Expert Sleepers
MIDI & OSC Scripting Manual
v1.0.0

Contents

MIDI & OSC Lua scripting

For most Expert Sleepers plug-ins it is possible to extend the plug-in's MIDI & OSC functionality via user-writeable scripts. Indeed, the plug-in's standard MIDI functionality is usually implemented using such a script, which you can use as reference for your customisations. Refer to the user manual of the individual plug-ins for information on their particular scripting implementation.

The language used for the MIDI scripts is Lua. You will find a complete description of the language, and some useful tutorials, at the Lua website: www.lua.org

All the standard language features of Lua are available in the scripts, plus some extra functions (documented below) specific to the Expert Sleepers system.

Learn by example

The best way to learn about scripting the MIDI & OSC functionality is to look at the existing examples, particularly the default script that ships with the plug-in. Just open up the plug-in bundle and find the midi.lua file within. (Windows users should download the Mac OS X version of the plug-in and get the script from there, since in the Windows version the script is munged into the plug-in as a Windows resource.)

Most of the example snippets in the documentation below are taken directly from the default midi script.

You should be able to find more scripts on the Expert Sleepers website.

Share your scripts!

You are encouraged to share your scripts with other users. For example, you could post them on the Expert Sleepers forum (linked from the website). Alternatively, email them to us, and we'll make the best of the bunch downloadable directly from the Expert Sleepers site.

Script locations

The plug-in looks for MIDI & OSC scripts in standard locations. Scripts must have the file-name extension ".lua".

Mac OS X

The plug-in looks for scripts in

```
Library/Application Support/Expert Sleepers/<plug-in name>/Scripts
```

E.g.

```
Library/Application Support/Expert Sleepers/Oomingmak/Scripts
```

Windows

The plug-in looks for scripts in

```
C:\Documents and Settings\<username>\Application Data\Expert
Sleepers\<plug-in name>\Scripts
```

E.g.

```
C:\Documents and Settings\<username>\Application Data\Expert
Sleepers\Oomingmak\Scripts
```

Overriding the default script

Normally any scripts that the plug-in finds are run in addition to (and after) the default script ('midi.lua') that comes with the plug-in itself.

However, if you name your own script 'midi.lua', then the default script is not run. This lets you completely replace the plug-in's default MIDI behaviour, rather than simply extend it.

MIDI & OSC Script Functions

The scripts are simply loaded and executed. You do not need to define any particular functions for the system to call.

The following functions are available for you to call to define your script behaviour.

getParameterID(param)

Returns the parameter ID of the named parameter. Use with setParameter()/ getParameter() (see below). E.g.

```
paramID_Pitch = getParameterID( "Pitch" )
```

getParameter(param)

Returns the value of the plug-in parameter. 'param' can either be the parameter name or the parameter ID (as returned from getParameterID()). Using the ID is more efficient. Typically you would obtain the ID in the main script body (which is only executed once) and then use it in a handler function (which can be called many times). E.g.

```
pitch = getParameter( "Pitch" )
pitch = getParameter( paramID_Pitch )
```

setParameter(param, value)

Sets the value of the plug-in parameter. See the description of getParameter() for the meaning of 'param'. E.g.

```
setParameter( "Pitch", 12.0 )
setParameter( paramID_Pitch, 12.0 )
```

getParameterMinMax(param)

Returns the minimum and maximum values allowable for a plug-in parameter. See the description of getParameter() for the meaning of 'param'. E.g.

```
local minv, maxv = getParameterMinMax( paramID_Pitch )
```

getParameterUnit(param)

Returns an integer value that indicates the unit of the parameter (e.g. Hz, db, seconds). The unit is one of the values defined by Apple's Audio Unit specification.

getParameterName(param)

Returns the name of the parameter. (This is the same name that appears at the top right of the GUI when the mouse is over a parameter's control.)

isParameterUsed(param)

Returns a boolean value indicating whether the given parameter number is used by the plug-in. You should not attempt to set or get the value of an unused parameter.

getNumParameters()

Returns the total number of parameters that the plug-in defines. More strictly speaking - returns one more than the largest parameter ID that the plug-in uses, since there may be unused parameter IDs.

setOthersParameter(id, param, value)

As setParameter(), but sets the parameter on another instance of the plug-in, not necessarily the one running the script. This allows you to control several instances of the plug-in from a single script.

The ‘id’ is matched against the OSC Port Offset of the plug-ins. Any plug-in that matches the id will have its parameter set.

Note that all the plug-ins must be loaded by the same host application. For controlling instances of the plug-in loaded by other hosts, or running on other computers, use the ‘sendOSC’ command (below).

E.g.

```
setOthersParameter( 2, paramID_Pitch, 12.0 )
```

getOthersParameter(id, param)

As getParameter(), but gets the parameter from another instance of the plug-in. See setOthersParameter() for a fuller explanation. E.g.

```
pitch = getOthersParameter( 2, paramID_Pitch )
```

sendOSC(address, path [, format] [, values])

Sends an OSC message. ‘values’ is an optional array of data items to be sent with the message. If ‘values’ is used, then ‘format’ is an optional string that indicates how the items in the values array should be interpreted. This is required because Lua treats all numbers as being of the same type, whereas OSC differentiates between integers and floating point values. The number of characters in ‘format’ should be the same as the number of values. Each character may be one of ‘i’ (integer), ‘f’ (float) or ‘s’ (string).

E.g.

```
sendOSC( "osc.udp://localhost:7001", "/foo" )
sendOSC( "osc.udp://localhost:7001", "/foo", { 3, 5.2, "hello" } )
sendOSC( "osc.udp://localhost:7001", "/foo", "ifs", { 3, 5.2, "hello" } )
```

Note that the second example sends two floats and a string; the third sends an integer, a float and a string.

requestAllNoteOn(function)

Request that the given function be called in response to any MIDI note on event. E.g.

```
local function handleNoteOn( channel, noteNumber, velocity )
   -- do stuff
end
requestAllNoteOn( handleNoteOn )
```

requestAllNoteOff(function)

Request that the given function be called in response to any MIDI note off event. E.g.

```
local function handleNoteOff( channel, noteNumber, velocity )
   -- do stuff
end
requestAllNoteOff( handleNoteOff )
```

requestAllCC(function)

Request that the given function be called in response to any MIDI continuous controller (CC) event. E.g.

```
local function handleCC( channel, cc, value )
   -- do stuff
end
requestAllCC( handleCC )
```

requestAllNRPN(function)

Request that the given function be called in response to any MIDI non-registered parameter number (NRPN) event. E.g.

```
local function handleNRPN( channel, nrpn, value )
   -- do stuff
end
requestAllNRPN( handleNRPN )
```

requestAllProgramChange(function)

Request that the given function be called in response to any MIDI program change event. E.g.

```
local function handlePC( channel, value )
   -- do stuff
end
requestAllProgramChange( handlePC )
```

requestAllPolyPressure(function)

Request that the given function be called in response to any MIDI poly pressure (polyphonic aftertouch) event. E.g.

```
local function handlePolyPressure( channel, key, value )
   -- do stuff
end
requestAllPolyPressure( handlePolyPressure )
```

requestNoteOn(note, function)

Request that the given function be called in response to a MIDI note on event matching the given note number. E.g.

```
local function handleNoteOn( channel, noteNumber, velocity )
   -- do stuff
end
requestNoteOn( 60, handleNoteOn )
```

requestNoteOff(note, function)

Request that the given function be called in response to a MIDI note off event matching the given note number. E.g.

```
local function handleNoteOff( channel, noteNumber, velocity )
   -- do stuff
end
requestNoteOff( 60, handleNoteOff )
```

requestCC(cc, function)

Request that the given function be called in response to the given MIDI continuous controller (CC) event. E.g.

```
local function handleCC( channel, cc, value )
   -- do stuff
end
requestCC( 20, handleCC )
```

requestNRPN(nrpn, function)

Request that the given function be called in response to the given MIDI non-registered parameter number (NRPN) event. E.g.

```
local function handleNRPN( channel, nrpn, value )
   -- do stuff
end
requestNRPN( 1000, handleNRPN )
```

requestProgramChange(pc, function)

Request that the given function be called in response to the given MIDI program change event. E.g.

```
local function handlePC( channel, value )
   -- do stuff
end
requestProgramChange( 2, handlePC )
```

requestPolyPressure(key, function)

Request that the given function be called in response to a MIDI poly pressure (polyphonic aftertouch) event on the given key. E.g.

```
local function handlePolyPressure( channel, key, value )
   -- do stuff
end
requestPolyPressure( 60, handlePolyPressure )
```

requestPitchWheel(function)

Request that the given function be called in response to a MIDI pitch wheel event. NB the value passed to the handler function is the raw 14 bit MIDI value, not e.g. a normalised ±1.0 value. E.g.

```
local function handlePitchWheel( channel, value )
   -- do stuff
end
requestPitchWheel( handlePitchWheel )
```

requestChannelPressure(function)

Request that the given function be called in response to a MIDI channel pressure (after-touch) event. E.g.

```
local function handleChannelPressure( channel, value )
   -- do stuff
end
requestChannelPressure( handleChannelPressure )
```

time()

Returns a value in seconds. The value itself is not particularly meaningful, but if you call it twice and subtract the two values you will have the time interval between the two calls. E.g.

```
time1 = time()
   -- do stuff
time2 = time()
print( "elapsed time", time2-time1 )
```

requestTimedCallback(interval, function)

Requests that the given function should be called after *interval* seconds. Returns a timer object that can be passed to cancelTimer() (see below). E.g.

```
local function timerTimeOut()
   print( "it is now 2 seconds later" )
end
requestTimedCallback( 2.0, timerTimeOut )
```

requestPeriodicCallback(interval, function)

Requests that the given function should be called every *interval* seconds. Returns a timer object that can be passed to cancelTimer() (see below). E.g.

```
local function timerCallback()
   print( "tick" )
end
requestPeriodicCallback( 1.0, timerCallback )
```

cancelTimer(timer)

Cancels a timer that was created with requestTimedCallback() or requestPeriodicCallback(). After this call the timer object is no longer valid and should not be used. E.g.

```
timer = requestPeriodicCallback( 1.0, timerCallback )

   -- timer is now firing every second

cancelTimer( timer )
timer = nil
```

setGUIBoolValue(id, value)

Sets a value that can be picked up by GUI objects. 'id' is a number between 0 and 7. 'value' is interpreted as a boolean value (true or false). See the GUI script method indicator() for a typical usage.

getGUIBoolValue(id)

Gets one of the values that can be set with setGUIBoolValue(). 'id' is a number between 0 and 7.

setOthersGUIBoolValue(other, id, value)

As setGUIBoolValue() but sets the value on other instances of the plug-in - see setOthersParameter() above.

getOthersGUIBoolValue(other, id)

As getGUIBoolValue() but gets the value from another instance of the plug-in - see getOthersParameter() above.

Pre-defined Global Values

The system defines some values before calling your script, which you can use to make the script's behaviour dependent on, for example, what kind of computer you're using. These values (which are pretty self-explanatory) are:

- isMac
- isWin
- isVST
- isAU
- majorVersion
- minorVersion
- dotVersion
- version

The plug-in's version number is of the form x.y.z (e.g. 2.1.4) where x is the major version number, y is the minor version number, and z is the dot version. The 'version' global variable contains a single value combining all three e.g. for version 2.1.4, 'version' is 20104. This is useful for making your scripts backwardly compatible - by testing for the version number and not trying to use features that were not present in a version of the plug-in older than the version you're testing for.

Debugging

You can use Lua's 'print' function to write out information to help you track what's going on (or what's not going on) in your script. Also any run-time errors, or errors in loading the script in the first place, are reported. In both cases, the output goes to:

Mac OS X

The system console.log. Use the standard Console utility (located in `Applications/Utilities`) to view it.

Windows

The system OutputDebugString API. Use an application like Sysinternal's DebugView to view it.

Version History

1.0.0 19/5/2009

- Initial version.

Contact

The Expert Sleepers website is here:

http://www.expert-sleepers.co.uk/

Or you can email

info@expertsleepers.co.uk

Or you can use the forum, which is here:
http://www.kvraudio.com/forum/viewforum.php?f=85

Acknowledgements

The software described in this manual makes use of the following open source projects. The author is greatly indebted to them for their efforts and generosity.

Below are reproduced the various copyright notices and disclaimers that accompany these software projects, in accordance with their terms of use.

Lua

oscpack

oscpack -- Open Sound Control packet manipulation library
http://www.audiomulch.com/~rossb/code/oscpack