StreamByter

General purpose programmable MIDI effects plugin for CoreMIDI and Audio Units V3.

Quick Start

First, insert the AU into a suitable host application or connect up the standalone app via CoreMIDI (a virtual input and output are advertised that you can connect to/from other apps).

Then, you instruct the StreamByter to operate on the MIDI events passing through using textual rules (defined in detail below). To apply your rules you press the 'Install Rules' button which checks them and if all correct they will then be operational.

Here are some useful and basic rules that you can copy/paste to get going straight away:

```
# filtering rules
X1-F = XX +B # only pass channel 1
NX = XX +B # block all note events
BX = XX +B # block all controller events
F8-C = XX +B # block all clock events

# remapping rules
XX = X0 # remap everything to channel 1
N9 = X0 # remap notes on channel 10 to channel 1
NX 24 = XX 28 # remap note C1 to E1

# cloning rules
NX = XX +D400 +C # echo each note with 400ms delay
BX 07 = XX 08 +C # clone CC7 to CC8

# split all notes below middle C to channel 2
# all notes from middle C and up to channel 3
NX 00-3B = X1
NX 3C-7F = X2

# create an overlapping split C-2 to B3 = CH3
# C-1 to B4 = CH 4
NX 00-53 = X2
NX 47-53 = X3 +C
NX 54-7F = X3
```

History and Introduction

The Stream Byter first appeared as a module in our MidiBridge app in order for us to extend its features 'out in the field' for customers (and was really only for internal use), but gradually became one of the most used (and infamous) aspects of that app. Based on our experiences and many suggestions, we extended the Stream Byter module for MidiBridge's successor app, MidiFire. Whilst retaining backwards compatibility, the MidiFire Stream Byter was redesigned and coded from the ground up and is much more advanced. This AU implements the MidiFire Stream Byter module and thus includes both the MidiBridge (Stream Byter I) and MidiFire (Stream Byter II) feature sets.

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

Setting up - AUv3

The Audio Unit is installed automatically when the app is downloaded. It advertises itself as both an Instrument and Effect processor. Generally, you would use the 'Effect' variant although some hosts you need to use the Instrument type. The StreamByter window is fully resizable, so where the host app permits you can increase/decrease the size of the window or use it fullscreen.

Each host app is different but essentially you use the hosts's Audio Unit setup to insert the StreamByter into your workflow. So far StreamByter has been tested and known to work in AUM, apeMatrix, Cubasis and sequencism. A couple of specific tips for usage:

- Cubasis - make sure the piano keyboard is showing when you want to edit StreamByter rules. This moves the plugin window upward; if you don't then the typing keyboard covers the edit box.
- Sequencism - use the 'Instrument' variant of the plugin.

In an AU environment, the host has to request events from the AU and this is done (generally) only while audio is running. This has an effect on SND rules inside an IF/LOAD statement. Depending on the host app, these can be delayed until the host (re)starts audio processing.

Finally, MidiFire users who are familiar with the SND rule and +I flag should note that this flag is not supported in an AU context.

Setting up - CoreMIDI

While the Audio Unit may be instantiated many times, the standalone CoreMIDI version is a singleton app and only one instance can be running on the device.

After you start the app, it will advertise an input and output virtual MIDI port pair which can be seen in other apps and can this be inserted into your workflow by making the appropriate virtual CoreMIDI connections in the 3rd party app(s). If you need to process hardware, bluetooth or network MIDI, then you will need a routing app like MidiFire to interconnect (although in this case MidiFire has the StreamByter module built in).

Stream Byter I (MidiBridge version)

This first section of the Stream Byter reference covers the MidiBridge version of the Stream Byter to which StreamByter 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.

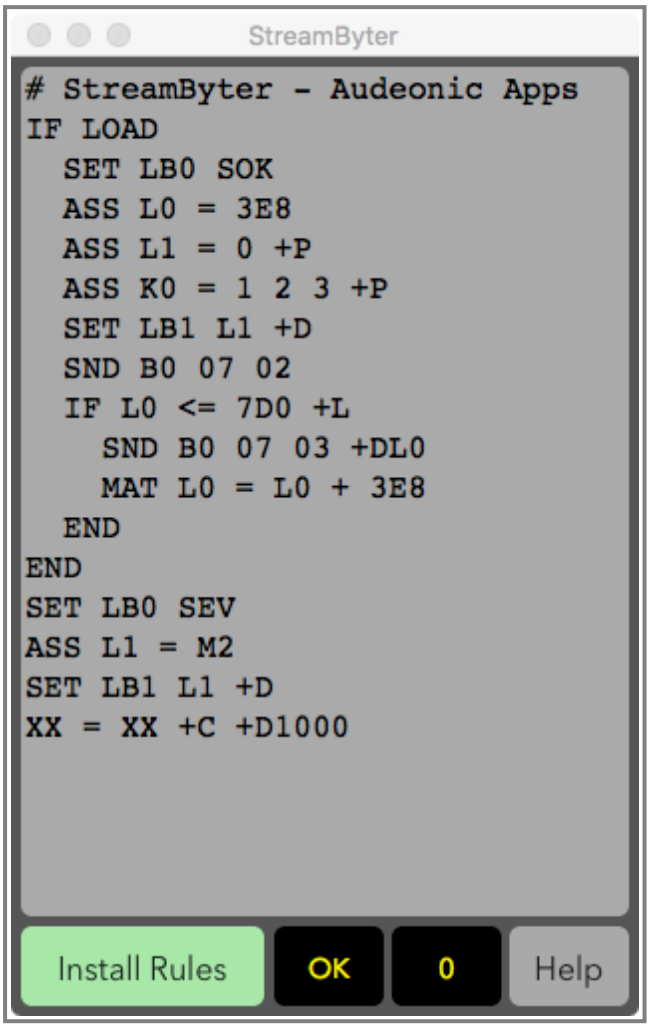
Some of the things you can do with the MidiBridge Stream Byter are:

- 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

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 9
X0-8 = 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)

StreamByter 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.

Let's start with some basics:

Variables

Each StreamByter has its own set of 4x256 16 bit unsigned integer local variable arrays (prefixed by I, J, K and L) and there is a 1x256 global variable array (prefixed by G) shared among all StreamByters per host app. 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 2 (or 4) letter hex number that marks the position in the array starting from 00. Some examples:

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

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.

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

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 value (00-FFFF)

Conditionals (IF/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. 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
END
```

The conditional expression (after the IF) is defined as

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

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.

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.

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
SND C0 01 +D2000
END
```

Send (SND)

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

```
SND <value> [<value> {<value> ...}] [+F] [+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
SND G0 L0 L1  # send a MIDI message constructed from global/local variables
```

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.

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.

Assign (ASS)

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

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

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
ASS 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)

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

```
MAT <variable> = <value> <operator> <value>
```

operators are:

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

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

Set label (SET)

You can set the value of the two 'info' labels on the StreamByter UI using a SET rule:

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

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
```

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
```

In Conclusion

top

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 StreamByter 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 StreamByter 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