

Lua extension

This document describes the Lua Extension for the Electra One MIDI Controller firmware. The extension introduces procedural programming capabilities to Electra One presets

Lua is a lightweight scripting language. You can find detailed information on the [Official Lua site](#). Alternatively, you can follow our [Lua Crash Course](#) — a short tutorial designed specifically for musicians and non-programmers.

With the Electra Lua extension, you can use the Lua programming language inside Electra One MIDI controllers to easily create, manage, and customize MIDI and music-related actions.

Note

To utilize the Electra One Lua Extension described in this document, you must have Firmware version 4.0 or later installed.

A brief overview

With the Electra One Lua extension, you can extend and customize your presets by adding Lua functions. It allows you to create features and behaviors that simply wouldn't be possible without it. Here are just a few examples of what you can do:

- Send and receive MIDI messages.
- Trigger Lua functions when control values change.
- Format display values.
- Modify the visibility, location, name, and color of controls.
- Execute custom patch dump request calls.
- Craft your own SysEx parsers.
- Calculate checksums and generating custom SysEx template bytes.
- Run Lua functions based on MIDI clock and transport control.
- Create sequences of MIDI data, clock messages, and MIDI LFOs.
- Visualize MIDI data on the controller's screen.

The core idea behind this extension is to clearly separate the static data defined in the declarative JSON preset from the dynamic processing handled at runtime through Lua scripting. The JSON preset acts as the foundation, pre-loading all pages, lists, devices, groups, and controls. Once the preset is loaded, the Lua extension takes over, allowing you to manipulate these objects for specific purposes.

It's important to note that the Lua extension cannot create new objects, but it can modify, reposition, and change the visibility of existing ones.

Uploading the scripts

To enable Lua extension functions within a preset, you must first upload a Lua script file(s). The uploaded script is then associated with the currently active preset. If a Lua script already exists for that preset, uploading a new one will overwrite it.

Normally, each preset uses a single Lua script. If needed, you can upload multiple Lua script files that work together as one larger Lua project.

This document covers the single-file setup. Multi-file configurations are explained in a separate guide about Electra One's file transfer and management protocol.

Uploading the scripts with the Preset Editor

You can create, edit, and upload Lua scripts directly from the [Preset editor](#) — the easiest and recommended way to work with Lua. If needed, you can also upload scripts to the Electra One MIDI Controller using a SysEx call.

Uploading the scripts with a SysEx call

```
0xF0 0x00 0x21 0x45 0x01 0x0C script-source-code 0xF7
```

`0xF0` SysEx header byte

`0x00` `0x21` `0x45` Electra One MIDI manufacturer Id

`0x01` Upload data

`0x0C` Lua script file

`script-source-code` bytes representing ASCII characters of the Lua script source code

`0xF7` SysEx closing byte

Executing a Lua command with a SysEx call

This is a call that executes arbitrary Lua commands, effectively serving as an API endpoint for controlling Electra One presets from external devices and applications.

It allows you to remotely manage Electra One presets using Lua commands, offering a powerful way to interact with the controller from external sources. The maximum allowed length of a Lua command is 65,353 bytes.

However, we recommend keeping commands short — commands shorter than 65 bytes are executed significantly faster than longer ones.

To optimize performance, it is better to use this SysEx call to trigger Lua functions defined in a previously uploaded Lua script, rather than sending large blocks of arbitrary Lua code.

```
0xF0 0x00 0x21 0x45 0x08 0x0D lua-command-text 0xF7
```

`0xF0` SysEx header byte

`0x00` `0x21` `0x45` Electra One MIDI manufacturer Id

`0x08` Execute command

`0x0D` Function

`lua-command-text` ASCII bytes representing the log message

`0xF7` SysEx closing byte

`lua-command-text` is a free-form string that holds the Lua command to be executed. It must respect the maximum length limit explained above.

An example of the lua-command-text

```
print ("Hello MIDI world!")
```

The structure of the script

The Electra One Lua Extension script is organized into four distinct building blocks:

- **The Setup Section:** This section is where you initialize and configure the settings and parameters needed for your script. It acts as the starting point for your script's execution and often includes setup tasks like defining global variables, establishing MIDI connections, or configuring other necessary resources.
- **The Standard Functions:** These are predefined functions included in the Electra One Lua Extension scripting environment. They provide the core functionality for interacting with the MIDI controller and its features. Standard functions can be used to send and receive MIDI messages, manipulate controls, and manage various aspects of the controller's behavior.
- **The Standard Callbacks:** Electra One provides a set of standard callback functions that allow your Lua script to respond to various events. These callbacks are invoked automatically by the system when specific events occur. For example, you can use callbacks to react to control value changes or button presses, adding dynamic and interactive behavior to your script.
- **The User Functions:** These are custom functions that you define to extend the functionality of your Lua script. User functions allow you to implement unique behaviors, process data, and create specific responses to tailor the script to your needs. They give you the flexibility to customize the Electra One experience according to your requirements.

Once you understand and use these four building blocks, you'll be able to create powerful Lua scripts that make your Electra One MIDI controller even more capable and flexible.

Let's use the following example to demonstrate it:

```
-- Display controls related to specific value of another control

-- a function to hide all controls within the groups
function hideAllGroups(groups)
    for groupId = 0, #groups do
        for i, controlId in ipairs(groups[groupId]) do
            control = controls.get(controlId)
            control:setVisible(false)
        end
    end
end

-- show given control group
function showGroup(groups, groupId)
    for i, controlId in ipairs(groups[groupId]) do
        control = controls.get(controlId)
```

```

        control:setSlot(i + 1)
    end
end

-- the callback function called from the preset
function displayGroup(valueObject, value)
    hideAllGroups(controlGroups)
    showGroup(controlGroups, value)
end

-- a standard callback function to handle PATCH REQUEST event
function patch.onRequest(device)
    print("Requesting patches from device " .. device.id);
    midi.sendProgramChange(PORT_1, device.channel, 10)
end

-- set the initial state. group 0 is displayed

-- define assignment of controls to groups
controlGroups = {
    [0] = { 20, 21, 22 },
    [1] = { 26, 27, 28 },
    [2] = { 32, 33 }
}

function preset.onLoad()
    showGroup(controlGroups, 0)
end

print("Lua ext initialized")

```

The setup

The setup section includes all source code that exists outside of any specific function and runs in the global context of the script. In this section, you can perform various tasks such as calling standard functions, executing user-defined functions, initializing global variables, and setting up resources.

Below is an example of a typical setup section from a script:

```

-- set the initial state. group 0 is displayed

-- define assignment of controls to groups
controlGroups = {
    [0] = { 20, 21, 22 },
    [1] = { 26, 27, 28 },
    [2] = { 32, 33 }
}

function preset.onLoad()
    showGroup(controlGroups, 0)
end

```

```
print("Lua ext initialized")
```

The primary purpose of the setup section is to prepare your extension to handle application events later on. It is executed immediately after the preset is loaded.

The location of the setup code within the script does not affect its functionality — it does not have to be placed at the top. However, if you plan to call your own user-defined functions in the setup section, it's recommended to either place the setup code after the function definitions or move it into the `preset.onLoad()` or `preset.onReady()` functions for better script organization. For more details, see the Preset Initialization section below.

The standard functions

Standard functions include functions from both the Lua standard libraries and the Electra One Extension libraries. They cover a wide range of tasks, such as printing messages, performing mathematical operations, sending and receiving MIDI messages, and interacting with user interface (UI) components.

You can find detailed descriptions of the Lua standard functions in the official [Lua documentation](#), and descriptions of Electra-specific functions in the API Reference later in this document.

As an example, the `print` function is a typical standard function you will use in your scripts:

```
print("Lua ext initialized")
```

The standard callbacks

The Electra One Lua Extension provides a set of predefined event handlers, often called callbacks. These callbacks are automatically triggered when specific events happen.

For example:

```
-- a standard callback function to handle PATCH REQUEST event
function patch.onRequest(device)
    print("Requesting patches from device " .. device.id);
    midi.sendProgramChange(PORT_1, device.channel, 10)
end
```

In this code snippet, the `patch.onRequest` function is a standard callback that responds to the 'PATCH REQUEST' event. When the event occurs, this callback runs the actions you have defined: printing a text message and sending out a Program Change MIDI message.

Standard callbacks like this allow you to customize how your Lua script reacts to different events, making your Electra One MIDI controller more interactive and adaptable.

The user functions

As a user, you have the creative freedom to define your own functions. In fact, you are encouraged to do so — user functions are the building blocks for creating more advanced and structured elements in your Lua script.

User functions help you organize your code and extend your script's capabilities. They let you group specific tasks or behaviors together, making your scripts more modular, easier to manage, and easier to reuse.

For example, the `displayGroup` function from the earlier source code example is a user-defined function that is linked to a callback hook inside the preset JSON.

```
-- the callback function called from the preset
function displayGroup(valueObject, value)
    hideAllGroups(controlGroups)
    showGroup(controlGroups, value)
end
```

Preset initialization

Some presets may require a carefully controlled sequence of actions during startup. The Electra One Lua Extension gives you ways to run your own functions at different stages of the preset loading process. It's important to understand the order in which these stages happen.

When a preset is loaded for the first time. For example, when you power on the controller — the following steps take place:

1. Everything defined in the global context of your Lua script (outside of any function) is executed.
2. The `preset.onLoad()` function is called, if defined.
3. All Lua functions linked to Control values are called, using the default values from the preset JSON.
4. The `preset.onReady()` function is called, if defined.
5. Electra One starts listening for external events, like user actions or incoming MIDI messages.

The same sequence of actions is performed when you load any preset for the first time — for example, when switching presets using the user interface. Once a preset has been loaded, switching back to it later will not re-initialize it. The preset is considered already initialized.

Lua Extension API Reference

This section provides a complete reference to all Lua functions available in the Electra One Extension library. You'll find descriptions, parameters, and usage details for each function, organized by category. These functions allow your script to interact with controls, MIDI messages, graphics, system settings, and more. Use this chapter as a practical guide while developing or exploring your Lua scripts.

Controls

The controls module provides functionality to manage preset controls. It is not intended for changing properties of individual controls. Individual controls are managed by manipulating the [Control object](#).

Functions

`controls.get(ref)`

Retrieves a reference to a Control object (Lua userdata). A control represents a fader, list, or other type of control. The reference number can be found in the control properties panel in the Preset Editor.

Parameters

ref number, a preset object reference number (1 .. 1023).

Returns

userdata, an object representing the control.

Example

```
-- Retrieving a reference to given control

local control = controls.get(1)
```

Control

A Control object represents a single control, like a fader or button. It stores its own data and provides functions to read and update its properties.

Functions

<control>:getId()

Retrieves an identifier of the Control. The identifier is assigned to the Control in the preset JSON.

Returns

number, the reference of the control (1 .. 1023).

Example

```
-- Retrieving a control and getting its Id

local volumeControl = controls.get(10)
print("got Control with Id " .. volumeControl:getId())
```

<control>:setVisible(shouldBeVisible)

Changes the visibility of the given control. The initial visibility is defined in the Preset JSON. The `setVisibility()` function can update the visibility at runtime.

Parameters

shouldBeVisible boolean, true when the Control will be visible.

<control>:isVisible()

Retrieves the visibility status of the control.

Returns

boolean, true when the control is currently visible.

Example

```
-- a function to toggle visibility of a control

function toggleControl(control)
    control:setVisible(not control:isVisible())
end
```

<control>:setName(name)

Sets a new name of the control.

Parameters

name string, the new name to assign to the control.

<control>:getName()

Retrieves the current name of the control.

Returns

string, the current name of the control.

Example

```
-- print out a name of given control

function printName(controlId)
    local control = controls.get(controlId)
    print ("Name: " .. control:getName())
end
```

<control>:setColor(color)

Sets a new color for the control. Although 24-bit RGB 888 is used, the controller internally converts it to 16-bit RGB 565.

Parameters

color number, a number representing the color as a 24-bit RGB value.

<control>:getColor()

Retrieves the current color of the control.

Returns

number, a number representing the color as a 24-bit RGB value.

Example

```
-- A callback function that changes color of the control
-- when its value exceeds 100

function functionCallback(valueObject, value)
    local control = valueObject:getControl()

    if (value > 100) then
        control:setColor(0xff0000)
    else
        control:setColor (0xffffff)
    end
end
```

<control>:setVariant(variant)

Sets a variant for the control. For a list of variants, see the [Globals](#) section below.

Parameters

variant number, a number representing the variant.

<control>:getVariant()

Retrieves the current variant of the control.

Returns

number, a number representing the variant.

<control>:setBounds({x, y, width, height})

Sets the bounding box (position and dimensions) of the control on the screen. The function expects an array to be passed as an argument. Use the `X`, `Y`, `WIDTH`, `HEIGHT` globals to access individual members of the array.

Parameters

<code>x</code>	number, X position on the screen.
<code>y</code>	number, Y position on the screen.
<code>width</code>	number, width of the control.
<code>height</code>	number, height of the control.

<control>:getBounds()

Retrieves the bounding box (position and dimensions) of the control on the screen. Use the `X`, `Y`, `WIDTH`, `HEIGHT` globals to access individual members of the array.

Returns

array, an array consisting of x, y, width, height boundary box attributes.

Example

```
-- print out position and dimensions of given control

local control = controls.get(2)
control:setBounds({ 200, 200, 170, 65 })
bounds = control:getBounds()
print("current bounds: " ..
      "x=" .. bounds[X] ..
      ", y=" .. bounds[Y] ..
      ", width=" .. bounds[WIDTH] ..
      ", height=" .. bounds[HEIGHT])
```

<control>:setPot(controlSet, pot)

Assigns the control to a specified control set and pot. See the [Globals](#) section below for available Control Set and pot identifiers.

Parameters

<code>controlSet</code>	number, a numeric identifier of the Control Set to assign.
-------------------------	--

pot number, a numeric identifier of the pot.

Example

```
-- Reassign the control to different controlSet and pot

local control = controls.get(1)
control:setPot(CONTROL_SET_1, POT_2)
```

<control>:setSlot(slot)

Moves the given control to a preset slot on the current page. The Control Set and pot values are assigned automatically, and the control is made visible.

Parameters

slot number, a numeric identifier of the page slot (1 .. 36).

<control>:setSlot(slot, page)

Moves the given control to a preset slot on a specified page. The Control Set and pot values are assigned automatically, and the control is made visible. This function is a variant of `setSlot(slot)`, allowing you to specify the target page as well.

Parameters

slot number, a numeric identifier of the page slot (1 .. 36).

page number, a numeric identifier of the page (1 .. 12).

Example

```
-- Change location of the control within the 6x6 grid

local control = controls.get(1)
control:setSlot(7)
```

<control>:getValueIds()

Retrieves a list of all valueIds associated with the control. The valueIds are defined in the preset JSON. Retrieved valueIds, eg. `value`, `attack` can be used as parameters for the `<control>:getValue(valueId)` function.

Returns

array, a list of value identifier strings.

Example

```
-- list all value Ids of a control

local control = controls.get(1)
local valueIds = control.getValueIds ()

for i, valueId in ipairs(valueIds) do
    print(valueId)
end
```

<control>:getValue(valueId)

Retrieves the [Value object](#) of the given control using the valueId handle. The valueId is defined in the preset JSON. If not specified, `value` will be used as the default valueId.

Parameters

`valueId` string, a text that identifies a specific value of a control.

Returns

userdata, a reference to the Value object.

Example

```
-- Display min and max display values

local control = controls.get(1)
local value = control.getValue("attack")

print ("value min: " .. value:getMin())
print ("value max: " .. value:getMax())
```

<control>:getValues()

Retrieves a list of all [Value objects](#) associated with the control. The value objects are defined in the JSON preset.

Returns

array, a list of references to userdata Value objects.

Example

```
-- list all value objects of a control

local control = controls.get(1)
local valueObjects = control:getValues()

for i, valueObject in ipairs(valueObjects) do
    print(string.format ("%s.%s", control:getName(), valueObject:getId()))
end
```

<control>:setPaintCallback(callback)

Assigns a function that defines how the control is drawn. This can only be used with Custom controls.

Parameters

callback function, a reference to a Lua function.

<control>:setTouchCallback(callback)

Assigns a function that will be called when a display touch event is received for the given Custom control.

Parameters

callback function, a reference to a Lua function.

<control>:setPotCallback(callback)

Assigns a function that will be called when a pot change event is received for the given Custom control.

Parameters

callback function, a reference to a Lua function.

<control>:repaint()

Schedules a repaint of the control. The function is meant to be used inside the Custom control touch and pot change callbacks.

<control>:print()

Prints all attributes of the Control object to the Logger output.

Try it yourself



Control module demo

Controller

The controller module provides functionality to query information about the controller hardware and the firmware, and to check compatibility.

Functions

getModel()

Retrieves information about the model of the controller.

Returns

enum, ['mk2', 'mini'].

getNumModel()

Retrieves numeric information about the model of the controller.

Returns

enum, [MODEL_MK2, MODEL_MINI].

getFirmwareVersion()

Retrieves version of the firmware currently installed in the controller.

Returns

string, a text representation of the firmware version.

getFirmwareNumVersion()

Retrieves numeric version of the firmware currently installed in the controller.

Returns

number, a numeric representation of the firmware version.

uptime()

Retrieves the number of milliseconds elapsed since the controller was reset.

Returns

number, time elapsed in milliseconds.

require(model, minimumVersion)

Raises an error when the model and minimum firmware version requirements are not met. The function is meant to be used in combination with an `assert()` function.

Parameters

`model` number, numeric information about the model of the controller.

`minimumVersion` string, minimum required version of the firmware.

isRequired(model, minimumVersion)

Checks the model of the controller and the version of the currently installed firmware. Returns information on whether the requirements are met.

Parameters

`model` number, numeric information about the model of the controller.

`minimumVersion` string, minimum required version of the firmware.

Returns

boolean, true if the requirements are met.

Example

```
-- Check if model and firmware requirements are met
-- Note: assert will terminate the script on a failed check
assert(
    controller.isRequired(MODEL_MK2, "4.0.0"),
    "Version 4.0.0 or higher is required"
)

-- Check if model and firmware requirements are met
```

```
-- Note: this will not terminate the script
local validToRun = controller.isRequired(MODEL_MK2, "4.0.1")

if validToRun == true then
    print("requirements met")
else
    print("requirements failed")
end

-- Query information about the controller and the firmware
print("model: " .. controller.getModel())
print("numeric model: " .. controller.getNumModel())
print("firmware version: " .. controller.getFirmwareVersion())
print("numeric firmware version: " .. controller.getFirmwareNumVersion())

-- Query system uptime
print("uptime: " .. controller.uptime() .. " msecs")
```

Try it yourself



Controller module demo

Data Pipe

Data pipes let presets share information with each other. Think of a data pipe as a named channel where one preset can send a stream of numbers, and another preset can receive them.

This is useful when you want different presets to work together. For example, you can create an LFO preset that continuously sends out modulation values. Another preset can then receive that stream and use it to control things like knobs or sliders — without changing its own internal setup.

This makes it easy to build flexible and creative setups where presets interact in real time.

Functions

pipe.acquire(pipeName)

Creates a new data pipe with a name. Other presets can listen to that pipe and receive any data sent by the preset that owns it.

Parameters

pipeName string, name of the data pipe channel.

Returns

number, a numeric identifier of the pipe channel.

pipe.send(pipeId, value)

Sends a floating-point number to a data pipe that was previously acquired using the `pipe.acquire()` function. The value will be delivered to all presets that are listening to that pipe.

Parameters

<code>pipeId</code>	number, a numeric identifier of the pipe to send the data to.
<code>value</code>	number, a floating-point number to send.

`pipe.release(pipeId)`

Releases a previously acquired pipe, making the pipeId available for other presets to use.

Parameters

<code>pipeId</code>	number, a numeric identifier of the pipe to send the data to.
---------------------	---

Devices

A Device represents a musical instrument connected to one of Electra's hardware ports and listening on a particular MIDI channel. The devices module provides functionality to query and create devices.

Functions

`devices.get(deviceId)`

Retrieves a reference to a Device object (Lua userdata). A Device represents a MIDI device that sends and receives messages with the controller.

Parameters

<code>deviceId</code>	number, a numeric identifier of the device (1 .. 32).
-----------------------	---

Returns

userdata, an object representing the device.

`devices.getByPortChannel(port, channel)`

Retrieves a reference to a Device object (Lua userdata) using the port and MIDI channel. A Device represents a MIDI device that sends and receives messages with the controller.

Parameters

<code>port</code>	enum, a numeric identifier of the port [PORT_1, PORT_2]
<code>channel</code>	number, a numeric identifier of the MIDI channel (1 .. 16).

Returns

userdata, an object representing the device.

`devices.create(deviceId, name, port, channel)`

Creates a new Device object and returns a reference to it.

Parameters

<code>deviceId</code>	number, a numeric identifier of device to be created.
<code>name</code>	string, a name to be assigned to the device.
<code>port</code>	enum, a numeric identifier of the port [PORT_1, PORT_2]
<code>channel</code>	number, a numeric identifier of the MIDI channel (1 .. 16).

Returns

userdata, an object representing the device.

Device

A Device object is used to manage the Device settings.

Functions

`<device>.getId()`

Retrieves an identifier of the Device. The identifier is assigned to the control in the preset JSON or created with the `devices.create()` function.

Returns

number, a numeric identifier of the device (1 .. 32).

`<device>.setName(name)`

Assigns a new name to the Device.

Parameters

<code>name</code>	string, a name to be assigned to the Device.
-------------------	--

`<device>.getName()`

Retrieves the name currently assigned to the Device.

Returns

string, a name currently assigned to the Device.

<device>.setPort(port)

Assigns a new MIDI port to the Device.

Parameters

port enum, a numeric identifier of the port [PORT_1, PORT_2]

<device>.getPort()

Retrieves the currently assigned MIDI port.

Returns

enum, a numeric identifier of the port [PORT_1, PORT_2]

<device>.setChannel(channel)

Assigns a new MIDI channel to the Device.

Parameters

channel number, a numeric identifier of the MIDI channel (1 .. 16).

<device>.getChannel()

Retrieves the currently assigned MIDI channel.

Returns

number, a numeric identifier of the MIDI channel (1 .. 16).

<device>.setRate(time)

Sets the time delay between sending individual MIDI messages.

Parameters

rate

number, the time delay in msecs to be set.

<device>.getRate()

Retrieves the current time delay between sending individual MIDI messages.

Returns

number, the time delay in milliseconds.

Example

```

-- This needs to reflect the preset device settings
local AccessVirusDeviceId = 2

-- Display info about the device
local device = devices.get(AccessVirusDeviceId)
print ("device port: " .. device:getPort())
print ("device channel: " .. device:getChannel())

-- A function to set channel of device with a Control
function setChannel(control, value)
    device = devices.get(AccessVirusDeviceId)
    device:setChannel(value)
end

```

Events

The Events library lets you control which notifications Electra One sends out and define callback functions to handle those events.

Functions

events.subscribe(eventFlags)

A function to instruct the controller what event notifications should be emitted. Subscribed events are result in calls to event callback functions and sending out SysEx event notifications.

Parameters

eventFlags

number, a byte where each bit represents an event type to be subscribed. Note, currently only 'PAGES' and 'POTS' are supported.

Example

```

-- Sending MIDI messages out

print ("Events demo")

events.subscribe(PAGES | POTS)
events.setPort(PORT_CTRL)

function events.onPageChange(newPageId, oldPageId)
    print ("old: " .. oldPageId)
    print ("new:" .. newPageId)
end

function events.onPotTouch(potId, controlId, touched)
    print ("potId: " .. potId)
    print ("controlId: " .. controlId)
    print ("touched: " .. (touched and "yes" or "no"))
end

```

events.setPort(port)

Sets MIDI port that will be used to send out event notifications.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2, PORT_CTRL].

events.onPageChange(newPageId, oldPageId)

A callback function that runs automatically when the user switches to a different page. The function is called only if Page change notifications have been subscribed to. For more details, see [Events](#) section.

Parameters

newPageId number, a numeric identifier of the Page being activated (1–12).

oldPageId number, a numeric identifier of the Page being left (1–12).

events.onPotTouchChange(potId, controlId, touched)

A callback function that is called when user touches or releases the controller knobs.

Parameters

potId number, a numeric identifier of the Page being activated (1–12).

controlId number, a preset object reference number (1 .. 1023).

touched boolean, when true the pot has an active touch, otherwise it was released.

Graphics

The Graphics module offers drawing functions for use inside paint callbacks. Drawing is limited to the area defined by each component's boundary box.

Functions

graphics.setColor(color)

Sets the color used by all following drawing functions.

Parameters

color a number representing the color as a 24-bit RGB value.

graphics.drawPixel(x1, y1)

Draws single pixel at (x, y) coordinates.

Parameters

x number, X position of the starting point of the line.

y number, Y position on the starting point of the line.

graphics.drawLine(x1, y1, x2, y2)

Draws a straight line between (x1, y1) starting coordinates and (x2, y2) ending coordinates.

Parameters

x1 number, X position of the starting point of the line.

y1 number, Y position on the starting point of the line.

x2 number, X position on the ending point of the line.

y2 number, Y position on the ending point of the line.

graphics.drawRect(x, y, width, height)

Draws a rectangle starting at the (x, y) coordinates, using the specified width and height.

Parameters

- x** number, X position of the left-top corner of the rectangle.
- y** number, Y position on the left-top corner of the rectangle.
- width** number, width of the rectangle.
- height** number, height of the rectangle.

graphics.fillRect(x, y, width, height)

Draws a solid-filled rectangle at the (x, y) coordinates with the given width and height.

Parameters

- x** number, X position of the left-top corner of the rectangle.
- y** number, Y position on the left-top corner of the rectangle.
- width** number, width of the rectangle.
- height** number, height of the rectangle.

graphics.drawRoundRect(x, y, width, height, radius)

Draws a rounded rectangle starting at the (x, y) coordinates, using the specified width and height.

Parameters

- x** number, X position of the left-top corner of the rectangle.
- y** number, Y position on the left-top corner of the rectangle.
- width** number, width of the rectangle.
- height** number, height of the rectangle.
- radius** number, radius to be applied at the corner.

graphics.fillRoundRect(x, y, width, height, radius)

Draws a solid-filled rounded rectangle at the (x, y) coordinates with the given width and height.

Parameters

- x** number, X position of the left-top corner of the rectangle.
- y** number, Y position on the left-top corner of the rectangle.
- width** number, width of the rectangle.

height number, height of the rectangle.

radius number, radius to be applied at the corner.

graphics.drawTriangle(x1, y1, x2, y2, x3, y3)

Draws a triangle by connecting the three specified coordinates (x1, y1), (x2, y2), and (x3, y3).

Parameters

x1 number, X-coordinate of the first vertex.

y1 number, Y-coordinate of the first vertex.

x2 number, X-coordinate of the second vertex.

y2 number, Y-coordinate of the second vertex.

x3 number, X-coordinate of the third vertex.

y3 number, Y-coordinate of the third vertex.

graphics.fillTriangle(x1, y1, x2, y2, x3, y3)

Draws a filled triangle by connecting the three coordinates points (x1, y1), (x2, y2), and (x3, y3) and filling the interior area.

Parameters

x1 number, X-coordinate of the first vertex.

y1 number, Y-coordinate of the first vertex.

x2 number, X-coordinate of the second vertex.

y2 number, Y-coordinate of the second vertex.

x3 number, X-coordinate of the third vertex.

y3 number, Y-coordinate of the third vertex.

graphics.drawCircle(x, y, radius)

Draws the outline of a circle centered at (x, y) coordinates with the specified radius.

Parameters

x number, X-coordinate of the center of the circle.

y number, Y-coordinate of the center of the circle.

radius number, radius of the circle.

graphics.fillCircle(x, y, radius)

Draws a filled circle centered at (x, y) coordinates with the specified radius.

Parameters

x number, X-coordinate of the center of the circle.

y number, Y-coordinate of the center of the circle.

radius number, radius of the circle.

graphics.drawEllipse(x, y, radiusX, radiusY)

Draws the outline of an ellipse centered at (x, y) coordinates with the specified horizontal and vertical radius.

Parameters

x number, X-coordinate of the center of the ellipse.

y number, Y-coordinate of the center of the ellipse.

radiusX number, horizontal radius of the ellipse.

radiusY number, vertical radius of the ellipse.

graphics.fillEllipse(x, y, radiusX, radiusY)

Draws a filled ellipse centered at (x, y) coordinates with the specified horizontal and vertical radius.

Parameters

x number, X-coordinate of the center of the ellipse.

y number, Y-coordinate of the center of the ellipse.

radiusX number, horizontal radius of the ellipse.

radiusY number, vertical radius of the ellipse.

graphics.fillCurve(x, y, radius, segment)

Draws a filled circular segment (curve) centered at (x, y) coordinates with the specified radius and segment.

Parameters

<code>x</code>	number, X-coordinate of the center of the curve.
<code>y</code>	number, Y-coordinate of the center of the curve.
<code>radius</code>	number, radius of the curve.
<code>segment</code>	enum, segment of the circle to be drawn [TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT, BOTTOM_RIGHT].

`graphics.print(x, y, text, width, alignment)`

Prints text starting at the (x, y) position, using the specified width and alignment.

Parameters

<code>x</code>	number, X-coordinate where the text starts.
<code>y</code>	number, Y-coordinate where the text starts.
<code>text</code>	string, the text to be printed.
<code>width</code>	number, the width of the box where the text is printed.
<code>alignment</code>	enum, text alignment inside the box [LEFT, CENTER, RIGHT].

Groups

The groups module helps you manage groups inside a preset. A Group is a graphical element that organizes and improves the layout of preset pages.

Functions

`groups.get(ref)`

Retrieves a reference to a Group object (Lua userdata).

Parameters

<code>ref</code>	number, a preset object reference number.
------------------	---

Returns

userdata, an object representing the Group.

Example

```
-- Retrieve a reference to given group
```

```
local group = groups.get(1)
```

Group

A Group object stores its own data and provides functions to update and manage it.

Functions

<group>:getId()

Retrieves an identifier of the Group. The identifier is assigned to the Control in the preset JSON.

Returns

number, the reference of the Group (1 .. 1023).

<group>:setLabel(label)

Sets a new label of the Group.

Parameters

label string, the new label to assign to the Group.

<group>:getLabel()

Retrieves the current label assigned to the Group.

Returns

string, the current label of the Group.

<group>:setVisible(shouldBeVisible)

Changes the visibility of the given Group. The initial visibility is defined in the Preset JSON. The

setVisibility() function can update the visibility at runtime.

Parameters

shouldBeVisible boolean, true when the Group will be visible.

<group>:isVisible()

Retrieves the visibility status of the Group.

Returns

boolean, true when the Group is currently visible.

<group>:setColor(color)

Sets a new color for the Group. Although 24-bit RGB 888 is used, the controller internally converts it to 16-bit RGB 565.

Parameters

color number, a number representing the color as a 24-bit RGB value.

<group>:getColor()

Retrieves the current color of the Group.

Returns

number, a number representing the color as a 24-bit RGB value.

<group>:setVariant(variant)

Sets a variant for the Group. For a list of variants, see the [Globals](#) section below.

Parameters

variant enum, a number representing the variant [VT_DEFAULT, VT_HIGHLIGHTED].

<group>:setBounds({x, y, width, height})

Sets the bounding box (position and dimensions) of the Group on the screen. The function expects an array to be passed as an argument. Use the **X**, **Y**, **WIDTH**, **HEIGHT** globals to access individual members of the array.

Parameters

x number, X position on the screen.

y number, Y position on the screen.

width number, width of the Group.

height number, height of the Group.

<group>:getBounds ()

Retrieves the bounding box (position and dimensions) of the Group on the screen. Use the **X**, **Y**, **WIDTH**, **HEIGHT** globals to access individual members of the array.

Returns

array, an array consisting of x, y, width, height boundary box attributes.

Example

```
-- change group slot and dimentions

-- Verical line only
local group1 = groups.get(1)

print("Label name: " .. group1.getLabel())
group1:setSlot(3, 2)

-- Renctangle group
local group2 = groups.get(2)

print("Label name: " .. group2.getLabel())
group2:setSlot(9, 2, 2)
```

<group>:setSlot(slot, width, height)

Moves the given Group to a preset slot on the current page and adjusts its horizontal and vertical span. A height of 0 creates a thin line; any other height setting forms a rectangle.

Parameters

slot number, a numeric identifier of the page slot (1 .. 36).

width number, a horizontal span of the Group in the slot units (1 .. 6).

height number optional, a vertical span of the Group in the slot units (0 .. 6).

<group>:setHorizontalSpan(width)

Changes the horizontal span of the group.

Parameters

width number, a horizontal span of the Group in the slot units (1 .. 6).

<group>:setVerticalSpan(width)

Changes the vertical span of the group. A height of 0 creates a thin line; any other height setting forms a rectangle.

Parameters

height number, a vertical span of the Group in the slot units (0 .. 6).

<group>:print()

Prints all attributes of the Group object to the Logger output.

Try it yourself



[Group module demo](#)

Helpers

The helpers library consists of helper functions to make handling of certain common situations easier.

Functions

slotToBounds(slot)

Converts a preset slot to a boundary box data table.

Parameters

slot number, a numeric identifier of the page slot (1 .. 36).

Returns

array, an array consisting of x, y, width, height boundary box attributes.

boundsToSlot({x, y, width, height})

Converts a bounding box (bounds) to slot.

Parameters

`x` number, X position on the screen.

`y` number, Y position on the screen.

`width` number, width of the control.

`height` number, height of the control.

Returns

number, a numeric identifier of the page slot (1 .. 36).

Example

```
-- Move control to given slot

control = controls.get(1)
control:setBounds(helpers.slotToBounds (6))
```

Info

The Info library lets you show custom text messages in the status bar at the bottom of the screen.

Functions

`info.setText(text)`

A function to to display the text in the bottom status bar.

Parameters

`text` string, a text message to be displayed.

Example

```
-- Display an info text

info.setText("Hello world")
```

Logger

Logging is a key element for understanding what is happening inside the controller. The Electra One Lua API provides the `print()` command, which sends text messages that can be viewed in the Electra One web application. Log messages created with the `print()` function are always prefixed with `lua:` text.

In fact, these log messages are SysEx messages sent to the CTRL port. They include both a timestamp and the text of the message. For more details about console logs, please review Electra One's MIDI implementation.

Because logging uses standard SysEx messaging, users can create their own log viewers or integrate Electra logs into their own applications.

The logger output can be enabled or disabled. When disabled, no log messages are sent over MIDI. By default, the logger is disabled for performance reasons. For more information on how to manage the logger, see the section on enabling and disabling logging.

Functions

`print(text)`

A function to print text to the Electra One web application Console log view.

Parameters

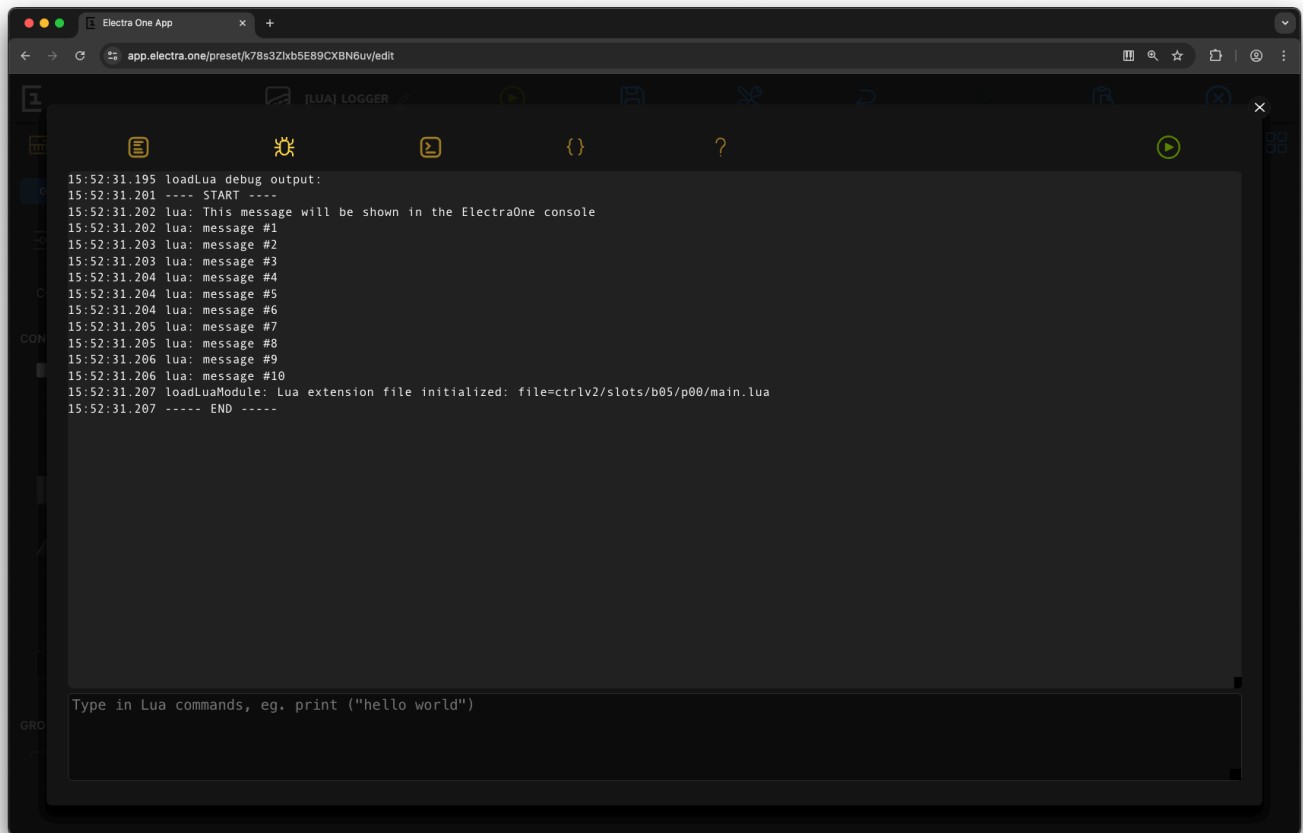
`text` string, a text message to be displayed.

Example

```
-- Printing to the console log
print("This message will be shown in the ElectraOne console")

for i = 1, 10 do
    print("message #" .. i)
end
```

The Example will produce following output in the Electra One web application Console



Try it yourself



Logger demo

Message

The Message object holds the actual MIDI or virtual message that Control's [Value object](#) sends and receives. Every Value object is linked to exactly one Message object.

Functions

<message>:setDeviceId(deviceId)

Sets the id of the device that will send and receive this message. Important: If this call creates a new [ParameterMap](#) entry, you must manually set its value.

Parameters

deviceId number, an identifier of the device to be assigned (1 .. 32).

<message>:getDeviceId()

Retrieves the identifier of the currently assigned device.

Returns

number, an identifier of the currently assigned device (1 .. 32).

<message>:setType(type)

Sets the type of the parameter that will be processed by the Message object. For a list of message types, refer to the overview in the [Globals](#) section.

Parameters

type number, an identifier of the Message parameter type (0 .. 16).

<message>:getType()

Retrieves the type of the parameter of the Message object. For a list of message types, refer to the overview in the [Globals](#) section.

Returns

number, an identifier of the Message parameter type (0 .. 16).

<message>:setParameterNumber(parameterNumber)

Sets the parameter number that will be processed by the Message object.

Parameters

parameterNumber number, a numeric identifier of the parameter (0 .. 16383).

<message>:getParameterNumber()

Retrieves the parameter number assigned to the Message object.

Returns

number, a numeric identifier of the parameter (0 .. 16383).

<message>:setValue(midiValue)

Sets the MIDI value inside the Message object. This value will be sent, processed by Lua functions, and used to update all related display values.

Parameters

`midiValue` number, the MIDI value to set and send (0 .. 16383).

`<message>:getValue()`

Retrieves the current MIDI value assigned to the Message object.

Returns

number, the current MIDI value (0 .. 16383).

`<message>:setMin(minumumValue)`

Sets the minimum MIDI value of the Message object.

Parameters

`minumumValue` number, the minimum MIDI value to be set.

`<message>:getMin()`

Retrieves the minimum MIDI value currently set for the Message object.

Returns

number, the current minimum MIDI value.

`<message>:setMax(maximumValue)`

Sets the maximum MIDI value of the Message object.

Parameters

`maximumValue` number, the maximum MIDI value to be set.

`<message>:getMax()`

Retrieves the maximum MIDI value currently set for the Message object.

Returns

number, the current maximum MIDI value.

<message>:setRange(minimumValue, maximumValue)

Changes the MIDI value range of the Message object. This function is a shortcut that avoids making two separate calls to Message `setMin()` and `setMax()` functions.

Parameters

`minimumValue` number, the minimum MIDI value to be set.

`maximumValue` number, the maximum MIDI value to be set.

<message>:setOffValue(offValue)

Sets the MIDI value that is sent when a State control, such as a Pad, is turned off.

Parameters

`offValue` number, a number representing the MIDI value used when the control is in the Off state.

<message>:getOffValue()

Retrieves the MIDI value currently set for the Message object associated with control in the Off state.

Returns

number, the current MIDI value used when the control is in the Off state.

<message>:setOnValue(onValue)

Sets the MIDI value that is sent when a State control, such as a Pad, is turned on.

Parameters

`onValue` number, a number representing the MIDI value used when the control is in the On state.

<message>:getOnValue()

Retrieves the MIDI value currently set for the Message object associated with control in the On state.

Returns

number, the current MIDI value used when the control is in the On state.

Example

```
-- Print info about the message

function valueCallback (valueObject, value)
    local message = valueObject:getMessage ()

    print ("Device Id: " .. message:getDeviceId ())
    print ("Type: " .. message:getType ())
    print ("Parameter Number: " .. message:getParameterNumber ())
    print ("Current value: " .. message:getValue ())
end
```

<message>:print()

Prints all attributes of the Message object to the Logger output.

Try it yourself



Message module demo

MIDI callbacks

MIDI callbacks are used to handle incoming MIDI messages. The general `onMessage()` callback is called for any incoming MIDI message, while specific callbacks are triggered only for particular message types.

When you define a callback function in your Lua script, the Electra One Lua interpreter automatically registers it. This registration adds a small processing overhead, so it's a good idea not to define empty callback functions that do nothing."

The first parameter of all callback functions is the `midiInput`. The `midiInput` is a data table that describes the origin of the message.

```
midiInput = {
    interface = "USB dev", -- a name of the IO interface where the messages was received
    port = 0              -- a numeric port identifier
}
```

The `midiMessage` data table is another key structure. It contains information about a MIDI message, broken down into its parts. While different types of MIDI messages use slightly different formats, every `midiMessage` includes the fields `channel`, `type`, `data1`, and `data2`.

For example, a Control Change message can be access either as:

```
midiMessage = {
    channel = 1,
```

```

    type = CONTROL_CHANGE,
    data1 = 1,
    data2 = 127
}

```

or

```

midiMessage = {
    channel = 1,
    type = CONTROL_CHANGE,
    controllerNumber = 1,
    value = 127
}

```

The full description of all `midiMessage` variants is provided later in this document.

Functions

`midi.onMessage(midiInput, midiMessage)`

A user-defined callback function to handle all types of MIDI messages.

Parameters

- `midiInput` data table, a table containing information about the source of the message, (see below).
- `midiMessage` data table, a structured representation of the incoming MIDI message.

`midi.onNoteOn(midiInput, channel, noteNumber, velocity)`

A user-defined callback function to handle Note On MIDI messages.

Parameters

- `midiInput` data table, a table containing information about the source of the message, (see below).
- `channel` number, a numeric representation of the MIDI channel (1 .. 16).
- `noteNumber` number, a numeric representation of the MIDI note (0 .. 127).
- `velocity` number, a numeric representation of the MIDI note velocity (0 .. 127).

`midi.onNoteOff(midiInput, channel, noteNumber, velocity)`

A user-defined callback function to handle Note Off MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>noteNumber</code>	number, a numeric representation of the MIDI note (0 .. 127).
<code>velocity</code>	number, a numeric representation of the MIDI note velocity (0 .. 127).

`midi.onControlChange(midiInput, channel, controllerNumber, value)`

A user-defined callback function to handle Control Change MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>controllerNumber</code>	number, a numeric representation of the Control Change parameter (0 .. 127).
<code>value</code>	number, a numeric representation of the Control Change value (0 .. 127).

`midi.onAfterTouchPoly(midiInput, channel, noteNumber, pressure)`

A user-defined callback function to handle Polyphonic Aftertouch MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>noteNumber</code>	number, a numeric representation of the MIDI note (0 .. 127).
<code>pressure</code>	number, a numeric representation of the MIDI pressure (0 .. 127).

`midi.onAfterTouchChannel(midiInput, channel, pressure)`

A user-defined callback function to handle Channel Aftertouch MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>pressure</code>	number, a numeric representation of the MIDI pressure (0 .. 127).

midi.onProgramChange(midiInput, channel, programNumber)

A user-defined callback function to handle Program Change MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>programNumber</code>	number, a numeric representation of the MIDI program number (0 .. 127).

midi.onPitchBend(midiInput, channel, programNumber)

A user-defined callback function to handle Pitch Bend MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>value</code>	number, an amount of Pitch Bend to be applied (-8191 .. 8192)).

midi.onSongSelect(midiInput, songNumber)

A user-defined callback function to handle Song Select MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>songNumber</code>	number, a numeric identifier of the song (0 .. 127).

midi.onSongPosition(midiInput, position)

A user-defined callback function to handle Song Position Pointer MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>position</code>	number, a numeric of beats from the start of the song (0 .. 16383).

midi.onClock(midiInput)

A user-defined callback function to handle Clock MIDI messages. There are 24 Clock messages per quarter note.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

midi.onStart(midiInput)

A user-defined callback function to handle Start MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

midi.onStop(midiInput)

A user-defined callback function to handle Stop MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

midi.onContinue(midiInput)

A user-defined callback function to handle Continue MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

midi.onActiveSensing(midiInput)

A user-defined callback function to handle Active Sensing MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

midi.onSystemReset(midiInput)

A user-defined callback function to handle System Reset MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

`midi.onTuneRequest(midiInput)`

A user-defined callback function to handle Tune Request MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

`midi.onSysex(midiInput)`

A user-defined callback function to handle SysEx MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

`sysexBlock` userdata, a reference to a SysexBlock object containing the SysEx data.

Example 1

```
-- Receiving MIDI messages
--
-- Receiving MIDI messages with a generic midi.onMessage() callback

function midi.onMessage(midiInput, midiMessage)
    if midiMessage.type == SYSEX then
        print("sysex message received: interface=" .. midiInput.interface)

        local sysexBlock = midiMessage.sysexBlock

        for i = 1, sysexBlock:getLength() do
            print (string.format ("data[%d] = %d", i, sysexBlock:peek(i)))
        end
    else
        -- generic approach using the data1 and data2
        print("midi message received: interface=" .. midiInput.interface ..
            " channel=" .. midiMessage.channel ..
            " type=" .. midiMessage.type ..
            " data1=" .. midiMessage.data1 ..
            " data2=" .. midiMessage.data2)

        -- Message type specific attributes
    end
end
```

```

        if midiMessage.type == NOTE_ON then
            print("noteOn received: interface=" .. midiInput.interface ..
                " channel=" .. midiMessage.channel ..
                " noteNumber=" .. midiMessage.noteNumber ..
                " velocity=" .. midiMessage.velocity)
        end
    end
end

```

Example 2

```

-- Receiving MIDI messages
--
-- Receiving MIDI messages with callbacks specific to MIDI message type

function midi.onControlChange(midiInput, channel, controllerNumber, value)
    print("controlChange received: interface=" .. midiInput.interface ..
        " channel=" .. channel ..
        " controllerNumber=" .. controllerNumber .. " value=" .. value)
end

function midi.onNoteOn(midiInput, channel, noteNumber, velocity)
    print("noteOn received: interface=" .. midiInput.interface ..
        " channel=" .. channel ..
        " noteNumber=" .. noteNumber .. " velocity=" .. velocity)
end

function midi.onNoteOff(midiInput, channel, noteNumber, velocity)
    print("noteOff received: interface=" .. midiInput.interface ..
        " channel=" .. channel ..
        " noteNumber=" .. noteNumber .. " velocity=" .. velocity)
end

function midi.onAfterTouchPoly(midiInput, channel, noteNumber, pressure)
    print("afterTouchPoly received: interface=" .. midiInput.interface ..
        " channel=" .. channel ..
        " noteNumber=" .. noteNumber .. " pressure=" .. pressure)
end

function midi.onProgramChange(midiInput, channel, programNumber)
    print("programChange received: interface=" .. midiInput.interface ..
        " channel=" .. channel ..
        " programNumber=" .. programNumber)
end

function midi.onAfterTouchChannel(midiInput, channel, pressure)
    print("afterTouchChannel received: interface=" .. midiInput.interface ..
        " channel=" .. channel ..
        " pressure=" .. pressure)
end

function midi.onPitchBendChannel(midiInput, channel, value)
    print("pitchBend received: interface=" .. midiInput.interface ..

```

```

        " channel=" .. channel ..
        " value=" .. value)
end

function midi.onSongSelect(midiInput, songNumber)
    print("songSelect received: interface=" .. midiInput.interface ..
        " songNumber=" .. songNumber)
end

function midi.onSongPosition(midiInput, position)
    print("songPosition received: interface=" .. midiInput.interface ..
        " position=" .. position)
end

function midi.onClock(midiInput)
    print("midi clock received: interface=" .. midiInput.interface)
end

function midi.onStart(midiInput)
    print("start received: interface=" .. midiInput.interface)
end

function midi.onStop(midiInput)
    print("stop received: interface=" .. midiInput.interface)
end

function midi.onContinue(midiInput)
    print("continue received: interface=" .. midiInput.interface)
end

function midi.onActiveSensing(midiInput)
    print("active sensing received: interface=" .. midiInput.interface)
end

function midi.onSystemReset(midiInput)
    print("system reset received: interface=" .. midiInput.interface)
end

function midi.onTuneRequest(midiInput)
    print("tune request received: interface=" .. midiInput.interface)
end

function midi.onSysex(midiInput, sysexBlock)
    print ("sysex message received: interface=" .. midiInput.interface)

    -- print the received data
    for i = 1, sysexBlock:getLength() do
        print(string.format ("data[%d] = %d", i, sysexBlock:peek(i)))
    end
end

```

```
end
end
```

MIDI functions

The MIDI library provides functions for sending raw MIDI messages. There are two ways to send MIDI messages: by composing a `midiMessage` data table and passing it to the generic `midi.sendMessage()` function, or by calling dedicated functions for specific message types, such as `midi.sendNoteOn()`.

MIDI messages can be sent to all of Electra's interfaces (USB Dev, USB Host, MIDI IO) or just one specific interface. This depends on whether you pass an Interface Type as the first argument to the function.

The following call will send the Note Off MIDI message to all available MIDI interfaces:

```
midi.sendNoteOff(port, channel, noteNumber, velocity)
```

while

```
midi.sendNoteOff(interface, port, channel, noteNumber, velocity)
```

will send it to a specific MIDI interface.

Functions

Note

For simplicity, functions will be listed here without the leading MIDI interface parameter.

`midi.sendMessage(port, midiMessage)`

A function to send a MIDI message defined in `midiMessage` data table.

Parameters

`port` enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

`midiMessage` data table, a structured representation of the incoming MIDI message.

`midi.sendNoteOn(port, channel, noteNumber, velocity)`

A function to send a Note On MIDI message.

Parameters

`port` enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>noteNumber</code>	number, a numeric representation of the MIDI note (0 .. 127).
<code>velocity</code>	number, a numeric representation of the MIDI note velocity (0 .. 127).

`midi.sendNoteOff(port, channel, noteNumber, velocity)`

A function to send a Note Off MIDI message.

Parameters

<code>port</code>	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>noteNumber</code>	number, a numeric representation of the MIDI note (0 .. 127).
<code>velocity</code>	number, a numeric representation of the MIDI note velocity (0 .. 127).

`midi.sendControlChange(port, channel, controllerNumber, value)`

A function to send a Control Change MIDI message.

Parameters

<code>port</code>	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>controllerNumber</code>	number, a numeric representation of the Control Change parameter (0 .. 127).
<code>value</code>	number, a numeric representation of the Control Change value (0 .. 127).

`midi.sendAfterTouchPoly(port, channel, noteNumber, pressure)`

A function to send a Polyphonic Aftertouch MIDI message.

Parameters

<code>port</code>	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
<code>channel</code>	number, a numeric representation of the MIDI channel (1 .. 16).
<code>noteNumber</code>	number, a numeric representation of the MIDI note (0 .. 127).
<code>pressure</code>	number, a numeric representation of the MIDI pressure (0 .. 127).

midi.sendAfterTouchChannel(port, channel, pressure)

A function to send a Channel Aftertouch MIDI message.

Parameters

port	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
channel	number, a numeric representation of the MIDI channel (1 .. 16).
pressure	number, a numeric representation of the MIDI pressure (0 .. 127).

midi.sendProgramChange(port, channel, programNumber)

A function to send a Program Change MIDI message.

Parameters

port	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
channel	number, a numeric representation of the MIDI channel (1 .. 16).
programNumber	number, a numeric representation of the MIDI program number (0 .. 127).

midi.sendPitchBend(port, channel, programNumber)

A function to send a Pitch Bend MIDI message.

Parameters

port	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
channel	number, a numeric representation of the MIDI channel (1 .. 16).
value	number, an amount of Pitch Bend to be applied (-8191 .. 8192)).

midi.sendSongSelect(port, songNumber)

A function to send a SongSelect MIDI message.

Parameters

port	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
songNumber	number, a numeric identifier of the song (0 .. 127).

midi.onSongSelect(port, songNumber)

A function to send a Song Select MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

songNumber number, a numeric identifier of the song (0 .. 127).

midi.sendSongPosition(port, position)

A function to send a Song Position Pointer MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

position number, a numeric of beats from the start of the song (0 .. 16383).

midi.sendClock(port)

A function to send single Clock MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

midi.sendStart(port)

A function to send a Start MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

midi.sendStop(port)

A function to send a Stop MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

midi.sendContinue(port)

A function to send a Continue MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

midi.sendActiveSensing(port)

A function to send a Active Sensing MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

midi.sendSystemReset(port)

A function to send a System Reset MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

midi.sendTuneRequest(port)

A function to send a Tune Request MIDI message.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

midi.sendSysex(port)

A function to send a SysEx MIDI messages.

Parameters

port enum, a numeric representation of the MIDI port [PORT_1, PORT_2].

sysexBlock userdata, a reference to a SysexBlock object containing the SysEx data.

midi.sendNrpn(port, channel, parameterNumber, value, lsbFirst, reset)

A function to send a NRPN MIDI message.

Parameters

port	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
channel	number, a numeric representation of the MIDI channel (1 .. 16).
parameterNumber	number, a numeric representation of the NRPN parameter (0 .. 16383).
value	number, a numeric representation of the Control Change value (0 .. 16383).
lsbFirst	boolean, when true the lsb and msb bytes will be swapped.
reset	boolean, when true the RPN reset will be sent at the end of the NRPN message.

midi.sendRpn(port, channel, parameterNumber, value)

A function to send a RPN MIDI message.

Parameters

port	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
channel	number, a numeric representation of the MIDI channel (1 .. 16).
parameterNumber	number, a numeric representation of the NRPN parameter (0 .. 16383).
value	number, a numeric representation of the Control Change value (0 .. 16383).

midi.sendControlChange14(port, channel, controllerNumber, value, lsbFirst)

A function to send a 14-bit Control Change MIDI message.

Parameters

port	enum, a numeric representation of the MIDI port [PORT_1, PORT_2].
channel	number, a numeric representation of the MIDI channel (1 .. 16).
controllerNumber	number, a numeric representation of the 14-bit Control Change parameter (0 .. 31).
value	number, a numeric representation of the Control Change value (0 .. 16383).
lsbFirst	boolean, when true the lsb and msb bytes will be swapped.

midi.flush()

A function to flush (force the sending of) all data from MIDI queues. Normally, outbound MIDI data is sent at regular time intervals. However, in certain situations, such as when a blocking call interrupts the processing users may wish to flush their MIDI data before entering the blocking call.

Example

```
-- Sending MIDI messages using the sendMessage ()

-- Control Change
midiMessage = {
    channel = 1,
    type = CONTROL_CHANGE,
    controllerNumber = 1,
    value = 127
}
midi.sendMessage(PORT_1, midiMessage)

-- Note On
midiMessage = {
    channel = 1,
    type = NOTE_ON,
    noteNumber = 60,
    velocity = 100
}
midi.sendMessage(PORT_1, midiMessage)

-- Note Off
midiMessage = {
    channel = 1,
    type = NOTE_OFF,
    noteNumber = 60,
    velocity = 100
}
midi.sendMessage(PORT_1, midiMessage)

-- Program Change
midiMessage = {
    channel = 1,
    type = PROGRAM_CHANGE,
    programNumber = 10
}
midi.sendMessage(PORT_1, midiMessage)

-- Pitch Bend
midiMessage = {
```

```

    channel = 1,
    type = PITCH_BEND,
    value = 513
}
midi.sendMessage(PORT_1, midiMessage)

-- Poly Pressure
midiMessage = {
    channel = 1,
    type = POLY_PRESSURE,
    noteNumber = 60,
    pressure = 100
}
midi.sendMessage(PORT_1, midiMessage)

-- Channel Pressure
midiMessage = {
    channel = 1,
    type = CHANNEL_PRESSURE,
    pressure = 64
}
midi.sendMessage(PORT_1, midiMessage)

-- Clock
midiMessage = {
    type = CLOCK
}
midi.sendMessage(PORT_1, midiMessage)

-- Start
midiMessage = {
    type = START
}
midi.sendMessage(PORT_1, midiMessage)

-- Stop
midiMessage = {
    type = STOP
}
midi.sendMessage(PORT_1, midiMessage)

-- Continue
midiMessage = {
    type = CONTINUE
}
midi.sendMessage(PORT_1, midiMessage)

-- Active Sensing
midiMessage = {

```

```

    type = ACTIVE_SENSING
}
midi.sendMessage(PORT_1, midiMessage)

-- System Reset
midiMessage = {
    type = RESET
}
midi.sendMessage(PORT_1, midiMessage)

-- Song Select
local ss = {
    type = SONG_SELECT,
    songNumber = 20
}
midi.sendMessage(PORT_1, ss)

-- Song Position
midiMessage = {
    type = SONG_POSITION,
    position = 10
}
midi.sendMessage(PORT_1, midiMessage)

-- Tune Request
midiMessage = {
    type = TUNE_REQUEST
}
midi.sendMessage(PORT_1, midiMessage)

```

Example

```

-- Sending MIDI messages out

print ("Sending MIDI out demo loaded")

-- Control change
midi.sendControlChange(PORT_1, 1, 10, 64)

-- Notes
midi.sendNoteOn(PORT_1, 1, 60, 100)
midi.sendNoteOff(PORT_1, 1, 60, 100)

-- Program change
midi.sendProgramChange(PORT_1, 1, 10)

-- Pitch bend
midi.sendPitchBend(PORT_1, 1, 513)

```

```

-- Polyphonic aftertouch
midi.sendAfterTouchPoly(PORT_1, 1, 60, 100)

-- Channel aftertouch
midi.sendAfterTouchChannel(PORT_1, 1, 100)

-- Send NRPN
midi.sendNrpn(PORT_1, 1, 512, 8192)

-- Send RPN
midi.sendRpn(PORT_1, 1, 1, 4096)

-- Send Control change 14bit
midi.sendControlChange14Bit(PORT_1, 1, 1, 2048)

-- Clock
midi.sendClock(PORT_1)

-- Start
midi.sendStart(PORT_1)

-- Stop
midi.sendStop(PORT_1)

-- Continue
midi.sendContinue(PORT_1)

-- Active sensing
midi.sendActiveSensing(PORT_1)

-- System reset
midi.sendSystemReset(PORT_1)

-- Song select
midi.sendSongSelect(PORT_1, 1)

-- Song position
midi.sendSongPosition(PORT_1, 200)

-- Tune request
midi.sendTuneRequest(PORT_1)

-- SysEx
midi.sendSysex(PORT_1, { 67, 32, 0 })

```

Overlays

The Overlays module provides functionality for managing preset overlays. An overlay is a list of MIDI values, with each entry containing a MIDI value, a text label, and optional bitmap data. Overlays provide options for List controls and can also replace display values for Faders

Functions

overlays.get(overlayId)

Retrieves a reference to an Overlay object (Lua userdata). The overlay id number can be found in the control properties panel in the Preset Editor.

Parameters

overlayId number, an overlay identifier.

Returns

userdata, an object representing the overlay.

overlays.create(overlayId, overlayData)

Creates a new Overlay object and returns a reference (userdata) to the object.

Parameters

overlayId number, an identifier of the overlay to be created.

overlayData table, a Lua table with the overlay items data.

Returns

userdata, an object representing the overlay.

The Overlay Lua table must be structured as shown below. **value** is the MIDI value, **label** is a text label associated with the MIDI value.

```
overlayData = {
  { value = 1, label = "Room" },
  { value = 2, label = "Hall" },
  { value = 3, label = "Plate" },
  { value = 4, label = "Spring" }
}
```

Overlay

An Overlay object stores the data and functions used to manage an overlay.

<overlay>.print()

Prints all attributes of the Overlay object to the Logger output.

Example

```
-- Define reverb and delay types with associated values and labels
local listReverbTypes = {
    { value = 1, label = "Room" },
    { value = 2, label = "Hall" },
    { value = 3, label = "Plate" },
    { value = 4, label = "Spring" }
}

-- Create a new overlay
overlays.create(2, listReverbTypes)
```

Try it yourself



Overlays module demo

Pages

The pages module allows you to get information about pages, check their status, and switch from one page to another.

Functions

`pages.get(pageId)`

Retrieves a reference to a Page object (Lua userdata). A Page represents a named collection of controls and groups displayed on the screen at the same time.

Parameters

`pageId` number, a numeric identifier of the Page (1 .. 12).

Returns

userdata, an object representing the page.

`pages.getAll()`

Retrieves an array of Page object references (Lua userdata) used in the preset.

Returns

array, a list of references to all pages in the preset.

`pages.getActive()`

Retrieves a reference to the currently active Page object (Lua userdata).

Returns

userdata, a reference to the currently active Page.

pages.display(pageId)

Changes the currently displayed page.

Parameters

`pageId` number, a numeric identifier of the Page (1 .. 12).

pages.setActiveControlSet(controlSetId)

Switches the active Control set on the current page. It acts as a shortcut so you don't have to manually query the active page first.

Parameters

`controlSetId` enum, a numeric identifier of the Control set [CONTROL_SET_1, CONTROL_SET_2, CONTROL_SET_3].

pages.getActiveControlSet()

Retrieves the information about currently selected Control set of the active page.

Returns

enum, a numeric identifier of the Control set [CONTROL_SET_1, CONTROL_SET_2, CONTROL_SET_3].

pages.onChange(newPageId, oldPageId)

A callback function that runs automatically when the user switches to a different page. The function is called only if Page change notifications have been subscribed to. For more details, see [Events](#) section.

Parameters

`newPageId` number, a numeric identifier of the Page being activated (1–12).

`oldPageId` number, a numeric identifier of the Page being left (1–12).

Example

```
-- Retrieve a reference to given page
```

```
local page = pages.get(3)
```

Page

A Page object stores its own data and provides functions to update and manage it.

Functions

<page>:getId()

Retrieves an identifier of the Page. The identifier is assigned to the Page in the preset JSON.

Returns

number, a numeric identifier of the Page (1 .. 12).

<page>:setName(name)

Sets a new name of the Page.

Parameters

<code>name</code>	string, the new name to assign to the Page.
-------------------	---

<page>:getName()

Retrieves the current name of the Page.

Returns

string, the current name of the Page.

Example

```
-- change name of a pge

local page = pages.get(1)

page:setName("LPF")
print("page name: " .. page:getName())
```

`<page>:print()`

Prints all attributes of the Page object to the Logger output.

Try it yourself



Page module demo

Parameter Map

The Parameter Map is the central part of the Electra Controller firmware. It keeps track of all parameter values across connected devices. Whenever a MIDI message is received, a knob is turned, or a value is changed by touch, the Parameter Map records the change, updates everything that depends on it, and sends out new MIDI messages.

Functions

`parameterMap.resetAll()`

Removes all entries from the Parameter Map, leaving it completely empty.

`parameterMap.resetDevice(deviceId)`

Resets all parameters of the given device to their initial (unset) state. Important: 'Unset' does not mean 0 or a default value; it means the value is empty and undefined.

Parameters

`deviceId` number, an identifier of the device to reset (1 .. 32).

`parameterMap.set(deviceId, type, parameterNumber, midiValue)`

Updates the MIDI value of a specific Parameter Map entry. This will automatically send the MIDI messages and call any related Lua function callbacks.

Parameters

`deviceId` number, an identifier of the device to reset (1 .. 32).

`type` number, an identifier of the Message parameter type (0 .. 16).

`parameterNumber` number, a numeric identifier of the Message parameter (0 .. 16383).

`midiValue` number, the MIDI value to set and send (0 .. 16383).

parameterMap.apply(deviceId, type, parameterNumber, midiValue)

Updates a specific Parameter Map entry by applying a MIDI value fragment. It uses a logical OR between the current value and the new fragment. Afterward, the system automatically sends the MIDI messages and calls any related Lua function callbacks.

Parameters

deviceId	number, an identifier of the device to reset (1 .. 32).
type	number, an identifier of the Message parameter type (0 .. 16).
parameterNumber	number, a numeric identifier of the Message parameter (0 .. 16383).
midiValueFragment	number, the MIDI value to apply (0 .. 16383).

parameterMap.modulate(deviceId, type, parameterNumber, modulationValue, depth)

Temporarily changes (modulates) the MIDI value of a Parameter Map entry. The modulated value is sent, but it is not saved in the Parameter Map or processed by Lua callbacks or value formatters.

Parameters

deviceId	number, an identifier of the device to reset (1 .. 32).
type	number, an identifier of the Message parameter type (0 .. 16).
parameterNumber	number, a numeric identifier of the Message parameter (0 .. 16383).
modulationValue	float, a modulation signal to be applied (-1.0 .. 1.0).
depth	number, a depth of the modulation signal (0 .. 100).

parameterMap.get(deviceId, type, parameterNumber)

Retrieves the current MIDI value stored in the Parameter Map entry. This function is equivalent to calling `message.getValue()` on a Message object with the corresponding attributes.

Parameters

deviceId	number, an identifier of the device to reset (1 .. 32).
type	number, an identifier of the Message parameter type (0 .. 16).
parameterNumber	number, a numeric identifier of the Message parameter (0 .. 16383).

parameterMap.getValues(deviceId, type, parameterNumber)

Retrieves a list of all [Value](#) objects linked to the Parameter Map entry. These are the Value objects that will be updated when the MIDI value changes.

Parameters

<code>deviceId</code>	number, an identifier of the device to reset (1 .. 32).
<code>type</code>	number, an identifier of the Message parameter type (0 .. 16).
<code>parameterNumber</code>	number, a numeric identifier of the Message parameter (0 .. 16383).

parameterMap.onChange(valueObjects, origin, midiValue)

Retrieves a list of all [Value](#) objects linked to the Parameter Map entry. These are the Value objects that will be updated when the MIDI value changes.

Parameters

<code>valueObjects</code>	array, a list of references to userdata Value objects associated with the ParameterMap entry.
<code>origin</code>	number, a numeric identifier of the change origin (see [Globals](./luaext.html#globals) for details).
<code>midiValue</code>	number, the current MIDI value of the ParameterMap entry (0 .. 16383).

Example

```
-- Display info about the change in the ParameterMap

function parameterMap.onChange(valueObjects, origin, midiValue)
    print(string.format ("a new midiValue %d from origin %d",
        midiValue, origin))

    for i, valueObject in ipairs(valueObjects) do
        local control = valueObject:getControl()
        print(string.format("affects control value %s.%s",
            control:getName(), valueObject:getId()))
    end
end
```

parameterMap.keep()

Saves the current state of the Parameter Map so it can be recalled later. The data stays safely stored in the controller even when it is powered off.

`parameterMap.recall()`

Recalls state that was previously saved with `parameterMap.keep()` function call.

`parameterMap.forget()`

Removes and forgets state that was previously saved with `parameterMap.keep()` function call.

`parameterMap.print()`

Prints all attributes of the ParameterMap entries to the Logger output.

Patch

This library helps you request patch dumps and process SysEx MIDI messages that contain patch data. The `patch.onResponse()` function is called automatically when a SysEx message matches the response header you defined in the preset JSON.

To use patch callbacks, you must first create a Patch object in the Device object defined in your preset JSON.

The example below shows the simplest Patch setup. Here, `patch.onResponse()` will be triggered whenever a SysEx message begins with the bytes `67`, `0`, `0`, `1`, `27`.

```

"patch": [
  {
    "responses": [
      {
        "id": 1,
        "header": [
          67,
          0,
          0,
          1,
          27
        ]
      }
    ]
  }
]

```

Functions

patch.onRequest(device)

A callback used to send a patch request to a specific device. The function is called when the [PATCH REQUEST] button is pressed, and it sends the request to all devices that have a patch request defined in their patch configuration.

Parameters

`device` data table, a device description data structure (see below).

patch.onResponse(device, responseId, sysexBlock)

A callback to handle incoming SysEx message that matched the Patch response definition.

Parameters

`device` data table, a device description data structure (see below).

`responseId` number, an identifier of the response, as defined in the preset JSON.

`sysexBlock` userdata, a reference to a SysexBlock object containing the received and matched SysEx message.

patch.requestAll()

Sends patch requests to all connected devices.

Example

```

-- Issue a patch requests
patch.requestAll()

-- Send a program change
function patch.onRequest(device)
    print ("Requesting patches...");

    if (device.id == 1) then
        midi.sendProgramChange(PORT_1, device.channel, 10)
    end
end

-- Parse an incoming response
function patch.onResponse(device, responseId, sysexBlock)
    -- print the header information
    print("device id = " .. device.id)
    print("device channel = " .. device.channel)
    print("device port = " .. device.port)

```

```

print("responseId = " .. responseId)
print("manufacturer Id = " .. sysexBlock:getManufacturerSysexId())

-- print the received data
for i = 1, sysexBlock:getLength() do
    print("data[" .. i .. "] = " .. sysexBlock:peek(i))
end

-- update two parameters
parameterMap.set(device.id, PT_CC7, 1, sysexBlock:peek(7));
parameterMap.set(device.id, PT_CC7, 2, sysexBlock:peek(8));
end

```

Device data table

```

device = {
    id = 1,           -- a device Id
    port = 0,         -- a numeric port identifier
    channel = 1,      -- a channel number
}

```

Preset

The preset library offers functions and callbacks to manage events that happen when working with presets.

Functions

preset.onLoad()

A callback function that is called immediately after the preset is loaded, but before the default values are initialized.

preset.onReady()

A callback function that is called when the preset is fully initialized and ready to be used.

preset.onExit()

A callback function that is called when preset is stopped or removed from the controller. Note, it is not called when user switches to another preset.

Tables

preset.userFunctions

The `preset.userFunctions` table allows you to define up to twelve custom Lua functions that can be triggered from the Preset Menu on the Electra One controller.

Each function is assigned to one of the predefined keys: `pot1` through `pot12`. These correspond to the twelve on-screen buttons in the Preset Menu and match the layout of the physical knobs.

Each entry is a table that defines:

- `call` – The Lua function that will be executed when the button is triggered. This is a required field.
- `name` – The label that will appear on the on-screen button. This is a required field.
- `close` – A boolean value that, when set to true, causes the Preset Menu to close after the function has been executed. This field is optional.

Only the assigned slots will be displayed in the menu. Buttons with a user function appear in blue, and when enabled in the configuration, functions can also be triggered by knob touch.

For more information on enabling knob touch interaction, see: Settings → Interface → Pot Touch Selections.

Example

```
-- Register the functions for use in the Preset Menu
preset.userFunctions = {
  pot1 = {
    call = printHello,
    name = "Hello",
    close = true
  },
  pot2 = {
    call = printHi,
    name = "Hi",
    close = false
  },
  pot12 = {
    call = printGoodBye,
    name = "GoodBye",
    close = false
  }
}
```

SysEx byte function

A SysEx byte function is used in SysEx templates, patch requests, and patch response headers to calculate and insert bytes at specific positions within a SysEx message.

The function is given information about the device and a parameter number, and it must return one byte containing a 7-bit value.

Example preset JSON

This example shows how Lua functions are used in both the patch request and the response header. Here, they are used to request and match a SysEx patch dump from a TX7 on a specific MIDI channel.

```
"devices": [
  {
    "id": 1,
    "name": "Yamaha DX7",
    "port": 1,
    "channel": 16,
    "patch": [
      {
        "request": [
          "43",
          {
            "type": "function",
            "name": "getRequestByte"
          },
          "00"
        ],
        "responses": [
          {
            "header": [
              "43",
              {
                "type": "function",
                "name": "getResponseByte"
              },
              "00",
              "01",
              "1B"
            ],
            "rules": [
              {
                "id": 136,
                "pPos": 0,
                "byte": 136,
                "bPos": 0,
                "size": 1,
                "msg": "sysex"
              }
            ]
          }
        ]
      }
    ]
  }
]
```

The following snippet shows how to use the Lua SysEx byte function in the SysEx template.

```

"values": [
  {
    "id": "value",
    "message": {
      "type": "sysex",
      "deviceId": 1,
      "data": [
        "43",
        {
          "type": "function",
          "name": "getChannelByte"
        },
        "00",
        "66",
        {
          "type": "value",
          "rules": [
            {
              "parameterNumber": 102,
              "bitWidth": 5,
              "byteBitPosition": 0
            }
          ]
        }
      ],
      "parameterNumber": 102,
      "min": 0,
      "max": 31
    },
    "min": 0,
    "max": 31
  }
]

```

Functions

<sysexByteFunction>(deviceObject, parameterNumber)

A function that calculates a SysEx byte and inserts it into a specific position of a SysEx message. It is given information about the device and the parameter number and must return a 7-bit value as a single byte.

Parameters

- | | |
|------------------------------|---|
| <code>deviceObject</code> | userdata, a reference to a userdata Device object. |
| <code>parameterNumber</code> | number, a numeric identifier of the Message parameter (0 .. 16383). |

Example

```
-- returns a byte that TX7 uses to identify the MIDI channel

function getChannelByte(device)
    return (0x10 + (device:getChannel() - 1))
end
```

SysexBlock

An object designed to handle SysEx messages.

In contrast to simple byte arrays, SysexBlock provides efficient tools for working with large SysEx messages, offering stream-like operations such as `read`, `write`, and `peek`.

Functions

<sysexBlock>:getLength()

Retrieves the total length of the SysEx message. The length includes the leading and trailing 0xF0 and 0xF7 bytes.

Returns

number, size of the SysEx message in bytes.

<sysexBlock>:getManufacturerSysexId()

Retrieves the SysEx manufacturer identifier from the SysexBlock object. It is either one byte-wide number or a number composed of the three bytes with LSB at position 3 and MSB at position 1.

Returns

number, a numeric identifier of the manufacturer's SysEx Id.

<sysexBlock>:seek(position)

Sets the SysexBlock's current position at given offset. The leading 0xF0 byte is at position 0.

Parameters

`position` number, the position of the read/write pointer in the SysexBlock.

<sysexBlock>:read()

Reads one byte from the current position within the SysexBlock. The read/write pointer is automatically increased after the read is completed.

Returns

number, a byte stored at current read/write pointer in the SysexBlock, or -1 on failure.

<sysexBlock>:peek()

Reads one byte from the current position within the SysexBlock. The read/write pointer is NOT automatically increased.

Returns

number, a byte stored at current read/write pointer in the SysexBlock, or -1 on failure.

<sysexBlock>:write(byte)

Writes one byte to the current position of the SysexBlock. The read/write pointer is automatically increased after the write is completed.

Parameters

byte number, a byte value to be stored at the position of the read/write pointer in the SysexBlock.

<sysexBlock>:close()

Closes the SysexBlock, stopping any further writing. You must always close a SysexBlock before using it.

System

The System module offers functions that control system behavior and manage tasks.

Functions

persist(table)

Saves a Lua table to non-volatile memory, allowing the data to be restored after the controller is powered off or restarted.

Parameters

`table` data table, a Lua table to be saved for later use.

`recall(table)`

Loads saved data from non-volatile memory and updates the provided Lua table with the retrieved values.

Parameters

`table` Data table, a Lua table to be updated with the restored data.

`yield()`

Suspends the execution of the current function, allowing other tasks to run. The function that called `yield()` will automatically resume once all higher-priority tasks have been completed.

`delay(millis)`

Waits for given amount of time.

Parameters

`millis` number, a time delay to wait milliseconds (1 .. 5000)

Timer

The timer library lets you run tasks repeatedly over time. It calls the `timer.onTick()` function at a set time interval or based on BPM (beats per minute). You can use it to create MIDI clocks, LFOs, and other ongoing processes. The timer is disabled by default, and the starting rate is 120 BPM.

Functions

`timer.enable()`

Enables the timer. Once the timer is enabled, the `timer.onTick()` is run at given time periods.

`timer.disable()`

Disables the timer. The period of the timer is kept.

timer.isEnabled()

Retrieves the information whether or not the timer is enabled.

Returns

boolean, true when the timer is enabled.

timer.setPeriod(period)

Sets the period in milliseconds at which the `timer.onTick()` function will be called.

Parameters

`period` number, a period specified in milliseconds (10..60000).

timer.getPeriod()

Retrieves the current period in milliseconds at which the `timer.onTick()` function is called.

Returns

number, a period specified in milliseconds (10..60000).

timer.setBpm(bpm)

Sets the timer rate in beats per minute (BPM) at which the `timer.onTick()` function will be called."

Parameters

`bpm` number, a BPM use to run the timer function.

timer.getBpm()

Retrieves the current timer rate in beats per minute (BPM) used to call the `timer.onTick()` function.

Returns

number, a BPM use to run the timer function.

timer.onTick()

A user function callback that will be run at the start of every timer cycle.

Example

```
-- A naive MIDI LFO implementation

faderValue = 0

timer.enable()
timer.setBpm(120 * 16)

function timer.onTick ()
    parameterMap.set(1, PT_CC7, 1, faderValue)
    faderValue = math.fmod(faderValue + 1, 127)
end
```

Transport

The transport library works like the timer but doesn't generate its own tick signal. Instead, it uses MIDI real-time system and clock messages from an external source. This lets you create processes that stay in sync with an external MIDI clock. The transport is disabled by default.

Functions

transport.enable()

Enables the transport. Once the transport is enabled, the `transport.onClock()` is run at according to the received MIDI clock messages.

transport.disable()

Disables the transport. The `transport.onClock()` will not be called until it is enabled again.

transport.isEnabled()

Retrieves the information whether or not the transport is enabled.

Returns

boolean, true when the transport is enabled.

transport.onClock(midiInput)

A user-defined callback function that is called on every incoming MIDI Clock message. There are 24 Clock messages per quarter note.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

transport.onStart(midiInput)

A user-defined callback function to handle Start MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

transport.onStop(midiInput)

A user-defined callback function to handle Stop MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

transport.onContinue(midiInput)

A user-defined callback function to handle Continue MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

transport.onSongSelect(midiInput, songNumber)

A user-defined callback function to handle Song Select MIDI messages.

Parameters

`midiInput` data table, a table containing information about the source of the message, (see below).

`songNumber` number, a numeric identifier of the song (0 .. 127).

transport.onSongPosition(midiInput, position)

A user-defined callback function to handle Song Position Pointer MIDI messages.

Parameters

<code>midiInput</code>	data table, a table containing information about the source of the message, (see below).
<code>position</code>	number, a numeric of beats from the start of the song (0 .. 16383).

Example

```
faderValue = 0

function preset.onLoad()
  if (not transport.isEnabled()) then
    transport.enable()
  end

  print("Transport enabled: " .. (transport.isEnabled() and "yes" or "no"))
end

function transport.onClock(midiInput)
  parameterMap.set(1, PT_CC7, 1, faderValue)
  faderValue = faderValue + 1

  if (faderValue > 127) then
    faderValue = 0
  end
end

function transport.onStart(midiInput)
  print("Start")
end

function transport.onStop(midiInput)
  print("Stop")
end

function transport.onContinue(midiInput)
  print("Continue")
end

function transport.onSongSelect(midiInput, songNumber)
  print("Song select " .. songNumber)
end

function transport.onSongPosition(midiInput, position)
  print("Song position " .. position)
end
```

MIDI input data table

For more details refer to [Data structures](#) section.

```

midiInput = {
  interface = USB_DEV, -- an numeric MIDI interface identifier
  port = 0           -- a numeric port identifier
}

```

Value

A Value object represents a single data value inside a Control. Each Value is identified by a valueId, and a Control can have one or more Values. The Value object describes the data users can change through interaction and provides functions to access and modify that data."

Functions

<value>:getId()

Retrieves the identifier of the Value - valueId. The identifier is assigned to the Value in the preset JSON.

Returns

string, a text that identifies a specific value of a control.

<value>:setDefault(defaultValue)

Sets the default display value of the Value object.

Parameters

defaultValue number, the default display value to be set.

<value>:getDefault()

Retrieves the default display value currently set for the Value object.

Returns

number, the current default display value.

<value>:setMin(minimumValue)

Sets the minimum display value of the Value object.

Parameters

`minumumValue` number, the minimum display value to be set.

<value>:getMin()

Retrieves the minimum display value currently set for the Value object.

Returns

number, the current minimum display value.

<value>:setMax(maximumValue)

Sets the maximum display value of the Value object.

Parameters

`maximumValue` number, the maximum display value to be set.

<value>:getMax()

Retrieves the maximum display value currently set for the Value object.

Returns

number, the current maximum display value.

<value>:setRange(minimumValue, maximumValue, defaultValue, applyToMessage)

Changes the display value range of the Value object and, if needed, updates the underlying [Message](#) object as well. This function is a shortcut that avoids making many separate calls to Value and Message setters.

Parameters

`minimumValue` number, the minimum display value to be set.

`maximumValue` number, the maximum display value to be set.

`defaultValue` number, the default display value to be set.

`applyToMessage` boolean, when true, the same range is applied to underlying Message object.

<value>:setOverlayId(overlayId)

Assigns an overlay list to the Value object. The overlayId is defined in the preset JSON or created using the `overlays.create()` function.

Parameters

`overlayId` number, an identifier of the overlay.

<value>:getOverlayId()

Retrieves the overlayId currently assigned to the Value object.

Returns

number, an identifier of the overlay.

Example

```

-- swap overlay lists of two controls

local listA = controls.get(1)
local listB = controls.get(2)

local valueA = listA.getValue("value")
local valueB = listB.getValue("value")

print("list A: " .. valueA:getOverlayId())
print("list B: " .. valueB:getOverlayId())

valueB:setOverlayId(1)
valueA:setOverlayId(2)

```

<value>:overrideValue(valueText)

Replaces the current value with custom text on the control's display. This custom text also overrides the output from [Value Formatters](#).

Parameters

`valueText` string, a text to be displayed instead of current display value.

<value>:cancelOverride()

Clears the overridden text set by `<value>:overrideValue()` and restores the display of the current value."

`<value>:getMessage()`

Retrieves the [Message object](#) assigned to the Value object.

Returns

userdata, a reference to the Message object.

Example

```
-- Get the message associated with the release value

local value = control:getValue("release")
local message = value:getMessage()
```

`<value>:getControl()`

Retrieves the [Control](#) object that this Value belongs to.

Returns

userdata, a reference to the Control object.

`<value>:getValue()`

Retrieves the current display value of the Value object.

Returns

number, the current display value.

`<value>:print()`

Prints all attributes of the Value object to the Logger output.

Try it yourself



Value module demo

Value formatters

A value formatter is a custom function that formats how a control's value is displayed. It receives a display value as input and returns a new value as a string. This allows users to customize how information appears on the screen in many different ways.

To use a formatter, you must assign it to a `Value` in the preset JSON by adding a formatter attribute to the `Value` object.

The value formatter runs automatically whenever the underlying MIDI value changes.

Example preset JSON

```
"values": [
  {
    "message": {
      "deviceId": 1,
      "type": "cc7",
      "parameterNumber": 2,
      "min": 0,
      "max": 127
    },
    "id": "value",
    "min": 0,
    "max": 127,
    "formatter": "formatFractions"
  }
]
```

For more detailed information about the preset JSON, visit the [Preset JSON format](#) page.

Functions

<formatterFunction>(valueObject, value)

A user-defined function that transforms the input display value into a text string shown on the screen.

Parameters

`valueObject` userdata, a reference to a userdata Value object that was changed.

`value` number, a new display value to be formatted.

Example

```
-- Convert number to a range with decimal numbers
function formatFractions(valueObject, value)
  return (string.format("%.1f", value / 20))
end

-- add percentage to the value
function addPercentage(valueObject, value)
```

```
return (value .. "%")
end
```

Value function callbacks

A value function callback is a user-defined function that lets you run custom actions whenever a control's value changes.

To use a callback, you must assign it to a Value in the preset JSON by adding a function attribute to the Value object. You can think of a callback as a flexible alternative to a Message:

while a Message sends a fixed MIDI command, a function runs dynamic Lua code when the value changes.

Example preset JSON

```
"values": [
  {
    "message": {
      "deviceId": 1,
      "type": "cc7",
      "parameterNumber": 2,
      "min": 0,
      "max": 127
    },
    "id": "attack",
    "min": 0,
    "max": 127,
    "function": "highlightOnOverload"
  }
]
```

For more detailed information about the preset JSON, visit the [Preset JSON format](#) page.

Functions

<customFunction>(valueObject, value)

A user-defined function that to execute Lua code whenever control's value changes.

Parameters

valueObject userdata, a reference to a userdata Value object that was changed.

value number, a new display value to be processed.

Example

```
function highlightOnOverload (valueObject, value)
  if (value > 64) then
    control:setColor (ORANGE)
```



```

else
    control:setColor (WHITE)
end
end

```

Window

The Window library gives you control over the graphic component repainting process.

Functions

`window.repaint()`

Forces repaint of the current window.

`window.stop()`

Stops automatic repainting of components. This helps when updating many controls at once. By using `window.stop()` before and `window.resume()` after your updates, you can speed up the process and display all changes together.

`window.resume()`

Resumes the process of automatic repainting of components. The complete repaint of whole window is forced at the moment when the repainting process is resumed.

Data structures

`midInput`

`midInput` is a data table that describes the origin of incoming MIDI messages. The consists of information about the MIDI interface and the port identifier.

- `interface` - integer, an identifier of Electra's MIDI interface. (see [Globals](#) for details).
- `port` - integer, a port identifier (see [Globals](#) for details).

Example

```

midiInput = {
    interface = MIDI_IO, -- a name of the IO interface where the messages was received
}

```

```
port = PORT_1          -- a numeric port identifier
}
```

midiMessage

The `midiMessage` data table carries the information about a MIDI message broken down do individual attributes. Different types of MIDI messages are represented with slightly different format of the `midiMessage` data table. The fields `channel`, `type`, `data1`, `data2` are, however, common to all types of messages.

For example, a Control Change message can be access either as:

```
midiMessage = {
  channel = 1,
  type = CONTROL_CHANGE,
  data1 = 1,
  data2 = 127
}
```

or

```
midiMessage = {
  channel = 1,
  type = CONTROL_CHANGE,
  controllerNumber = 1,
  value = 127
}
```

- `channel` - integer, a numeric representation of the MIDI channel (1 .. 16).
- `type` - integer, an identifier of the MIDI message type (see [Globals](#) for details).
- `data1` - integer, the first data byte of MIDI message (0 .. 127).
- `data2` - integer, the second data byte of MIDI message (0 .. 127).
- MIDI message type specific attributes are listed below.

Attributes specific to MIDI message types

MIDI message type	Attributes
<code>NOTE_ON</code>	<code>noteNumber</code> <code>velocity</code>
<code>NOTE_OFF</code>	<code>noteNumber</code> <code>velocity</code>
<code>CONTROL_CHANGE</code>	<code>controllerNumber</code> <code>value</code>

MIDI message type	Attributes
POLY_PRESSURE	noteNumber pressure
CHANNEL_PRESSURE	pressure
PROGRAM_CHANGE	programNumber
PITCH_BEND	value
SONG_SELECT	songNumber
SONG_POSITION	songPosition

Globals

The global variables are used to identify common constants that can be used instead of numbers.

Hardware ports

Identifiers of the MIDI ports.

- PORT_1
- PORT_2
- PORT_CTRL

Interfaces

Types of MIDI interfaces.

- MIDI_IO
- USB_DEV
- USB_HOST

Change origins

Identifiers of the sources of the MIDI value change. Origin is passed as a parameter of the ParameterMap `onChange` callback.

- INTERNAL
- MIDI
- LUA

Parameter types

Types of Electra MIDI parameters. These types are higher abstraction of the standard MIDI message types.

- PT_VIRTUAL
- PT_CC7
- PT_CC14

- `PT_NRPN`
- `PT_RPN`
- `PT_NOTE`
- `PT_PROGRAM`
- `PT_SYSEX`
- `PT_START`
- `PT_STOP`
- `PT_TUNE`
- `PT_ATPOLY`
- `PT_ATCHANNEL`
- `PT_PITCHBEND`
- `PT_SPP`
- `PT_RELCC`
- `PT_NONE`

Control sets

Identifiers of the control sets. The control sets are groups of controls assigned to the pots.

- `CONTROL_SET_1`
- `CONTROL_SET_2`
- `CONTROL_SET_3`

Pots

Identifiers of the hardware pots. The pots are the rotary knobs to change the control values.

- `POT_1`
- `POT_2`
- `POT_3`
- `POT_4`
- `POT_5`
- `POT_6`
- `POT_7`
- `POT_8`
- `POT_9`
- `POT_10`
- `POT_11`
- `POT_12`

Colors

Identifiers of standard Electra colors.

- `WHITE`
- `RED`
- `ORANGE`
- `BLUE`
- `GREEN`
- `PURPLE`

Variants

- `VT_DEFAULT`
- `VT_HIGHLIGHTED`
- `VT_THIN`
- `VT_VALUEONLY`
- `VT_DIAL`
- `VT_CHECKBOX`

Bounding box

Identifiers of individual attributes of the bounding box (bounds).

- `X`
- `Y`
- `WIDTH`
- `HEIGHT`

MIDI message types

Identifiers of standard MIDI messages.

- `CONTROL_CHANGE`
- `NOTE_ON`
- `NOTE_OFF`
- `PROGRAM_CHANGE`
- `POLY_PRESSURE`
- `CHANNEL_PRESSURE`
- `PITCH_BEND`
- `CLOCK`
- `START`
- `STOP`

- CONTINUE
- ACTIVE_SENSING
- RESET
- SONG_SELECT
- SONG_POSITION
- TUNE_REQUEST
- TIME_CODE_QUARTER_FRAME
- SYSEX

Controller events

Flags indentifying individual types of events.

- NONE
- PAGES
- CONTROL_SETS
- USB_HOST_PORT
- POTS
- TOUCH
- BUTTONS
- WINDOWS

Touch events

Identifiers of touch events used in the Touch callbacks.

- DOWN
- MOVE
- UP
- CLICK
- DOUBLECLICK

Curve segments

Identifiers of the curve segments used in the graphics module.

- TOP_LEFT
- TOP_RIGHT
- BOTTOM_LEFT
- BOTTOM_RIGHT

Controller models

Identifiers of the Electra One hardware models.

- `MODEL_ANY`
- `MODEL_MK1`
- `MODEL_MK2`
- `MODEL_MINI_MK1`

Horizontal alignment

Text alignment modes

- `LEFT`
- `CENTER`
- `RIGHT`