MIDIio X-Tra 1.0

Last Updated 1st August 2013.

What is MIDIio?

MIDIio is an X-tra (plug-in) for Adobe Director that lets you send and receive MIDI messages in real-time. MIDIio provides MIDI input and output functions that enable Lingo control of external MIDI synthesizers or other MIDI devices, and Lingo response to externally generated MIDI messages.

MIDIio is available for Mac OS X and Windows. An earlier version supporting Mac OS 9 is available upon request

Availability

Information about the latest release is available from the MIDIio home page at:

http://www.rossbencina.com/midiio-x-tra

You can contact the author at rossb@audiomulch.com

Licensing and Pricing

MIDIio is distributed in an unregistered form that only allows it to be used within the Director authoring environment.

A developer license is available for US$99 per platform. This permits a single user on a single machine to author using MIDIio. This license includes unlimited runtime distribution rights on the licensed platform(s). When a license is purchased a key will be supplied which unlocks MIDIio to function outside the Director authoring environment.

No Warranty

MIDIio X-tra (The Software) is provided with no warranty:

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Lingo Methods Overview

The MIDIio X-tra implements the following methods. Note that all methods require the X-tra instance value as the first parameter (not shown).

```
Initialisation
Register( name, regCode )
Init()                    --call this first, returns error information

Timer
GetTime()                 -- returns internal timer time in milliseconds
ResetTimer()              -- resets internal timer to 0

MIDI Input
GetInputPorts()           -- returns a list of available MIDI input ports
OpenInput( portNumber, bufferSize )           -- begin MIDI input
CloseInput()                                  -- terminate MIDI input

SetSystemFilter( messageTypes )               --filter system messages
SetChannelFilter( messageTypes, channels )    --filter channel messages

MessagesPending()   -- returns true if the input queue contains messages
GetNextMessage()    -- retrieve next message from the input queue
GetPendingMessages()      -- retrieve a list of all pending messages
InputOverflowed()         -- returns true if the input queue has overflowed
FlushInput()              -- flush all pending messages from the input queue

MIDI Output
GetOutputPorts()          -- returns a list of available MIDI output ports
OpenOutput( portNumber )  -- begin MIDI output
CloseOutput()             -- terminate MIDI output

SendMessage( message )    -- send one MIDI message
```

Lingo Methods Reference

X-tra Initialisation

`Init()`

`Returns: property list: error information`

Call **`Init()`** directly after instantiating a new MIDIio object (or after calling `Register()` if you are a licensed developer). This method performs initialisation. You should check the result for possible error information.

`Register( string: name, string: regCode )`

`Returns: property list: error information`

If you are a licensed developer, call **`Register()`** with your registration details directly after instantiating a new MIDIio object ***before*** calling `Init()`. `Register()` unlocks the X-tra so that it will operate within projectors and shockwave movies. Unregistered copies of MIDIio only operate within the Director authoring environment.

An example of how to call `Register()` is provided when you purchase a licence.

Timer

`GetTime()`

`Returns: integer: current time in milliseconds`

`ResetTimer()`

`Returns: nothing`

MIDIio maintains a timer to time-stamp incoming MIDI messages and to be used as a time reference within Lingo scripts that need to send MIDI notes with specific rhythms.

`GetTime()` returns the current timer time in milliseconds. The timer begins at time 0 when the X-tra is instantiated and may be reset to 0 at any time by calling **`ResetTimer()`**. The accuracy of this timer may vary from platform to platform – with a worst case granularity of 20ms.

MIDI Input

Listing, Opening and Closing MIDI Input Ports

`GetInputPorts()`

`Returns: list of strings: available MIDI input port names`

`OpenInput( integer: portNumber, integer: bufferSize )`

`Returns: property list: error information`

`CloseInput()`

`Returns: property list: error information`

You need to open a MIDI input port to receive MIDI messages. A MIDIio object instance can only have one MIDI input port open at any time. To use multiple ports simultaneously you can instantiate multiple instances of the MIDIio X-tra.

The number and names of available input ports will vary from system to system. For example, Apple's CoreMidi allows the user to create any number of virtual ports. On Windows each installed MIDI device will appear as one or more ports.

`GetInputPorts()` returns a list containing the names of all available MIDI input ports.

`OpenInput()` opens the specified MIDI input port and immediately begins queuing MIDI messages. The *`portNumber`* parameter selects the port to open. `PortNumber` is the 1-based index of the desired port in the port list returned by `GetInputPorts()`. For example, if `GetInputPorts()` returns `["MOTU Input", "Midisport Input"]`, `OpenInput(2)` will open the "`Midisport Input`" port. The *`bufferSize`* parameter specifies the size (in bytes) of the input queue used to buffer events before they are passed to Lingo by a call to `GetNextEvent()`. Note than each full MIDI message is stored with a 4-byte time stamp.

Call **`CloseInput()`** to close the input port.

Receiving Incoming MIDI Messages

`MessagesPending()`

`Returns: boolean: *True if there are any pending input messages*`

`GetNextMessage()`

`Returns: property list: *one MIDI message*`

`GetPendingMessages()`

`Returns: list of property lists: *list of MIDI messages*`

`InputOverflowed()`

`Returns: boolean: *True if the input buffer has overflowed*`

`FlushInput()`

`Returns: nothing`

MIDIio places incoming MIDI messages into an internal input queue (buffer). **`MessagesPending()`** returns True if the input queue contains one or more messages to process.

`GetNextMessage()` returns the next available MIDI message from the input queue and removes the message from the queue. Each message is represented by a property list. See the MIDI Message Data Format section below for an explanation of the format of the property list returned by `GetNextMessage()`. If no messages are pending `GetNextMessage()` will return an empty list.

`GetPendingMessages()` returns a list containing all pending messages in the input queue and removes these messages from the queue.

If a large number of MIDI messages are received between calls to `GetNextMessage()` it is possible that the input queue will overflow and some MIDI messages will be lost. The **`InputOverflowed()`** method will return True if the input queue has overflowed and False otherwise.

`FlushInput()` clears all pending events from the input queue.

Note: MIDIio translates incoming note-on messages with a value of 0 to note-off messages.

Note: to receive MIDI messages, first open an input port by calling `OpenInput()`.

Filtering Incoming MIDI Messages

SetSystemFilter(list: *allowableMessageTypes*)

Returns: property list: *error information*

SetChannelFilter(list: *allowableMessageTypes*, *channels*)

Returns: property list: *error information*

SetSystemFilter() and **SetChannelFilter()** can be used to specify which received MIDI messages will be passed through to GetNextMessage() and GetPendingMessages(). This can be useful to limit the amount of data that has to be processed by Lingo. You should set the filters to only pass the messages that you intend to respond to.

Use SetSystemFilter() to specify which MIDI system messages will be passed. Use SetChannelFilter() to specify which MIDI channel messages will be passed. You can optionally filter different channel messages on each MIDI channel.

The *allowableMessageTypes* parameter is either a single message type symbol, or a list of message type symbols. These symbols may be any of those returned by GetNextMessage() plus a number of special symbols that represent families of related messages. See the MIDI Message Data Format section below for a complete list of messages and their corresponding families.

SetSystemFilter() accepts the following standard message types in the *allowableMessageTypes* list: #clock, #start, #stop, #continue, #activeSensing, #systemReset, #songPositionPointer, #songSelect, #tuneRequest, #systemExclusive. Additionally, the following message family specifiers are accepted: #none, #all, #realTime, #common.

SetChannelFilter() accepts the following standard message types in the *allowableMessageTypes* list: #noteOn, #noteOff, #polyKeyPressure, #controlChange, #programChange, #channelPressure, #pitchBend, #localControlOff, #localControlOn, #allNotesOff, #omniModeOff, #omniModOn, #monoModeOn, #polyModeOn. Additionally, the following message family specifiers are accepted: #none, #all, #note, #voice, #mode.

MIDIio can filter different MIDI messages for different input channels, the *channels* parameter selects which MIDI channel(s) a call to SetChannelFilter() applies to. *Channels* can be an integer from 1 to 16 specifying a single channel, a list of channel numbers, or 0 to indicate that the call applies to all channels. The filter settings for channels that are not addressed by the *channels* parameter are left unchanged. This means that allowing messages from only one channel requires two calls to SetChannelFilter(), one call to mask all messages on all channels, and one call to select the required messages on the required channel (see Example 2 below).

Filtering Incoming MIDI Messages Example 1

```
-- allow only note on/note off messages on all channels:
MidiObj.SetSystemFilter([]) -- no system message
MidiObj.SetChannelFilter( [#notes], 0 ) -- note on/off on all channels
```

Filtering Incoming MIDI Messages Example 2

```
-- allow system real-time messages plus note on/off and pitchBend messages on
channels 3 and 4:
MidiObj.SetSytemFilter( #systemRealTime ) -- list not needed for single syms.
MidiObj.SetChannelFilter( [], 0 ) -- reset all channels to pass nothing
MidiObj.SetChannelFilter( [#note, #pitchBend], [3,4] )
```

MIDI Output

Listing, Opening and Closing MIDI Output Ports

`GetOutputPorts()`

`Returns: list of strings: available MIDI output port names`

`OpenOutput( integer: portNumber )`

`Returns: property list: error information`

`CloseOutput()`

`Returns: property list: error information`

You need to open a MIDI output port to send MIDI messages. A MIDIio object instance can only have one MIDI output port open at any time. To use multiple ports simultaneously you can instantiate multiple instances of the MIDIio X-tra.

The number and names of available MIDI output ports may vary from system to system.

`GetOutputPorts()` returns a list containing the names of all available MIDI output ports.

`OpenOutput()` opens the specified MIDI output port. The *`portNumber`* parameter selects the port to open. `PortNumber` is the 1-based index of the desired port in the port list returned by `GetOutputPorts()`. For example, if `GetOutputPorts()` returns `["MOTU Output", "Midisport Output"]`, `OpenOutput(2)` will open the `"Midisport Output"` port.

Call **`CloseOutput()`** to close the output port.

Sending MIDI Messages

`SendMessage( property list: `*`message`*` )`

`Returns: property list: `*`error information`*

Call **`SendMessage()`** to send a MIDI message. The message can be specified as a property list or as an ordered list of values. See the MIDI Message Data Format section for an explanation of the message formats accepted by the MIDIio X-tra.

Note: to send MIDI messages, first open an output port by calling `OpenOutput()`.

MIDI Message Data Format

`GetNextMessage()`, `GetPendingMessages()`, and `SendMessage()` receive and transmit MIDI messages using a specially formatted property list. The first property in the list is always the *`type`* property, which contains a symbol identifying the message type. Subsequent properties differ depending on the value of the *`type`* property.

MIDI events returned by `GetNextMessage()` contain a *`timeStamp`* property as their final element. If present, the *`timeStamp`* property is ignored by `SendMessage()`. The timestamp indicates the time that the event was received relative to the MIDIio timer (see `GetTime()` and `ResetTimer()`).

The table below provides a complete list of valid values for the *`type`* property and required additional properties for each message type. All properties other than *`type`* are integers, with the exception of the *`data`* property of `#systemExclusive` messages which is a list of integers. The value of *`channel`* properties range from 1 to 16, all other properties range from 0 to 127 unless otherwise indicated.

Important Note: the specific ordering of elements in a MIDI message property list is ***required*** for calls to `SendEvent()` and is guaranteed for values returned from `GetNextMessage()` and `GetPendingMessages()`. This permits the use of the more efficient ordered array operations `getAt()` and `[ ]`.

Table: MIDI Message Property List Keys

Message type	Additional properties	Family(s)
Channel voice messages		
#noteOn	#channel, #number, #velocity	#note, #voice
#noteOff	#channel, #number, #velocity	#note, #voice
#polyKeyPressure	#channel, #number, #pressure	#voice
#controlChange	#channel, #number, #value	#voice
#programChange	#channel, #number	#voice
#channelPressure	#channel, #pressure	#voice
#pitchBend	#channel, #amount (+/-2^13)	#voice
Channel mode messages		
#localControlOff	#channel	#mode
#localControlOn	#channel	#mode
#allNotesOff	#channel	#mode
#omniModeOff	#channel	#mode
#omniModOn	#channel	#mode
#monoModeOn	#channel, #numChannels	#mode
#polyModeOn	#channel	#mode
System real-time messages		
#clock		#realTime
#start		#realTime
#stop		#realTime
#continue		#realTime
#activeSensing		#realTime
#systemReset		#realTime
System common messages		
#songPositionPointer	#position	#common
#songSelect	#number	#common
#tuneRequest		#common
#mtcQuarterFrame	#subFrame (0-7), #data (0-15)	#common
System exclusive		
#systemExclusive	#manufacturerID, #data	#common

MIDI Message Format Examples

Note on message, channel 3, note 60, velocity 128 at time 2048:

```
[ #type:#noteOn, #channel:3, #keyNumber:60, #velocity:128,
#timestamp:2048 ]
```

Clock at time 45670:

```
[ #type:#clock, timestamp:45670 ]
```

Use of property lists allows the use of dot operator syntax in Director 7 and later, for example:

```
msg = GetNextMessage()
If msg.type = #noteOn then
       HandleNoteOn(msg.channel, msg.keyNumber )
End if
```

Error Information Format

Methods that return error information return a property list with two items: `#code` (an integer) and `#text` (a string). A non-zero `#code` value indicates an error.