StreamByter

General purpose programmable MIDI effects plugin for CoreMIDI and Audio Units V3 (macOS and iOS).

Version 1.6, October 2019

Quick Start

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

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 original 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 StreamByter 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 StreamByter has a bit of a learning curve and you also need to understand the MIDI protocol. Many customers have mastered the StreamByter but we do recognise it is somewhat esoteric. Therefore we offer to provide assistance in writing StreamByter 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 later in this dosument 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/Flags, UI Controls, Macros/Subroutines/Includes, Logging, QWERTY keystrokes (mac)

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 (iOS only), 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, Audiobus, 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.

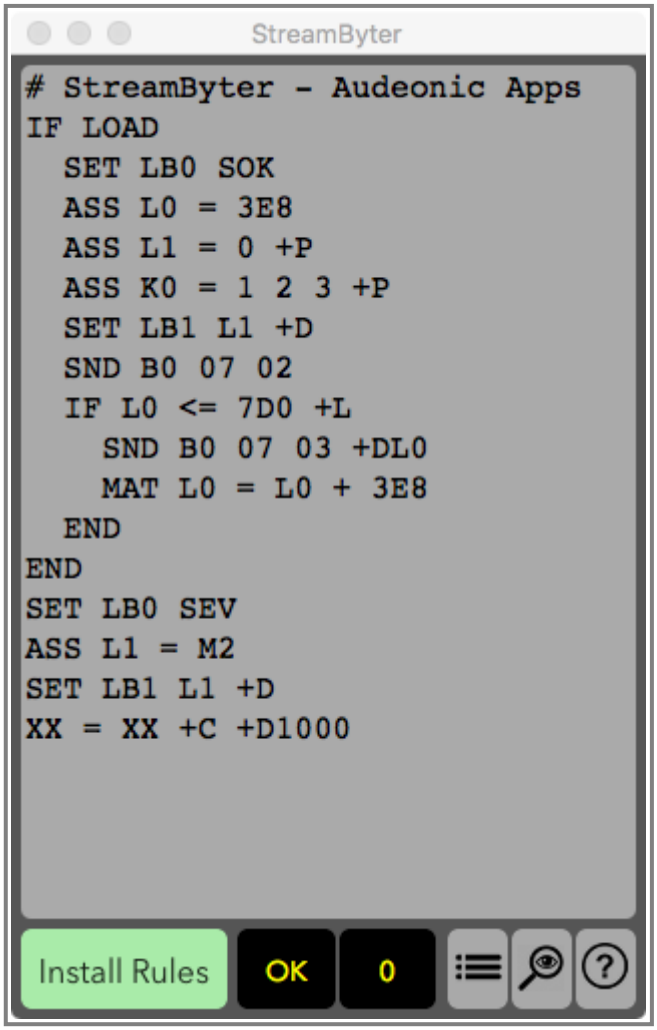
In an AU environment, the host has to request events from the AU and this is done (generally) only while audio is running. This has an effect on SND rules inside an IF/LOAD statement. Depending on the host app, these can be delayed until the host (re)starts audio processing.

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).

Here is the main interface panel and description of each element:



StreamByter

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.

Next to the 'Install Rules' button are two labels whose values can be set programatically within your StreamByter rules.

At the bottom right you will find some extra control buttons:

 - Presets Management

 - MIDI Event Monitor

 - This Help Page

Presets Management

You can manage StreamByter presets internally if you wish. Presets can be saved locally or to iCloud. The factory presets (read only) are also available in the preset manager. Use the dropdown menu at the bottom to switch between the different locations.

Presets stored to 'Local' in the AU variant are available on the local device to each instance of the AU in any host. In the standalone variant, local presets are available only to the standalone app.

iCloud presets are available across all devices (mac or ios) provided that iCloud Drive is active and logged in with the same credentials.

Use the 'Save', 'Load' and 'Delete' buttons to manage the presets in the current location. For Load and Delete you must select a preset in the list first.

Press 'Done' to return to the main panel.

MIDI Event Monitor

The MIDI Monitor panel displays all events entering and leaving the StreamByter in textual and hex format along with the events' timestamps.

Use the 'Clear' button to clear the current events from both the 'IN' and 'OUT' sub-monitors.

Press 'Done' to return to the main panel.

Tip - For efficiency, monitoring of events does not start until you have opened the monitor panel at least once.

The remainder of this manual details the StreamByter rule syntax and semantics.

Stream Byter I (MidiBridge version)

This first section of the StreamByter 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.

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)

Stream Byter 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 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. Sliders (see below) manage a 16 slot array of 16 bit signed integers prefixed by Q. 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
I$43       - 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!)
M$1234      - message byte number 1235 (using decmial)
Q0          - 1st slider
```

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 (if known) of the host or in the standalone version of any MIDI clock received. 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 host transport 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
```

Tip - Generally you will want to set aliases inside an IF LOAD

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
SND C0 01 +D2000
END
```

Extra rules (BLOCK/EXIT)

In version 1.4, 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 StreamByter 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.

The +I flag indicates that the message is not sent to StreamByter's output, but is injected into StreamByter's input where it will then be processed by the current ruleset. Injected events can also be delayed (see below)

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.

New in version 1.4 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

Labels/Flags (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[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 1.4 are some extra things that can be set from a script:

NVR - with version 1.4 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 in the case of an AU host (eg AUM) that name will be shown under the icon. This can be set dynamically.

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

SLI[DER_DISPLAY] - you can expose or show the controls panel from a script by setting this flag to 1 (expose 1st panel), 2 (expose 2nd panel) or 0 (hide). Helpful if you want the controls to appear automatically when your script is loaded. This can be set dynamically.

```
SET SLIDER_DISPLAY <0|1|2>
```

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

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

Sliders, Menus and Buttons

Introduced in version 1.3, the controls panel (exposed by touching the round icon on bottom right) is a set of 16 UI widgets (sliders, buttons or menus) that can control your script or be set from within your script (ie. bi-directional). The widgets are available as two separate pages of 8. Touch the controls icon to move from page 1 to 2. Note that the second page of controls will not normally be available if the code does not use controls Q8 to QF.

Controls Panel

Widget values are controlled in a script using the 'Q' array (0 to F). If a Q array value is set inside a script, then the widget will change according to the new value. Adjusting the widget will update the current value in the corresponding Q array.

In addition, when you adjust a widget an internal sysex message is fed into the script, so you can track when a widget is changed and do something. The format of the internal sysex message is:

```
F0 7D 01 nn F7
```

Where 'nn' is a value from 00 to 0F representing the widget that was changed

'nn' above can also be the value 7A, 7B or 7C which represents (respectively) start, continue and stop events either from MIDI clock or the host transport.

Tip - the internal sysex message never leaves StreamByter so no need to block it.

You can give the widget a name (10 character max, single word) and change the widget's range from the default of 0-127 using the SET rule as follows:

```
SET Q[0-F] <name> [min [max]] [+B[UTTON]|M[ENU]|N[OTE]|Y[ESNO]|T[OGGLE]|H[IDE]]
```

By default, the controls panel consists of all sliders, but you can replace any slider with either a push button or a drop down menu. Drop down menus can be just numeric (including negatives), note names (range limited if required) or booleans. You use the + flags shown above to setup a widget.

In the case of a button or toggle widget, the Q value will toggle between 0 and 1 on each press. A toggle is highlighted whenever its current value is 1. In the case of a boolean dropdown, the Q value will be 0 or 1. All other widgets will set Q to the value shown, or in the case of a note dropdown it will be the note number that corresponds to a note.

The +HIDE flag will hide the control from the controls box. There is no need to explicitly hide all controls on the second (Q8-16) page if they are not used.

Here is a full example of how to configure, set and track widgets:

```
IF LOAD
  # configure widgets name and range
  SET Q0 CHANNEL 1 $16 +MENU
  SET Q1 SWITCH +YESNO
  SET Q2 NOTE 3C 47 +NOTE
  SET Q3 PUSH_ME +BUTTON
  SET Q4 OCTAVES $-4 $4 +MENU
  SET Q5 DELAY_MS $100 $2000

  # set widget initial values
  ASS Q0 = 1 0 3D 0 0 $500
END

# handle widget movements
IF M0 == F0 7D 01

  # show widget number in left label
  SET LB0 M3 +D

  # show widget value in right label
  IF M3 == 2
    SET LB1 QM3 +NOTE
  ELSE
    SET LB1 QM3 +DECIMAL
  END
```

```
END
```

Each widget is exposed as an AU parameter, so remote control of controls via the host are possible. Note, that the widget range via AU is fixed at 0 to 127, but AU parameter change messages are scaled to fit the actual widget range.

Macros, Subroutines and Includes

Version 1.4 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 ANY_SLIDER F7 7D 01
  DEFINE SLIDER0_MOVED MSG0 == ANY_SLIDER 00
END

IF SLIDER0_MOVED
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
```

Includes is a mechanism for pulling in other code from StreamByter's presets or the web. You may have a library of handy subroutines that you use in many of your scripts. If so, you can save those subroutine(s) to a preset and include those subroutines in other scripts at Install Rules time.

```
SET INC[LUDE] F[ACTORY]|L[OCAL]|I[CLOUD]|U[RL] <preset_name>|<url>
```

Include presets must be saved as one, single word in order to be recognised. A url must be https and point to an .sbr file. Includes cannot be nested; an include inside an include is simply ignored.

A factory icnlude set 'standard_includes' is shipped with StreamByter which gives some common message types some standard names and re-usable subroutines. Just open up the standard_includes preset to see what is available.

```
IF LOAD
  # pull in the standard factory includes
  SET INCLUDE FACTORY standard_includes

  # retrieve some code includes locally
  # (local preset 'AUDEONIC_INCLUDES' must exist a local preset)
  SET INCLUDE LOCAL AUDEONIC_INCLUDES

  # retrieve some code includes from a URL
  # (url must resolve to a valid .SBR file)
  SET INCLUDE URL https://audeonic.com/midifire/club/FOO_INCLUDES.SBR
END
```

When an Include is resolved it is as if the rules in the include file replace the include line, so be aware of placement of your includes.

An include line will be flagged in error if the included code could not be found or if the included code itself has an error.

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

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
```

Log entries are shown on the output event logger. 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).

Send Keystroke (KEY) (mac standalone 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

```
# convert a 16 bit unsigned value to
# signed 32 bit value while preserving
# implied sign
ASS P0 = L0
IF P0 >= 8000
  MAT P0 = P0 | FFFF0000
END


# 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 where you will find many code examples as well as ready-to-run presets that you can copy/paste.

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