Deleting Individual HID Inputs

To delete individual inputs one-by-one from the OSC Stream Editor window:

3) Click in the **Port** text box for the input you wish to delete.
4) Click the minus (**-**) symbol at the bottom-left of the window. The item will be deleted.

Erasing the Entire List

The Clear List function will permanently delete all addresses from the Stream Setup window. This operation is not undoable!

To permanently clear the input list:

3) Click the **Clear List** button.
4) Isadora will show a dialog asking if you are sure you want to delete all the items. Click the OK button to confirm your choice and delete all items.

Serial Input/Output

Isadora can transmit and receive data via standard serial (RS-232, RS-485) hardware installed on your computer using the Send Serial Data, Serial In Watcher – Binary and Serial In Watcher - Text actors.

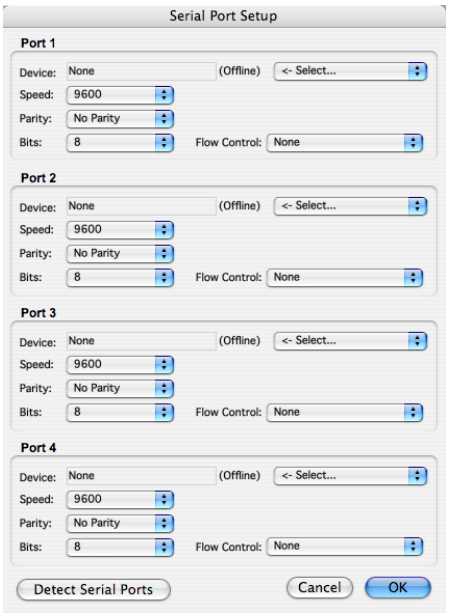
Hardware Interface & Drivers

Before you begin, you must have a hardware serial interface that allows you to connect serial devices to your computer. Before using the interface with Isadora, you must install its drivers. To do this, please follow the installation instructions in the interface's manual. Note: Isadora should not be running when you install the drivers, otherwise the interface may not be recognized.

After you've installed any required drivers, you should connect the install the serial device or connect it to your computer as appropriate.

Serial Port Setup

Ensure that your serial input/output interface is connected to/installed on your computer. Then start Isadora and choose **Communications > Serial Port Setup**. A dialog will appear that looks like this:



This window determines communications settings of Isadora's two ports, and whether or not the port is enabled.

Preparing a Port for Serial Communications:

1) Select the desired output device from the popup menu at the right. After you do so, the name of the device will appear next to the caption **Device** on the left side. It will also indicate if the device is currently online.
2) Set the speed, parity, and number of bits to transmit.
3) If necessary, set the "Flow Control" popup to Hardware to enable hardware handshaking, or Xon/Xoff to enable software handshaking.
4) Repeat if necessary for Ports 2, 3 or 4.
5) Click OK to confirm your settings.

Enabling for Serial Communications:

To enable serial communications, choose **Communications > Enable Serial Ports**. Isadora will report an error if there are any problems initializing the serial ports. Otherwise you can assume that serial communications have been enabled. Note that this setting is saved with the Isadora document. If you save the document with the serial ports enabled, Isadora will attempt to open those ports automatically the next time the document is opened.

Disabling for Serial Communications:

To disable serial communications, choose **Communications > Enable Serial Ports**. Note that this setting is saved with the Isadora document. If you save the document with the serial ports disabled, Isadora will not attempt to open those ports automatically the next time the document is opened.

Receiving Serial Data: (v1.3)

For information on how to receive data from to the serial port, see the documentation for the text and binary versions of Serial In Watcher actor starting on page 455.

Sending Serial Data:

See the documentation for the Send Serial Data actor on Page 453 for information on how to send data to the serial port.

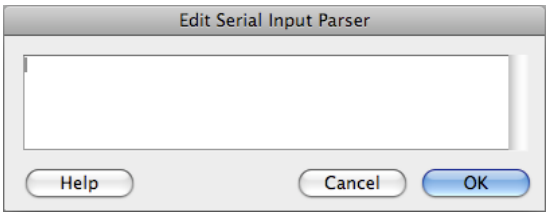
Input Data Parsing – Overview

Several actors allow you to parse incoming data streams from external sources like serial input devices or data acquired over the internet via a TCP/IP connection. All of these actors use a common system for being able to extract values and other meaningful data from within these streams. In Isadora 1.3, these actors include the Serial In Watcher- Binary, Serial In Watcher - Text, TCP InWatcher - Binary, TCP In Watcher - Text. This section explains the parsing system and how to use it.

The first step is to define a "pattern" – a special set of codes used by Isadora to specify the format of the data coming from the external device and how to extract parameters from it.

To do this, you must first get information about the format of the data that will be received by Isadora from your external device. For hardware devices, you can usually find this information in the device owner's manual, on the Internet, or by asking the device vendor. For data streams coming over the internet, you may need to seek online documentation or to examine the data stream itself.

Once you understand the incoming data stream, you can develop a pattern that matches data coming from the device, and tells the plugin what data to assign to what output parameters, if any. To enter or edit the pattern, double-click the actor to open the “input parser” dialog box and enter the pattern in the text field at the top.



The Input Parser Dialog for a Serial In Watcher Actor

Each time a block of data received, an attempt is made to match it against your pattern. If the match is successful, parameters are parsed out of data stream and sent to the output properties defined by the pattern.

For example, say you have a light level sensor attached to your serial port and it sends a continuous stream of messages reporting the light level. The message comes in the form of ASCII text: the “#” sign, followed by a two –digit hexadecimal

followed by a carriage return character. The stream of data might look something like this in a terminal program:

#0A
#0F
#15

Because the data is text based, and because it ends with an “end of line” marker (the carriage return) you would choose the Serial In Watcher - Text actor to read and interpret the data. Within your pattern, you would also define an output parameter (say, "light level") that will be added to the Serial In Watcher - Text actor, and that will output the light level whenever a valid message is received.

In addition to any output parameters you have defined, the "msg rcv" output parameter is always defined and sends a signal out every time a block of data matches your pattern.

Text vs. Binary Actors

There are two forms of each actor that pares input data: a "Text" one and a "Binary" one. The only difference between the two is the text actor reads input data until the some delimiter character is hit, and then matches that data with the pattern, while the binary actor reads input data in fixed size blocks. For both the text and binary actors, *the pattern syntax and matching rules described below are exactly the same*.

The text version of the actor has an input named "eom char". This is the delimiter character. Typically you would set this to a newline or a carriage return but any values are acceptable. The current default is 13 (carriage return).

The binary version of the actor has three inputs "msg len", "timeout", and "reset". The "msg len" parameter specifies the length of a block of data in bytes. Each time this many bytes is read from the input, the data is matched to your pattern. If no bytes arrive within the "timeout" period, then any partial input data is discarded, the pattern is not matched, and the actor continues to wait for new input. The "reset" input can be used to force a reset just as if the timeout period expired.

Data Parsing: Patterns

Patterns are composed of a series of one or more "elements", where an element defines the format of one part of the input data. Pattern elements can match, say, a word or a number in the input data, a sequence of bytes, etc.

There are two flavors of pattern elements: **text** and **binary** (indicated in the table below). **Text** elements are typically used when matching human-readable, text input data (for example, a video player that sends the string "FRAME 2482 SPEED 1.0"). **Binary** elements are used to match bytes of raw data (such as 2 byte integers, or bitfields). You can mix both text and binary elements together in a single pattern. *Text and binary elements can be used interchangeably in both the text and binary versions of the data input parsing actors!*. Again, the only difference between the "text" and "binary" actors is what defines a block of input data.

Between any two text type elements, any amount of whitespace (tabs, spaces, etc, but not the delimiter character) in the input data is ignored. Between any two binary type elements, or a text and binary type element, the pattern must match the input exactly. While this rule sounds a little convoluted, in the end it (hopefully) allows matching text data to be more convenient and intuitive (because you don't have to explicitly write rules to match whitespace between words in text data -- the actor takes care of that for you).

The value of any element can also be assigned to user-defined actor output parameters. The syntax for assigning values to output parameters is detailed below. You define the parameter name and type, and the value of the element is interpreted depending on the parameter type you define.

Data Parsing: Elements

Separate each element in your pattern string with whitespace. Here is the syntax and description for all of the pattern elements you can define. In the syntax column below, **bold** things are literal characters/words that you type, *italic* items are variables and parameters you make up, and square braces [] enclose optional items.

Syntax	Type	Example	Description
" *string* "	Text	"hello"	Case-insensitive string match (example also matched "HELLO", "HeLlO", etc). Question marks and backslashes have special meaning in these strings (see table below).
' *string* '	Text	'hello'	Case-sensitive string match. Question marks and backslashes have special meaning in these strings.
[*n*] [**.** [*m*]] **#** [*n*] [**.** [*m*]] **digits**	Text	8.2# 4 digits # .#	Matches a decimal number. Specifically, matches an optional +/- at the start of a number, and up to *n* digits to the left of the decimal point and *m* digits to the right. If you leave out *n* or *m*, any number of digits matches but keep in mind that if you leave out the period, then this element will **not**

			accept input with a decimal point in it (i.e. ".#" signifies a number with an optional decimal point, of any length, "#" specifies an integer only). The words "digit", "digits", and "#" all mean the same thing, "#" is just a shortcut.
[*n*] **X** [*n*] **hex**	Text	8X x 4 hex	Matches a hexadecimal number, up to *n* digits. If *n* not specified, then matches any number of digits (making the longest possible match). Both "x" and "hex" mean the same thing.
[*n*] **A** [*n*] **letters**	Text	8A 8 letters 4 hex	Matches up to *n* letters (depends on current locale but these are **letters in the alphabet only**, of any case). If *n* not specified, then makes the longest possible match. The keywords "a", "letter", and "letters" all mean the same thing.
[*n*] **C** [*n*] **chars**	Text	4c characters 4 hex	Matches up to *n* characters (any printable character; pretty much any character with a picture on your keyboard except for whitespace, this includes letters, numbers, and punctuation). If *n* not specified, then makes the longest possible match. The keywords "c", "char", "chars", "character", and "characters" all mean the same thing.

[*character_set* **]**	Text	[a-zA-Z] [^ 0-9] [abc 4\r\n]	Matches one or more characters in *character_set* (see below).
[[*character_set* **]]**	Text	[[a-zA-Z]] [[^ 0-9]]	Matches exactly one character in *character_set*.
eol	Text	eol	Matches any sequence of characters (possibly none at all) up to and including a CR/LF, then matches all following CR/LF characters. This can be used to match the remainder of a line of text. Please note that for the text versions of actors that parse input data, the use of the delimiter character makes this element a little bit ambiguous. You may want to avoid this with the text actor for the time being unless you are sure you want to use it... if all you want to do is say "this input data ends with a linebreak", consider using the text actor with the "eom char" input parameter instead.
[*n*] **?** [*n*] **bytes**	Binary	4? 17 bytes	Matches a sequence of *n* bytes of any value. If *n* is left out, 1 is assumed. "?", "byte", and "bytes" all mean the same thing.
{ *byte_set* **}**	Binary	{ 00-20 F0 F1 }	Matches one or more characters in *byte_set* (see below).
{{ *byte_set* **}}**	Binary	{{ 00, 01, 08 }} {{0A}}	Matches exactly one character in *byte_set*.

(*bitfields* **)** [**:** *type*]	Binary	*See below.*	Matches some number of bytes in the input data, and extracts integer bitfields from the matched bytes. This can be used to read bitfields from binary integers. The *type* specifies the endianness of the matched integer, is optional, and is either "B" or "L". Default if not specified is "B". See below for more details.

Data Input Elements: String

Some characters or character sequences in a *string* (inside single or double quotes) have special meaning:

?	This matches any character in the input at all.
\?	Use this to match an actual question mark (since a lone question mark has a special meaning).
\"	A double quote.
\'	A single quote.
\n	A linefeed (ASCII 10).
\r	A carriage return (ASCII 13).
\t	A tab (ASCII 9).
\\	A backslash.

So the element **"?ack\?"** will match any of the following input data (for example): **back? HACK? pAcK?**

Data Input Elements: Character_set

A *character set* consists of a set of characters and character ranges. These are specified inside square braces. If you want to include a literal hyphen in a character set, specify that hyphen first. If you want to invert the character set (i.e. all characters not specified), put a caret (^) as the first character. Here are some examples (these examples are inside single square braces; [[]] double square braces use the same syntax). Note that spaces inside a character set are significant -- they match spaces in the input. Escape sequences like to the ones listed in the table above can be used in character sets to identify special characters (like tabs and newlines) (note that a ? is just a plain old ? in character sets, though, unlike in strings).

[ABCD]	The characters A, B, C, or D.
[A-D]	The characters A, B, C, or D.
[a-zA-Z]	Any lowercase or uppercase letter (English locale, ASCII).
[-xyz]	A hyphen, x, y, or z.
[-0-9]	A hyphen or a numeric digit.
[^a]	Any character except a lowercase a.
[^-0-9]	Any character except hyphens and numeric digits.
["ABCD\r\t]	A space, a double quote, an A, B, C, or D, a carriage return, or a tab.
[^-abc 24M-O]	Anything except a hyphen, a, b, c, space, 2, 4, M, N, O, or tab.

Data Input Elements: byte_set

A *byte_set* is similar in spirit to a character set except you specify 2-digit hexadecimal numbers instead of characters. You can separate individual values with spaces or commas (whichever you prefer, they are treated the same). These appear in curly braces. Here are some examples, in single curly braces (but double curly brace syntax also uses these). To invert the set, specify a caret (^) as the first character.

{ 00 01 }	The bytes 0 or 1.
{ 0A, 0D, 10 }	The bytes 0A, 0D, or 10 (hex).
{ 03-08 F0-FF }	The bytes 03 to 08 or F0 to FF.
{ ^ 30 - 39 }	Anything except the bytes 30 to 39 (hex).
{^30-39}	Same as above, just showing that you can leave out the spaces.

Data Input Elements: bitfields

Frequently, devices with binary formatted data pack multiple values into a single integer, covering only a few bytes. You can use bitfield elements to specify such bitfields and optionally assign the values of certain fields to output parameters. Bitfields are slightly different than the other elements in that the parameter assignment rules described in the following section do not apply. Instead, multiple parameters may be specified in a single bitfield element. For more information on how to define actor output parameters for the other element types, read "Parameter Assignment" below.

Bitfield syntax is like so:

([*name* **=**] *start* **-** *end* **,** [*name* **=**] *start* **-** *end* **,** ... **)** [**: B** | **L**]

The *name* is optional and defines an actor output parameter to assign the value of the bitfield to. The output parameter type will be an integer, and the rules for name are the same as described in "Parameter Assignment" below (must not start with number, etc). The *start* and *end* values specify the start and ending bit index into the

integer, with 0 being the least-significant bit. The number of bytes of input matched is implied by the highest bit index here! So if the highest bit index is 15, then this actor assumes the integer size is 16 bits, and so matches 2 bytes of input data (1 byte is 8 bits). The "B" and "L" on the end are optional and specify the byte endianness of the binary integer: big-endian or little-endian, respectively. If neither is specified, "B" is assumed. The input bytes are read, converted to an integer value according to the specified endianness, and then bitfield values are extracted. Note that multiple fields may cover the same bits, if you want.

This syntax sounds complicated but it's not. Hopefully some examples will clear it up. Say you are using the Monome button box (see http://wiki.monome.org/view/SerialProtocol). The Monome button box has 2 output messages, each are 2 bytes long. To extract the values of "press" messages:

(address = 15-12 , state = 11-8 , x = 7-4 , y = 3-0)

If you don't care about "state", you can just leave it out entirely:

(address = 15-12 , x = 7-4 , y = 3-0)

In both cases, the maximum bit index is 15 so 2 bytes of input data are matched, and those 2 bytes are treated as a big-endian integer. Now let's say you don't care about "address" or "state". You're still going to want to leave a placeholder there so that Isadora knows you need 2 bytes:

(11-10 , x = 7-4 , y = 3-0)

The "11-10" is a silly value but illustrates a point: The number of bits required is rounded up to the nearest byte, and also it's OK if there are bits that aren't part of bitfields. 11 rounded up to the nearest byte is 16 bits. Bits 12-15 and 8-9 are unused. Here is a slightly more complex example that specifies a little-endian 64-bit (8 byte) integer (you are not limited to 32 bits -- the highest bit index you can use is unlimited, however if a single field spans more than 32 bits, you will likely encounter some problems). Note the overlapping bitfields and the 1-bit value "c":

(a=60-40, b=20-15, bb=18-15, c=14-14, d=13-2) : L

As mentioned below, this actor has no built-in support for filtering out certain values (such as only responding to Monome button messages with a certain "address"). You will have to use the techniques described below to accomplish this.

Data Input: Parameter Assignment

Pattern elements match specific portions of the input data. Sometimes you will want this actor to output the values that it matches. To do this, use the syntax **name:type=element** where **name** is any parameter name you define, **type** describes the parameter type and some details about the format of the matched data, and **element** is the pattern element as described above (can be any element except a bitfield). For example, **light_level:integer=8 digits** defines an actor output parameter named "light level" with an integer type, matches up to 8 digits in the

input, and converts the matched digits to an integer, outputting the value from the actor.

Parameter names may consist of letters, numbers, and underscores, but must not start with a number ("_value" and "something2" are valid, "2cool" is not valid). The type specifies the type of the actor's output parameter, and for certain binary-flavor elements, describes how the binary data is interpreted. The following types are valid:

string	A text string.
float	A decimal number. Please note that binary floating-point numbers are not currently supported.
integer	An integer. For binary types, the same as "integer".
binteger	An integer. For binary types, input treated as big-endian integer.
linteger	An integer. For binary types, input treated as little-endian integer.

You do not need to specify the entire type, you only need to specify a unique prefix, so "str" is the same as string, "int" is the same as "integer", "f" is the same as "float", etc. Here are some examples:

value : integer = 3 digits	Matches up to 3 digits, assigns to integer "value".
mixlevel:float=.#	Matches a decimal number, assigns to float "mixlevel".
position:float=8.2#	Matches a decimal number, assigns to float "position".
word:string="hello"	Matches the string "hello" and assigns to string parameter "word".
id:string=3 digits	Matches up to 3 digits, assigns to *string* "id".
choice:string=[[a-z]]	Matches a single lowercase letter, assigns to string "choice".
color:int=?	Matches a single byte of any value, assigns to integer "color".
volume:bint=4 bytes	Matches 4 bytes of any value, treated as big-endian integer and assigned to "volume".
fname:string=letters	Matches a sequence of 1 or more letters, assigns to string "fname".

If you assign a text element to an integer parameter, the actor attempts to convert as much as possible to an integer. So if you have "a:int=characters", and the input is "123abc", **a** will be assigned a value of **123**. Similarly, if you have "b:int=letters", **b** will always have the value **0**.

Some technical info: You are not limited to standard machine sizes for binary integers. You can do "a:int=3?" to match 3 bytes, treat as a little-endian 3-byte integer, and assign the value to **a**. You can also do "b:int=15?" to match 15 bytes, but keep in mind that internally, the actor can't really represent values wider than 4 bytes, so the matched value will be truncated if it is too large.

Now, sometimes you will want to only respond to input matches if certain parameters you define have certain values. For example, say a text-mode device sends out a single character at the start of each line specifying the line type. This actor currently does not have built-in support for doing that kind of filtering. You have two options to do this. The first option is, if possible, to make sure that your pattern elements are specified in such a way that only the data you are interested in is matched (for example, if you are only interested in lines where the first word is "a", the specify an "a" element to take care of this). Another option is to combine this actor with other Isadora actors that filter out certain values; so you might take the first character at the start of each line and assign it to an output parameter, and then run that output parameter into a Comparator actor to check for the values you want to respond to.

Data Input: Examples

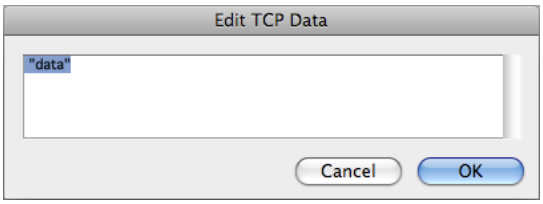
"?????"	Matches any 5 bytes.
[^*] "*"	Matches any string ending in an asterisk.
'Hello'	Matches the string "Hello", case-sensitive.
"moveto" x:float=.# y:float=.# z:float=.#	Matches strings like **MoveTo 6.8 -2.3 89** or **MOVETO +9 -13. +.1**.
a:int=2 digits [^|] "|" b:int=2 digits [^|] "|" c:int=2 digits	Matches a string such as **43 apples | 8 oranges | 98 monkeys**, assigning the values 43, 8, and 98 to the integer output parameters a, b, and c, respectively.
hue:int=2x "," sat:int=2x "," val:int=2x	Matches a string like **4F , 8B , 9A**, converting the hexadecimal values to integers and assigning to output parameters.
"#" r:int=2x g:int=2x b:int=2x	Match and parse the values of an HTML color such as **#FF0020**.

Data Output Formatting

Several actors allow you to format one or more input parameters into a resulting string of text characters. The system used to accomplish that formatting is the same for each of these actors. In Isadora 1.3, the actors include the Text Formatter, Send Serial Data, TCP Send Data.

First, you should set the params input on the actor, as this will add input parameters (param 1, param 2, etc.) that will be used when generating the final output. Up to nine parameters may be accepted by any one actor.

To edit the formatting specifier, double-click this actor's icon and an editor dialog will appear.



In it's simplest form the editor will accept a text string enclosed in single quotes. In the example above, each time the actor was triggered to generate output, the four ASCII characters 'd', 'a', 't', 'a' would be generated.

You may also include two-digital hexadecimal values, *outside* the quotes to specify characters that cannot be represented on a keyboard. For instance, the formatting specifier

"hello" 0D 0A

would send the ASCII characters "hello" followed by a Carriage Return (hex 0D, decimal 13) and a Line Feed character (hex 0A, decimal 10).

To format and include a values sent to one of the actor's input properties, you would use the notation ***Px***, where x is a number from 1 to 9, indicating which parameter you wish to include. When using this notation for numeric parameters, there are a number of extra options

Px	Use the default formatting. For integer numbers, output the ASCII text of the number in decimal; for numbers with decimal points, output the ASCII text of the number and all the digits after the decimal point; for text inputs, output the text itself Examples: The integer 12 outputs the characters '1', '2' The floating point number 3.141 outputs the characters '3', '.', '1', '4', '1' The text "hi!" outputs the characters 'h', 'i', '!'.
Px:n.m	Output the number, with a maximum of ***n*** digits to the left of the decimal point and ***m*** digits to the right. If the input parameter is text, ignore ***n.m*** and just output the text.
Px:Zn.m	Same as above, but add leading zeros to ensure a total of ***n*** digits appear to the left of the decimal point.
Px:nX	Output the ASCII representation of the number as *n* hexadecimal digits. If the input parameter is a floating point number, the digits after the decimal are ignored. If the input parameter is text, ignore the ***nX*** and just output the text. Example: Px:2X applied to the decimal value 254 outputs 'F', E'
Px:ZnX	Same as above, but add leading zeros to ensure a total of *n* digits.
Px:C	Output the character as a single byte of data.

	Examples: The number 65 gives 'A' The number 13 gives a carriage return character

To send ASCII text, you must enclose the text in double-quotes, i.e. to send the word **hello** you would enter "**hello"**. When specifying ASCII text, you can send various control characters by using one of the special "escape" sequences shown below.

\a	0x07 (Bell)
\b	0x08 (Backspace)
\f	0x0C (Form Feed)
\n	0x0A (New Line)
\r	0x0D (Carriage Return)
\t	0x09 (Tab)
\\	Backslash
\"	Double-quote
\0	0x00 (Null)

Note that to include a double-quote or a backslash inside of ASCII text, you must precede it with a backslash as well.

Some further examples:

0E 11 C0 – sends the three hexadecimal bytes, 0E 11 C0, which are decimal 14, 17 and 192 respectively.

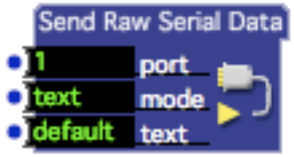
"pl\r" – sends the characters 'p' and 'l' followed by a carriage return (hex 0D, decimal 13)

05 "hello" 0A 0D – sends eight bytes, starting with 5, then the characters 'h', 'e', 'l', 'l', 'o', followed by hex 0A and 0D (which are 10 and 13 in decimal.)

FF "?" P1:C P2:C – sends four bytes, starting with hexadecimal FF (255 decimal) followed by the character "?", and ending with two bytes that give the value input parameters 1 and 2 (*Param 1, Param 2*) respectively.

- **param 1, param 2, etc:** variable value parameters that will be inserted into the message. See description above for more about using variable parameters.

Send Raw Serial Data (v1.3)

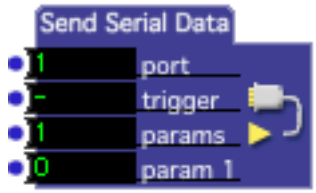


Sends raw data to the specified serial port. This actor has two modes: 'text' and 'hex'.

Input Properties

- **port**: Specifies the serial port to which the data will be sent when triggered, from 1 to 8. This port is configured using the Serial Port Setup dialog found in the Communications menu.
- **mode**: When set to 'text', the text received at the text input is sent directly to the serial port. When set to 'hex', the text must consist of hexidecimal characters (0-9, A-F). Each pair of characters is converted to its single byte equivalent and the result sent to the serial port. The latter option is essential if you need to send data that includes the value 0 within the block of data, as this is the marker for the end of a text string.
- **text**: The text to send to the serial port, interpreted according to the 'mode' setting.

Send Serial Data (v1.1)



Formats and sends data to the specified serial port.

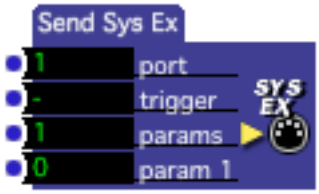
To specify the precise format of the data sent to the serial port, you must double-click this actor and change its formatting specifier. To learn more about how to control formatting, see "Data Output Formatting" on page 249.

Input Properties

- **port**: Specifies the serial port to which the data will be sent. This port is configured using the Serial Port Setup dialog found in the Communications menu.
- **trigger**: When a trigger is received on this port, the data is sent to the specified port.
- **params**: The number of variable parameter inputs. Increasing this number adds parameter inputs, decreasing it removes them.
- **param 1, param 2, etc:** variable value parameters that will be inserted into the output data. See "Data Output Formatting" on page 260 for more information on how to format the data from these inputs.

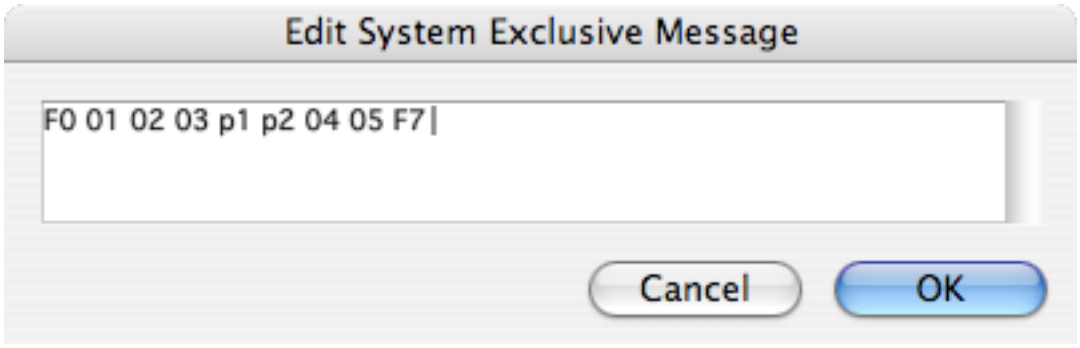
  The param inputs of this actor are mutable. Each input will change its data type to match that of the first link made to it. The inputs will become mutable again if all of its links are disconnected. (To learn more about mutable inputs and outputs, please "Mutable Inputs and Outputs" on page 107.)

Send Sys Ex



Sends a MIDI System Exclusive message, which may contain variable values, each time a trigger is received.

To specify the contents of the System Exclusive message, double-click this actor's icon. A dialog allowing you to edit the System Exclusive message will appear:



Messages must be entered in hexadecimal, starting with a hex F0 (start of exclusive) and ending with hex F7 (end of exclusive).

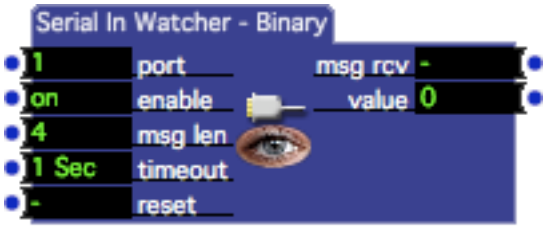
You can insert up to nine variable values in the message by using the codes P1 – P9 instead of a valid hexadecimal number. After you do this, you should then set the

params input to the desired number of variable parameters, and the specified number of parameter value inputs will appear (e.g, param 1, param 2, etc.) When a message is sent, the codes P1 – P9 specified in the editor will be replaced with the current value of the matching parameter inputs (i.e. "P2" in the message above is replaced by the value of the param 2 input).

Input Properties

- **port**: The MIDI port on which the message will be sent. These port numbers correspond to the Destinations shown in the MIDI Setup dialog.
- **trigger**: Sends a MIDI System Exclusive message each time a trigger is received on this input.
- **params**: The number of variable parameter inputs. Increasing this number adds parameter inputs, decreasing it removes them.
- **param 1, param 2, etc:** variable value parameters that will be inserted into the System Exclusive message. See description above for more about using variable parameters.

Serial In Watcher Binary



Reads a fixed length binary data block from the specified serial port using a user-specified pattern matching specification.

(Note: To read data consisting of variable length messages marked by a delimiter, use the Serial In Watcher - Text actor.)

Values within the data are parsed and output from this actor according to a user-specified pattern-matching specifier. To edit this specifier, double-click this actor's icon, and the editor will open. For documentation on parsing input streams, see "Data Input Parsing" on page 251

Input Properties

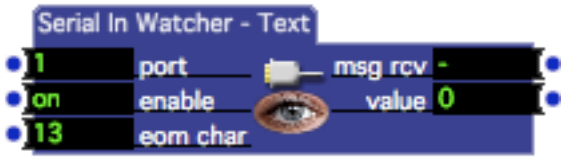
- **port**: Specifies the serial port from which to receive data.
- **enable**: When turned on, reads all data from the serial port and attempts to match the specified pattern. When turned off, ignores data from the serial port. This should be used with caution as enabling this input in the middle of a message may result in the data being read erroneously.

- **msg len**: The length of the data blocks to be received by this watcher. Each time the specified number of bytes arrives on the specified serial port, an attempt will be made to use the pattern matching specifier to decode the incoming data.
- **timeout:** Specifies a timeout for the input buffer. If more than this amount of time passes between receiving any two bytes, the input buffer will be cleared and the incoming message length count reset to zero.. This is to help avoid erroneous messages should the serial input cable be accidentally disconnected, etc.
- **reset:** Clears the input buffer when triggered and resets incoming message length count is reset to 0.

Output Properties

- **trigger**.: Sends a trigger when a valid message has been parsed and it matches the pattern specified by the pattern-matching specifier.
- **value outputs**.: The output for one of the parsed values. (The names and number of these outputs are based on the pattern-matching specifier.)

Serial In Watcher Text



Reads a variable length data block from the specified serial port using a user-specified pattern matching specification.

(Note: To read data consisting of fixed length messages with no delimiter, use the Serial In Watcher - Binary actor.)

Values within the data are parsed and output from this actor according to a user-specified pattern-matching specifier. To edit this specifier, double-click this actor's icon, and the editor will open. For documentation on parsing input streams, see "Data Input Parsing" on page 251

Input Properties

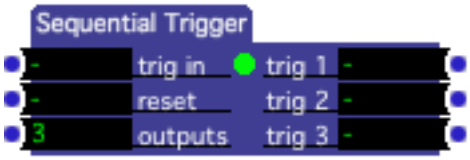
- **port**: Specifies the serial port from which to receive data.
- **enable**: When turned on, reads all data from the serial port and attempts to match the specified pattern. When turned off, ignores data from the serial port. This should be used with caution as enabling this input in the middle of a message may result in the data being read erroneously.
- **eom char**: The character that signifies the end of a message (i.e. eom = end of message.) Whenever this character is received, the data accumulated in the buffer

is parsed using the pattern-matching specifier, and values are sent to the outputs if a match is successfully made.

Output Properties

- **trigger**.: Sends a trigger when a valid message has been parsed and it matches the pattern specified by the pattern-matching specifier.
- **value outputs**.: The output for one of the parsed values. (The names and number of these outputs are based on the pattern-matching specifier.)

Sequential Trigger



Sends triggers out of multiple outputs sequentially.

Input Properties

- **trig in**: Sends a trigger to the next trigger output. The first trigger that is received after the actor is activated will be sent to the trig 1 output. Each subsequent input trigger will send a trigger out of the next available output. When the last output has been triggered, the sequence starts over at the beginning.
- **reset**: Resets the sequence over so that the next trigger received at the trig in input will send a trigger to the trig 1 output.
- **outputs**: determines the number of trigger outputs. Setting this value to a higher number will add more trigger outputs to the actor, setting it lower will remove trigger outputs.

Output Properties

- **trig 1, trig 2, etc**.: The output triggers.
-