

MIDIio X-Tra 1.0

Copyright ©1999 Ross Bencina. All rights reserved.

Last Modified 3rd November 1999.

Please note that this is a preliminary document and may change without notice.

Overview

MIDIio is a Macromedia™ Open Architecture Scripting X-tra for real-time input and output of MIDI messages. MIDIio provides MIDI input and output functions to enable Lingo control of external MIDI synthesizers or other MIDI devices, and Lingo response to externally generated MIDI messages.

MIDIio is developed for PPC Macintosh and 32bit Windows computers. The Macintosh version of MIDIio requires Opcode's OMS (Open Music System) version 2.0 or later (available from the Opcode web site.)

Availability

Information about the latest releases is available from the MIDIio home page at:
<http://www.audiomulch.com/midiio/>
or you can contact the author at rossb@audiomulch.com.

Licensing and Pricing

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

A developer license will be 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 will allow MIDIio to function outside the Director authoring environment.

Lingo Methods

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

```
Register( name, regCode )
```

Initialisation

```
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
```

MIDI Message Data Format

Error Information Format

Register(string name, string regCode)

Returns: property list, error information

Register() unlocks the X-tra so it will operate within projectors and shockwave movies. Unregistered copies of MIDIio only operate within the Director authoring environment.

Init()

Returns: property list, error information

Call this method directly after instantiating a new MIDIio object. This method performs initialisation. On the Macintosh it checks that OMS is present and returns an error if it isn't.

GetTime()

Returns: integer, current time in milliseconds

ResetTimer()

Returns: nothing

MIDIio maintains a timer to timestamp incoming MIDI messages and to be used as a time reference within Lingo scripts that need to play notes with specific rhythms. The current timer time can be retrieved by a call to GetTime() and is measured in milliseconds. The timer begins at time 0 when the X-tra is created and may be reset to 0 at any time by a call to ResetTimer(). The accuracy of this timer may vary from platform to platform – with a worst case granularity of 20ms.

GetInputPorts()

Returns: list of strings, available input port names

OpenInput(integer portNumber, integer bufferSize)

Returns: property list, error information

CloseInput()

Returns: property list, error information

Before MIDIio can receive MIDI messages an input port must be opened. A single instance of a MIDIio object allows only one input port to be opened at any time, multiple ports may be opened by using multiple instances of MIDIio. The number of available input ports may vary from system to system – OMS allows the user can create any number of virtual ports. On Windows there is one or more ports for each installed MIDI device. GetInputPorts() returns a list containing the names of all available input ports.

OpenInput() opens a specific port for input and immediately begins queuing MIDI messages. The *portNumber* parameter selects the port to open, and is the 1 based index of the desired port in the port list returned by GetInputPorts(). For example if GetInputPorts() returns ["modem", "printer"], Open(2) will open the printer port. The *bufferSize* parameter specifies the size of the input queue used to buffer events before they are processed by a call to GetNextEvent() – note that each full MIDI message is also stored with a 4-byte time stamp.

Call CloseInput() to close the input port.

SetSystemFilter(list allowableMessageTypes)

Returns: property list, error information

SetChannelFilter(list allowableMessageTypes, channels)

Returns: property list, error information

SetSystemFilter() and SetChannelFilter() may be used to specify which received MIDI messages will be passed to Lingo by GetNextMessage(). Use SetSystemFilter() to specify which system messages will be passed. Use SetChannelFilter() to specify which channel messages will be passed.

The *allowableMessageTypes* parameter is a list of message type symbols. These symbols may be any of those returned by GetNextMessage() plus a number of special symbols representing families of related messages. See the **MIDI Message Data Format** section 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 not specified by

the *channels* parameter are left unchanged, this means that allowing messages from only one channel requires two calls to `SetChannelFilter()`, one to mask all messages on all channels, and one to select the required messages on the required channel (see example 2 below.)

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

Example 2

-- allow system real-time messages plus note on/off and pitchBend messages on channel 3 and 4...

`MidiObj.SetSystemFilter([#systemRealTime])`

`MidiObj.SetChannelFilter([], 0)` –clear filters on all channels

`MidiObj.SetChannelFilter([#note, #pitchBend], [3,4])`

MessagesPending()

Returns: boolean, *are there any pending input messages?*

GetNextMessage()

Returns: property list, *MIDI message*

GetPendingMessages()

Returns: list of property lists, *MIDI messages*

InputOverflowed()

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

FlushInput()

Returns: nothing

MIDIio maintains an input queue into which it places each incoming message.

`MessagesPending()` returns true if the input queue contains messages to process.

`GetNextMessage()` returns the next available message in the input queue in a property list. If no messages are pending `GetNextMessage()` will return an empty list. See **MIDI Message**

Data Format for an explanation of the format of the property list returned by

`GetNextMessage()`. MIDIio translates incoming note on messages with a value of 0 to note off messages. `GetPendingMessages()` returns a list of all pending messages in the input queue.

If a large amount of MIDI data is received between calls to `GetNextMessage()` it is possible that some events will be lost. The `InputOverflowed()` function will return `True` if the input buffer has overflowed.

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

GetOutputPorts()

Returns: list of strings, available output port names

OpenOutput(integer portNumber)

Returns: property list, error information

CloseOutput()

Returns: property list, error information

Before MIDIio can transmit MIDI information an output port must be opened. An instance of the MIDIio X-Tra may open a single output port. The number of available output ports may vary from system to system. `GetOutputPorts()` returns a list containing the names of all output ports available on the current system.

`OpenOutput()` allows one of the available ports to be opened for output – the *portNumber* parameter is the 1 based index of the desired port in the port list returned by `GetOutputPorts()`. For example if `GetOutputPorts()` returns ["modem", "printer"], `Open(2)` will open the printer port.

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

SendMessage(property list message)

Returns: property list, error information

`SendMessage()` also accepts an ordered list of values.

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 and contains a symbol identifying the message type of the message. Subsequent properties differ according to the value of the *type* property. MIDI events returned by `GetNextMessage()` contain a *timeStamp* property as their final element, the *timeStamp* property (if present) is ignored by `SendMessage()`. The timestamp indicates the time the event was received relative to the MIDIio timer (see `GetTime()` and `ResetTimer()`.)

The following is a complete list of valid values for the *type* property, and required additional properties for individual message types. 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 shown.

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	#realTime
#songSelect	number	#realTime
#tuneRequest		#realTime
system exclusive		
#systemExclusive	manufacturerID,	data

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
```

Note that the order of elements in a MIDI message property list is enforced for calls to `GetNextMessage()`, `GetPendingMessages()` and `SendEvent()` allowing the use of the more efficient ordered array operations `getAt()` and `[]`.

Error Information Format

Methods that return error information return a property list with two items: `#code` (an integer) and `#text` (a string), the following errors are currently defined:

Code	Description
0	No Error

A non-zero code value indicates an error.

TODO:

- Clean up description of get input ports and get output ports
- Check and document the possibility of omitting parameters meaning `#none` in `Set*Filter`, or using an empty list to mean none, or using a symbol instead of a list of symbols when only one symbol needs to be passed. Perhaps allow this in `SendMessage` too as in `SendMessage(#clock)`.