend.

The types of things you can do with this module are up to your imagination! You could cross fade between sounds as you move up/down the keyboard/fretboard, open up a delay (and close) as you pitchbend up or increase chorus the softer you play. It's like having a roadie with unlimited fingers automatically adjusting knobs on your gear based on your playing.

Tip - you can have loads of Robotic Knobs all monitoring different aspects of your playing and adjusting in real time for some very crazy control of your sound.

Parameters:

- trigger type (default: Note On Velocity)

  The trigger type parameter tells the Robotic Knob what type of event you wish to have monitored. The possible values are:

  - Note On Velocity

    The Knob will generate CC messages based upon how hard you play notes.

  - Keyboard Position

    The Knob will generate CC messages based upon where in the MIDI keyboard range you are currently playing (from low to high).

  - FTP Fretboard Position

    For owners of a Fishman Triple Play unit, this generates CC messages based upon which fret you are currently playing (from 0 to 24).

  - Pitch Bend

    This generates CC messages based on pitch bend information coming from your controller (from -64 to +64)

- trigger from, to (default: depends upon type)

  This parameter specifies the trigger event range that you wish the Knob to react to. Only when incoming events are within that range will CCs be generated.

- delta limit (default: none)

This parameter allows you to set a maximum limit within which CCs will be issued. If the current trigger event is 'delta limit' greater than the previous trigger event, then no complementary CC is generated. You can use this parameter to prevent sudden CC changes when you move a large distance on the fret or keyboard.

- oob clamp (default: Both)

  This parameter determines what will happen if a trigger event is outside of the from, to range. If the oob (out of bounds) clamp is full, then events outside of the range will trigger the minimum or maximum CC to be sent. If set to None, then no CC is sent if the trigger is out of range. If set to Lower then the minimum value is sent if the trigger is less than minimum. If set to Upper then the maximum value is sent if less than maximum.

- output CC, channel (default: CC7, channel 1)

  Use these two parameters to set the CC number and channel of the complementary CCs that will be issued.

- initial CC value (default: none)

  You can set the initial CC value that you want sent when the module is loaded. ie. set the initial CC value on load.

- output from, to (default: 0 - 127)

  This sets the range of the value byte of the complementary CCs. The Knob will scale the input event (based on trigger from, to) to this range.

- output smoothing (default: None)

  As complementary CCs are generated you can generate 'inbetween' CCs as the CC value moves from one value to another to make smooth graduations in change.

- smoothing speed (default: Immediate)

  If output smoothing is enabled then you can specify how quickly the inbetween CCs are issued during the smoothing process. With this you can control how slowly or quickly the complementary CCs ramp up or down.

OSC Exchange

modules

The OSC Exchange accepts OSC data, packages it up into MIDI sysex and then transmits over a MIDI channel to another MidiFire instance which then unpacks and forwards on the OSC messages at the other end.

Use this module to link OSC apps/gear over a wifi, bluetooth, DIN cable or USB MIDI connection (eg. musicIO).

Parameters:

- udp send port (default: 0)

  Packaged OSC data received by the module is sent out on this UDP port. If the value is 0, then nothing is sent.

- udp receive port (default: 0)

  The module will listen to OSC packets coming from this udp port and then package off and send the OSC data in sysex packets out of the module for forwarding elsewhere. If the value is 0, then the module will not be listening.

- max MIDI rate (default: 8000 bytes/s)
  In some circumstances you may need to limit the bandwidth used in sending the OSC data. Specify a value here if packets are getting lost, or set it to 0 for no rate limiting at all.

- wrap data in sysex (default: yes)
  OSC and other non MIDI data is generally wrapped in a sysex message to pass on to another MidiFire instance over a MIDI link. If, however, the data you are sending/receiving over UDP is just ordinary MIDI (say you want to give a scripting language MIDI in/out via UDP), then you would turn off this option.

In Conclusion

top

You've made it to the bottom of what has turned out to be a very long and in-depth manual! If you're still having problems or just have questions, please do contact us at apps@audeonic.com or join us on the Audeonic Soapbox (forum) at http://soapbox.audeonic.com

As a parting note, we hope you find MidiFire useful and would like to thank you for downloading it.

We would really appreciate it if you could take a little bit of time and rate and review MidiFire on the App Store to assist others who may be considering downloading the app and of course (hopefully) augmenting the development team's egos.

Tip - HTML was authored by hand in Dublin, Ireland

-- end