# LEMUR

## USER GUIDE

## Table of Contents

# Chapter 1 - Welcome

Thanks for choosing Liine's Lemur, a groundbreaking iOS app for controlling music and media applications. Its unprecedented modularity will forever change the way you compose, perform, produce and, in more general terms, interact with your work environment.

## What Lemur is . . .

- Lemur is a controller: its purpose in life is to provide hands-on control over software running on a computer, or other connected devices such as samplers, synthesizers etc.
- Lemur uses two communication protocols: MIDI and Open Sound Control. Software or hardware must comply with at least one of those protocols to be controlled by Lemur.
- MIDI hardware can be connected directly to an iOS device running Lemur through a CoreMIDI compatible device such as iConnect MIDI or Alesis iO Dock, or via the Apple Camera Connection Kit and a compliant USB-Midi device.

… and what it is not.

Lemur does not produce any sound by itself. There is plenty of wonderful software and hardware out there perfectly doing that job. Lemur is here to make the use of that software and hardware more efficient, intuitive and hands-on. It is not restrained like conventional hardware.

## About this Manual

Considering the great variety of applications that Lemur can control, it would be nearly impossible to exhaustively detail here all the interfacing possibilities. Therefore, instead of providing step-by-step tutorials for every single piece of software you might use, this manual focuses on the general features and concepts one needs to master in order to work with Lemur.

Besides this manual, there exists a host of external material, including additional documentation, Projects and videos meant to make your introduction to Lemur work as smooth as possible. Documentation improvements and revisions are ongoing. Be sure to check out

Liine website: http://liine.net

User Library: http://liine.net/en/community/user-library/

Liine Forum: http://liine.net/forum

# Chapter 2 - Concepts

## 2.1 Lemur Editor

Lemur is provided with a dual-platform (Mac OS X and Windows) interface design utility, the Lemur Editor. Download the latest version of Lemur Editor at http://liine.net/en/support/

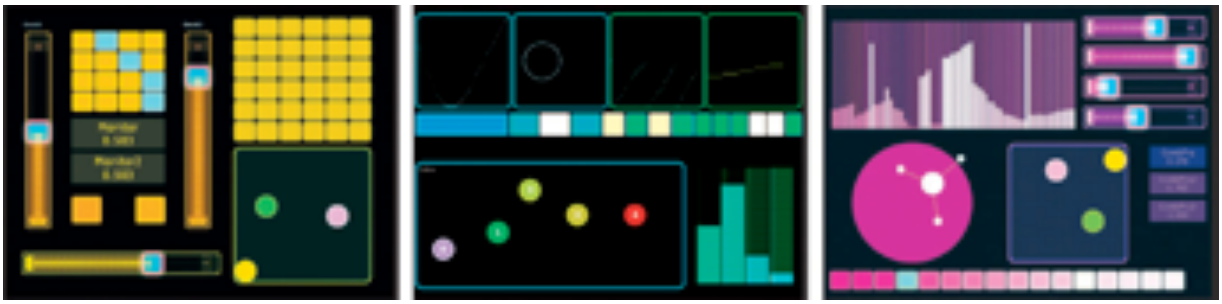
One of Lemur's main characteristics is that you can populate the screen with as many different virtual objects as you need. This way, you can design the interface layouts that will perfectly fit both your application and your hands.

## 2.2 Projects

The Lemur Editor lets you open or create complete control Projects for Lemur. All Project files the Lemur Editor produces can be stored on your computer's hard drive as .jzml files. Projects can also be stored in your iOS device's internal memory after being transferred there by the Editor or via iTunes app file management. You can access the Lemur Project browser at the top of the Lemur Settings menu on your iPad or iPhone.

## 2.3 Interfaces

A Project contains your Interface (also called Pages when there are several), which can each contain multiple Objects, which can also be organized in Containers if desired. The number of Interfaces and Objects is limited only by memory, which gives you plenty of room for your creations.



The figure above shows a typical Lemur project, comprised of three different interfaces or pages. Once a Project is loaded on your iOS device, the different interfaces can be browsed with the grey tabs at the top of the screen. Tap a tab to jump to that interface.

## 2.4 Modules

A Module is a set of reusable interface elements saved as JZLIB files. A module can feature objects, containers and interfaces. Additionally, you can import pre-built parts of a project as Modules. Modules are available from Liine's website. You can also create your own library of reusable items. Indeed, any part of a Lemur Project can be exported as a Module for future use in other Projects. Thanks to this feature you never have to develop the same thing twice. Just save your building blocks as Modules to produce a growing library of re-useable, object-oriented tools.

The User Library on Liine's website, http://liine.net/en/community/user-library/, is a great resource for stacking up both on Projects and Modules.

## 2.5 Objects

Objects are the main message generators of Lemur: they provide the values you control with your fingers. There is a multitude of different Objects available that all have their special capabilities:


**Breakpoint**


**Container**


**CustomButton**


**Fader**


**MultiBall**


**Knob**


**SurfaceLCD**


**Leds**


**Text**


**LemurMenu**


**Monitor**


**MultiSlider**


**Pads**


**Range**


**RingArea**


**SignalScope**


**Switches**

Objects can be arranged on screen at variable sizes and the functionality that you want can be added to them via a simple yet comprehensive menu-driven system of attributes and variables., or through multi-line scripting which allows more complex behaviors to be programmed.

Auto-mapping of new Objects to Continuos Controllers or MIDI Notes allows for quick and easy layout of almost instantly functional interfaces.

Objects may also be created and modified in real-time via OSC messaging allowing suitable OSC programs to manipulate and modify your interface.

Objects are capable of bi-directional communication and can control as well as be controlled.

## 2.6 Variables

A Variable is simply a storage location for some information and an associated name for referencing that value. The stored value may be constant or it may change over time inresponse to user input, programming or external control. The Variable name is used to get or set the value.

Most Objects have their own set of Variables that reflect the Objects' states, and change when you touch them with your fingers. A Fader's x Variable reflects the position of its cap for example, or that of a Switch its on/off state. All these Variables can also be modified by external software, by Scripts or by other Objects dwelling in the Project.

In addition to these Built-in variables, the Lemur Editor lets you create your own User-defined Variables, which you can equate to your own mathematical or logical expressions.

A Variable can be defined **locally**, living in a specific Object and accessed from the outside through its address. A Variable can also be defined **globally** for a Project, enabling multiple Objects to use its values directly. Variables can also be declared within the scope of a running Script.

Variables can be multi-dimensional too. A Knob, for instance, has only one Variable defining its output with a single number, while A MultiBall Object has three Variables: the X and Y positions of the balls, and a third Z Variable representing their Brightness. The MultiBall's Variables are Vectors with as many components as there are balls defined for the Object.

The built-in Time Variable deserves a special mention. It presents the time in milliseconds since Lemur was switched on. Combined with Lemur's functions and mathematical operators, it can produce a great variety of time-varying number sequences.

## 2.7 Scripts

Lemur has a C-style scripting language that allows you to program additional functionality into your interface. Scripts can be used to instruct Lemur to perform specific tasks at specific times, or in response to user interaction, program state or external input.

From simple actions such as changing a MultiSlider colour when a Switch is pressed, to complex actions such as commanding a Multiball's balls to follow set trajectories. Lemur has a rich set of built-in functions, including the famous Physics engine, and of course, you are able to define your own expressions and functions to extend its capabilities. The possibilities are endless.

## 2.8 OSC

Lemur is compliant with the Open Sound Control protocol, a network-based standard with significant advantages over MIDI: low latency, higher data capacity, 32-bit numerical precision, flexibility and easy set-up. Supported by a growing number of high profile applications (Max/MSP, Logic 9, Monome, Reaktor, Circle, Modul8, Resolume Avenue…), OSC has opened a new era in the field of real-time control and human-machine interface.

OSC is an open-ended, dynamic URL-styled symbolic naming scheme capable of using symbolic and high-resolution numeric data. Pattern-matching can specify multiple recipients of messages. OSC gives you great flexibility in the kinds of data and hardware you can communicate with.

## 2.9 MIDI

Naturally, Lemur also complies with the MIDI protocol and can transmit and receive all MIDI messages. Any of these messages can directly be assigned to any Built-in or User-defined Variables, sent through scripts, or activate a script execution upon reception by Lemur.

MIDI can be transmitted wirelessly to the Lemur Daemon on any accessible computer, via Network MIDI sessions. Wired MIDI may be sent via iPad compatible MIDI docks or via Class compliant MIDI interfaces attaches with an Apple Camera Connection Kit.

MIDI may also be sent to compliant Core-MIDI apps running on your iPad or iPhone.

## 2.10 Lemur Daemon

Lemur Daemon is a little helper application running in the background on your computer. It handles the MIDI data flow between Lemur and the MIDI ports installed on the computer.

The Lemur Daemon automatically scans all available MIDI ports on the computer, be they physical or virtual, and makes them available to Lemur(s) connected to the network.

On Mac, the Lemur Daemon also creates its own set of virtual ports, the Daemon Inputs and Outputs, which are accessible in your applications' MIDI settings.

On PC, third party virtual ports such as those offered by loopMIDI, loopbe or MIDI Yoke are needed in order to link to the Lemur Daemon and use MIDI with Lemur.

## 2.11 Targets

A Lemur Daemon Target consists of a pair of MIDI Inputs and Outputs.

An OSC Target is characterised by its IP address (or hostname) and its port number.

No matter how complex your setup may be, you can control everything from a single Interface on Lemur by individually assigning the different Variables to different Targets, or by using Scripts containing Output functions.

The networkability of Lemur endows it with the unique capacity to simultaneously control several applications running on the same computer, or even to control several pieces of software launched on different computers. Each application or device controlled by Lemur being represented by a "Target". Lemur features eight Lemur Daemon Targets and eight OSC Targets.

**Note**: You can connect to the MIDI ports of any computer on the network provided they have a Lemur Daemon running. Use the Lemur Daemon Targets settings page on Lemur to get a list of running Lemur Daemon's on the network and to connect to specific MIDI ports on the respective machines. You can also communicate with any computer on your network using OSC.

# Chapter 3 – Connection & Setup

Connection and setup of the Lemur app is simple and easy. We've created easy to follow videos on connecting Lemur over Wi-Fi and using OSC or MIDI, as well as connecting over USB with CoreMIDI. These videos are available on the Liine website at http://liine.net/en/support/

As a reference, basic instructions and recommendations follow.

## 3.1 Wi-Fi MIDI or OSC

- Make sure that Bluetooth is off on both your iOS device and the laptop.
- Ad-hoc Wi-Fi connections are recommended over connections through a router. Latency and jitter will be much lower via an Ad-hoc Network.
- OSC communication is **only** possible over Wi-Fi.
- Lemur Daemon must be running to use MIDI over Wi-Fi.

## 3.2 USB MIDI (CoreMIDI)

- USB MIDI is the lowest latency way to connect Lemur.
- Liine recommends using iConnect MIDI, although many other devices will also work. Be sure to familiarize with the specifications of your MIDI interface. The Alesis iO Dock, for example, blocks all SYSEX messages.
- The Apple Cammera Connection Kit and a compliant MIDI interface may be used.

## 3.3 Connection Problems?

- Lack of connectivity could be due to a number of factors. Carefully following this list of actions should remedy the problem:
- Check that your iPad is connected to the same network as the computer. Check that you have the latest version of Lemur, Lemur Editor and Lemur Daemon installed and running.
- Check that Bluetooth is turned off on the iPad. Close and restart the Lemur Daemon app. To quit Lemur on the iPad/iPhone, kill the app from the multitasking bar. Restart Lemur.
- Check that you don't have any Firewalls blocking OSC communications.
- Make sure no other applications are using ports 8000-8003 on your iPad or iPhone. Also make sure those ports are not busy on your computer with, for example, a Network MIDI Session on Mac, rtpmidi session on Windows or an application such as Max or Osculator.
- If the connection still fails, please contact support@liine.net

# Chapter 4 - Software Installation

## 4.1 The Lemur Daemon

For Lemur to become fully functional, you will need to run the latest Lemur Installer package downloaded from Liine's website at http://liine.net/en/support/ to install the necessary Liine software on your computer.

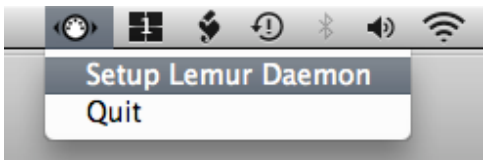
Lemur Installer contains the **Lemur Editor**, for creating and transferring your Projects to Lemur, and the **Lemur Daemon**, a small application that keeps running in the background, and is required to handle MIDI communication and transmit keyboard and mouse commands from Lemur.

## 4.2 The Lemur Daemon on Mac OS X

You can find the Lemur Daemon on the Menu Bar of your MacOS X desktop;

It doesn't appear in your Dock while it's running,.

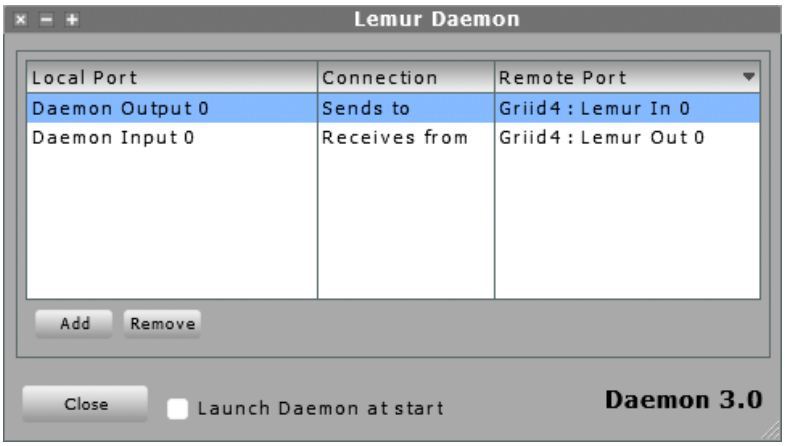
A click on the Lemur Daemon opens a menu with two entries:




- **Setup Lemur Daemon** opens the MIDI connection browser of the Daemon. Please have a look at Chapter 8 "Targets Setup" for details.
- **Quit** shuts the Lemur Daemon down.

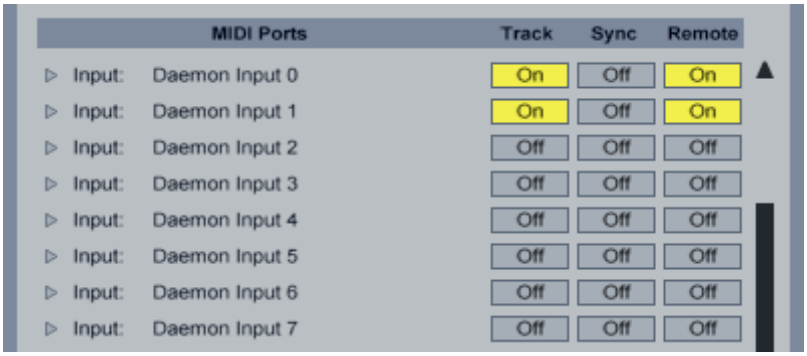
Please note that the Lemur Daemon must be running if you want to use MIDI with Lemur.

Use the small X in the upper left corner of the MIDI port browser for closing it without shutting the Daemon down




If you want the Daemon to be automatically started, enable the **Launch Daemon at startup** checkbox.

On Mac, the Lemur Daemon automatically creates 8 virtual MDI Inputs and 8 virtual MIDI Outputs for you to link with Lemurs. They are conveniently named Daemon Input and Daemon Output. Once connected to the Ins and Outs of Lemur, you can use to communicate with any MIDI application on your Mac.




## 4.3 Windows Install

Download the latest software from the web, double-click the **exe** file and follow the instructions.

By default the Installer will create a folder named Liine in your Program Files (C:\Program Files\Liine). Click on **Browse** to select a different folder.

## 4.4 The Lemur Daemon on Windows

By default the Lemur Daemon will be loaded automatically at boot time of the computer and appear in the System Tray on your desktop.

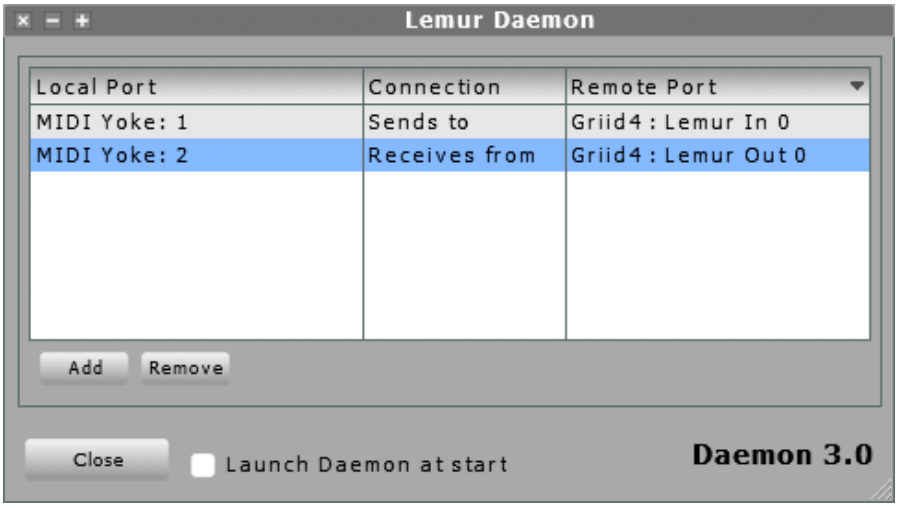

A click on the Lemur Daemon icon opens a menu with two entries:

- **Setup Lemur Daemon** opens the MIDI port browser of the Daemon. Please have a look at chapter 8.2 for details.
- **Quit** shuts the Lemur Daemon down.

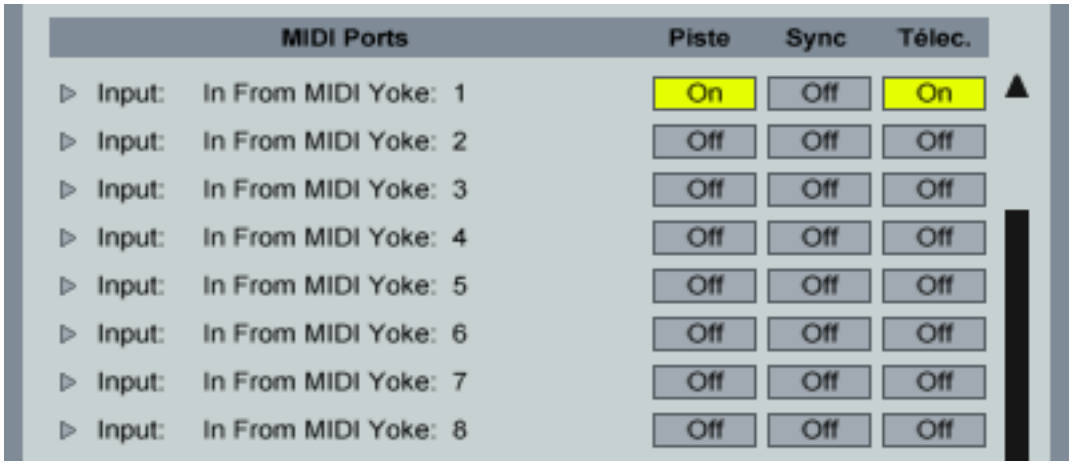
Please note that the Lemur Daemon must be running if you want to use MIDI with Lemur. Use the small **X** in the upper left corner of the MIDI connection browser for closing it without shutting the Daemon down.f you don't want the Daemon to be automatically started, uncheck the **Launch Daemon at Startup** flag and the next time you boot your computer the Daemon won't be started.

On Windows, you will need third-party virtual MIDI ports, such as **loopMIDI** or **loopbe**, in order to link your applications to the Lemur Daemon. Both drivers are freeware and easily downloadable from the Internet.

In the software application you want to control with Lemur, choose the virtual MIDI port to which Lemur is connected. As you can also have bi-directional communication via MIDI - i.e. the MIDI-enabled software can also control Lemur Objects - use a second, distinct virtual port to make the reverse connection to Lemur. Using a distinct port prevents MIDI feedback problems.




Once you've connected them to the Ins and Outs of Lemurs, you can use them in any MIDI application on your PC.

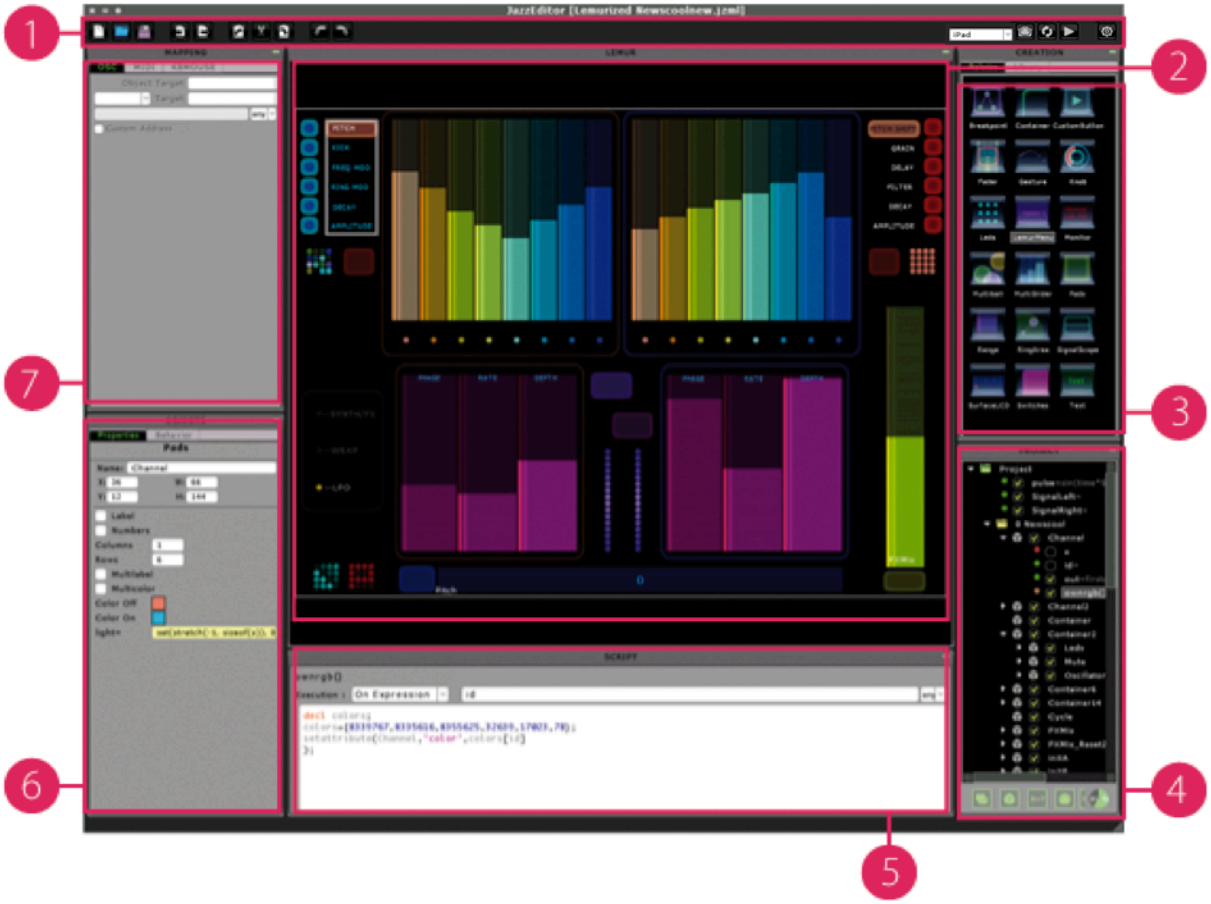


# Chapter 5 - Introducing the Lemur Editor

Double-click the Lemur Editor application icon to start editing. All editing and building of Interfaces is done via the Lemur Editor software. Here you create, modify and save your Projects, or simply open them in order to transfer them to Lemur.

## 5.1 Overview

The Lemur Editor's Workspace consists of seven main areas:




**1. Header:** provides icons for basic functions.

**2. Lemur panel:** this is where the virtual objects are placed to build Interfaces.

**3. Creation panel:** contains the Palette and Library tabs.

**4. Project panel:** provides a hierarchical view of your current project.

**5. Script panel:** for defining an expression or a multi-line script.

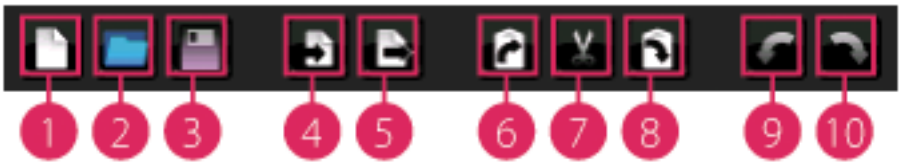
**6. Objects panel:** for defining the selected object's Properties and Behavior.

**7. Mapping panel:** for assigning OSC or MIDI messages to your object's variables.

It is useful to know that the layout of the Lemur Editor Workspace can be customized to suit your preferences. Just grab a panel by its title bar, and drag it to another spot on the Workspace. Everything can be rearranged.

## 5.2 Header

The Header section provides a series of icons that allow quick and easy access to some of the Lemur Editor's basic functions.




**1. New Project:** Creates an empty Project.

**2. Open Project:** Presents you with a dialog for opening Projects from your file system.

**3. Save Project:** Opens a dialog for saving your current Project to the file system. Choose an appropriate folder to store your Projects.

**4. Import Module:** Opens a dialog for importing a Module into the currently opened Interface at the level of hierarchy you choose.

**5. Export Selection:** Exports the selected group of objects as a Module and save it to the file system.

**6. Copy:** Copies selection to clipboard.

**7. Cut:** Cuts selection to clipboard.

**8. Paste:** Pastes from clipboard.

**9. Undo:** Rolls back an unlimited number of Lemur Editor commands.

**10. Redo**: Redo last command.


**11. Resolution Menu:** Opens a menu of the possible Project resolutions. Projects open at the resolution they were created or saved at. Changing the Resolution of an open Project will present a '**Change Resolution'** dialog box allowing you to choose whether to automatically stretch all objects to the new Project size. Some manual editing may still be necessary and Object minmum sizes are enforced

**12. MIDI Mapping:** Opens a list of the current Projects MIDI mappings.

**13. Synchronization:** When the Lemur Editor is connected to a Lemur, any changes to Objects' states on Lemur will also be reflected in the Lemur Editor. This can be very handy for saving a Project including the state of all objects.

**14. Connection:** Opens a dialog listing the accessible Lemurs on the network for connecting the Lemur Editor to a Lemur and transferring the current project.

**15. Settings:** Opens the Lemur Editor settings menu.

## 5.3 Lemur Panel

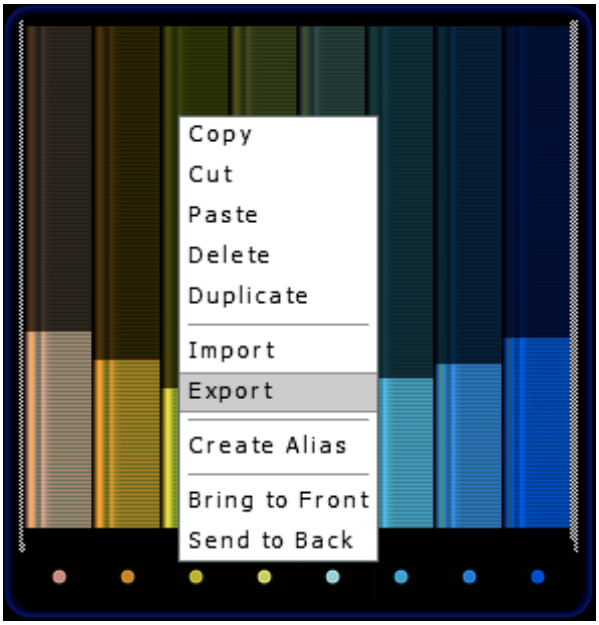
Lemur Panel is the canvas on which your Objects are laid down and assembled to construct Interfaces. In terms of size, it replicates that of Lemur's screen (1024x724 for iPad, 480x276 for iPhone and 800x600 for Legacy). Lemur panel always displays the Interface currently selected in the Project panel (see corresponding section below).


Objects are created either by dragging and dropping the desired icons from the **Palette**, or by clicking on the **Create Object** icon underneath the **Project panel**. Fetch the Objects with your mouse and displace them to the ideal spot. You will notice that when your mouse hovers over an Object, its corners highlight. Simply grab a corner and move the mouse to resize it.

Multiple Objects can be selected by holding the Shift key and clicking on them one after the other. You can also drag a square around them to get the same effect. By clicking an Object again while still holding the Shift key you remove it from the selection.

There are two different modes available concerning interaction with Objects in the Lemur Editor. In default **Edit** mode, you can't change the state of the Objects laying on Lemur panel. In **Run** mode, you can use your mouse to change the state of the Objects, as if you were using your fingers on Lemur. Simply press "**E**" on your computer keyboard and hold it to switch to Run mode. As long as you hold the key you can use your virtual Interface. This comes very handy when debugging your work without having Lemur connected.

The Editing Area also features a **contextual menu** which is displayed by right-clicking (PC)/ ctrl-clicking (Mac) on an Object. It offers an additional way of executing commands such as **Copy** or **Export**:




Objects can overlap in the Interface. The rule of thumb is that the Object created last will cover older Objects. The commands **Bring to Front** or **Send to Back** change the order of Objects in a "pile". Some Objects (like the Monitor) can have a transparent background, leading to various possibilities for labels and captions. When the Transparency flag is activated, only parts of the Object will remain opaque and any Objects lying underneath can be seen.

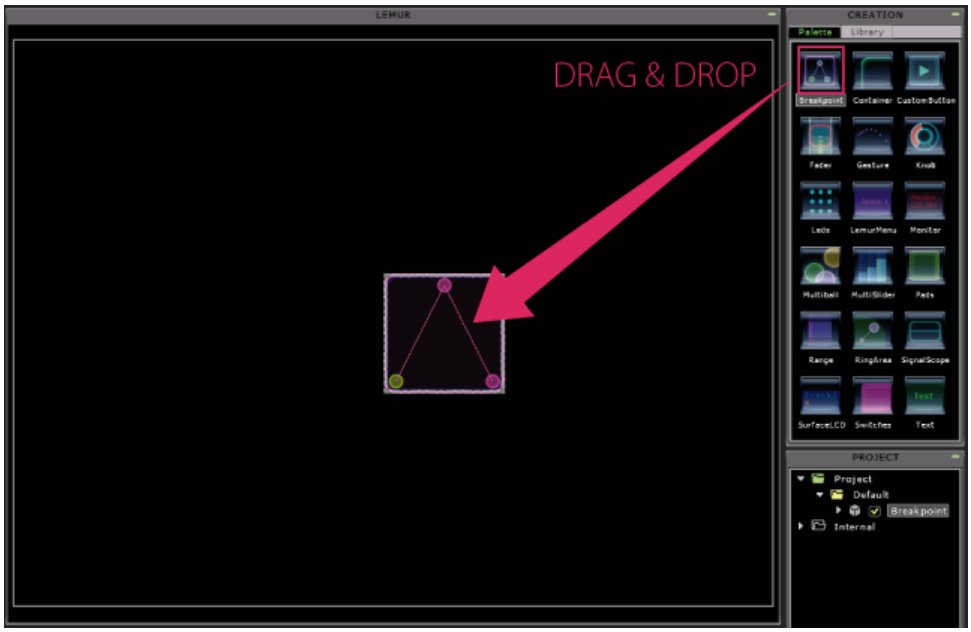
Note that a lot of the commands and functions of the Lemur Editor can also be controlled via keyboard **shortcuts**. The shortcuts are displayed in the Tool tips that pop up when your mouse pointer hovers above the respective commands. Please refer to **Appendix I** for a complete list of shortcuts.

## 5.4 Creation Panel

The Creation panel consists of 2 tabs: the **Palette**, for creating Objects by dragging them from the Objects Icons lists and dropping them on Lemur panel, and the **Library**, for dragging modules to and from Lemur panel.

### 5.4.1 Palette

The Palette tab displays the built-in Lemur Objects. To create an Object, select an icon and drag it anywhere on Lemur panel. Double-clicking creates the Object in the top left corner of the screen.

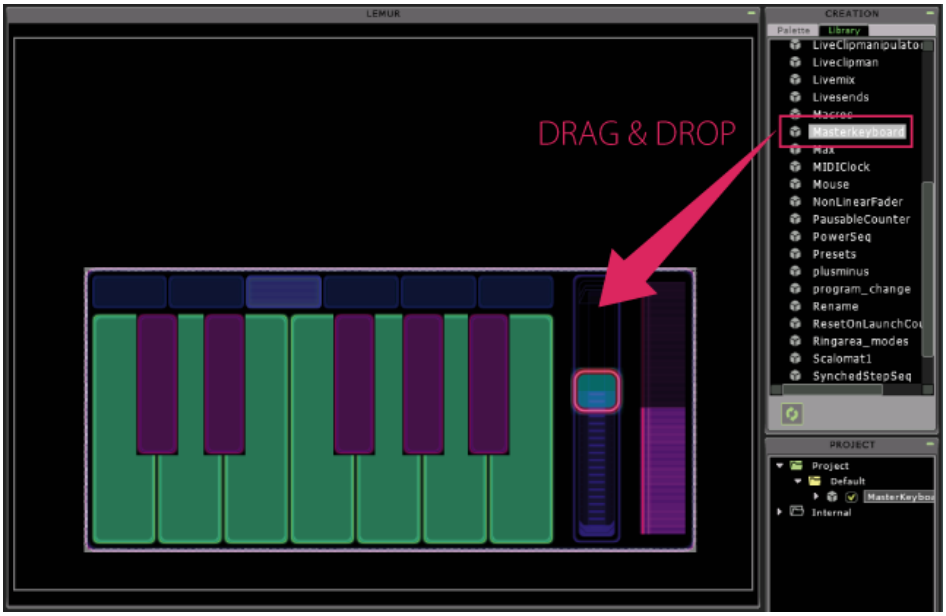



As we'll see later, you can also create an Object by clicking on the **Create Object** icon underneath the **Project panel, or pressing Command/Control+Shift+O**. The choice is yours.

### 5.4.2 Library

The Library tab displays the user-installed Lemur Modules. The **Library** works on the same principle, however before being able to use the Library, a file path for your module folder must be specified in the **Lemur Editor Settings** window (click on the top-right corner's icon).

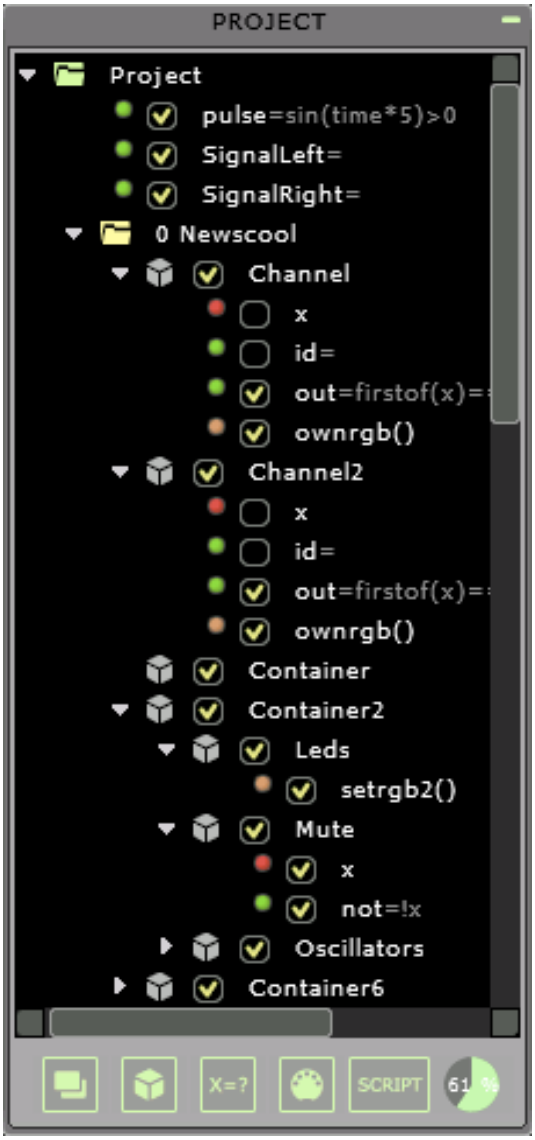
When you're done, the modules present in your folder should be listed and available to be dragged and dropped to the Lemur panel.




This also works the other way around. Any Object selection on Lemur panel can be dragged to the Library panel by holding the **alt** key and be directly saved as a module.

## 5.5 Project panel

The Project panel displays all elements of the current Project in a **hierarchical tree** structure.




Click on the small **disclosure triangles** in front of the symbols or double-click on a symbol to expand or collapse the next deeper level of the tree.

The various entries in the Project Browser are color-coded as follows:

- **Red** dots stand for the Object's main parameters (x for a Fader; x, y, z for a MultiBall, etc.)
- **Orange** dots depict Custom MIDI messages or Scripts created by the User.
- **Green** dots indicate User-defined Variables.
- **Blue** dots indicate User-defined Functions.
- **Grey** dots indicate internal Functions

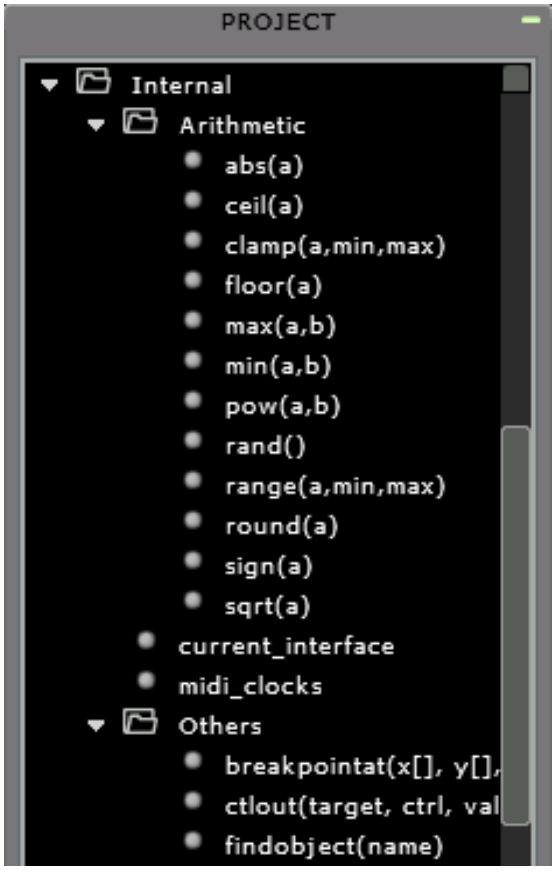
The **green folder** at the top of the tree structure represents the current parent Project **Yellow folders** represent the different Interfaces (pages) contained in the Project.

These yellow Interface folders contain the Objects, depicted by the small **grey cubes**, together with their built-in Variables, and any additional User defined Variables, Functions, Custom Midi messages or Scripts, all represented by **colored dots**. Note that in some instances it may be necessary to create User defined Variables, Functions, Custom Midi messages or Scripts directly at the root of the Project folder so that they can be used globally, but more will be said on this later.

The small **checkboxes** play an important role in Lemur Projects. If a box is checked, it means that the MIDI or OSC messages associated with the entry, being a single variable or a whole Object, are transmitted to the Targets. In the case of a Script, the checkbox can be used to activate or deactivate the execution of that Script.

There might be at least two reasons for not sending the output value of a particular object. One would be if you were using an Object merely to display information coming from your computer. The MultiBall object could represent the state of something in an application for instance.

The other reason could be that you transmit the data from the Object to your computer via a User-defined Variable containing a mathematical expression. In this case the built-in Variable (e.g. the x) is only referenced by the Expression in another Variable. This is often used to scale values into a more appropriate range. We'll learn more about this technique later.




Use the five **Create** command iconss to create new Interfaces, Objects, Expressions, Custom MIDI mappings, and Scripts.

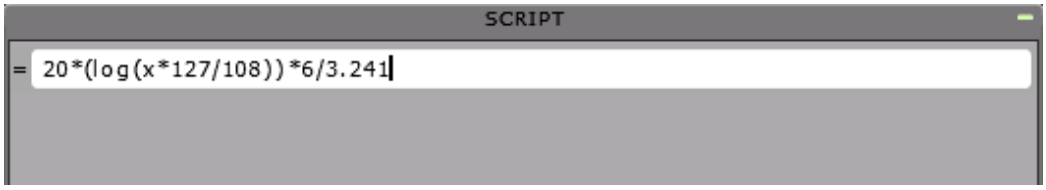

To the right, the **Memory Display** shows you the percentage of Lemur's memory that your Project will occupy when it is instantiated, it is not the size of the Project itself. This is for reference only.

## 5.6 Script Panel

The Script panel is multi-purpose. It allows for the single-line definition of **User-defined Variables** or **Functions** created with the **Create Expression** icon underneath the Project Browser. For instance, imagine we need to derive a logarithmic value from a Fader's built-in x Variable. Click on the **Create Expression** icon and name it appropriately:

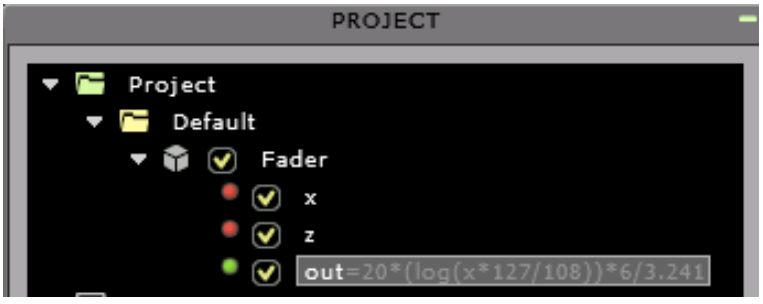


Then type in the mathematical expression to define your own Variable with respect to the original x variable:

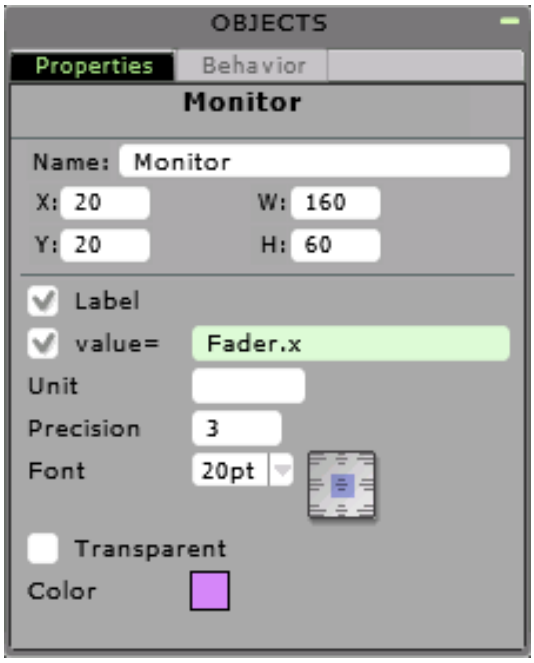


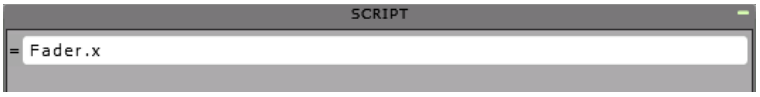

Notice here that we've used a **log** function. This is one of the many handy built-in functions, available to make your life easier when building complex Interfaces. Please refer to the Parser reference on Chapter 13 for insights into built-in functions and operators.

This User-defined Variable we just created is then displayed in the Project panel and ready to be used.

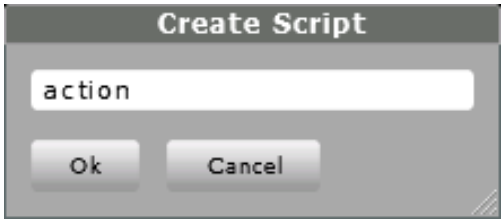


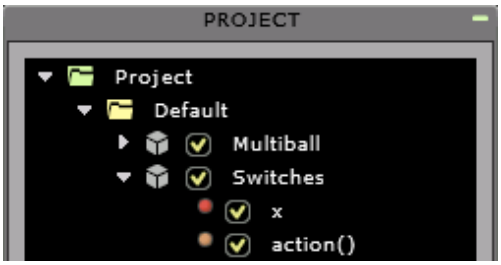

The Script panel can also be used to enter a single-line expression for some User-defined **Object parameters**, typically the Light or Value parameters:

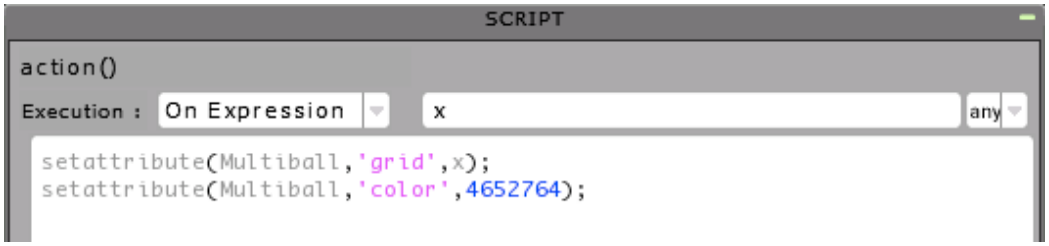



Finally, the **Script** panel also allows for the **multi-line** coding of a **Script** created with the **Create Script** icon. Multi-line scripts can be used, in conjunction with a set of powerful **built-in functions**, to instruct Lemur to perform specific tasks at specific times. A typical application of this powerful feature could be the real-time manipulation of an object' **attributes**, such as its dimensions, color or physics. A basic example would be to instruct a Switch to activate a Multiball's grid, or a Text object to change its content upon reception of an OSC or MIDI message.


Please refer to Chapter 11 and 12 for more information about **multi-line scripts**.

## 5.7 Objects Panel

The Objects panel contains two tabs: the **Properties** and **Behavior** tabs. The Properties tab provides access to the basic properties of Lemur's Objects, while the Behavior tab concerns their physical responses. Details of all the Objects's parameters can be found in the Object Reference section of this manual.

### 5.7.1 Properties Tab

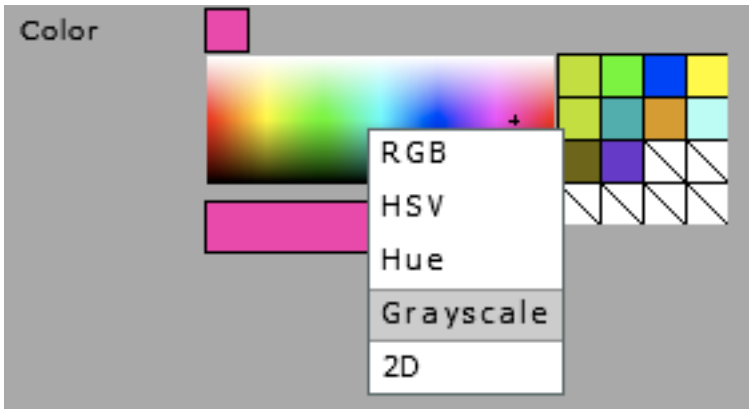

In essence, the Properties tab deals with the appearance of the Objects such as their color for instance. This is also where you name an Object (up to 64 characters) or type in its size and position with greater precision if you need to. Additional parameters may vary depending on the type of Object selected.

By default, the **Color** parameter of your Objects is set through a 2D color picker, which appears when you click on the square color sample:

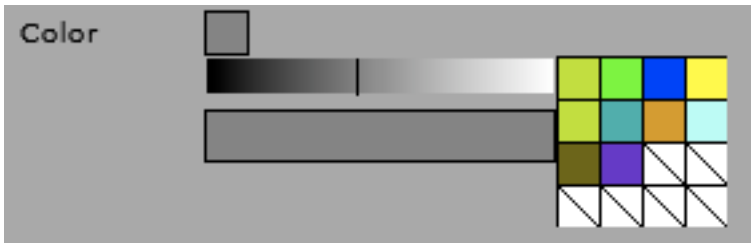

The grid to the right of the color picker is for storing your favorite colors. Just drag the large color sample bar underneath the color picker to a slot in the grid, and that color will be available for direct selection in the grid next time you use the color picker.

Note that the small square color sample next to **Color** can be directly dragged to other Objects present on Lemur panel to colour them with the exact same colour.

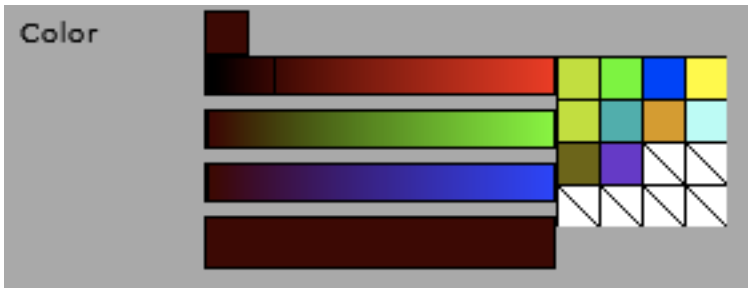
If needed, the colour selection mode system can be changed by right-clicking (PC) or ctrl-clicking (Mac) on the colour picker:



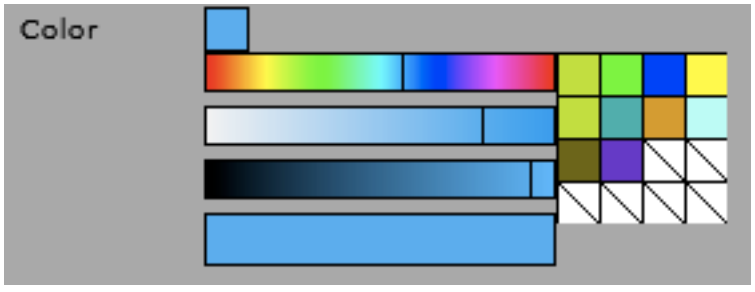

Choosing **Grayscale** for instance will leave you with a black and white selection tool.




Choosing **RGB** leaves you with a classical RGB additive color selection tool, in which the red, green and blue sliders are added together to produce the desired color.

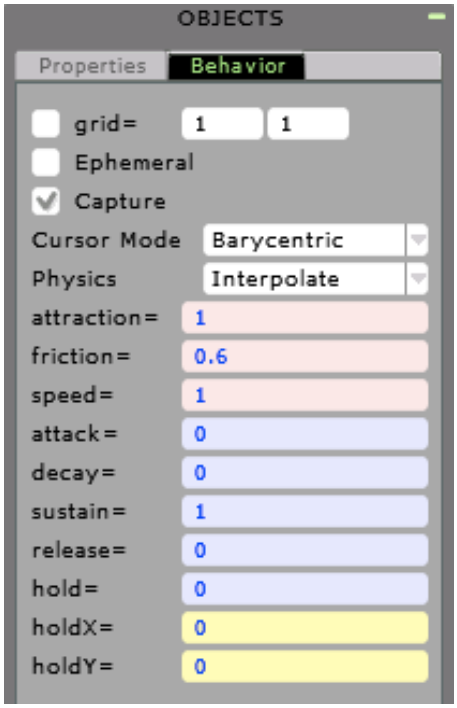



**HSV** stands for hue, saturation, value, where the hue slider lets you pick up the "pure" color, saturation the perceived "intensity" and value the "lightness".




Choosing **Hue** leaves you with the traditional, simple color swatch tool:

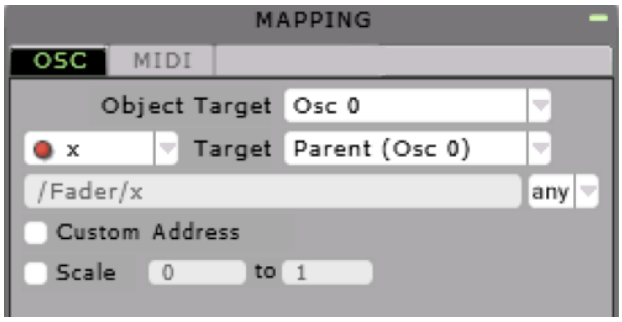
### 5.7.2 Behavior Tab




The Behavior tab deals with the physical characteristics of the Objects. Here you find parameters like friction and tension and the different physics modes of the Object.

## 5.8 Mapping Panel

### 5.8.1 OSC Tab

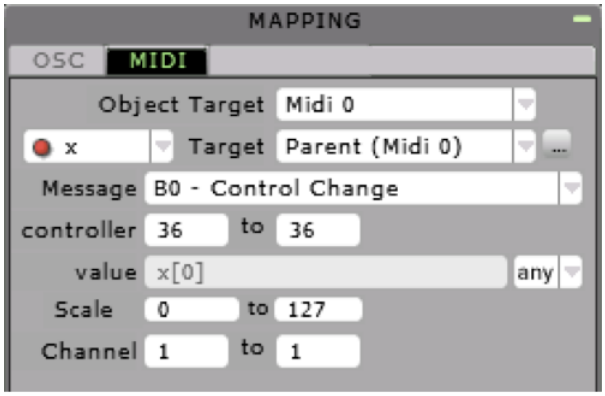



On the OSC Panel you can define the routing of the different Variables to OSC **Targets**.

You have a menu for the Variables and one for the eight possible OSC Targets.

This panel also serves for controlling Lemur objects via OSC.

### 5.8.2 MIDI Tab



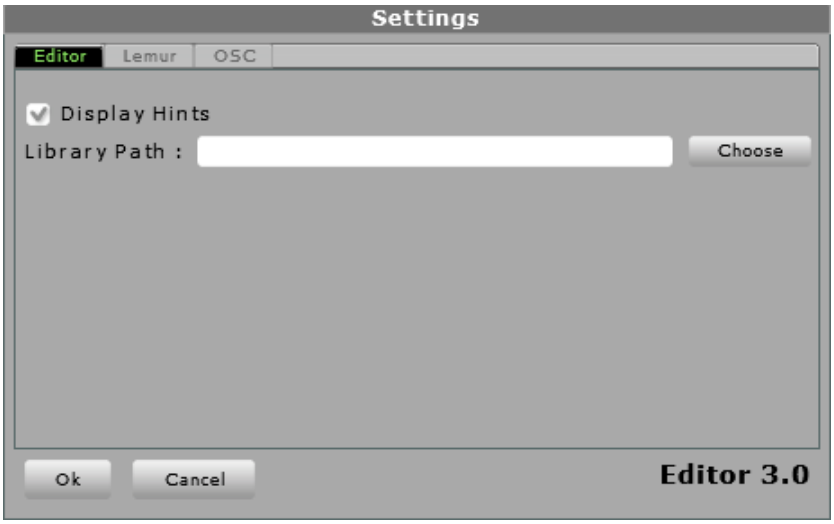

The MIDI panel lets you assign **MIDI messages** to your Variables and route them to the MIDI Targets.

This panel also serves for controlling Lemur Objects via MIDI.

## 5.9 Settings Menu

### 5.9.1 Editor Tab

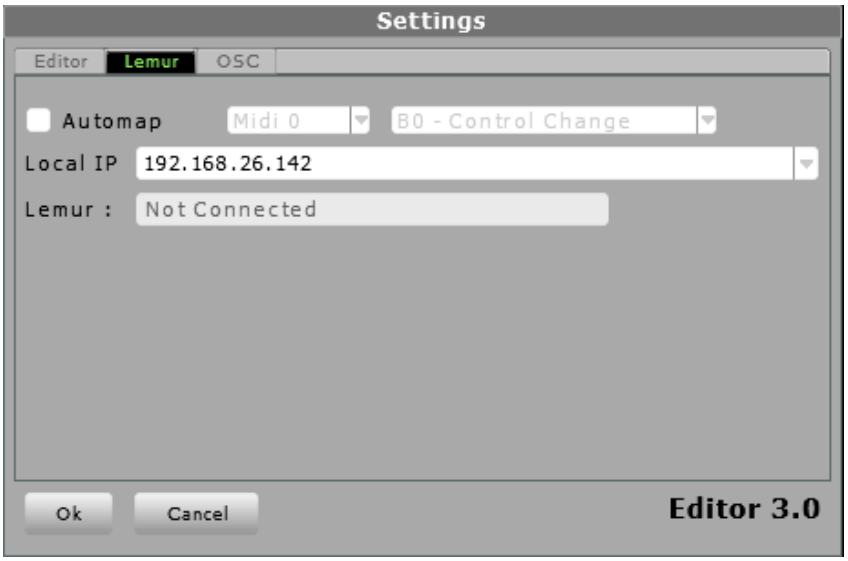
The first panel of the Editor tab deals with two basic settings. The first is the **Display Hints** checkbox: when active, the description of the interface item alongside with the eventual keyboard shortcut is displayed when you hover with your mouse pointer over the various interface elements.




The second is the path selection for the Creation panel's **Library**. Here you tell the Lemur Editor where to look for your Modules collection, so that afterwards you may simply drag and drop them to Lemur panel.

### 5.9.2 Lemur Tab

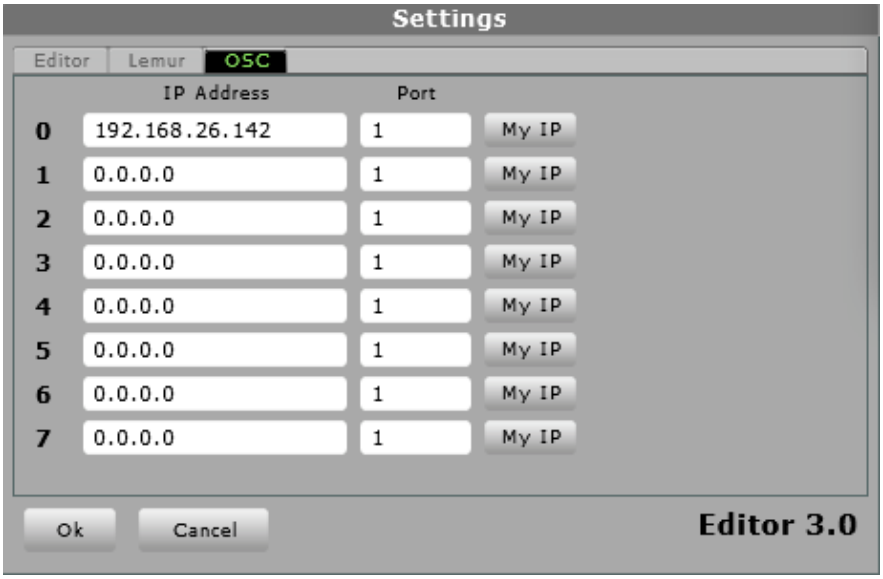
The second panel of the Settings window, called Lemur, deals with two other settings. The first is the **Automap** checkbox: when active, each new Object will be automatically mapped to the chosen MIDI Target and Message:




The **Local IP** setting lets you choose which of your computer's local IP addresses the Lemur Editor will use for connecting to Lemur. This menu may contain multiple listings for the different networks you have connected to. The IP of the currently connected Lemur is also displayed.

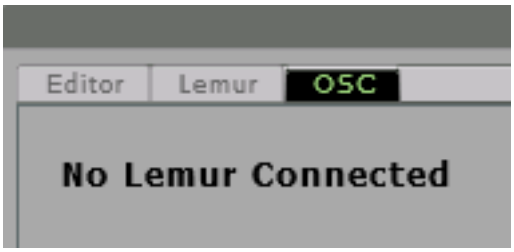
### 5.9.3 OSC Tab

The third panel of the Settings window lets you set your **OSC Targets** provided the Lemur Editor is connected to Lemur. This panel mirrors the OSC Targets settings page on Lemur, and either can be used to configure your OSC Targets.




Just fill in the IP addresses of the Targets you want to communicate with. If it's a piece of software running on the same computer as the Lemur Editor, click the **My IP** button, this will automatically fill in the IP of the currently used interface.

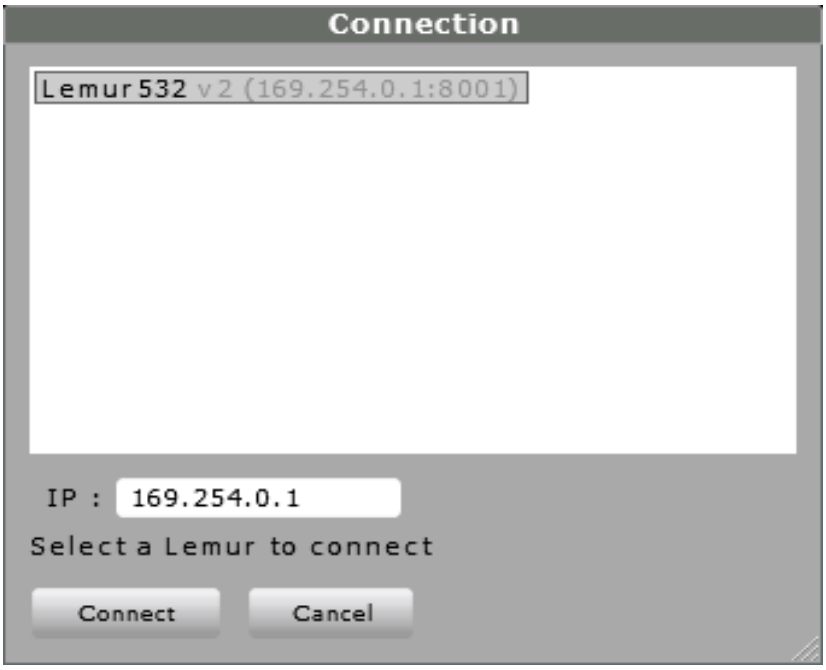
If there is no connection, the Lemur Editor states this and does not show any OSC settings:



# Chapter 6 - First Steps

## 6.1 Connecting Lemur

The first step naturally consists in opening the connection between Lemur Editor and Lemur. To do so, click on the **Connect** button on the toolbar, which should result in opening the following window:




Provided your network is set up correctly (see Chapter 3 and Appendix I), the connected Lemurs should be listed with their IP addresses and port numbers. You can select a Lemur on the list and click Connect to establish a connection. A double-click on the list entry does the same trick.

If you just connected Lemur or had it switched on shortly before, wait a few seconds for the list to update.

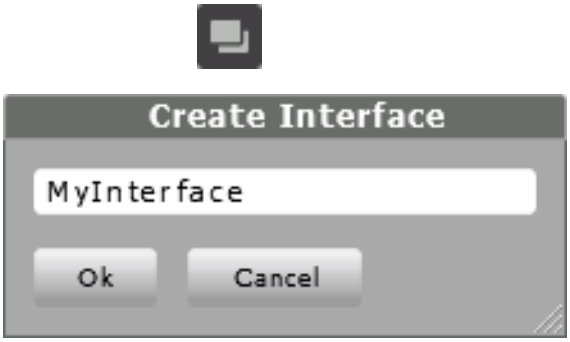
Note : Be careful! If you have a Project residing on Lemur and connect to it with a different Project open in the Lemur Editor, the one on Lemur will be overwritten. A connection always automatically transfers the currently opened Project to Lemur.

The **IP field** below the list allow manual entry and remembers the last setting of Lemur connection. It can also be handy for connecting through a proxy.

The current connection status can be seen from the displayed Connection icon. A **Play** button means there is no Lemur currently connected. If you connect to a Lemur the **Stop** button appears and a click leads to a disconnection of Lemur.

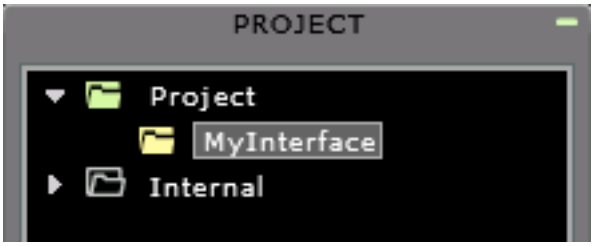
## 6.2 Creating an Interface

Now that Lemur and Lemur Editor are connected together, let's create our very first interface. To do so, just click on the **New Interface** button, located below the **Project panel**.

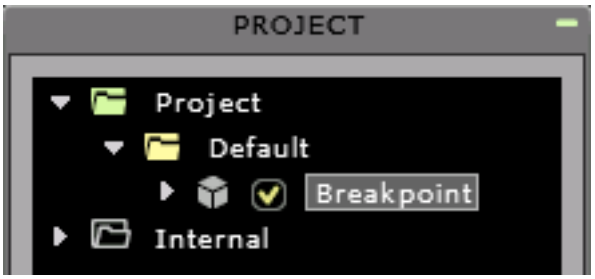



This opens a small dialog asking for the name of the new Interface. You can type in any name you like. Click **OK** and your new Interface is created.

Both Lemur panel and Lemur screen suddenly turn black. There is nothing to worry about: we just created a blank interface. Notice that the new interface appears as a **yellow** folder in the Project panel, as shown below.




Alternatively, beginning a new project simply by creating / dropping an Object on Lemur panel will automatically create a Interface named "Default".

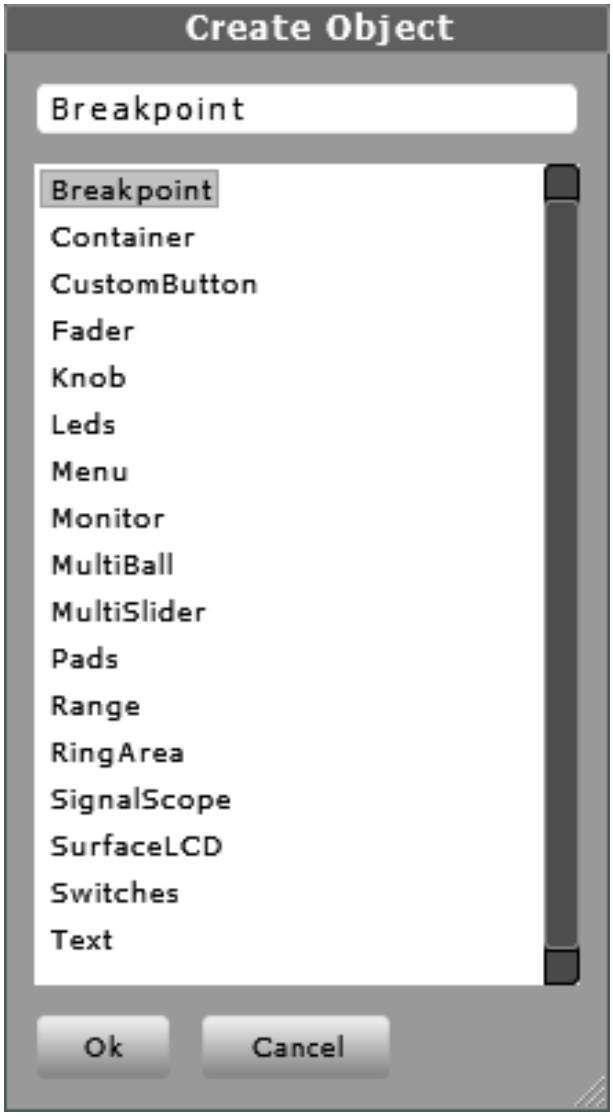



Your project may contain more than one interface; simply create and name them as required. Interfaces are arranged alphabetically within the Project. When a Project contains multiple interfaces the separate pages are indicated by grey tabs at the top of the iPad screen. Individual interfaces are labelled according to their name in the Project.

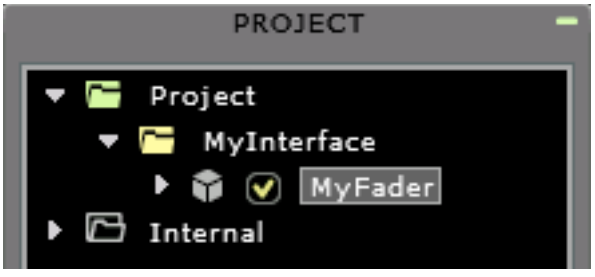
The individual interfaces may be selected by tapping the appropriate tab, programmatically via the **selectinterface(index)** built-in function or via OSC commands.

## 6.3 Creating Objects

Now to start filling this empty space, you can either use the Palette and drag and drop the desired Object onto Lemur panel, or click on the **Create Object** button located at the bottom of the Project panel. Clicking on the Create Object button opens the Create Object window, listing the available Object types by alphabetical order:




To create a new object, we just have to choose one among the list, to give it a name of our liking then to click the **Ok** button. For our first experience with interface building, I would suggest to start with the most common: the Fader. Once created, our newbie Fader appears on the upper left corner of both the editing window and Lemur's screen. It's also referenced in the Project panel.



When you touch the Fader on Lemur, you will notice that the actions are not mirrored in Lemur Editor.

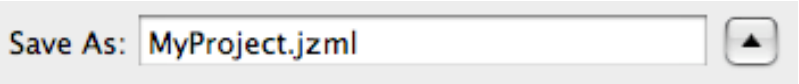
Click the **Synchronization** button to change this. Now everything you do on Lemur will be reflected on your computer. This helps if you have to save a project in a defined state for future use.

Already bored with your lonely fader? That's no problem; let's repeat the very same procedure described above and create a Switch and a MultiBall Object.

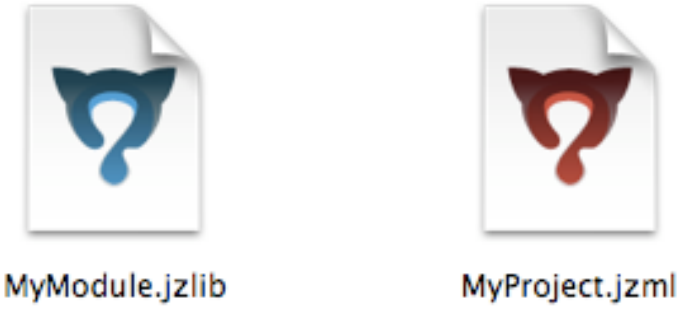
Not surprisingly, these will appear on both the editing window and Lemur screen. And as good news never come alone, the two objects are also listed in the Project panel, sorted in alphabetical order.

## 6.4 Saving your Project

You might want to keep this work of art ready for future loading and editing. Just hit the **Save** button in the toolbar and give a name to your project.




All files the Lemur Editor produces, both Project files or Modules, reside on your hard disk, Project files with the extension **.jzml** and Modules with the extension **.jzlib.**




Now, let's open our project again, by hitting the **Open** button. You might have noticed that the **Connect** Button has changed is status. That's normal behavior. When opening a new Project in Lemur Editor (either by loading an existing one from disk or by creating an empty Project), the connection to Lemur is dropped.

If you connect Lemur again, the Project loaded in Lemur Editor will automatically be transferred to Lemur, overwriting the one displayed on Lemur.

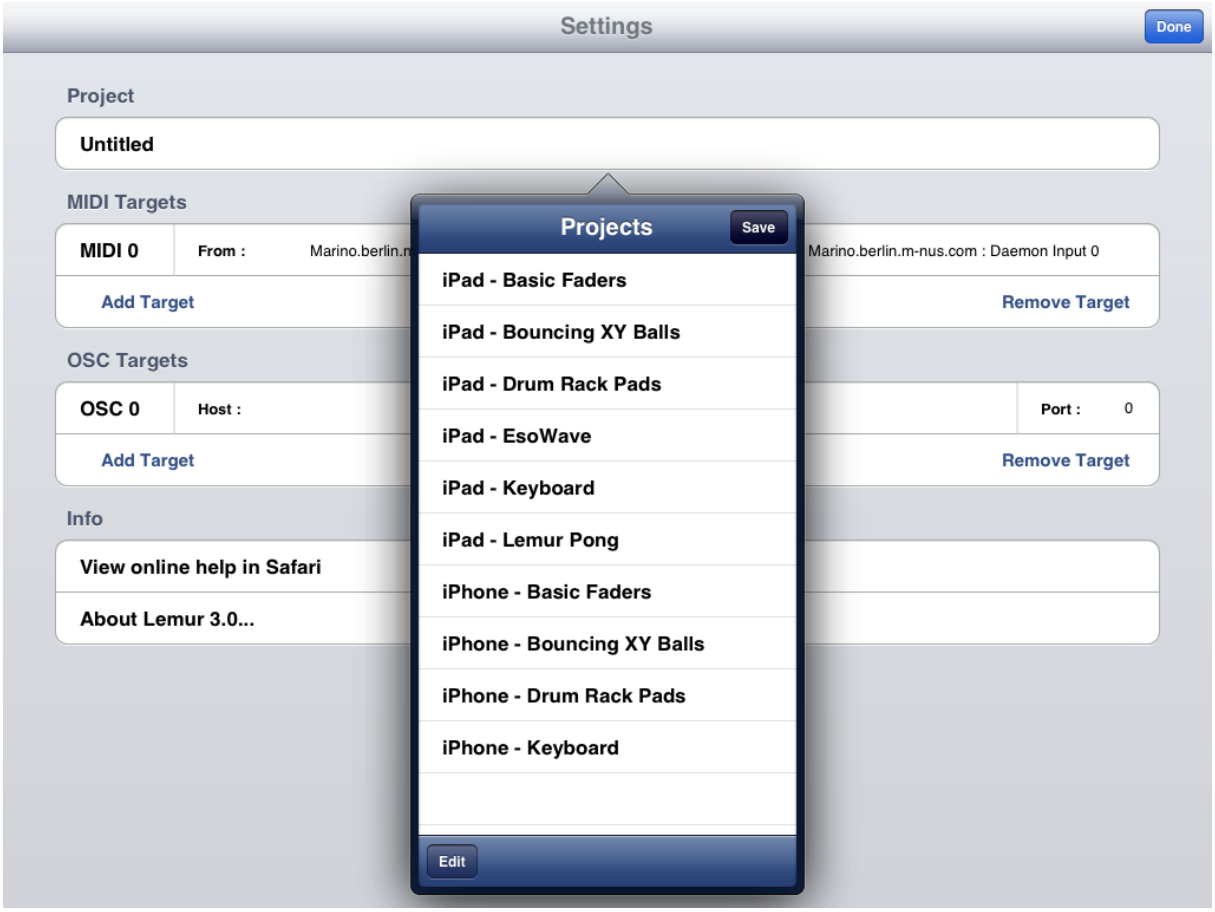
Make sure to save your work in the Lemur Editor before closing the Project.

Once your Project is transferred to Lemur, you may close the Lemur Editor and use Lemur to produce OSC data. It will happily connect to all defined OSC Targets. Remember that if you use MIDI in your Project, the **Lemur Daemon** has to run as it is needed to translate the OSC data from Lemur into MIDI messages.

## 6.5 Lemur's memory for Projects

Lemur has a memory for all the Projects you throw at it. You just have to tell it to remember them. This frees you from always having to open the Lemur Editor when you just want to work with your finished Interface.

In the app, go to the Settings menu and tap the Project list. It displays a list of Projects currently stored in your Lemur's memory.



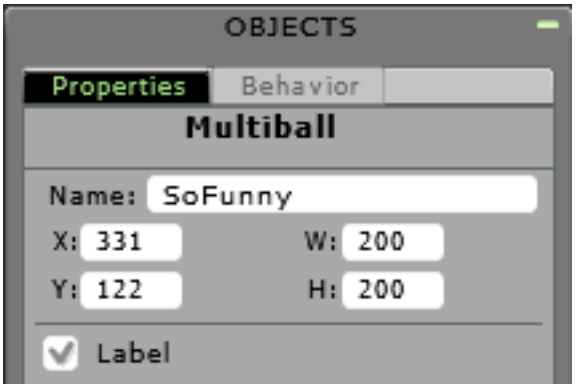

We want to save the important project we have built, so we tap the **Save** command. You can then enter a name for the project. You will see the project appear on the list. You can load a project by tapping on it, or tap **Edit if you need** to delete a project.

## 6.6 Changing Object Appearance

First we are going to resize the Fader to have more space for our fingers. Grab the lower right corner of the Fader and drag it down to make the Fader longer and maybe a bit wider too. Note how the Width and Height Parameters in the Properties change as you do it, and vice versa: if you type a value in one of the dimension fields, the fader will move and resize accordingly.

Talking about the Properties tab, what about changing the colour of our fader? Sure, a yellowish green can be nice, but what about a deep blue? Click on the color picker and pick a color.

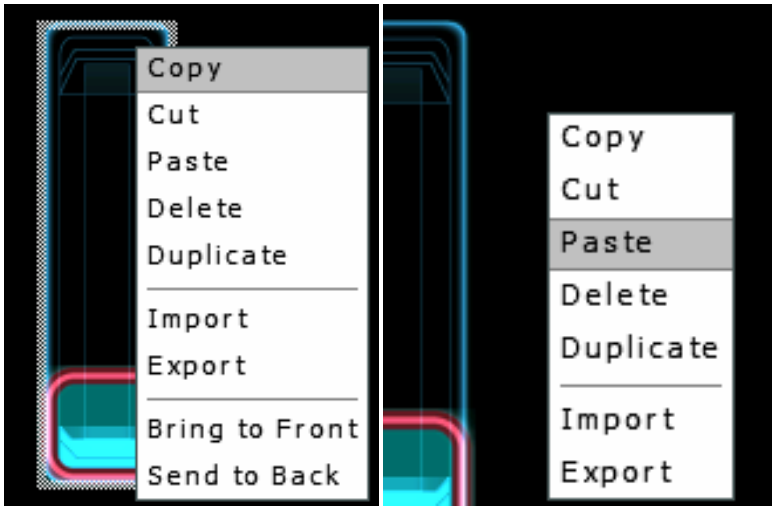
Sometimes it's a good idea to display the name of your Objects in the Interface. Click into the Name field of the MultiBall's Properties tab and change it. Then check the **Label** checkbox.





## 6.7 Groups

Now we want two Faders. We can do this with a simple **Copy** and **Paste**.

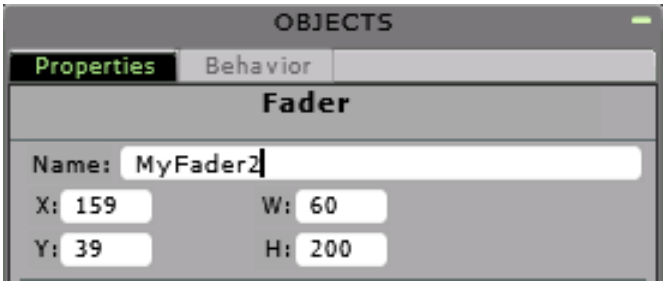



Right-click (Command-click for Mac) on the Fader and choose **Copy** from the context menu. Then right-click on the background of Lemur panel and choose **Paste** from the menu. Those commands are also available on the Toolbar, via shortcuts, or as an alternative, by holding the **alt** key and dragging the mouse to the desired location.

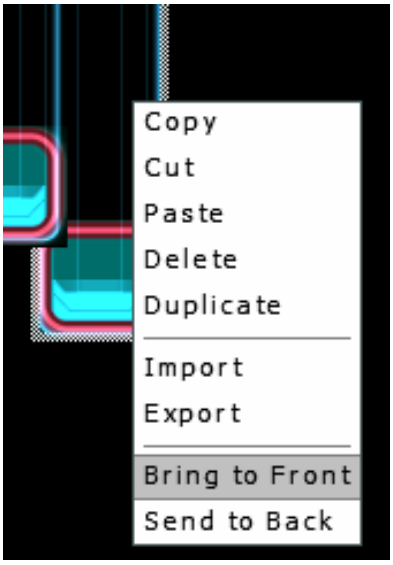
Your Fader is duplicated. Grab it with your mouse and position it to the right of the first Fader.

Have a look at the **Properties** tabs of your Faders. The Lemur Editor automatically named the new Fader "MyFader2" to avoid a name conflict. This is important, as the names are also used as addresses of the Objects for OSC mapping and variable accessing. As a consequence, there cannot be any identical names in any Lemur Project – except by protecting objects inside containers, which we'll cover later in this manual.

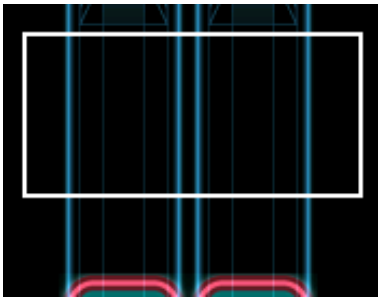
Of course you can also change the names by entering one of your choice into the **Name** field of the Properties.



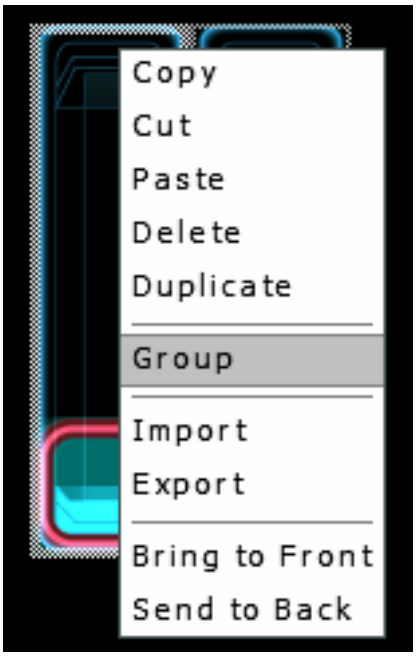

If you have several Objects lying on top of each other, the Lemur Editor provides a way to control the layering order. A right-click on a selected Object opens a menu. Choose **Bring to Front** to make the selected Object the topmost. **Send to Back** puts it to the bottom of the Object pile.




We now want to change the colour of the two Faders. As we already saw, we could do it by changing every Fader on its own. But it's tough to hit the exact same colour twice, so we'll select them both and simultaneously change their colours by dragging a square that touches both Objects with the mouse.



Objects can also be grouped permanently via the context menu by right-clicking (command-clicking for Macs) on the selected Objects and choosing **Group**.





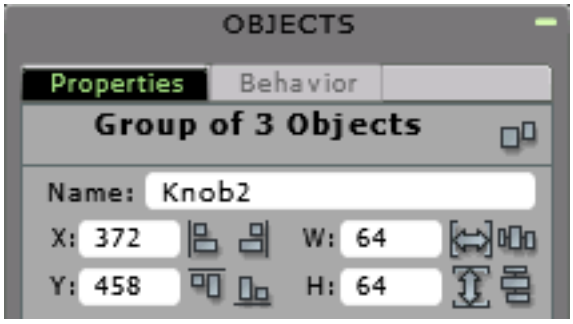
This group command is also available on the Properties panel if several Objects are selected. Do this for your Faders and every time you click on one of them both will be selected. Double-click on your new Group to display the Properties panel. Note that since different kind of objects can grouped together, only the properties common to all objects are displayed. Now dial in a nice purple with the color picker. Both Faders now have the exact same color. If you want to ungroup the Faders, click on the Ungroup symbol in the properties panel.

When you select several Objects or have them grouped, you'll notice additional commands appearing on the Properties panel that deal with layout and size of the Objects.


The commands are:

- Align to Left, Align to Right
- Align to Top, Align to Bottom
- Make same Width
- Make same Height

They always refer to the Object you selected first in the group. These commands are of great help when it comes to tidying up your Interfaces. Experiment a bit with those to get the hang, and remember that the **Undo** command is at your disposal to roll back changes.

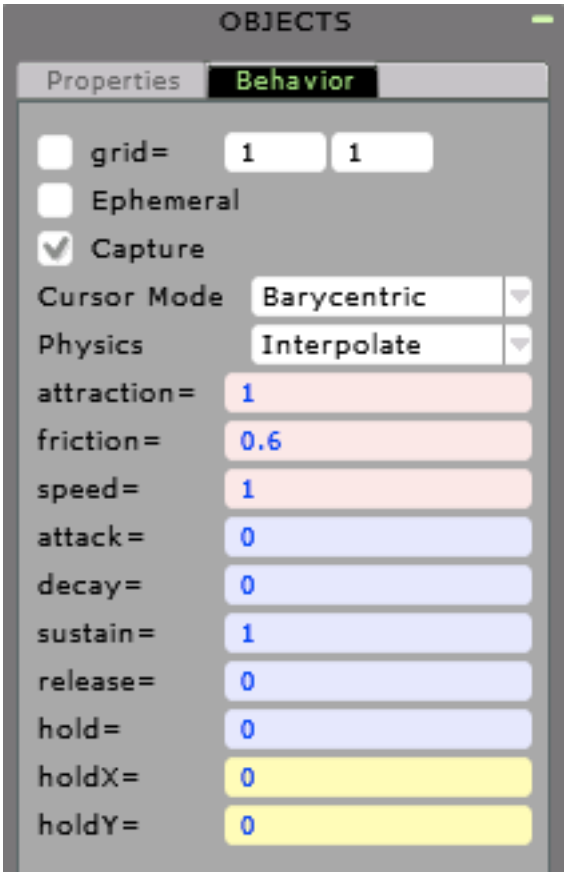



If you select three or more Objects (or have a group of three or more), two additional commands are available:

- **Distribute horizontally** distributes the Objects evenly. The left- and rightmost Objects are the boundaries for the line of Objects.
- **Distribute vertically** does the same, only on the vertical axis.

## 6.8 Configuring Object's Behaviors

Now, let's glance at one of the most powerful feature of Lemur's objects: their configurable behaviors. Select the MultiBall object with the mouse, and look at its **Behavior** tab:




The **Physics** menu has three modes defining the behavior of the balls. When Physics is set to **None**, the ball moves immediately to the position of your finger and stays there. It also follows your finger around as you move it on the touch screen, and immediately stops when you lift up your finger.

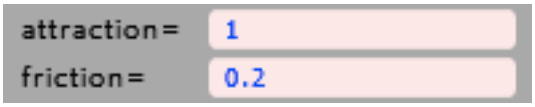
Make sure the Physics menu is set to **Interpolate**, which should be the default setting.


Touch Lemur screen somewhere in the MultiBall, but not on top of a ball. The ball will not move to your finger immediately but according to the setting of the **Attraction** parameter. Let's modify

Attraction to make the ball react more slowly. Enter a value of 0.1. Experiment again with the Ball's reaction. It should follow your finger slowly.

Now let's try the last behavior mode. Choose **Mass-Spring** from the **Physics** menu.

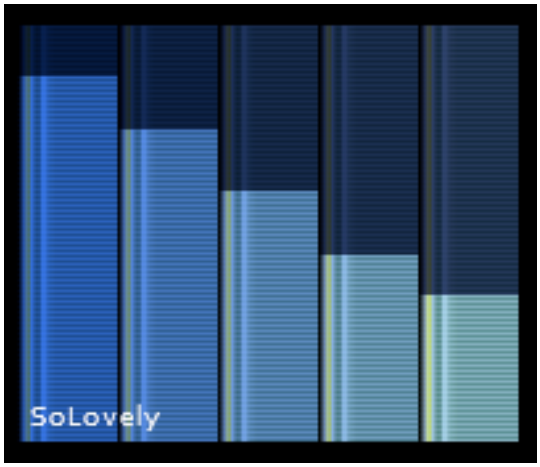

Change the **Attraction** value to 1.0 and the **Friction** to 0.2.




Try moving the ball now. It bounces off the "walls" and eventually slows down. The Mass-Spring mode is similar to Interpolation, except that the ball has friction (or a lack of it), can bounce off walls, and in certain cases, the ball may oscillate before coming to a complete rest. You can also control the speed of the ball via the **Speed** parameter.

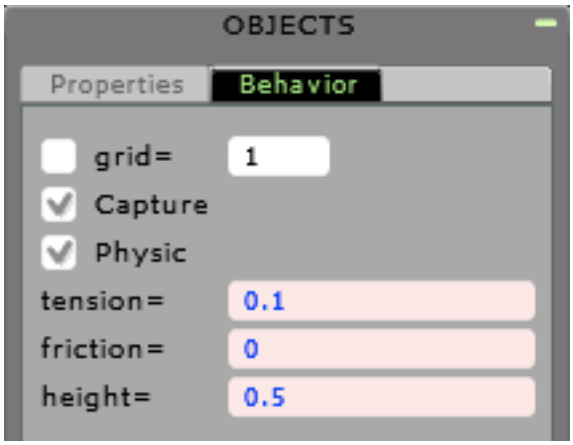
Physics parameters are great for producing complex time-varying values with little effort. We will see later on that we can obtain even more enjoyable effects by controlling the Physics properties with other objects.

As another example we create a MultiSlider Object with 5 sliders and name it *SoLovely*:

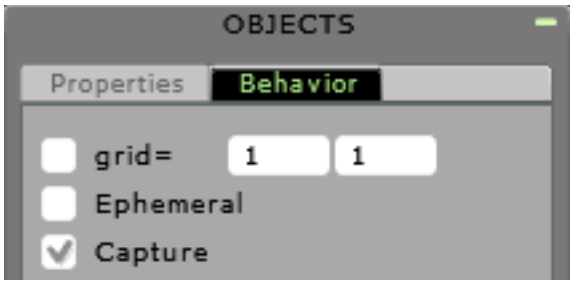

We then drag it to a free spot on the Interface, and make it bigger so we can interact easily with the individual sliders.

By default the MultiSlider Object doesn't display any physical behavior. The sliders simply follow your fingers and stay where you leave them.




Now change to the **Behavior** panel and activate the **Physic** checkbox. Also, change the tension to 0.1 and the friction to 0. Now it's a completely different story. If you move the sliders they will continue to wiggle forever. They behave as if they are connected with springs and oscillate around a centre value defined by the **height** parameter.

Again, we are only scratching the surface here. Please have a look at the Object Reference Chapter for details about all Object Properties.

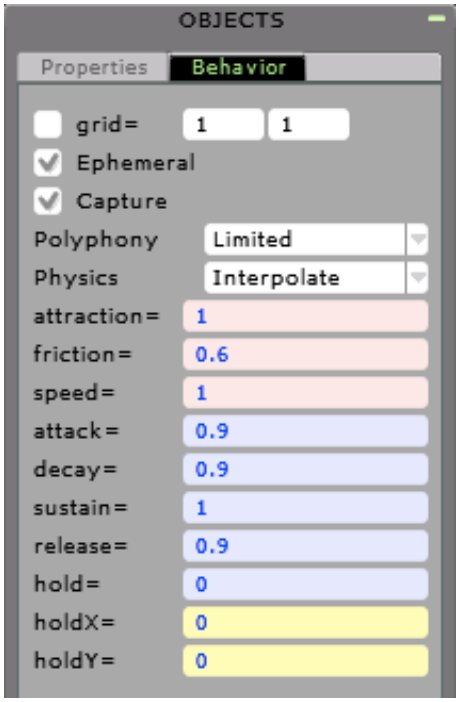



Let's come back to our MultiBall Object. The **Capture** parameter on the Behavior tab has a great influence on the possible finger actions. When you are inside of the Object's boundaries, the Capture mode doesn't make any difference. But if you move the ball and leave the borders of the Object, the control of the ball will stop when Capture is off. If Capture is on, the ball will continue being tied to your actions. No matter if you move your finger across other Objects and even if you switch to a different Interface, the ball will still cling to your finger and the finger has no effect on other Objects. Just experiment a bit with the two modes and you will quickly get the hang of it.

The MultiBall Object is a complex critter. It also sports an **ADSR+H** envelope for its brightness parameter or z Variable. You might be familiar with envelopes from using synthesizers. They represent an easy way to produce defined parameter changes over time.

The MultiBall ADSR acts just like its cousins from the synthesis world. When you touch Lemur the brightness of the ball will be faded in according to the Attack value, decay down to the sustain level and fade out with the release time when you lift your finger from the surface.

Objects supporting ADSR envelopes (MultiBall and Pads for the time being) also have a **hold** parameter. Its effect is similar to a sustain pedal, freezing the object's state as long as its value is 1. When set to 0, if has no effect. Interesting effects can be achieved when setting the hold parameter to an expression depending on other objects' state. We'll cover such methods later on.




Hereis an example of brightness ADSR. Please note that the Ephemeral mode has to be active for the envelope to work. In **Ephemeral** mode the balls appear when you touch the surface and they vanish when you lift your finger.

Sometimes you don't want the full resolution of an Object because the target parameter in your software is quantized to only a few values. You can quantize the values sent via OSC by using variables and expressions, but the steps introduces by the quantization won't be reflected in the behavior of the Object.

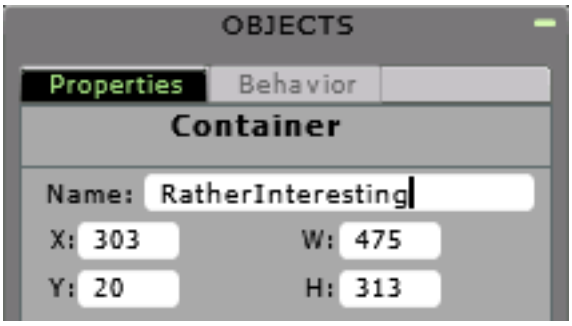
Enter the **Grid** parameter. This parameter is available for several objects, and "quantizes" the movement of your Object into multiple **steps.** The number of steps can be chosen in the adjacent fileds. Test it with the *SoFunny* MultiBall object. Open its Properties and check the grid flag after setting its x-value to 3 and the y-value to 5.


If you now have a look at the Object you see that the MultiBall area displays a grid and the Ball can only move to distinct points on each axis.


The values produced by the *SoFunny.x* and *SoFunny.y* variables are of course also quantized to produce only three steps on the x-axis (0.000, 0.5000, 1.000) and five steps on the y-axis (0, 0.25, 0.5, 0.75, 1).

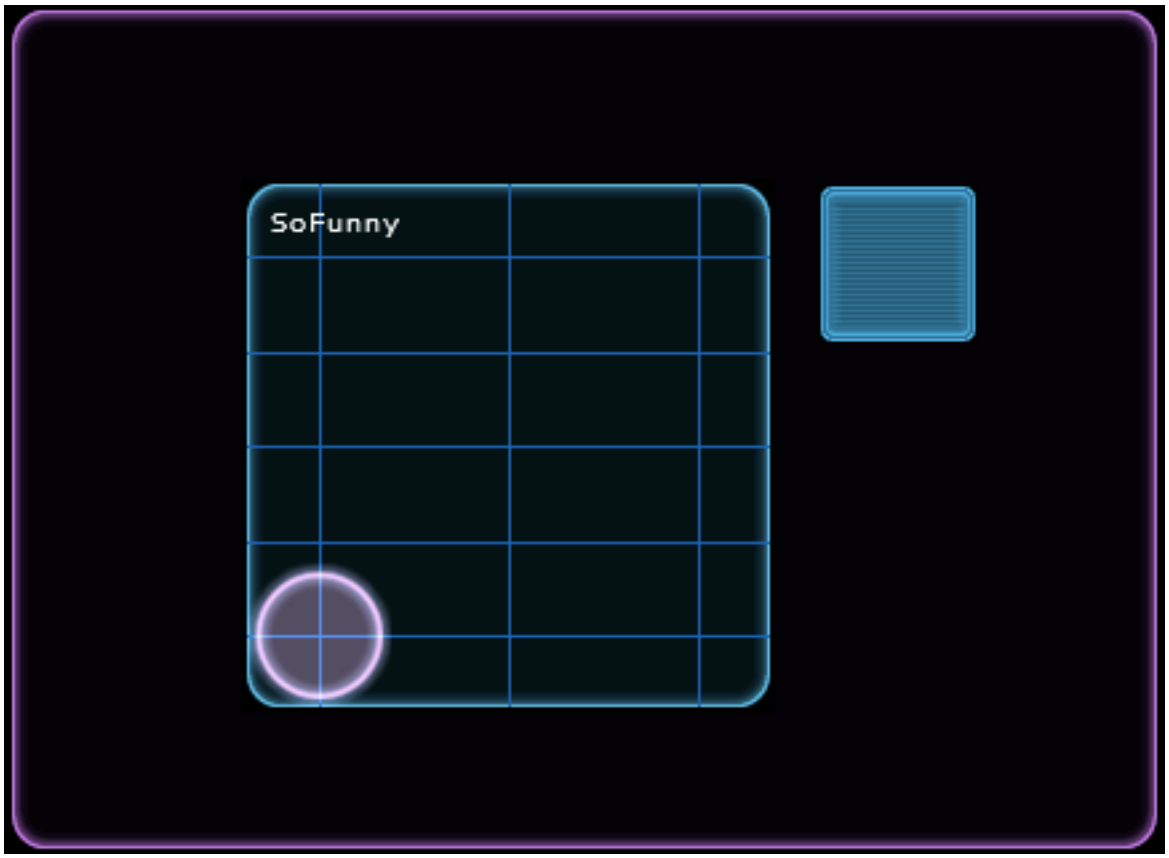
## 6.9 Using Containers

The next thing we want to do is put some of the Objects into a Container to separate them visually from our Faders. To do this, drag a square around the MultiBall and the Switch Object to select them. Now right-click (ctrl-click on Mac) on the selection and choose **Cut** from the menu. The Objects disappear. They are not gone, though, but just got copied into the clipboard of the Lemur Editor.

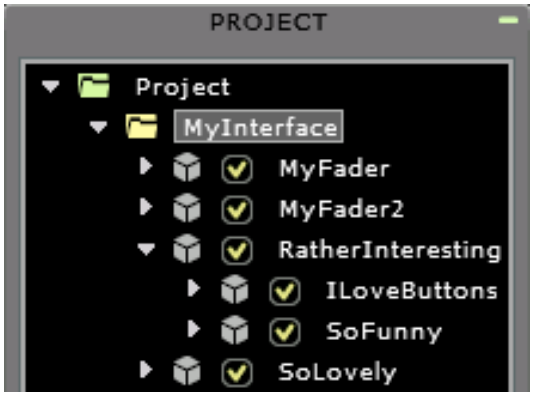



Click the **Create Object** button and pick a Container from the displayed list, or drag and drop it from the Palette. Name it and position the Container to the right of the Faders, then resize it to encompass the complete right part of the Interface.

Right-click (Command-click for Macs) the Container and choose **Paste** from the menu. This puts the two Objects we have cut out into the Container.

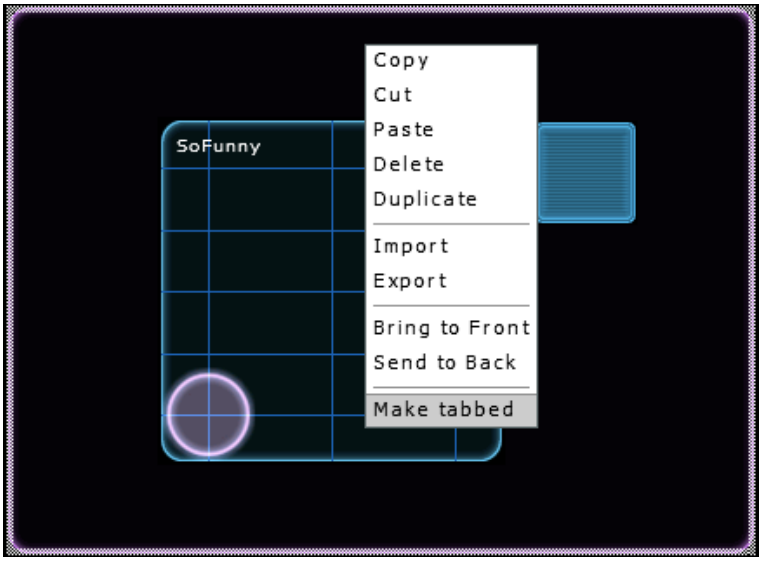


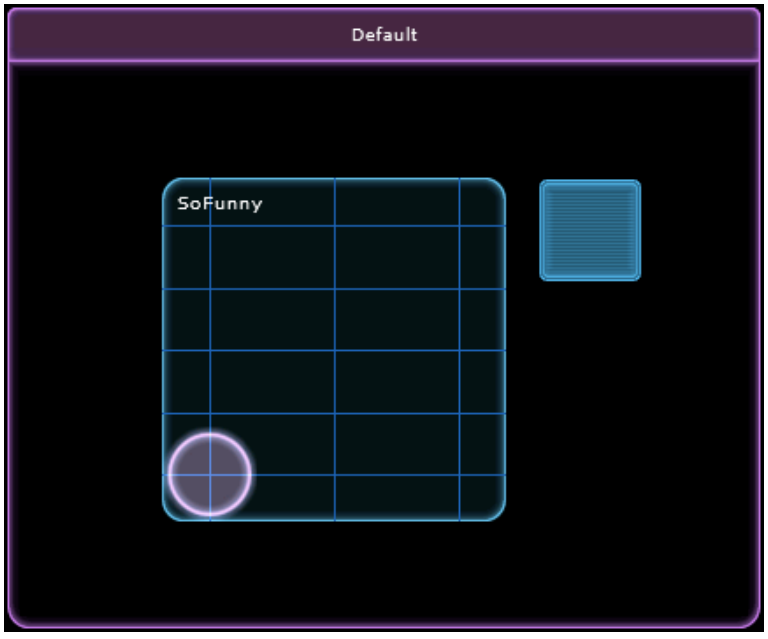

Now it's time to have a look at how all this is mirrored in the Project panel. The Container can be opened in the Project panel by clicking on the **disclosure triangle** in front of the Container. You see that the two Objects in the Container are grouped on a lower level in the Project hierarchy.



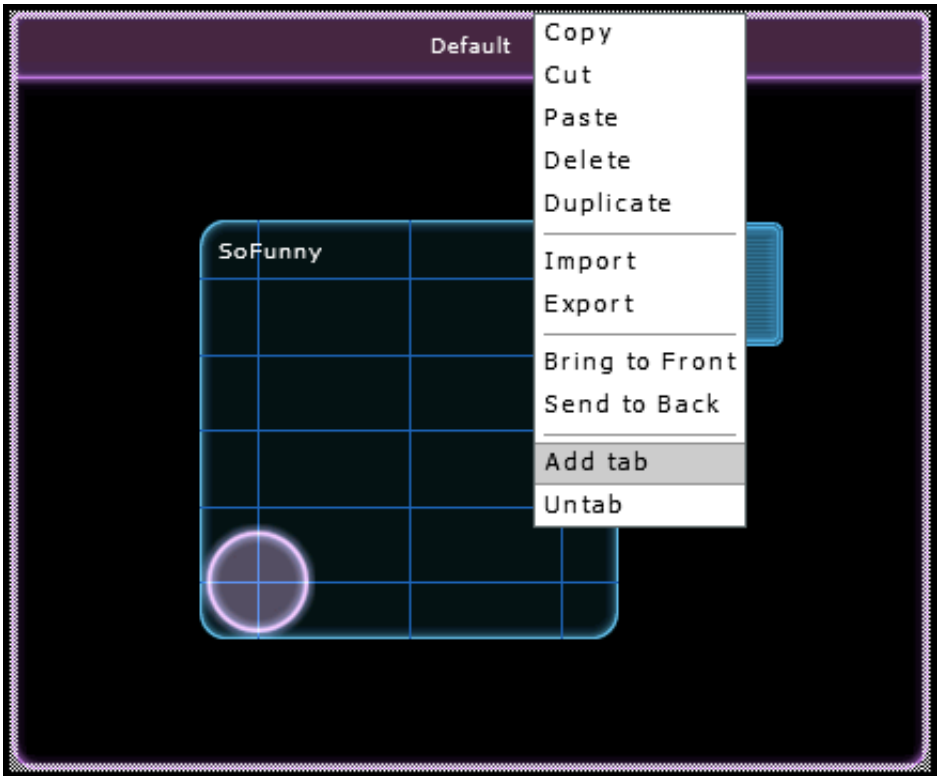

Note that the use of Containers also has implications for the Variable addresses of Objects living inside of the Container. Indeed, within the Container, the local names of both objects are still the same. From the outside however, there are now known as *RatherInteresting.ILoveButtons* and *RatherInteresting.SoFunny* respectively. Do you feel a little bit lost? Well, that's normal at this point, but don't worry, we will come back to this later.

A Container Object can also be made tabbed and additional tabs inserted in the Object. To make your Container tabbed, simply by ctrl-click (Mac)/ right-click (PC) onto the standard Container object and choose the option **Make tabbed**.



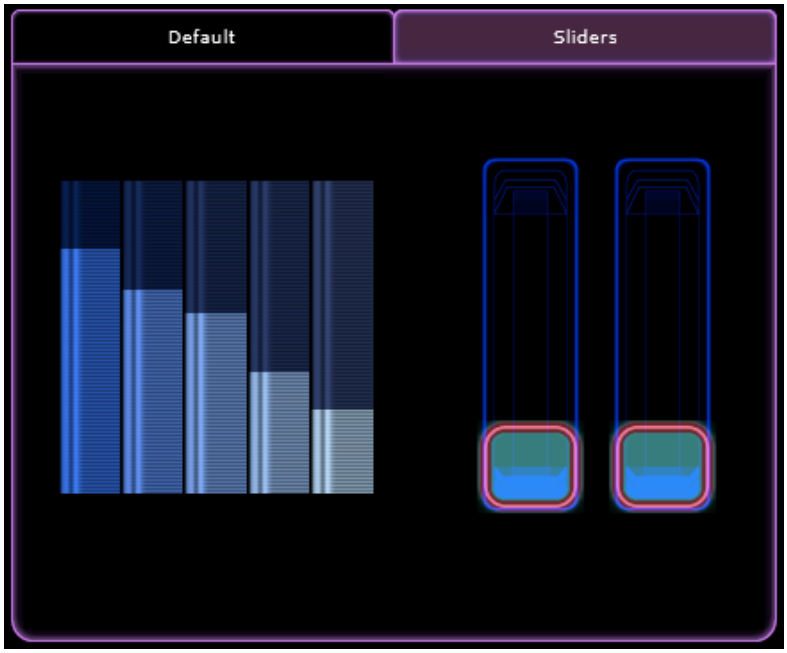



Additional tabs can then be inserted with the option *Add tab*:

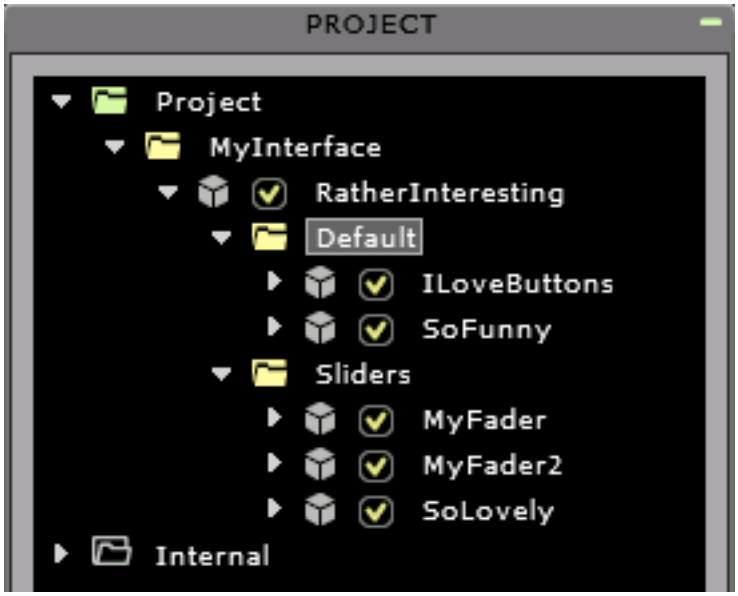



A small window then appears to name the new tab. Call it *Sliders* for instance :


Now select the remaining Objects outside the Container, cut them and paste them inside the new tab:

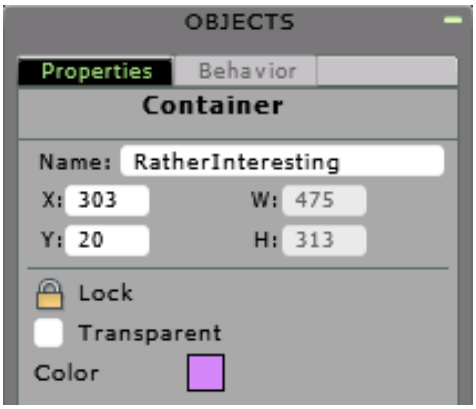



In the Project panel, each tab is represented as a folder inside the Container Object. To switch from one tab to the next in the Lemur Editor, click on the desired tab in the Project panel, or enter into Run mode by holding the "**e**" key on your keyboard and clicking on the corresponding **TabBar**.

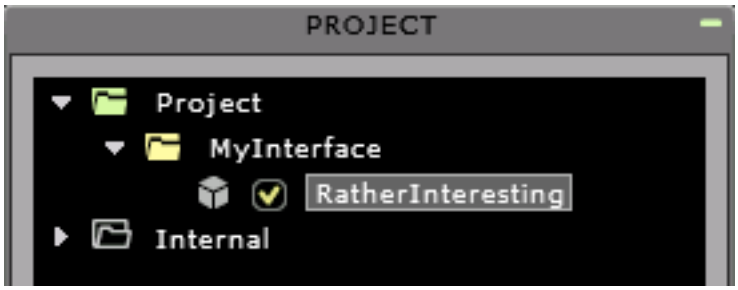



If needed, the tabs 'order can be modified by clicking within a Container and choosing the **Bring up** or **Bring down** actions.

In large Projects the Project panel can quickly become crowded with Objects, leading to a lack of overview. In that case you may want to use the Container Object's **Lock** feature. Just try it with our *RatherInteresting* Container. Check the Lock flag in the Properties and you will see the MultiBall and Switch Objects vanish from the list.




A locked Container doesn't show its contents in the Project panel. This of course also means you can't edit the Objects inside of it. Simply unlock it again, if you need access to the contents.




## 6.10 Import and Export of Modules

Now let's suppose you would like to reuse later some portions of the interface in a different project. For that purpose, the Lemur Editor's Toolbar provides the handy **Import Module** and **Export Selection** commands for the integration and creation of Modules. Any imported Module is incorporated into the currently opened Interface and you can select any group of Objects for export as a Module to the file system (the file extension for Modules is **.jzlib**).

Please note that you can also import and export elements via the **contextual menu** (right-click/control-click on the Editing Area). Alternatively, you may drag your selection directly to the Library panel while holding the **alt** key, provided you've defined a path for your Library files in the Lemur Editor's settings page.

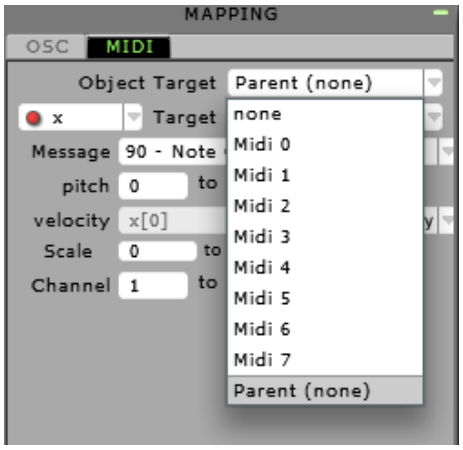
Make sure that you do not destroy any dependencies in terms of Variables and expressions when you export parts of your Project. It is good practice to encapsulate the exported Module in a Container and have the variables and functions that are important for the Module's functionality created locally to that Container. If you define them globally, they will not get exported and the functionality of the Module will be broken.

# Chapter 7 - Mapping

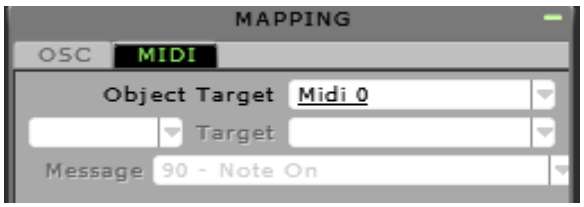
## 7.1 Setting up MIDI messages

The simplest way to assign MIDI messages to your Objects' Variables is to use the Mapping panel's MIDI tab and select a Variable / Target mapping.

The top of the MIDI panel features the **Object Target** menu, which lets you select a default MIDI Target for all built-in or User defined Variables local to the Object. You can choose *none*, any of the 8 available MIDI Targets, or **Parent**, a Parent being a higher hierarchy level in the Project panel's tree. An Object's Parent can be a Container or the Project itself. A Variable's Parent can be an Object, a Container or the Project itself…

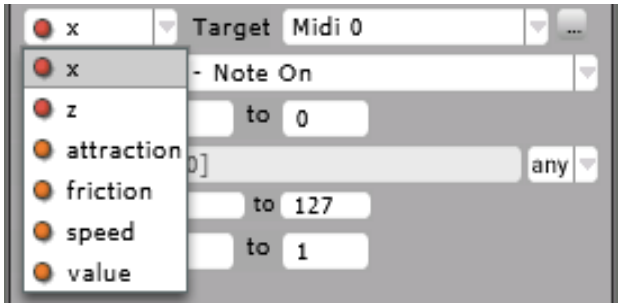



If an Object Target has been set up at a higher hierarchy level, this will be stated in the pull-down menu inside the brackets next to *Parent*. Let's check it out and set up a Target at the Project level. To do that we select the Project folder in the Project panel, and set its Object Target in the Mapping panel to, say, Midi 0:

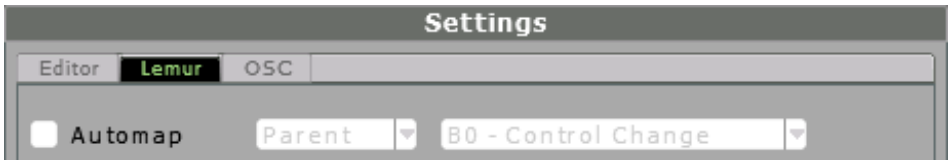

Now, for any object created prior to this, the Parent will appear in the Object's Target menu. As for any new Object, it will automatically be set to the Parent's Target.

Below the Object Target's menu, you find the **Variable menu**, for choosing a local Variable to pair with a MIDI message, and the **Variable's Target menu**, for selecting a Target, which may different from the Object Target if needed. Note, it is this value that defines MIDI transmaission.

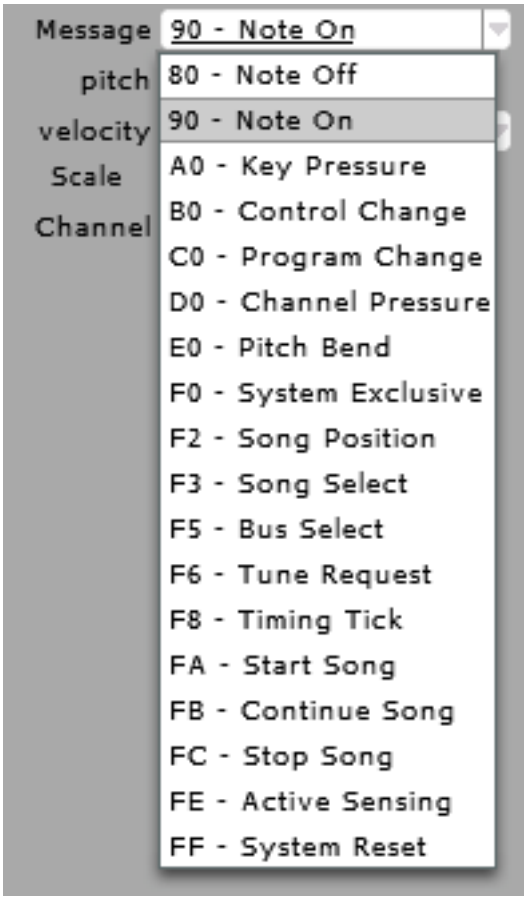




Clicking on the **Quick Map** button (the tiny icon next to the Variable's Target menu) automatically sets the Variable's Target and MIDI message selection to those chosen in the Lemur Editor settings window's Lemur tab for the **Automap** feature. This can be thought of as a handy "semi-automatic" mapping function.

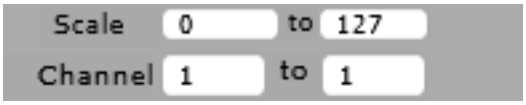


The next element is the **MIDI Message menu,** whose purpose is, quite naturally, to let you select the MIDI message you want to pair with your Variable. Messages range from Note messages over Controllers to System Realtime messages, like Song Start/Stop or Active Sensing (the different messages type are preceded by their MIDI status byte in hexadecimal form). In most cases though, you will either choose a Control Change or a Note On MIDI message.




Depending on the chosen message type, the MIDI tab displays different parameters to tweak, such as "pitch" for Note On messages, or "controller" for Control Change messages. Please refer to the MIDI Message reference on Chapter 14 for details.

The **Scale** parameter, typically set from 0 to 127, ensures that the 0-1 values generated by the Object's Variables are scaled to the MIDI values, which are typically integers (whole numbers) between 0 and 127




It is important to understand that the communication with MIDI Targets set up in the MIDI tab is **bidirectional**. In other words, a Variable to which a MIDI message has been assigned in the MIDI tab can transmit this message but will also react to the same incoming message. This explains why you also find Variables in the Variables menu that do not output values.

By the way you might already have noticed the different colours for the Variables in the menu:

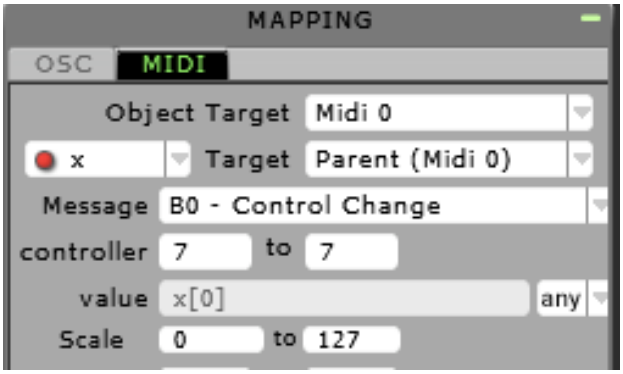
- **Red** stands for the Object's main parameters (x for a Fader; x, y, z for a MultiBall, etc.)
- **Orange** depicts a properties/behavior parameter (like friction, attraction, value, etc.)
- **Green** entries are Expressions created by the user.
- **Blue** entries are Functions created by the user.

The use of the **Trigger Mode menu**, common to three Mapping panel's tabs, is detailed later on in this chapter. It allows for choosing when and how often the messages are actually transmitted (triggered).

## 7.2 Simple MIDI mapping examples

Let's have a look at a simple example:

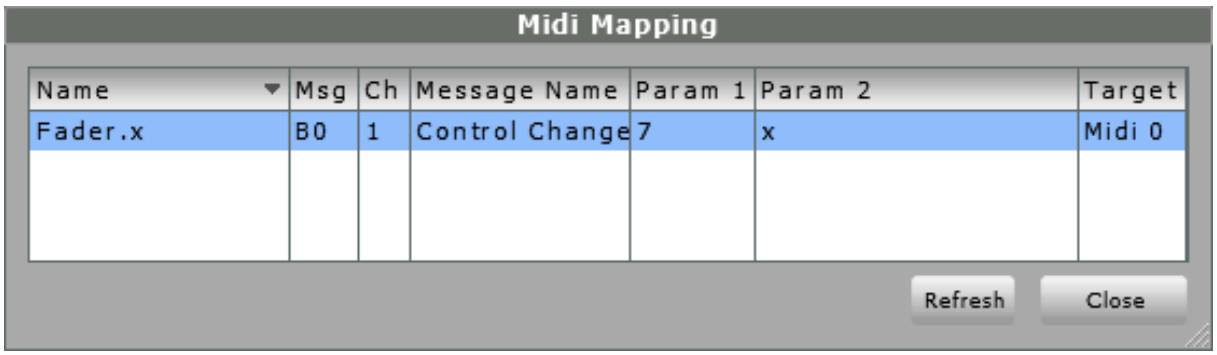
- Select a Fader in your project. To assign a MIDI message to this fader, click its MIDI tab within the Mapping panel.




- Change the settings of the dialog to those shown above (Message is Control Change, controller number is 7, and value is x). The Scale fields should be set from 0 to 127. Set the MIDI channel according to the channel of the desired Target, which is chosen via the Target Menu to the upper left. These settings configure the Fader to transmit MIDI controller 7 (volume) with the fader's 0-1 floating-point values scaled to 0-127.

- If you wish to, you can click on the **MIDI Map** button located on the Lemur Editor's Header to verify the MIDI assignment:

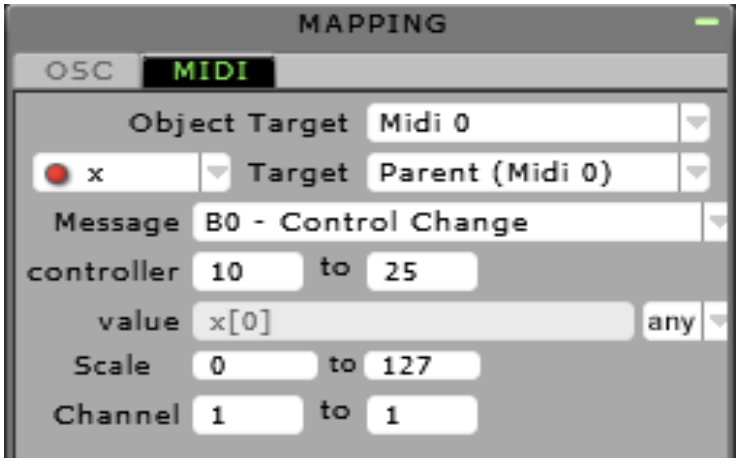
The MIDI Mapping window then appears, displaying the assignment we just configured:



In the next example, we'll use a MultiSlider object and automatically assign different control change messages for each slider.

- Create a MultiSlider object. Using the Properties tab, set the number of sliders to 16.


- Now click on the MultiSlider's MIDI panel

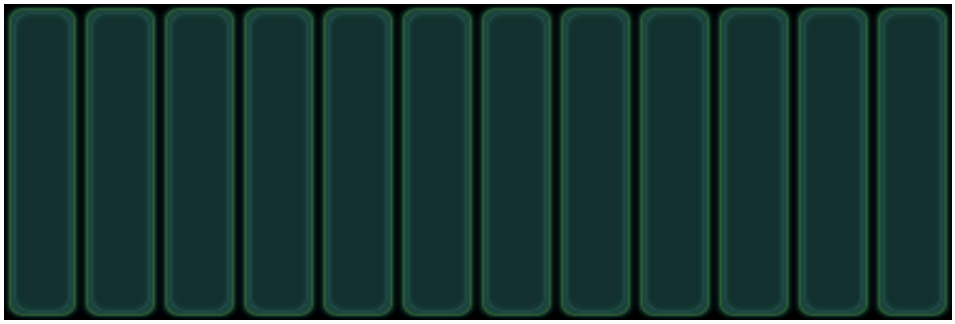



- Choose **B0 Control Change** and **MIDI 0** as Target for the x Variable.

- Type controller number **10** into the first controller field. The second controller field is automatically set to **25** to extrapolate for the 16 sliders. Indeed the Lemur Editor automatically assigns each slider to successive MIDI controller values. The same thing happens for other multi-value Lemur variables (such as MultiBall's' parameters).

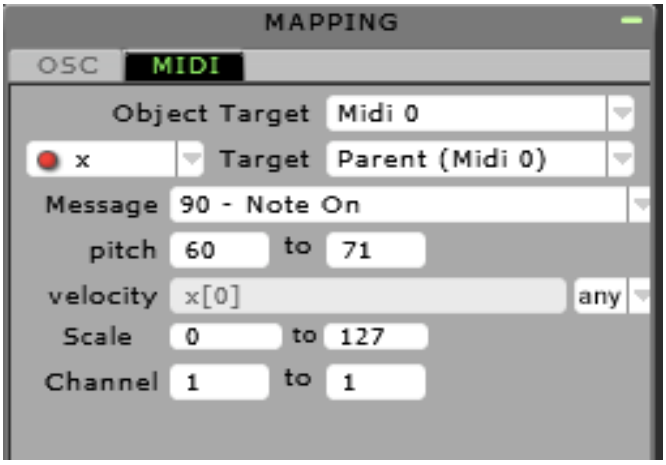
The following example uses the same automatic mapping feature to create a chromatic one-octave MIDI "pad" on Lemur that will send MIDI note messages.

- Create a Pads object. Give it 12 columns and one row.

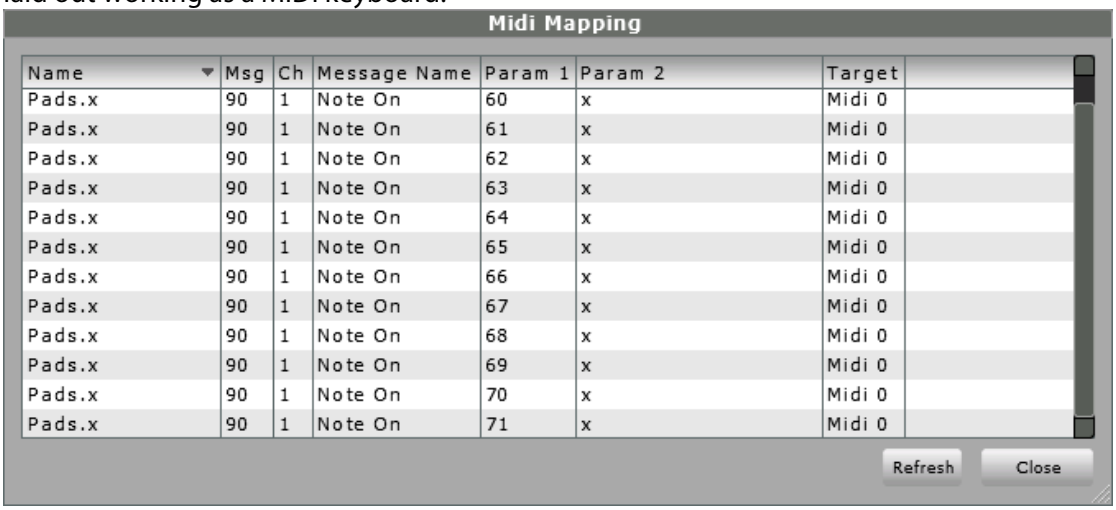



- Select the Object and go to the Mapping panel's MIDI tab. Choose the x Variable and Note On as the MIDI message type.

- Set the first pitch field to 60 and the second will automatically be expanded to 71 (reflecting your 12 Pads). Your MIDI assignment should resemble the one shown below.

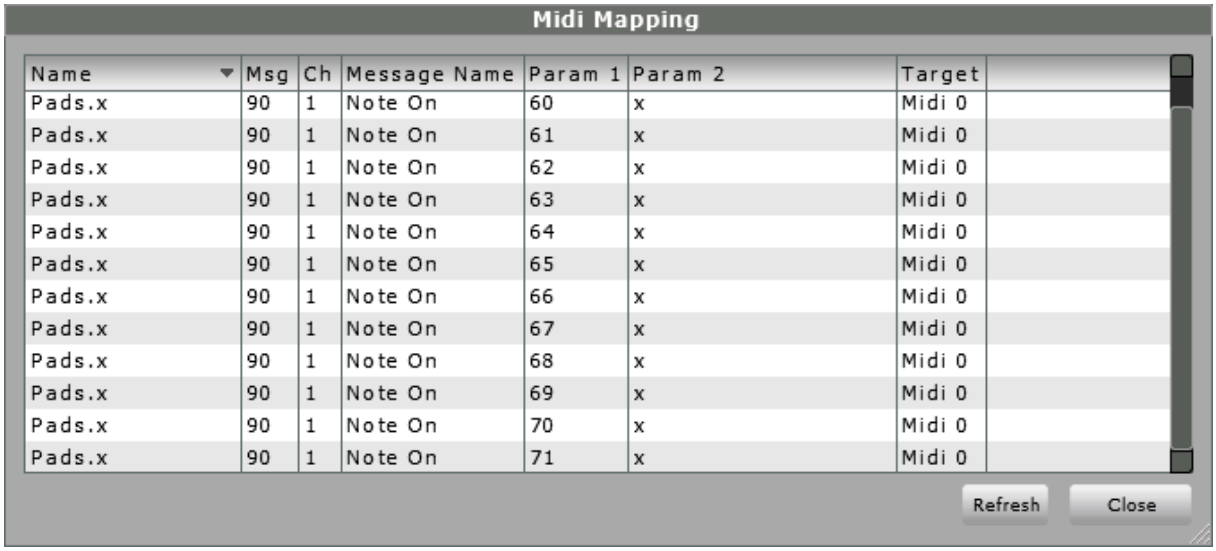



Click the MIDI Map icon to look at the MIDI Mapping window. You see the 12 Pads nicely laid out working as a MIDI keyboard.




You can use more than 12 pads to create a bigger keyboard if you wish.

The Midi Mapping Window consists of seven columns:

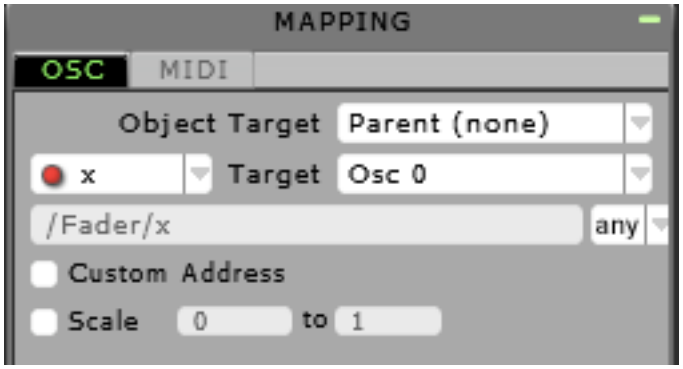



- The **Name** column shows the name of the Variables used.
- The **Msg** (Message) column displays the numbers of the different MIDI messages used for the individual mappings.
- In the **Ch** (Channel) column you find the associated MIDI channel for the individual mappings.
- The **Message Name** column writes out the full name of the different MIDI messages.
- The two **Parameter** columns show the MIDI parameters associated with the chosen message type, i.e. the CC number and the CC value for a MIDI controller message.
- The **Target** column shows which of the MIDI Targets that are defined on the Settings window have been chosen for the respective mapping.

The **Refresh** button updates the list of MIDI mappings and you can close the window by hitting the **Close** button or the escape key on your computer keyboard.

## 7.3 OSC

The **Open Sound Control** protocol specifies the transmission of messages between two devices. Rather than attaching specific meanings to these messages - as it is done with MIDI - Open Sound Control allows you to define your own system of messages. With Lemur, the names of objects you create and their "path" in the project hierarchy constitute their default "address" for OSC messages.



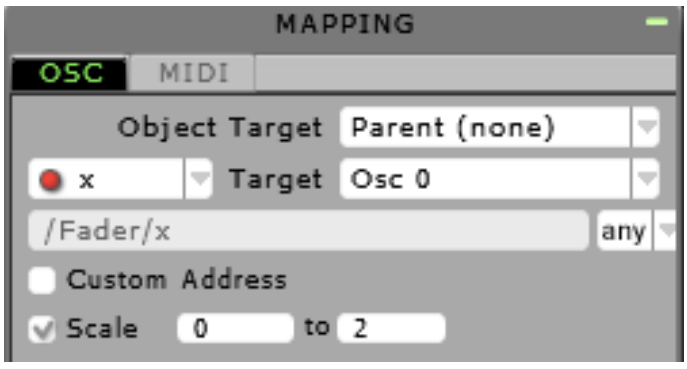

As in the MIDI tab, the top of the **OSC** tab features the **Object Target menu** which lets you select a default OSC Target (communication path to an application) for all built-in or User defined Variables local to the Object.

On the next line, you see the **Variable menu** and to its right the **Variable's Target menu**. Let's say we want to transmit the Fader's x Variable via OSC. If you pull down the Variable menu, you see that it lists all available Variables of the Fader: Choose the x. From the OSC Targets menu, choose the Target that you have set up in the general OSC Settings.

If you now move your Fader it will send the value of x to the OSC address /Fader/x. Use whatever OSC software you have to check it.

If you want to change the OSC address use the **Custom Address** flag and type in whatever address you need. This may be useful in case of naming conflicts between Lemur Projects and projects running on the Target side.

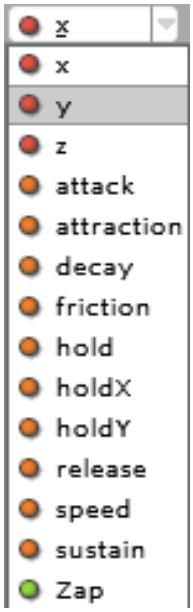
Default parameters of objects range between 0 and 1 (such as the x variable of a Fader). However, you can choose to scale their OSC output to your own range. Enable the **Scale** checkbox and enter other integer boundaries for the output.




The communication with OSC Targets is **bidirectional**. You define the pairing for both incoming and outgoing OSC data with the various parameters of your Objects on the OSC tab. That's why you also find Variables on the menu that don't output values: they can be remote controlled from the displayed OSC address.

Main object's parameters react to incoming OSC according to their OSC range. Sending /Fader/x 1.0 to Lemur will bring the Fader's cap to the top if the default range is used. If you've customized the range to 0..2 as explained above, the Fader's cap will instead go midway, since 1.0 is the centre of the 0..2 range.

**Note** that OSC scaling is only enabled to object's main parameters, and not to user-created expressions, since those have undefined range by essence.



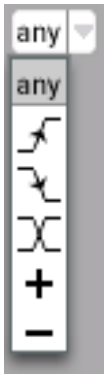

You might already have noticed the different colours for the Variables in the menu. By the way, the above menu is from a MultiBall Object.

- **Red** stands for the Object's built-in Variables (x for a Knob; x, y, z for a MultiBall, etc.)
- **Orange** depicts a properties/behavior parameter (like friction, attraction, value, etc.)
- **Green** entries represent User-defined Variables.
- **Blue** entries represent User-defined Functions.

The use of the Trigger Mode menu is detailed later on in this chapter. It allows for choosing when and how often the messages are actually transmitted (triggered).

## 7.4 Trigger Modes

The purpose of the Trigger Mode menus is to define when a message should be transmitted with respect to a change in a Variable's state.




**any**: The message is sent whenever the parameter changes.

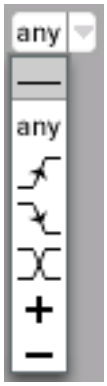
**up**: The message is sent each time the value rises from 0 into the positive value range.

**down**: The message is sent each time the value reaches 0 from the positive value range.

**up and down:** The message is sent each time the parameter reaches 0 OR rises from 0.

**+** : The message is sent each time the parameter increases above its previous value.

**-** : The message is sent each time the parameter decreases below its previous value.




For **Custom MIDI Messages** (see Chapter 10 for an explanation) there is an additional item on the Trigger Mode Menu: None

**Note**: the first entry ("**None**") means that a change in the attached value or expression will not trigger a transmission of the MIDI message. It might, however still get triggered by values or expressions associated with other parameters of the Custom MIDI message that don't have their Trigger Mode set to None.

# Chapter 8 - Targets Setup

## 8.1 Lemur Daemon Targets

The Lemur Daemon handles the MIDI data flow between Lemur and the MIDI ports installed on the computer.

The Lemur Daemon automatically scans all available MIDI ports on the computer, be they physical or virtual, and make them available to Lemur(s) connected to the network. On Mac, the Lemur Daemon also creates its own set of virtual ports, the Daemon inputs and outputs, which are accessible in your applications' MIDI settings. On PC, third party virtual ports such as those offered by loopMIDI, loopbe or MIDI Yoke are needed in order to link to the Lemur Daemon and use MIDI with Lemur.

It doesn't end there. You can also have several computers on the same network all running Lemur Daemons and all available MIDI ports of all computers will be accessible from Lemur MIDI Targets setup. This means maximum flexibility in terms of MIDI topology. Regardless of the complexity of your MIDI network, you can reach every single device via Lemur, as long as it is connected to a computer on the same network as Lemur.

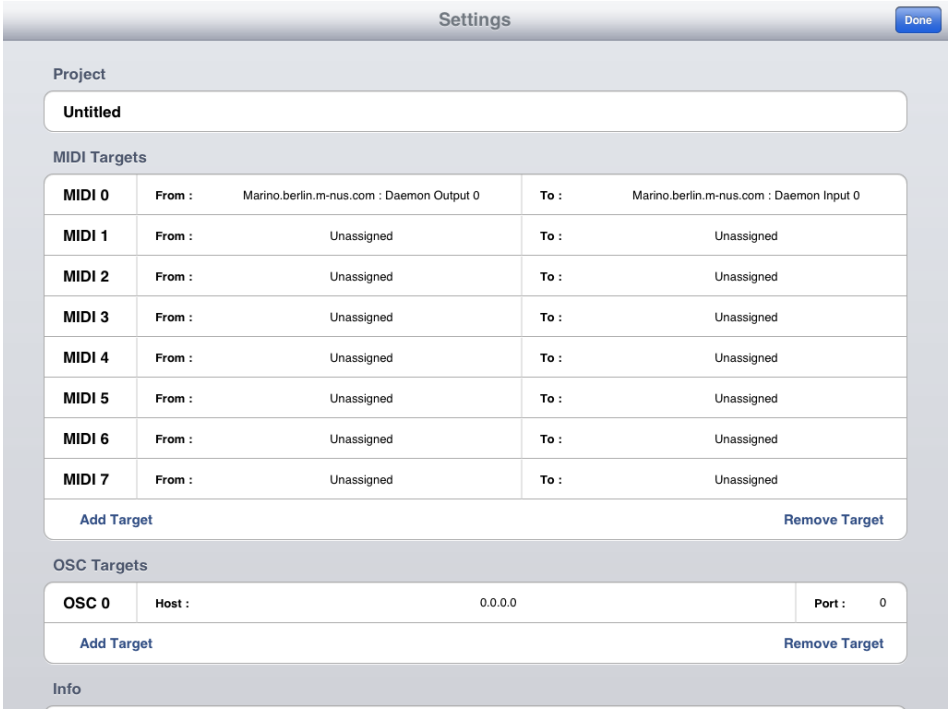
Lemur Daemon Targets can either be set up on Lemur itself or via the Lemur Daemon running on a computer.

There are some thoughts to be invested about being able to make MIDI connections on Lemur and making them from the Lemur Daemon itself. The two methods lead to different behavior when starting Lemur or the Daemon, respectively:

- Connections created from Lemur are saved in its internal memory. Lemur will then automatically attempt to reconnect to the Daemons at boot. If the connections can't be re-established (due to a missing Daemon on the computer), the entries on Lemur's Lemur Daemon Targets settings are greyed out. As soon as the awaited Daemon appears on the network, Lemur makes the corresponding connection.

- Connections created from the Lemur Daemon itself are saved in its preferences file. They will be automatically remade when the Lemur Daemon is launched. If the connections can't be re-established (because Lemur is connected), the entries on the Daemon's settings are greyed out. As soon as the awaited Lemur appears on the network, the Daemon makes the corresponding connection. If you need to have several Lemur Daemon setups depending on which computer Lemur is hooked to, it is recommended to create the connections from the Lemur Daemons: this way, whenever you plug Lemur to any of those computers, the relevant connections will be made.

On Lemur:




The Lemur Daemon Targets configuration screen on Lemur displays the currently defined **MIDI Targets.** There can be up to eight of each kind. The MIDI communication is bi-directional, meaning you can choose a MIDI input port (incoming to Lemur) and a MIDI output port (outgoing from Lemur). Keyboard and Mouse communication is one-way only (outgoing from Lemur to a computer). The ports available in the menu depend on the ports detected by the Lemur Daemons currently running on the network.

By default only a single target is visible. Tap **Add Target** to view more. Whether visible or not, Targets are always active.

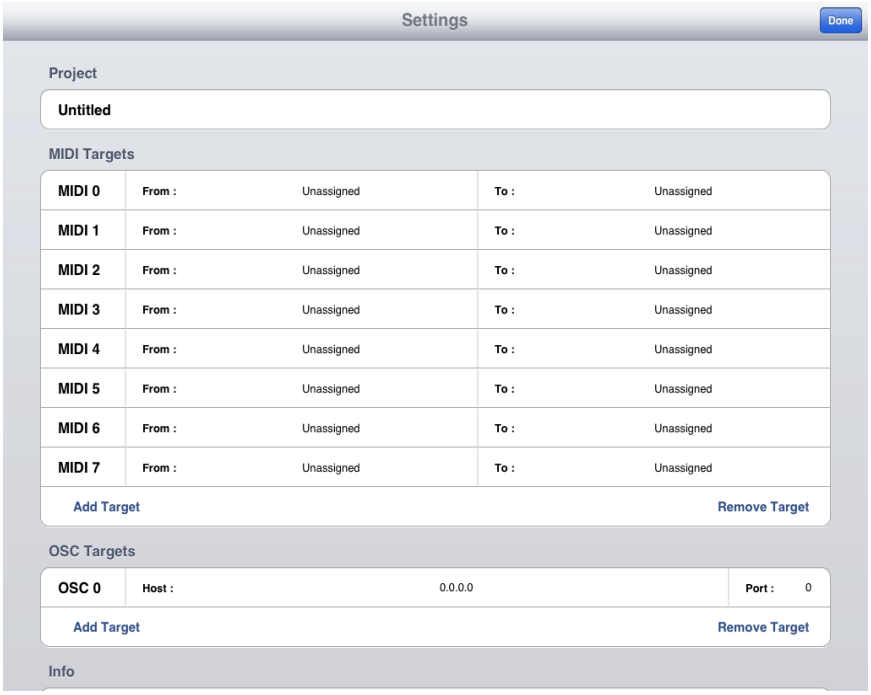
Touch the Input or Output of a Target to be changed on the MIDI screen of Lemur. You get a list of currently available Daemons on the network.

## 8.2 Lemur Daemon Targets Setup Example

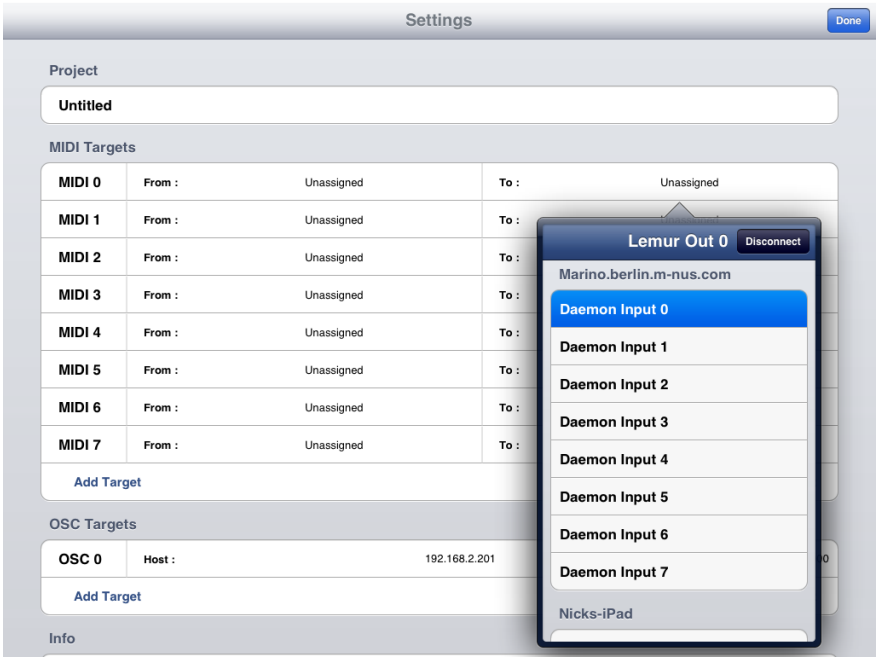
Let's go through the steps to get a MIDI connection up and running:

- On PC, start the Lemur Daemon by going to the Liine Start menu folder and clicking on the Lemur Daemon icon. On Mac, click on the Lemur Daemon icon in your Applications folder.

- Push the **Settings** button on Lemur (top right button).




- Now touch the **Outputs** button for the Lemur Daemon Target 0 and see the list of Daemons detected by Lemur.




- Choose one of the MIDI ports (see below for details) and touch the Connect button. You should now see the selected port appearing in the Targets list.
- You should now see the selected port appearing in the Targets list.
- Now let's see how the Lemur Daemon register these connections. Click on the Lemur Daemon's icon on your computer and choose **Setup Lemur Daemon**.
- A window appears, listing the ports connections we've just configured.

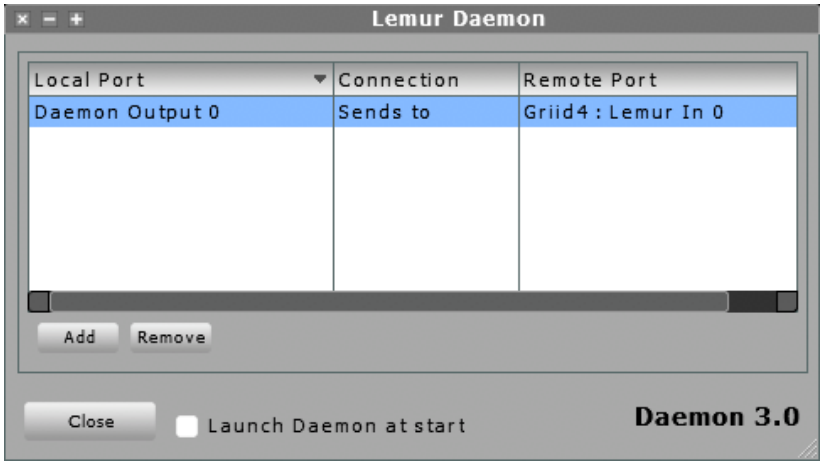
- You can now check if the MIDI connection works. Connect an application to the virtual MIDI port. Create a Fader and send some controller data to Target 0 or take a Pad Object and send some Note On messages.

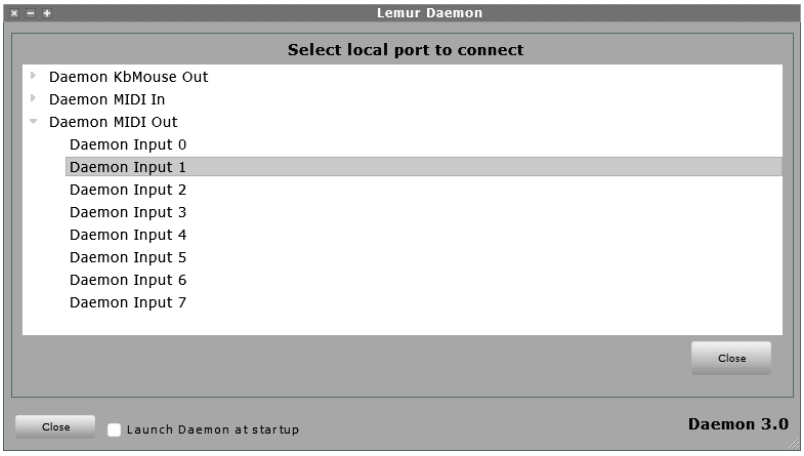
- Turn Lemur off.

- Turn it back on and have a look at the Lemur Daemon Targets configuration page on Lemur and at the Settings of the Daemon. Everything is still there and works as before.

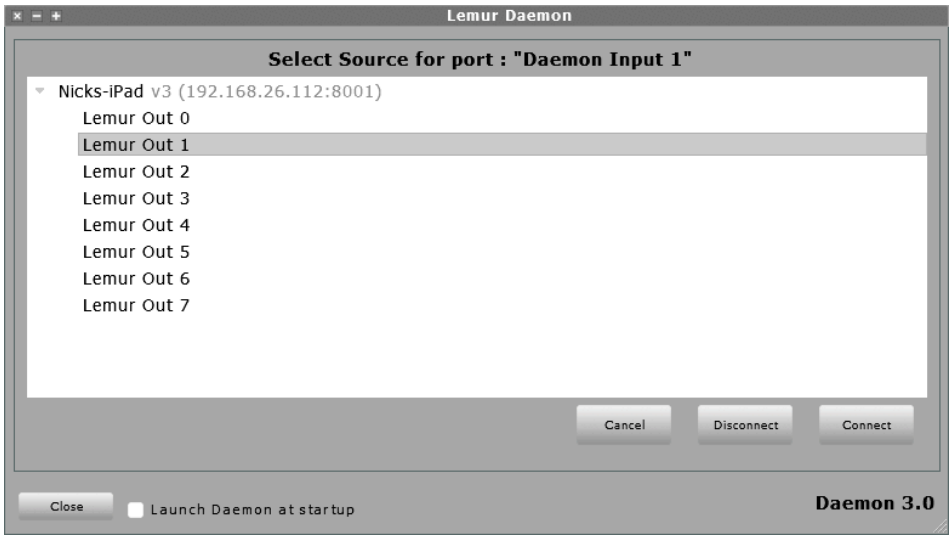
Connections to MIDI ports can also be configured from the Lemur Daemon:

- Click the **Add** button in the Lemur Daemon window. You are now presented with all the MIDI ports and the Mouse and Keyboard port of the machine. You can choose any In or Out port to connect to a Lemur. (Note that Mac Users will see there the Daemon Input and Daemon Output virtual ports that are automatically created by the Lemur Daemon at launch). Any port in the Daemon In section can be connected to the output of a Lemur. Any port in the Daemon Out section can be connected to the input of a Lemur.




- Double click on the local port you want to connect. Here, we want to receive all MIDI data coming from Lemur's Target 1 on our virtual port named Daemon Output 1. You can now browse the network and look for a Lemur to connect to. Choose Lemur Out 1 as a source, and double click the item, or hit the **Connect** button.



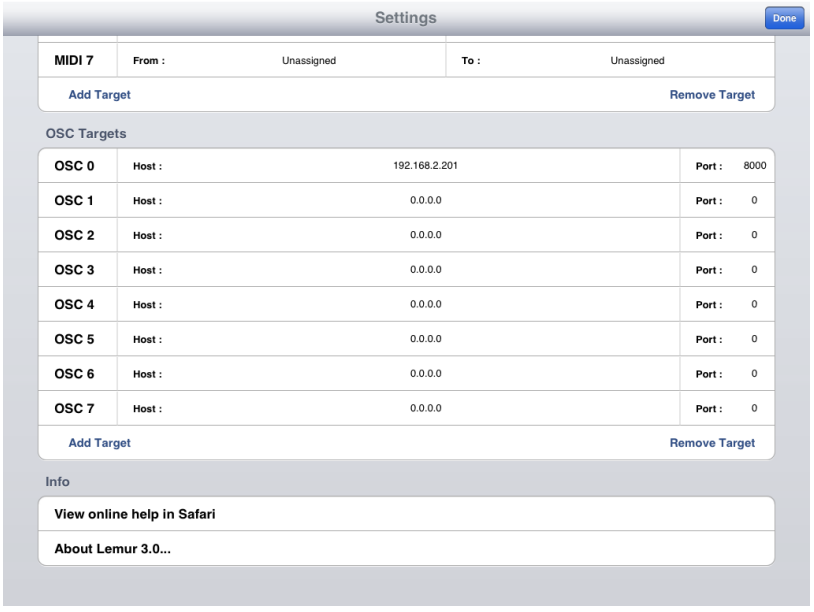

- Congratulations, you have successfully created a connection from the Lemur Daemon! Now if you quit and re-launch the Daemon, or reboot your computer, it will automatically redo the connection from IAC 1 to Lemur Out 0. Use this technique if you often swap computers connected to Lemur, and need a unique configuration for each of those.

## 8.3 OSC Targets

OSC Targets can be software applications on your local computer, somewhere on the network, or OSC-enabled hardware connected to the network. They are called Targets because they are what Lemur targets its messages at. Always remember that OSC is bi-directional and that OSC Targets can also control Lemur's Objects. Lemur is hard-coded to listen on Port 8000.

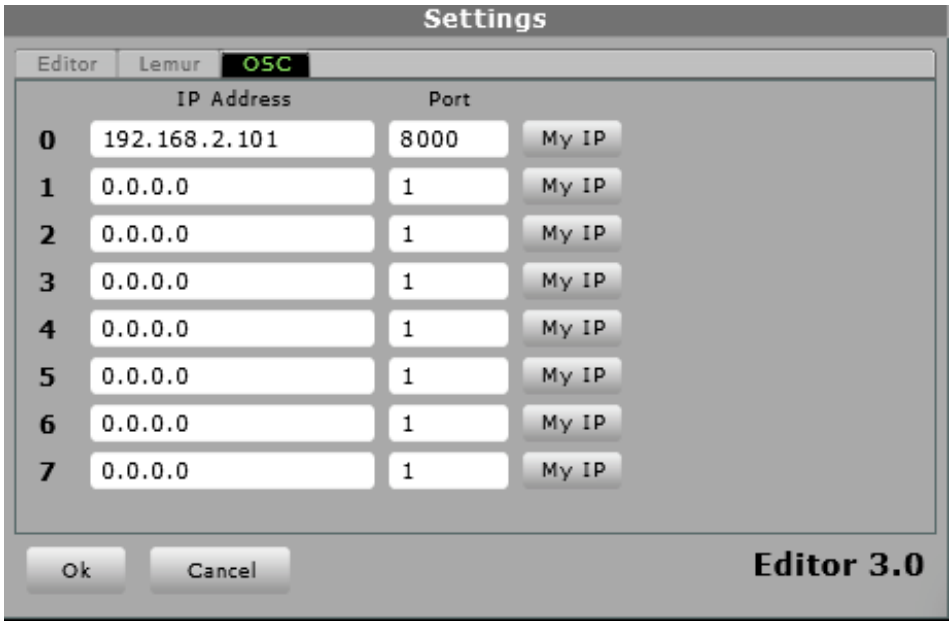
OSC Targets can be configured from Lemur's **OSC Targets** configuration screen, which is the second part of Lemur's Settings window. OSC Targets are also mirrored in the Lemur Editor 's settings window if a Lemur is connected. You can use both screens to set up your OSC Targets.

On Lemur, simply fill in the **IP Address** and **Port** number pad for each Target you want to communicate with.



The OSC settings are automatically saved into Lemur's memory. Whether you set up your OSC targets from the Lemur Editor or from Lemur, they will always be saved to memory and available at each reboot. The **Done** button closes the Settings window and returns you to your project.

If you're configuring your OSC Targets from the Lemur Editor's settings window, and if your Target is a piece of software running on the same computer as the Lemur Editor, click on the **My IP** button to automatically fill in the IP of your computer.




The OSC Targets' **Port** setting is dependent on the corresponding setting in the targeted application. Ports are something like channel selectors that allow multiple applications to share the same IP address without conflicts. The port number configuration is determined by the software you are using:

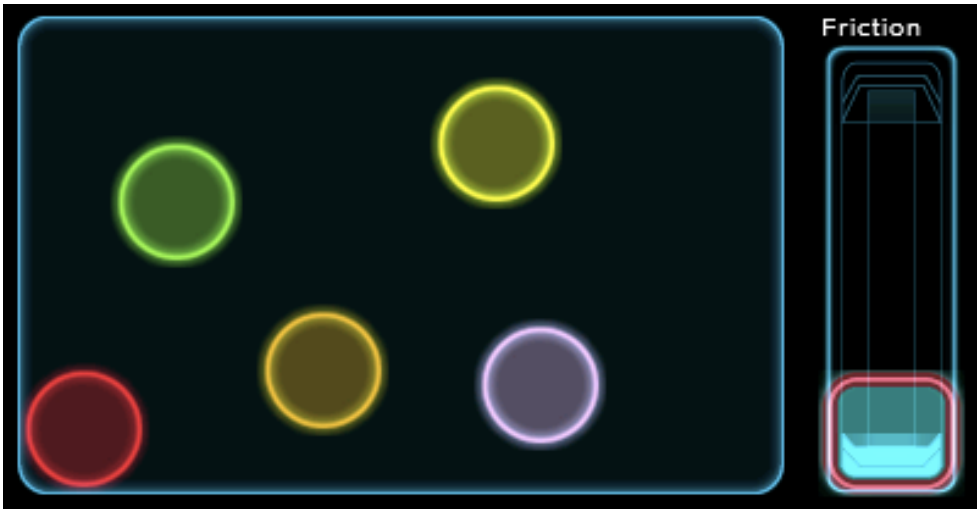
- **Port 8001 and 8002 should not be used**, because the Lemur Editor and Lemur Daemon use those ports to establish connections with Lemur. If you try to use another application while the Lemur Editor is open, the application will not be able to access ports 8001-8002. Similarly, if you launch the Lemur Editor while another application is using ports 8001-8002, it will be unable to connect to Lemur. Lemur is hard-coded to listen on Port 8000.
- For Cycling 74's **Max/MSP**, the port number can be anything. **8000** is the typical value used.
- For Native Instruments' **Reaktor**, the default port number is 10000.
- For other OSC-compatible software or hardware, consult its documentation for details on port settings.
- **Lemur** is hard-coded to listen on Port **8000**.

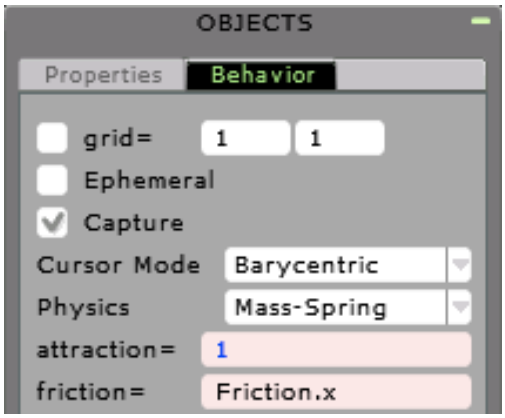
Once you have loaded a Project from the Lemur Editor onto Lemur (which is done automatically on Connect), or opened a Project from Lemur's internal memory, the OSC messages are handled by Lemur. This means you can close the Lemur Editor and let Lemur communicate with OSC Targets.

# Chapter 9 - Going further with the Lemur Editor

## 9.1 Control your Objects with your Objects

One of the many exciting features of Lemur is the ability to control the behavior of your Objects via other Objects in the Interface. Let's go through a basic example: create a Multiball and a Fader, and name the Fader Friction. Check the Fader's Label flag in its Properties tab to make things clearer.

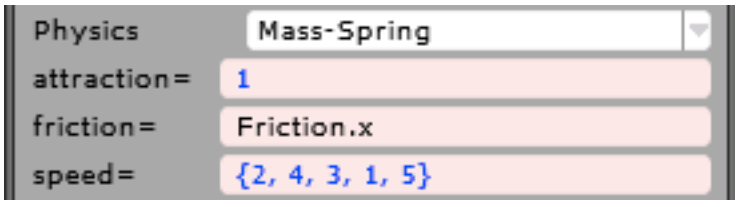



Now switch to the Behavior tab of the MultiBall Object and change the Physics parameter to **Mass-Spring**. Now type in **Friction.x** (the Fader's x Variable) as the value for friction:

Now you control the MultiBall's friction parameter via the Fader, it's as simple as that. Use any value of any of the Objects dwelling in Lemur Interface to control any of the parameters. This can lead to very complex and interactive Interfaces …

If you have sub-objects you can address their behavior properties individually by using a **vector** or array instead of a singleton. If you have five balls in a MultiBall Object and use a vector with 5 components to denote, say, **speed**, each ball will at its own speed value. Of course this also works with attraction, friction or any other parameter of Objects with multiple sub-objects.




You could have the Fader display the chosen friction value simply by checking its **value** flag. But sometimes you want to have the value displayed at a different place on your interface. You can even display values on top of Objects if you wish, and that's what we're going to do now. Create a Monitor Object and position it right in the middle of the MultiBall area.

Name the Object *Grip*, check the **Label**, **Transparent** and **value** flags. Fill in **Friction.x** for value because we want to display the friction parameter set via our Fader.


Now set the **Font** size of the Monitor to **13pt** and choose a nice, contrasting colour. You should get something like this:


If you move the Friction Fader, its value is displayed via the Grip Monitor. You can still use the MultiBall Object, as if the Monitor wouldn't exist, because the Monitor is transparent to touch. Don't mix this feature up with the **Transparent** flag of the Monitor which is only making the background of the Monitor invisible. You can always "reach through" a Monitor Object, even if its Transparent flag is unchecked. This can come handy if you want to hide Objects in your Interface but still use them with your fingers.

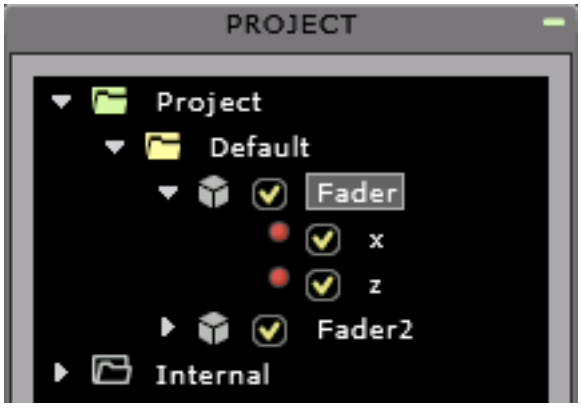
## 9.2 Making your own Object Variables

Object variable are declared by clicking on the **Create Expression** button below the Project panel, or by pressing Command/Control(Mac/PC) + Shift + E keyboard shortcut. They may be declared at different levels in the project by selection within the Project hierarchy. Object variables declared this way persist over time, unlike variable defined within scripts.

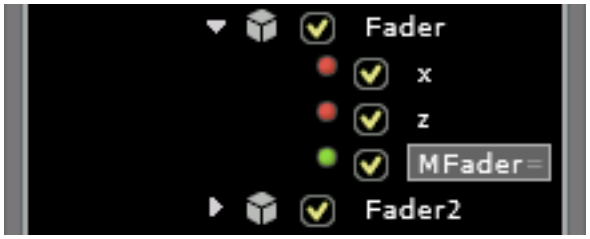
Now, let's say you have that special Reaktor patch with a custom parameter having an extra wide range. You want fine-grained control with a Fader in the lower range but you also need the possibility to control it over the complete range. How would you do that?

By creating two Fader Objects, and use second as a multiplier for the Value of the first one through a User-defined Variable that gets transmitted via OSC and MIDI.

So let's create two Faders, and select the first one in the Project panel:



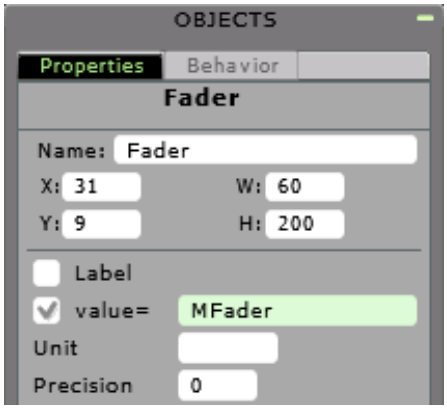

Now click on the **Create Expression** button below the Project panel.


Type in the name *MFader* into the dialog and click OK. You should see the Variable named *Mfader* in the tree, local to the Fader, and color-coded with a Green dot.

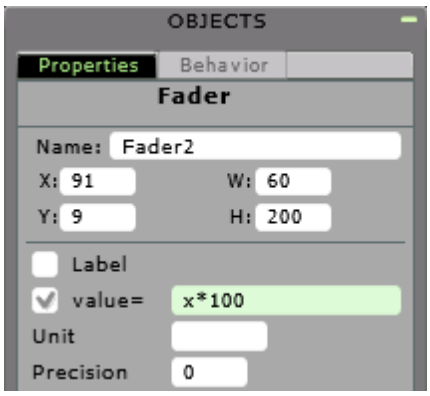
We want to scale the output of the first Fader with the value of the second. Do this by clicking on the fresh Variable in the Project panel. Type **x*(Fader2.x*100)** into the Script field of the Variable. Make sure that the checkbox in front of the local Variable is checked. This ensures that the Variable is actually transmitted via OSC and MIDI. It should look like this:


You might wonder why we addressed the Fader's value with x directly and not via Fader.x. This is possible because the Variable we defined is local to the Fader so that it "knows" the address of its Parent. If we had created the User-defined Variable at the project level, the full address (Fader.x) would be required.

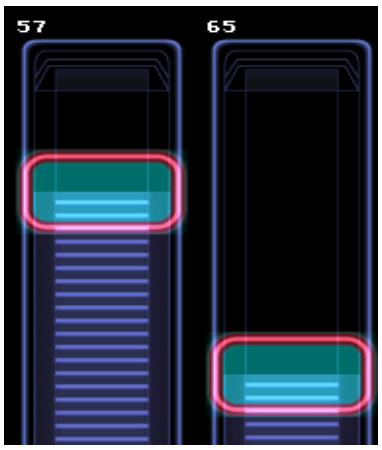



Of course you want to see the output of the Variable. Open the Properties tab of the first Fader and check the **value** flag. Type **MFader** into the **value** field. As we don't care about decimal places this time, set the **Precision** to 0.

For the second Fader also check the value checkbox and type x*100 into the value field. **Precision** can also be set to 0.



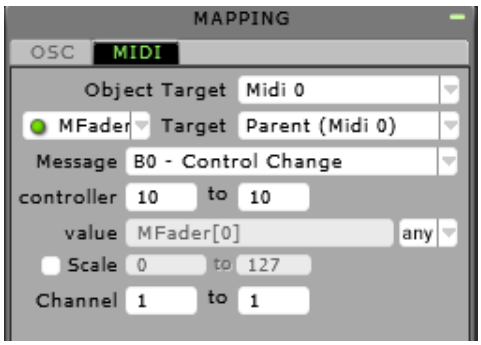

If you now move the Faders you will see the relevant values displayed above. Move the second Fader and the value of the Variable will change, because it gets multiplied by a factor between 0 and 100 depending on the value of the second Fader (Fader2.x*100).




Please note that the displayed values are not those actually produced by the Faders. They both still produce values between 0 and 1. Those values are combined in the calculation of the local Variable **MFader** that is displayed above the first Fader. The factor that's used for multiplication is displayed above Fader 2.

Now, what if you want to multiply with multiples of 10, only? The Fader has **grid** parameter for quantization of the output.

We want to transmit the local Variable *MFader* via MIDI. This is easily done via the MIDI panel of the first Fader. The Variables menu contains the Variable *MFader* and we can choose it to be transmitted to MIDI Target 0 (or any other Target we want).




Choose **Control Change** as Message and **Controller 10** as controller. Make sure that you uncheck the **Scale** checkbox, as we want the values to be transmitted as they are and not extrapolated to values between 0 and 127. Now Lemur will output values between 0 and 100 to MIDI Target 0 depending on the state of the two Faders.

## 9.3 Using Vector Variables

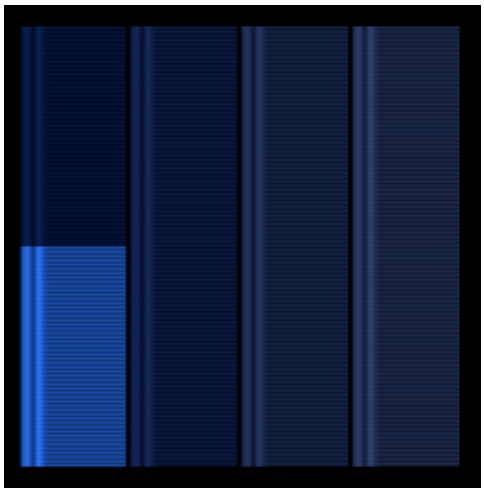
Vector variables are also declared by clicking on the **Create Expression** button below the Project panel, or by pressing Command/Control(Mac/PC) + Shift + E keyboard shortcut.

Lemur doesn't only know data structures containing single values. There are also vector/list variables that consist of more than one value. See chapter 13.1 for further information

Vector Variables are used by the Breakpoint, Leds, MultiSlider, MultiBall, Pads, and Switches Objects.

This can be convenient when you need controls for ADSR envelopes or other tightly interrelated groups of parameters. Let's use a MultiSlider to produce values for an envelope working somewhere in a Synthesizer and triggered with a Pads Object:

- Create a **Pads** Object and name it **Trig**, then create a MultiSlider and name it **Env**.
- Click the MultiSlider's Properties tab, and set the number of sliders to 4. Move and resize the MultiSlider so you can control each slider easily. Maybe something like this or a little bigger?



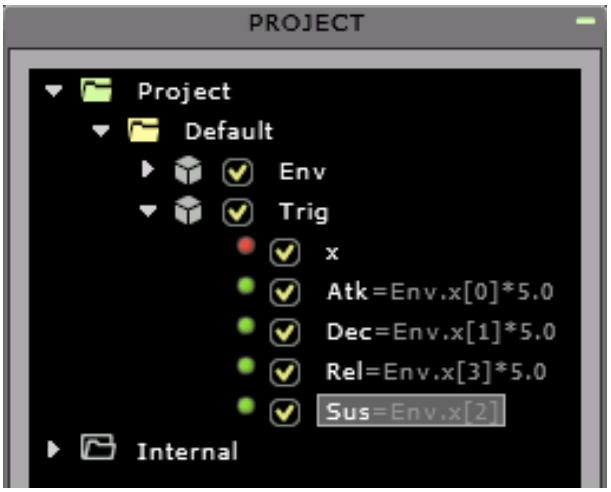
We're going to use the first slider for **Attack,** the second for **Decay**, the third for **Sustain**, and the fourth for **Release**. As with the x variables of all Lemur objects, the MultiSlider x variable ranges between 0 and 1. This is not really enough of a range for the time values of our envelope, so we need to scale these values. We'll also need to create three Expressions, one for each of the envelope's time values.

- Click on the Trig Object (Pads) in the Project panel so that its name is selected.
- Click the Create Expression button to create a new User-defined Variable.
- Name the expression **Atk** and press return
- In the corresponding script field enter the Expression shown below:

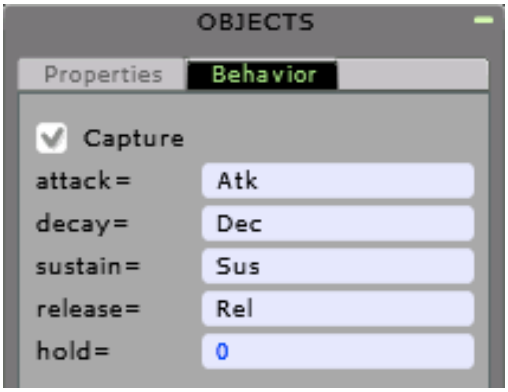

This means that the Atk Variable will use the value of the first slider (with the index of 0) of the MultiSlider we just created.

Create similar Expressions for **Dec** (decay), which will use **Env.x[1] * 5.0, Sus** (sustain), which will use **Env.x[2]**, and **Rel** (release), which will use **Env.x[3] * 5.0**. Note that we do not scale the value for Sustain, since this is just a value between 0 and 1.

When you're done, your Project panel should look like this:




The final step is to enter these Variable names in the Behavior Properties of the *Trig* Object. Enter **Atk** for Attack, **Dec** for Decay, **Sus** for Sustain, and **Rel** for Release as shown below. As the Variables are local to the *Trig* Object, you don't have to write out the complete addresses (which would be *Trig.Atk*, *Trig.Dec*, etc.).




Now you can test your envelope by hitting the Pads Object. You don't even need an attached Synthesizer to see what's going on. Dial in the envelope values via the MultiSlider and you will see how the changes of the sliders affect the brightness envelope of the pad.

## 9.4 Using Custom MIDI Messages

Cutom MIDI Messages are created by clicking the **Create Custom MIDI** icon below the Project panel, or by pressing Command/Control + Shift + M keyboard shortcut.

An extended Mapping panel with additional triggering modes and options is used to construct your Custom MIDI messages. Unlike simple MIDI messages set in the standard Mapping panel Custom MIDI messages are not bi-directional.

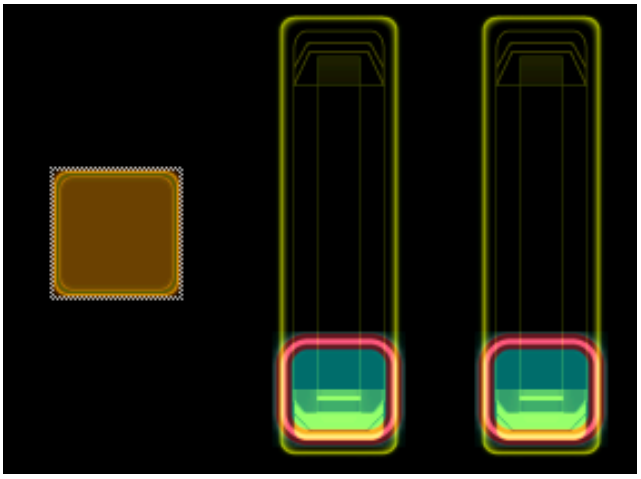
Custom MIDI messages provide the means to generate very complex MIDI data by entering Expressions for all the parameters of the chosen MIDI message. When using the standard Mapping panel's MIDI tab, you can only assign the selected Variable to one parameter of the message. With Custom MIDI messages you gain complete freedom to assign any value or expression to any one of the different parameters of the message. And remember that expressions can use all values of all Objects in your Project allowing the construction of very complex messages.

For a Note On message the parameters would be:

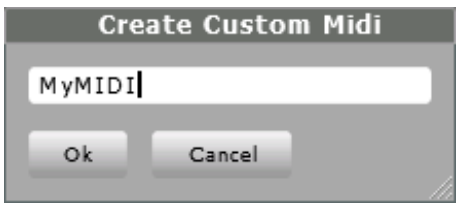
- Pitch
- Velocity
- MIDI Channel

You can even use an additional expression to control the triggering of the MIDI message. A **Trigger Mode** menu is also available for each of the Custom MIDI's parameters. Please read Chapter 7.3 for details on Trigger modes.

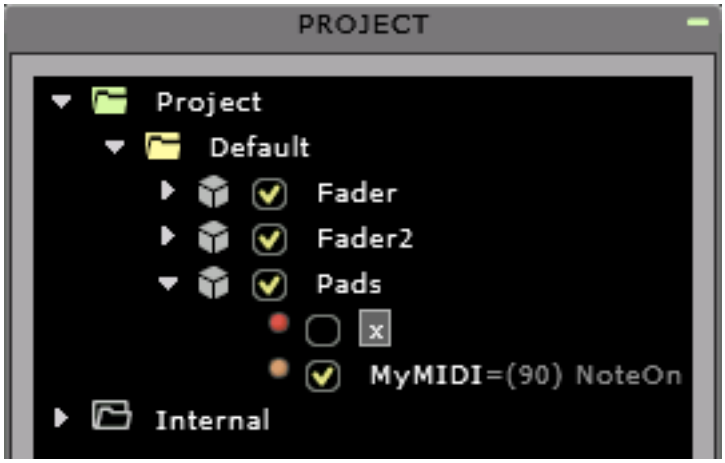
As a small example, let's create a Pads Object and two Faders, then select the Pads Object in the Project panel.

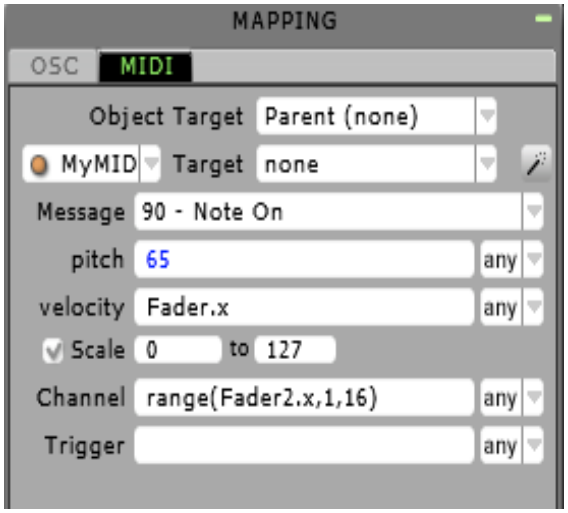


Next, click on the **Create Custom MIDI** icon  below the Project panel to create a customized MIDI message local to the Pads Object. and call it **MyMIDI**.



The Custom MIDI object will appear in the PROJECT panel




In the MAPPING panel for the Custom Midi message select an appropriate Target MIDI port.

For Message type we choose **Note On** from the Message menu.

We want a fixed pitch for the generated note messages and set the pitch field to **65**.

For **velocity** we use the value of the first Fader (Fader.x), scaled to the MIDI range of 0 to 127 via the **Scale** fields. Note that we've set the corresponding Trigger modes to "none" (change of value of the Fader doesn't trigger message sending).

The **Trigger** field is set to **x**, which is the state of the Pad. By setting the corresponding Trigger mode field to "**up**", the MIDI message is sent when x leaves 0, i.e when the Pad is pressed.

Now we can trigger notes with the Pad and control the velocity with the first Fader. As a playful addition we assign the other Fader's value to the MIDI channel. Here we use range function to get values between 1 and 16.

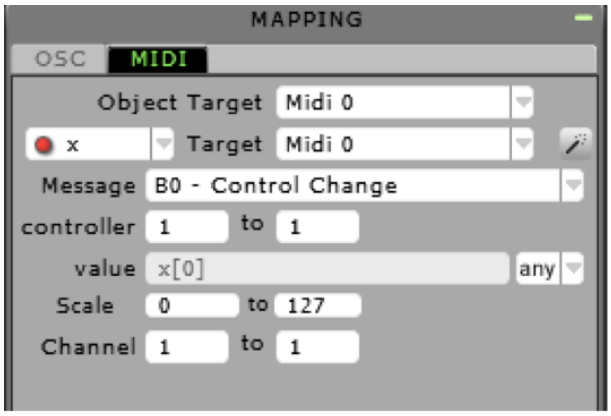
Now you can choose the MIDI Channel for the outgoing note messages. This might be an exotic application but it's fun – stack up a pile of sounds in multimode in your favourite synth and trigger different sounds by moving the channel fader on Lemur. Pay attention to the different Trigger Modes we use. Pitch, Velocity and Channel are set to None. Only a change in the Pads.x value triggers transmission of the MIDI message.

It's important to note that a Variable assigned to a Custom MIDI can output data but will not react to the corresponding incoming message. Custom MIDI messages are **not bidirectional**.

## 9.5 Bidirectional Control

Not only can the Variables of a Lemur Project output data, but they can also be controlled by external software or hardware via MIDI or OSC. This is what we refer to as **bidirectional control**.

If you want to control a Variable via MIDI, just set up the correct MIDI port for a Target in the Lemur Daemon Target settings. Then choose that MIDI Target for the Variable on the MIDI panel.




If you have, for example, a Fader's x value set up for transmission of MIDI controller 1 on MIDI Target 0, it will also be controllable via incoming controller 1 data on the input port chosen for that MIDI Target.

Similarly, you can control all values via OSC. The different values are controlled via their OSC addresses you already know.


For controlling a Fader via OSC, just set up the OSC Target for the x value on the OSC panel and produce the OSC messages on the Target side.

In the Fader's case you would send values between 0 and 1 to the OSC address /Fader/x to the current IP and port 8000 of Lemur. If you've set up a custom OSC range for Fader.x, it will respond to values inside that range.

Please make sure that you think about the **hierarchy** of Objects on Lemur. If the Fader is inside of a Container the address would be **/Container/Fader/x**.

Note that it's also possible to switch Interfaces via OSC. Just send an OSC message in the format "/interface *InterfaceName*" to Lemur's IP (port 8000). In our case this would look like "/interface 1 Demo" for switching to the first one. For names that include spaces there is an additional consideration to be made. The OSC software has to send the name as a single string. This might involve putting it in quotes in the software. This is the case for Max/MSP.

Control of Lemur Objects can also be achieved by using multi-line scripts set up to execute on either 'On MIDI' or 'On OSC'. The relevant OSC or MIDI data is then available as a Vector called either OSC_ARGS or MIDI_ARGS for use within Lemur. Individual bytes of this data are obtained via array notation ie. OSC_ARGS[0], OSC_ARGS[1] etc

For more about working with OSC, MIDI and bi-directional control please have a look at the extensive workshop chapters we offer on our website. There you'll learn everything about how Lemur loves Max/MSP, Reaktor and other OSC enabled applications.

## 9.6 Defining and Using Functions

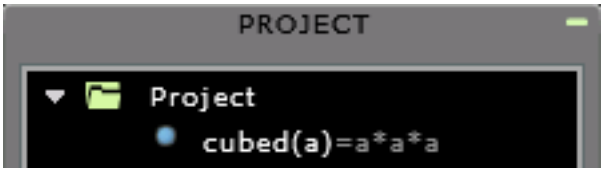
In addition to built-in functions and operators, you may define your own mathematical **functions** to use in User-defined Variables and expressions. Here's how to do it:

Select your projects folder in the Project panel, and click the Create Expression button to create a new Expression.

The Expression is created at the level of the hierarchy you have currently selected. If you have an Object selected it will be an Expression local to that Object. If you have the Pojects folder selected the Expression will go there and be available for use anywhere in the Project.

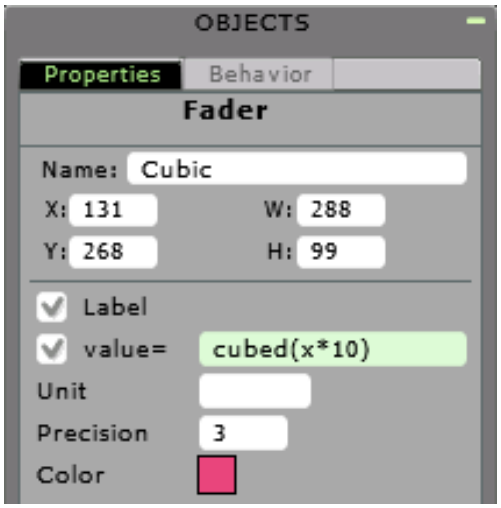
- Type the name of the function and its arguments in parentheses. For example, to define a function named *cubed* that takes one argument, you would type:


- Click OK and go to the Script panel to type in the definition of your function as an expression. You can reference other variables if needed. The example below just multiplies the function argument a by itself twice to raise the input to the third power.

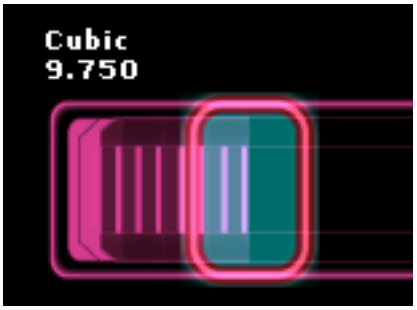




The newly declared function appears in the Project panel with a Blue dot to indicate it is a function.

How to use the new function? Let's try it out on a simple Fader: create a Fader Object, make it horizontal for a change (by stretching it horizontally) name it **Cubic** and check both its **label** and **value** checkboxes in its properties tab.




Set the value field to **cubed(x*10).** You are now using the Function we just created to raise the x value (which is multiplied by 10 beforehand) to the power of three. The variable argument of the Function is replaced by the Expression x*10.




Move the Fader and you will see that it displays an exponential range from 0 to 1000.

# Chapter 10 - Introducing Multi-line Scripts

Multi-line scripting is one of the most powerful features of the Lemur Editor. It can be used, in conjunction with a set of powerful built-in functions and operators, to instruct Lemur to perform specific tasks at specific times. From simple actions such as changing a MultiSlider color when a switch is pressed, to complex actions such as commanding a Multiball's balls to follow set trajectories... possibilities are truly endless.

Scripting can be quite easy, but it can also be developed into very complex systems. This section is not designed to teach you everything you need to know. It is recommended that you should learn some fundamental programming principles to fully utilise Lemur's scripting possibilities.

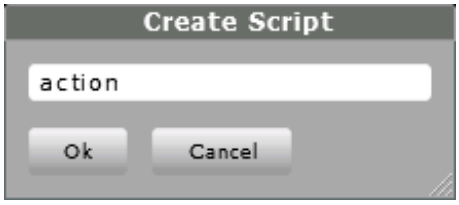
## 10.1 Creating a Script

Multi-line Scripts are created by clicking the **Create Script** icon below the Project panel, or by pressing Command/Control + Shift + S keyboard shortcut.

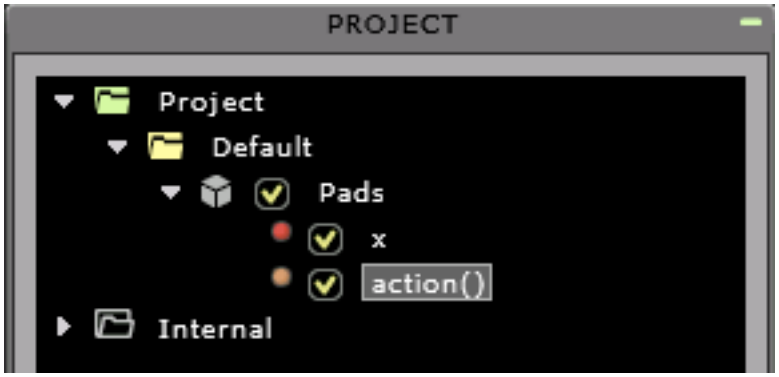
To create a new script, the first step is to select the element you want to associate the script with in the Project panel. Scripts may be be associated globally across the entire project, or a specific interface, or locally to a particular container or object,

Select the required element in the Project panel and click on the **Create Script** icon.


A dialog window will appear, asking you to name the script appropriately. Avoid names taken by internal functions, such as *set*, *firstof*, *replace* and so on, as your script would then temporarily become a substitute for that particular function. You can refer to the Internal folder in the Project panel for a quick look at the built-in functions list.




When created, your script will appear in the Project panel, preceded by an orange dot, and followed by brackets containing the script's arguments, if any, representing the values passed to the script at the time of call.

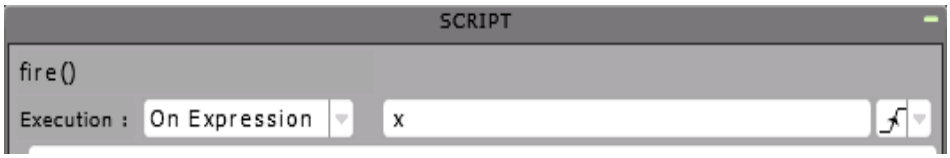


## 10.2 Script Execution

All Lemur scripts are executed for every frame displayed, 60 times a second, roughly every 16msec.

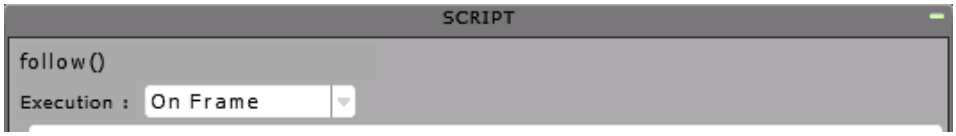
Script Execution occurs according to selections set in the Execution toolbar of the Script panel. Before beginning to encode a multi-line script, it is essential to define precisely when Lemur should perform the desired task and set the script's execution mode accordingly.

To this end, a choice of 6 different **Execution** modes is available to cover the range of possible situations:




- **On Expression**: the script's execution is triggered when the value of the variable or expression entered in the adjacent field changes in the manner specified by the **Trigger menu**. A valid value must be entered here in order for the script to execute.

- The Trigger menu offers several options: the script can be triggered whenever the variable or expression changes, when it rises from 0, when it reaches 0, when it reaches or rises from 0, each time it increases above its previous value, or each time it decreases below its previous value. This is the same Trigger menu that we find in the MIDI or OSC tabs when we assign OSC or MIDI messages commands to objects.

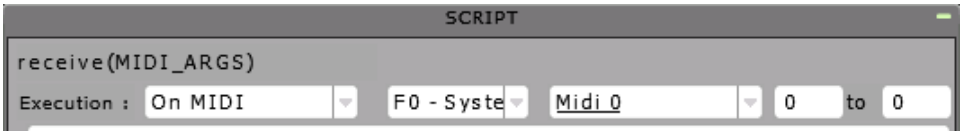



- **On Frame**: the script's execution is triggered each Lemur frame, that is, each time Lemur scans an interface to recalculate the states of the objects on display. The rate at which Lemur scans an interface is inversely proportional to the amount of elements to be calculated. With a maximum rate of 60 frames per second for a blank interface, this rate slows down with increasingly complex Interfaces.

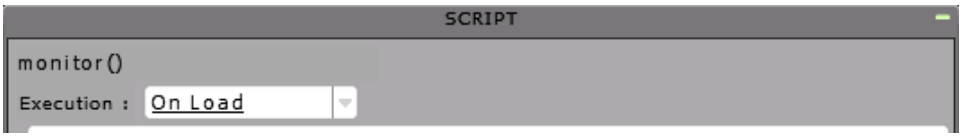

- **On OSC**: the script's execution is triggered upon reception of the OSC message displayed in the adjacent field, the script's arguments (OSC_ARGS).

- By default, this address is set to the script's name (*Container/Fader/Scriptname*), but can be customized by ticking the **Custom Address** checkbox.

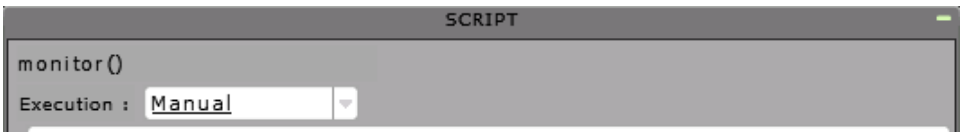
- The OSC message's arguments are then available inside the script with the OSC_ARGS array. Individual values are accessed OSC_ARGS[0], OSC_ARGS[1] etc, Please consult your device or application documentation for specific details on argument format




- **On MIDI**: the script's execution is triggered upon reception of the MIDI message (MIDI_ARGS) defined in the adjacent fields: MIDI Message type, MIDI Target, MIDI channel.
- The MIDI Message selection defines what is received and the particular arguments of the selected MIDI message are then available inside the script with the MIDI_ARGS array. See Chapter 14 MIDI Mapping Reference for further details of individual messages.
- In the case of a SYSEX message, this array can be of varying length, to a maximum of 256 bytes, and is typically terminated by the F7 character. You may also use sizeof() to check the length of known messages., and subarray() to access subsets within it.
- The first SYSEX byte is MIDI_ARGS[0], the second is MIDI_ARGS[1], etc. A typical System Exclusive message will consist of Manufacturer, Device and Command numbers that are specific to each device, followed by data specially formatted for that particular device. Please consult your devices documentation for further details.




- **On Load**: the script's execution is triggered when the project is loaded on Lemur.
- **Note: On Load** scripts relying on external numerical constants or arrays of constant numerical data should operate correctly, however there can be issues when depending on more complex variables, or calling other scripts, due to variable evaluation order.




- **Manual**: the script's execution is triggered when called up in another script or expression.
- **Arguments** may be declared in the script's name and will be available within the script .

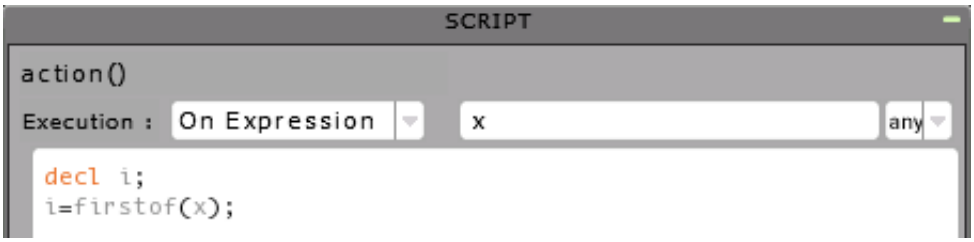
**Remember**, it is essential to define when Lemur should perform the desired task and set the script's execution mode or else it may not execute the way you intend or at all.

## 10.3 Script Variables

The Multi-line Script panel allows you to declare (create) variables specific to your Script, allowing you to store local values while the script is in use. These "declared" Variables are only visible inside the Script panel (they do not appear in the Project panel), and expire when the script is not in use.

In addition to these Script Variables, all the built-in or User-defined Object Variables alike can be recalled in your code and combined with functions and operators to create your script.

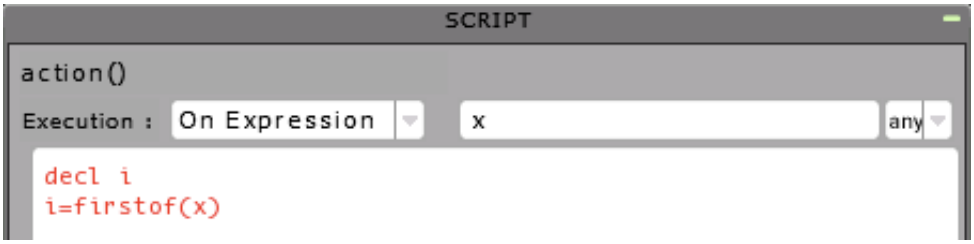
To declare a script Variable, use the statement **decl** :

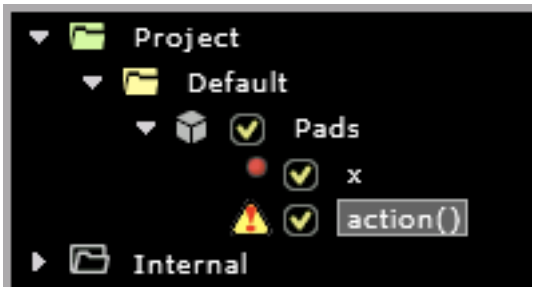



decl will appear in orange once the line is correctly completed with a semi-colon;

A script Variable can be declared and equated to an expression in a single line too:


**Note that the Multi-line Script syntax requires that each line of code ends with a semicolon (;).** If any error, such as forgetting a semicolon, is present in the script, the lines of code will become red, and a "caution" triangle appears in the Project panel next to the script.



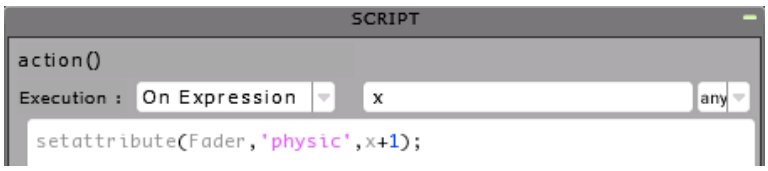
## 10.4 Attributes

The Objects' Attributes include all those Object parameters which are editable in real-time, either through instructions in a script with the function **setattribute(object,'name', value),** or directly through an OSC command in the form **/Container/Fader @grid 1** (activate the Fader's Grid parameter), **/Container/Fader @rect 300 100 60 200** (change the object's position to X=300 and Y=100, and the object's dimension to W=60 and H=200), or **/Container/Text @content "Hello !!!"** (change the displayed text). The Lemurs iPad OSC listening port is 8000.

Let's look at the Fader's attributes for instance. The table below presents a list of all its attributes, giving the possible values these attributes can take, together with a brief description. A list of all the attributes for each Object can be found in the Object Reference Chapter (Chapter 12).

| **The Fader Object's attributes** | | |
|---|---|---|
| Attribute | Value | Parameter |
| capture | 0 or 1 | capture off/on |
| color | integer* | color of the object |
| cursor | 0 to 3 | cursor mode |
| grid | 0 or 1 | grid off/on |
| grid_steps | 1 to 33 | number of grid steps |
| label | 0 or 1 | label off/on |
| physic | 0 to 2 | physics mode |
| precision | 0 to 6 | decimal places of displayed value |
| rect | {X,Y,W,H} | object's position and dimensions |
| unit | text | unit of displayed value |
| value | 0 or 1 | value off/on |
| zoom | 1 to 50 | zoom in object |

Imagine, for instance, that you would like to change your Fader's physic mode from Interpolate to Mass-Spring while you're playing, by pressing a Switch on your interface. Your script, created locally to your Switch object would consist of the following line:




With this line, we instruct Lemur to set the 'physic' attribute of Fader Object to the value x+1, x being the x Variable of the switch since the Script was created local to the Switches Object. This means that the value of the 'physic' attribute will be equal to 1 when the Switch is off (x=0), and to 2 when the switch is on (x=1). Note that the Execution mode is set to "On expression x", with Trigger mode "any", so that the script is executed each time the switch is pressed on or off.

By the way, it is useful to know that when you are working in the Lemur Editor, the function **getattributelist(object)** can be used to return a list of attributes for the chosen object in a Monitor object on Lemur

## 10.5 Built-in Functions and Operators

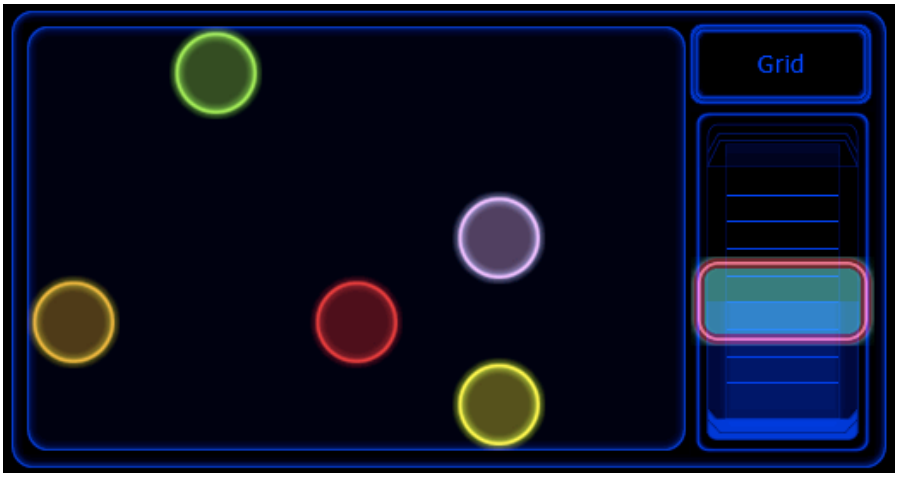
The Lemur Editor features a wide range of handy built-in functions, all listed in the Project panel's **Internal** folder, whose purposes are to make your life easier when defining a single-line expression for a User-defined Variable or when encoding a multi-line script. Some of these functions, such as the **firstof(x)** function (returns the position of the first non-null item in a vector) can be used in both single-line expressions and multi-line scripts, while others, like the **setattribute(object,'name', value)** mentioned previously, are specific to multi-line scripting.

In addition, the multi-line Script panel also understands a wide range of operators such as the **arithmetic**, **comparison** and **logic** operators, but also operators denoting assignment or bitwise operations. Basically assignment operators allow the same variable name to contain different values at different times during the script execution, while bitwise operators allow operations on array patterns at the level of their individual elements.

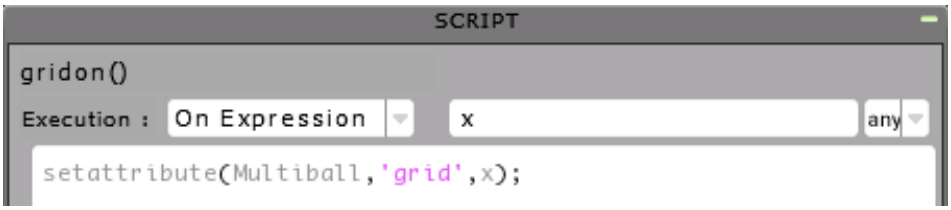
Please refer to the Parser Reference on Chapter 13 for descriptions of all built-in functions and operators.

## 10.6 Examples

We have a Multiball, together with a CustomButton and a Fader. We'd like to be able to switch the Multiball's grid on or off with the CustomButton, and set the Multiball's number of balls with the Fader.

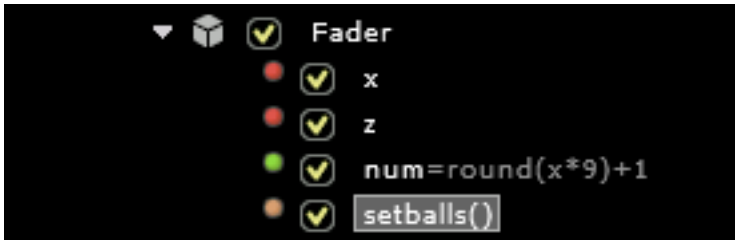


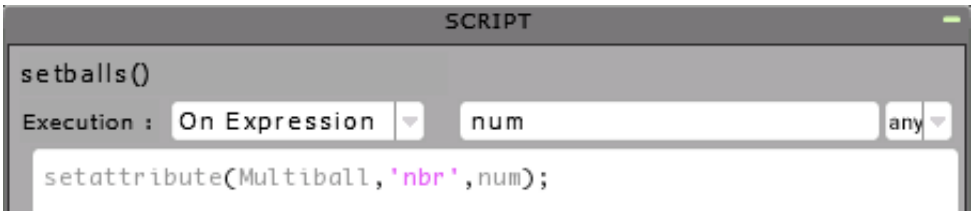

Let's take care of the Multiball's grid first. In parser terms, switching a Multiball's grid on or off means setting its 'grid' attribute to 0 (off) or 1 (on). For the script to be executed when the CustomButton is pressed, we need to associate the Script with the CustomButton's x variable. To do this, we'll first select the CustomButton in the Project Browser and click on the Script icon. We'll name this script *gridon* and set its execution mode to **On Expression**. We then fill the field adjacent to On Expression with x, and keep the trigger menu to any: the script will be executed whenever the CustomButton's x variable (its on or off state) changes. As far as the code is concerned, we'll use the **setattribute(object,name,value)** function, where *object* will be Multiball, *name* the attribute's name 'grid' and *value* will be x, the CustomButton's state:




Let's take care of the Fader next. Again we'll use the **setattribute(object,name,value)** function, but this time the attribute's name will be 'nbr': the number of balls.

To make this practical, we'll use a 10 steps grid for the Fader, so that each step upward adds a ball, and each step downward takes one off. We can translate this into a number with an expression local to the Fader: select the Fader in the Project Browser, click on the Create expression icon, name it num for instance and equate it to the expression **round(x*9)+1**. That way num will take integer values between 1 and 10, values that we'll use in our script with the **setattribute(object,name,value)** function to set the number of balls. We'll associate the script with the Fader object, to be executed **On Expression num** (each time the Fader modifies the number of balls) as follows:




An alternative would be to replace the Fader object with two pads: one for incrementing the number of balls, the other for decrementing. To make it more interesting, we can also add a MultiSlider for setting the Multiball's color, and a Fader to zoom in the Multiball for increased precision.

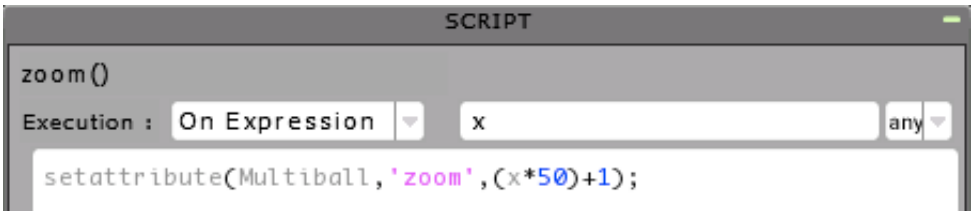
We'll begin with the + Pad together with a script we'll call *increment*. Again the execution mode will be **On Expression x**, but this time we'll set the corresponding trigger menu so that the script gets executed only when x goes from 0 to positive (we only want the script to be triggered when the pad is pressed, not when it's released).

Our script begins by declaring a new local variable b, which will correspond to the current number of balls, i.e, the value that we'll increment. We can use the function **getattribute(object,name)** to get this info. We'll then use the arithmetic operator **++** (which simply means "increment") to increment our b variable before using it with the **setattribute(Multiball,'nbr',b)** function.


Similarly, we'll create a second pad together with a script called *decrement*. The difference here will be the use of the arithmetic operator **--** (which means "decrement").


The MultiSlider object will be used to set the Multiball's color, with three sliders for mixing the primary colors (Red, Green and Blue) according to the RGB additive color model. In the script, we'll use the **setattribute(object,name,value)** function to set the Multiball's 'color' attribute with a value computed by the **RGB(r,g,b)** function, which will convert the 3 sliders (each between 0 and 1) into a single color value that the parser understands. Note that the value of the color attribute can range from 0 to 8355711: ((R*127 x 2^16) + (G*127 x 2^8) + B*127), where R, G and B are values between 0.00 and 1.00.


Finally we'll create a script for the Fader object in order to control the Multiball's **'zoom'** attribute. Since the 'zoom' attribute takes values between 1 and 50, we'll need to scale the Fader's x variable by multiplying it by 50 and adding 1:



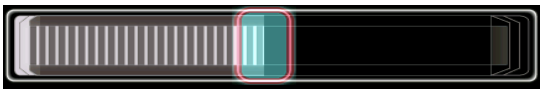

The **setattribute(object,name,value)** can also be used to change the dimensions of an object and/or move it around by changing its **'rect'** attribute. The'rect' attribute takes values in the form {X,Y,W,H}, where, as in the Lemur Editor's Properties tab, X is the horizontal position of the object, Y the vertical position, W the width of the object and H the height.

In this last example, we've got large Container called *Containerbig* filled with Multiball objects, and in order to gain space, we've enclosed it within a smaller Container called *Containersmall* and add a Fader to scroll up and down the Multiballs.

The script, local to the Fader object, and whose execution is initiated by any change in its x variable (each time the Fader moves up or down), uses the value of x to multiply the Y element of *Containerbig*'s rect attribute, thereby changing its vertical position within the smaller container:


Lastly, we can add an horizontal Fader object called Fader2 whose width is recalculated as the Fader is displaced. Maybe not the kind of Fader you'd like to use for controlling your master volume in a Live situation, but interesting nonetheless. Try changing the Fader's Physics to Mass-Spring to make it more fun.

Here we begin the script by declaring a new script Variable *temp*, then use the function **getobjectrect(object)** to obtain the existing value of the **rect** attribute in the form {X,Y,W,H}. In the third line of code, the Fader's new position modifies the third element of the vector tmp (tmp[2]), which corresponds to the width of the object. Finally the freshly calculated variable *tmp* is injected in the **setobjectrect(object,rect[])** to modify the width of the object.




## 10.7 Color Coding in Scripts

The Lemur Editor performs syntax coloring checks of your scripts and expression. The color of the coding will show if there is a syntax error or not. If there is no syntax error, different colors are assigned to different programming objects.

**Scripts**

Red – Error: The script contains syntax errors and will not compile

Orange – Commands; decl statement in declaration of variables in scripts. If/then/else and do/while etc commands,

Green – Comments: Starting with '//' and ending with a line feed/carriage return, multine-line coments starting with /* and ending with */

Blue – Literals and Strings, recognized as constants at compile-time and therefore will not be reevaluated

Purple – Character strings for attribute names

Grey – Commands, Call of Scripts, Variables

**Expressions**

Red – The expression contains syntax errors

Black – The expression is Ok

Blue – Literals and Strings, recognized as constants at compile-time and therefore will not be reevaluated

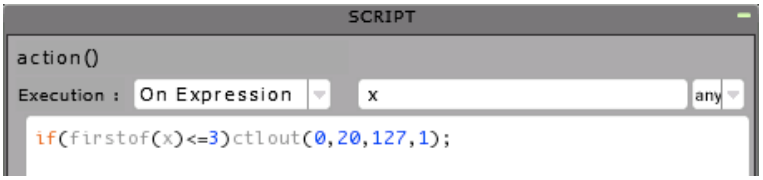
# Chapter 11 - Advanced Scripting

## 11.1 Conditional statements

- **if**

When the parser finds an if in the Script panel, it expects a boolean condition, that is, a logical proposition having one of two values: **true** (non-zero) or **false** (zero). If the condition is true, the parser executes the single line or block of code immediately after the condition, which, in the case of a block of code, will be enclosed within braces {}. If the condition is false, the statement in the single line or block of code is ignored.

The **if** structure in its simplest form:




(if **firstof(x)** is smaller or equal to 3, Lemur sends a Control Change MIDI message on Target 0 for controller 20, with a value of 127, on MIDI channel 1. )

Several instructions can follow the boolean condition provided the block of code is enclosed within braces:


(if **firstof(x)** is smaller or equal to 3, Lemur sends two Control Change MIDI messages: one for controller 20, one for controller 21. )

- **else**

It's also possible to make use of the **else** statement. If the condition is false, the execution jumps to the instruction immediately after else :


(if **firstof(x)** is smaller or equal to 3, Lemur sends a Control Change MIDI message for controller 20. If firstof(x) is greater than 3, it sends a CC for controller 23. )

The else statement can also be a block of code contained within braces:

SCRIPT

action()

Execution : On Expression x any

```
if(firstof(x)<=3)
{
    ctlout(0,20,127,1);
    ctlout(0,21,127,1);
}
else
{
    ctlout(0,22,127,1);
    ctlout(0,23,127,1);
}
```

(if **firstof(x)** is smaller or equal to 3, Lemur sends two CC MIDI messages for controller 20 and 21. Else if it's greater than 3, Lemur sends two CC MIDI messages for controller 22 and 23. )

- **else if**

By using **else if**, it is possible to combine several conditions. Only the statement(s) following the first condition that is found to be true will be executed, and all other statement(s) will be skipped.

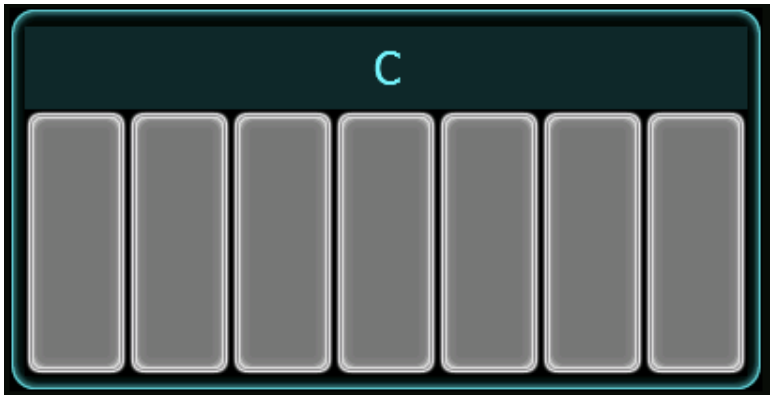
Let's go through an example. We're using a row of pads as a rudimentary keyboard, and we'd like a Text object to display the notes' names as we're playing: *A,B,C...* For this we'll use a script local to the Pads object, to be executed "on x" (each time a pad is pressed), and consisting of several else **if** statements. The **firstof(x)** function is used to return a value corresponding to the position of the pad being pressed, and the **setattribute()** function sets the Text Object's 'content' attribute according to the value of the firstof(x) function.

SCRIPT

action()

Execution : On Expression x any

```
decl text, val;
val = firstof(x);
if (val==0) text = 'C';
else if (val==1) text = 'D';
else if (val==2) text = 'E';
else if (val==3) text = 'F';
else if (val==4) text = 'G';
else if (val==5) text = 'A';
else if (val==6) text = 'B';
setattribute(Text, 'content', text);
```



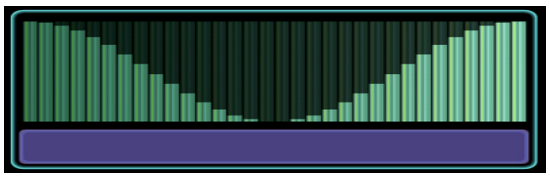
## 11.2 Loops

Note: The time variable is evaluated at each frame, which typically happens 60 times per second, and less with very heavy templates or lots of scripts. So a loop in a script that waits for time to cross a certain mark will never exit, since time is "frozen" in a way. Actually there's a watchdog that makes sure script eventually exit and the app is not stuck indefinitely)

- **for**

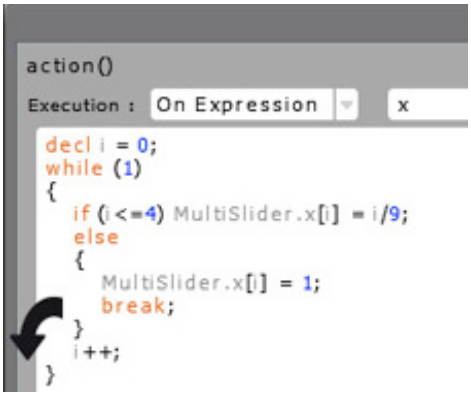
The **for** loop is an **iteration** statement which allows code to be repeatedly executed a specified number of times. The type of for loop used in the Script panel is characterized by three control expressions separated by semicolons: the **initializer** expression, the **loop test** expression and the **counting** expression. The initializer sets up the initial value of the variable within the looped block of code. The loop test expression is evaluated at the beginning of each iteration through the loop, and determines when the loop should exit. Finally, the counting expression is responsible for altering the loop variable.

Let's go through an example. In the script below, a **for** loop is used to set up the value of a MultiSlider's individual sliders on the basis of a cosine function. The initializer i=0 sets up the the initial value of the variable i to 0, which means that the first time, the loop block is executed for MultiSlider.x[0], setting up the value of the first slider. The variable **i** is then incremented by the counting expression **++** to i=1, and the code within braces repeated for MultiSlider.x[1] . The variable will be incremented and the code repeated as long as **i< sizeof(MultiSlider.x)**, which means for each slider up to MultiSlider.x[31].

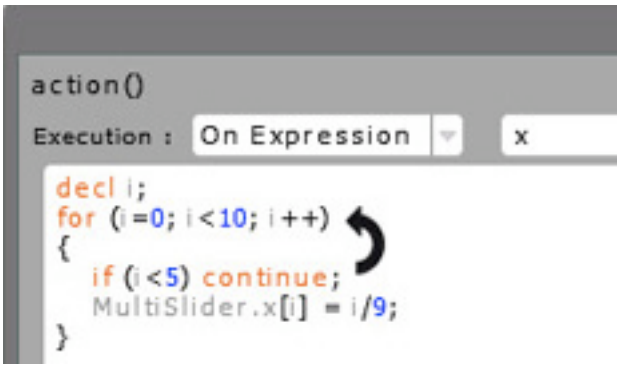
- **break**

A break statement interrupts the execution of the current for/ while loop and jumps to the next statement.



- **continue**

The **continue** statement is used to **skip** the remainder of the loop body and continue with the next iteration of the loop. The effect is to prematurely terminate the innermost loop body and then resume as normal with the next iteration. If the iteration is the last one in the loop, the effect is to terminate the entire loop early.

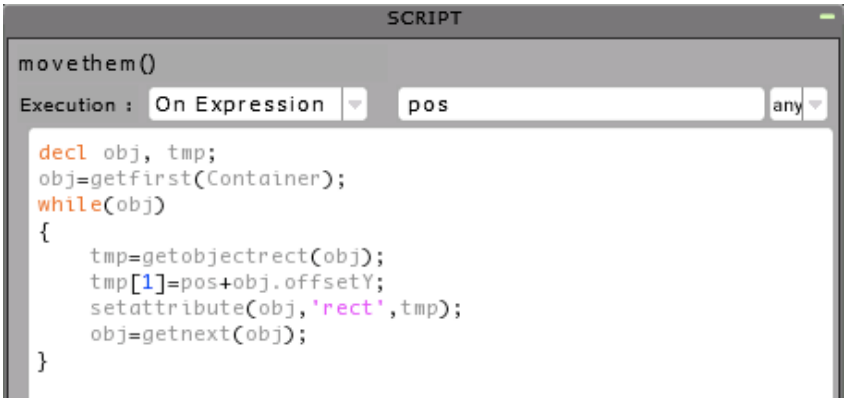



- **while**

A **while** loop allows code to be executed repeatedly based on a given boolean condition. The while structure consists of a condition and a block of code. The condition is first evaluated, and if the condition is true, the block of code within braces is then executed. This repeats until the condition becomes false.

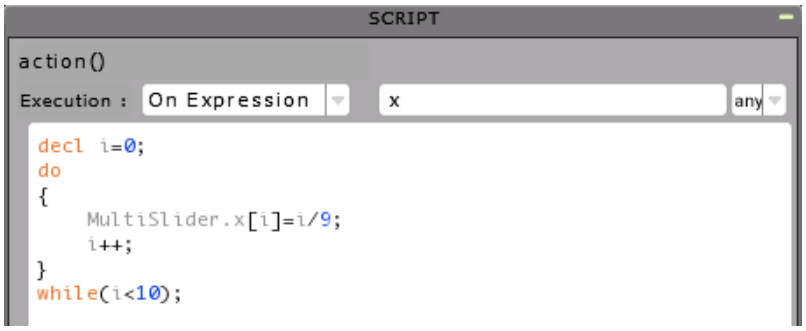
Let's go through another example. In the script below, the **getfirst(Container)** function is used to select the first object within the Container object's hierarchy. Since there're several objects in the Container, the boolean condition (obj) will necessarily be true, and therefore the statements within braces executed a first time. The **getobjectrect(obj)** function will extract the position and dimensions of the Container's first object, and its vertical position (tmp[1]) recalculated on the basis of the Fader's position, and object's offset (the offset variable defines the vertical position within the Container, and is manually entered for each object).

Once the **setattribute(obj,'rect',tmp)** function has changed the object's vertical position, the **obj** variable is replaced with the next object down the Container's hierarchy (this is done with the **getnext(obj) function**), and the statements within braces repeated. Those statements will be repeated for each object within the Container, that is, as long as the getnext(obj) returns a non-zero value and keep the boolean condition (obj) true.



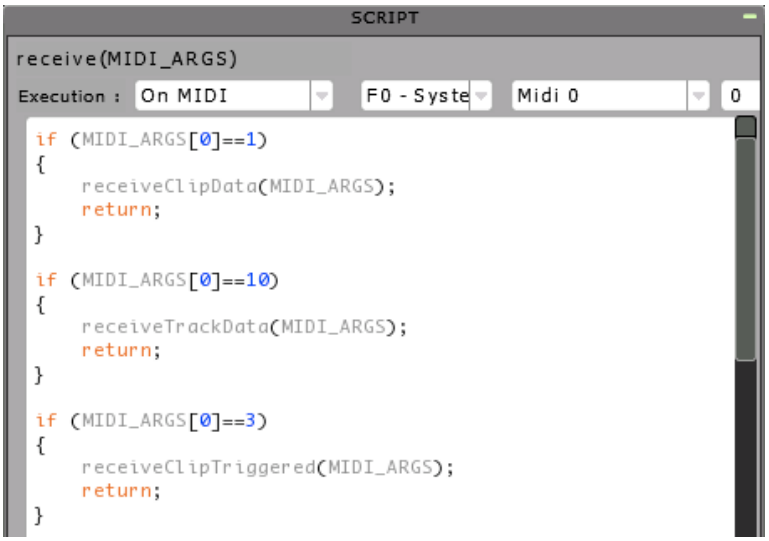
- **do while**

The **do while** structure also consists of a condition and a block of code, but the difference here is that the block of code within braces is executed first, and then the condition evaluated. If the condition is true, the block of code within braces is executed again. This repeats until the condition becomes false. The do while loop can be referred to as a post-test loop, and the while loop as a pre-test loop.




## 11.3 Return

The **return** statement isn't specific to conditional or loop structures, but can be used anywhere to instruct the execution to leave the current script. For instance, it can be used to optimize a script's efficiency by preventing the execution to read the entire script needlessly:




Now let's go through a last example. The script below can be used to return the maximum value reached by an array (vector). Notice that the script's execution mode is set to **Manual**, so it can be recalled at any time, in any parts of the Project. For instance, it can be recalled in a Monitor Object to display the maximum value reached by a MultiSlider's sliders. This is done by changing the script's arguments (array) by (MultiSlider.x).

SCRIPT

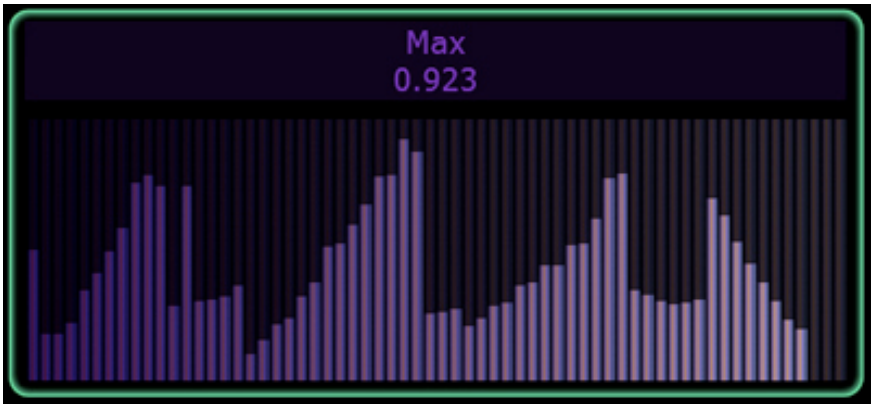
max_array(array)

Execution : Manual

```
decl size = sizeof(array);
decl i, value;
value = array[0];

for (i=1; i<size; i++)
{
    if (array[i]>value) value = array[i];
}

return value;
```




OBJECTS

Properties | Behavior

**Monitor**

Name: Max

X: 3 W: 530

Y: 0 H: 51

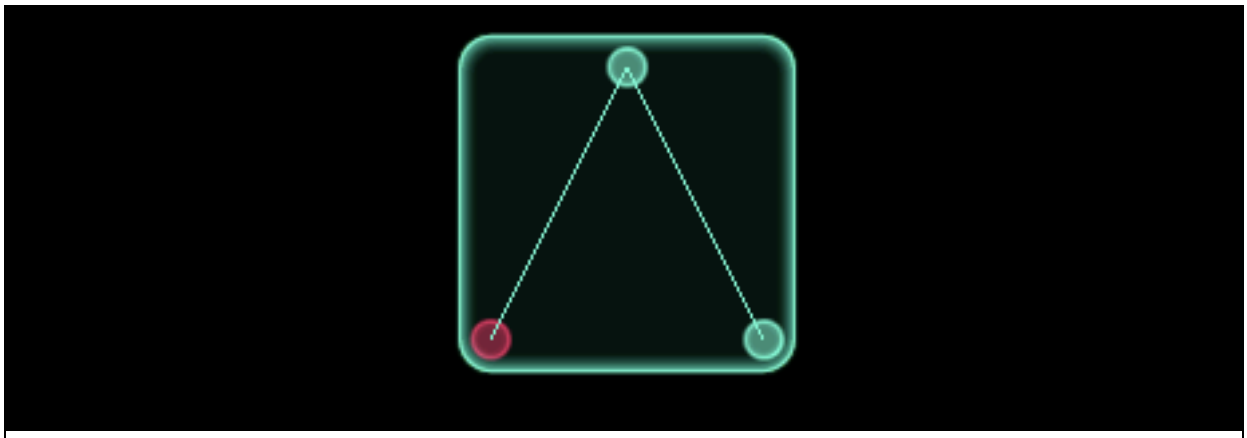
Label

value= max_array(MultiSlider.x)

Unit

# Chapter 12 - Object Reference

## 12.1 Breakpoint



The Breakpoint Object is a multi-segment envelope editor, which assigns each finger to track one of a number of points (up to 64) in a rectangular space. If the edit Option is activated, new points can be created in real-time by double-touching the object's surface. By default the object constraints a point's horizontal displacement so that it does not travel further than its left neighbour (x[2]>=x[1]>=x[0]), but that can be deactivated by ticking the free form Option.

| **Variables** | |
|---|---|
| x | A vector containing the points' horizontal positions. |
| y | A vector containing the points' vertical positions. |
| **Properties** | |
| Name | Name of the Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed on the Object. |
| Live Edition | If checked, new points can be added by double-touching the object's surface. |
| Points | Number of points (1 to 64). |
| Coordinates | If checked, x and y coordinates are displayed when touched. |
| Background | Choose the color of all the points except the first, together with the outline around the Breakpoint. |
| First point | Choose the first point's color with the color picker. |
| Light | Can be a constant, a vector or any mathematical expression and controls the luminosity of the points. -2 means black, +2 means white, and you can choose any decimal number in-between. |

| **Behavior** | |
|---|---|
| Grid | If checked, the range of values produced by the Breakpoint is quantized into [grid] steps. The maximum number of steps for the Breakpoint's axis is 33. |
| Free Form | By default the object constraints a point's horizontal displacement so that it does not travel further than its left neighbour (x[2]>=x[1]>=x[0]). If the Free Form option is checked, points can be displaced freely on the horizontal axis. |
| Physics | |
| • None | Points follow finger positions immediately. Attraction and Friction parameters are ignored. |
| • Interpolate | Points follow the fingers' positions according to the value of Attraction. Larger values for attraction (up to 1) cause a point to move to a finger position more quickly. As the Attraction value is lowered, the points take longer to arrive at the finger position. When Attraction is set to 0, points cannot be moved by your fingers. |
| • Mass-Spring | Attraction and Friction are both active. Attraction works as described above under Interpolation. Friction ranges between 0 and 1. Lower values of friction mean that if a point is moving it will tend to keep moving. With a value of 0, the points will essentially never stop moving. At a value of 1, a point will move only where you place with your finger. Values of 1 for Attraction and Friction are essentially the same as if Physics is set to Interpolate. |
| • Super-Spring | The segments between points also behave as springs, whose initial and resting lengths are set by the Rest parameter. |
| Attraction | The amount of attraction the cursors (your fingers) have on the points. |
| Friction | The amount of friction the Object's surface applies on the points. |
| Speed | This value multiplies the points' speed after Physics computation by a user-defined expression. Input a singleton for the same effect on all points, otherwise input a vector. |
| Rest | When Super-Spring mode is enabled, the rest (0 to 1) is the value of the initial and resting lengths of the segments between the points. |
| HoldX | If 0, this has no effect. Any value greater than 0 freezes the respective point on its current position on the x-axis. The y-axis remains active. Use a vector, if you want to affect specific points. |
| HoldY | If 0, this has no effect. Any value greater than 0 freezes the respective point on its current position on the y-axis. The x-axis remains active. Use a vector, if you want to affect specific points. |

| **Attributes** | | |
|---|---|---|
| color | {integer*,integer*} | color of the background and first point |
| coordinates | 0 or 1 | coordinates off/on |
| editable | 0 or 1 | live edition off/on |
| free | 0 or 1 | free form off/on |
| grid | 0 or 1 | grid off/on |
| grid_steps | {1 to 33, 1 to 33} | number of grid steps on the x and y axis |
| label | 0 or 1 | label off/on |
| name | text | get object name only |
| nbr | 1 to 64 | number of points |
| physic | 0 to 3 | physics mode |
| rect | {X,Y,W,H} | object's position and dimensions |
| zoom | 1 to 50 | zoom in object |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.2 Container


Containers can enclosed any number of objects inside them, including other Containers. A Container can also be made tabbed by ctrl-clicking/right-clicking on it and choosing the option *Make tabbed*. Additional tabs can then be adjoined with the option *Add tab*, and their respective content displayed by touching the corresponding tab. The tabs' order can be modified by ctrl-clicking/right-clicking within a Container and choosing *Bring up* or *Bring down*.

| **Properties** | |
|---|---|
| Name | The Container's name (used as a prefix for addressing the Objects inside the Container, as in *Container/Fader.x*). |
| Lock | A locked Container prevents editing and display of its content in the Project panel. |
| Transparent | If checked, the Objects lying "under" the Container (but not belonging to its content) are shown and the Container's frame is not displayed anymore. |
| Color | The color of the Container's frame. |

| **Attributes** | | |
|---|---|---|
| color | integer* | color of the container |
| label | 0 or 1 | label off/on |
| name | text | Container name |
| rect | {X,Y,W,H} | object's position and dimensions |
| tabbar | 0 or 1 | tabbar off/on |
| transparent | 0 or 1 | transparent off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

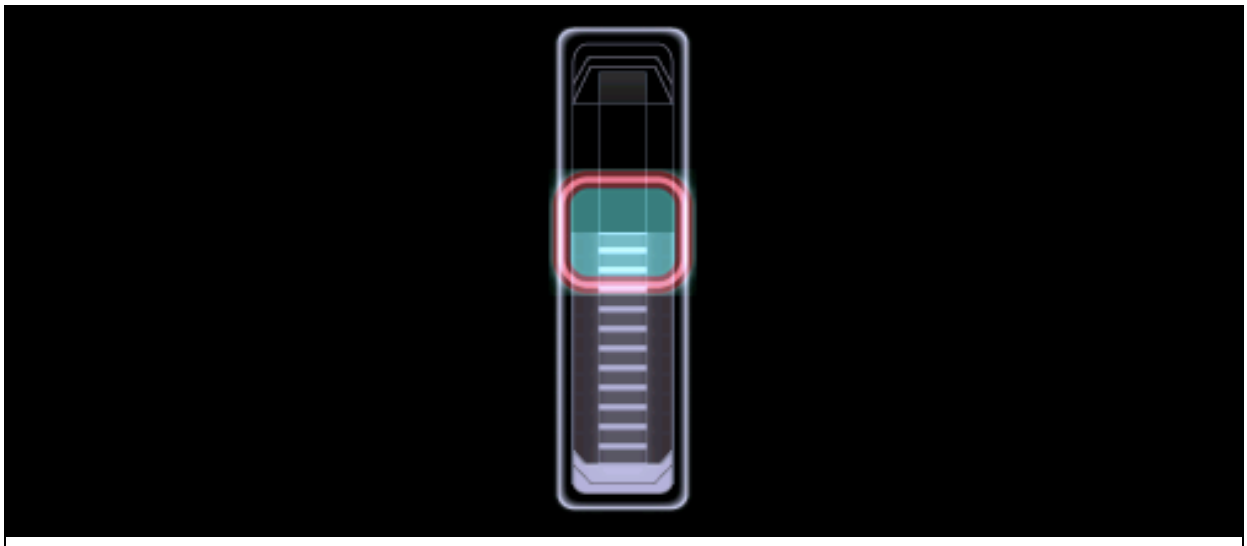
## 12.3 Custom Button

| | |
|---|---|
| The Custom Button can act as a pad or switch. You can set the text of the button for on and off state. Another option is to have the state displayed by two different geometric forms to be chosen among. | |
| **Variables** | |
| x | The state of the button. |
| **Properties** | |
| Name | The name of the Object, also used as its default OSC address. |
| Style Off | A menu for choosing how the Off state is depicted. Choose between text or symbols. If text is chosen, a whitespace can be used in the text entry. |
| Style On | A menu for choosing how the On state is depicted. |
| Font | A menu for choosing the font size for the text. The font size ranges from 8pt to 24pt. |
| Alignment | A graphical menu for choosing the position of the text within the boundaries of the Object. You get a choice between 9 different positions. This feature does not work for the symbols. |
| Color Off/On | Dial in the color for the two respective button states. |
| Light | Can be a constant or any mathematical expression and controls the luminosity of your button. -2 means black, +2 means white, and you get to choose any decimal number in-between. |
| **Behavior** | |
| Capture | If Capture is checked, an Object will only react to cursors created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, the original behavior is restored, meaning an Object will react to whatever cursor is present at any moment in its area. |
| Mode | A menu for choosing if the Button works as a Switch or a Pad. A Switch changes state on touch and doesn't change back on removal of the finger. A Pad changes its state back if you remove your finger. |

| **Attributes** | | |
|---|---|---|
| behavior | 0 or 1 | switch or pad mode |
| bitmap | {0 to 14,0 to 14} | decoration bitmap in off state, in on state |
| capture | 0 or 1 | capture off/on |
| color | {integer*,integer*} | color off state, color on state |
| label_off | text | button text in off state |
| label_on | text | button text in on state |
| name | text | Object name |
| outline | 0 or 1 | outline off/on |
| rect | {X,Y,W,H} | object's position and dimensions |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.4 Fader



The Fader tracks your finger with a virtual cap and generates a value corresponding to the position of the cap on the fader. The Fader can be oriented vertically or horizontally. Grab and drag a corner to change its orientation.

| **Variables** | |
|---|---|
| x | The location of the cap. When the cap is at the top or right-most position of the fader (depending on orientation), the value is 1 by default. When it is at the bottom or at the left-most position, respectively, the value is 0. |
| z | A flag variable for detecting if the Fader is being touched. 1 if touched, 0 if untouched. This is useful to emulate fader touch of control surfaces. |
| **Properties** | |
| Name | The name of the Breakpoint Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed in the Interface. |
| Value | If checked, the current value of the Fader is displayed in the Interface. In addition, you can enter a formula for how the value is displayed. This does not affect the actual value sent by the Fader, which remains between 0 and 1. |
| Unit | This user-specified text is appended at the end of the value display. Use it to specify the type of value, as in dB or ms. |
| Precision | Specifies the number of decimal places for the value display. The default value is 3 and the maximum number is 6. This setting has no influence on the actual output of the Object. You have to scale the output using expressions or on the Target side. |
| Color | Pick a color for the fader's body, its cap always keeps the pink outline. |

| **Behavior** | |
|---|---|
| Grid | If checked, the range of values produced by the Fader is quantized into [grid] steps. The maximum number of steps for the Fader is 33. |
| Capture | If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, the old school way from previous versions is restored, meaning an Object will react to whatever cursor is present at any moment in its area. |
| Cursor Mode | |
| • Limited | The Fader will respond to a new cursor only if the original one has been destroyed (i.e. finger is raised) |
| • Get Newer | Whenever a new cursor appears inside the Object's area, it gains full control of the Object |
| • Barycentric | Each cursor, old or new, has the same amount of influence on the Object. |
| • Cap Only | The Object acts like a conventional fader that doesn't react to cursors outside of the cap area |
| Physics | |
| • None | The Fader cap tracks one finger immediately. If other fingers touch the fader, they are ignored. |
| • Interpolate | The cap moves according to the value of Attraction from its current location to the location of your finger. Larger values for attraction (up to 1) cause the cap to move to the new finger location more quickly. When Attraction is set to 0, the cap cannot be moved by your finger. |
| • Mass-Spring | Attraction and Friction are both active. Friction ranges between 0 and 1. Lower values of friction mean that if the cap is moving it will tend to keep moving. With a value of 0, the cap will never stop moving. At a value of 1, the cap exactly follows your finger. Values of 1 for Attraction and Friction are essentially the same as if Physics is set to None.<br><br>Please consider that Lemur will send the Fader's position constantly with lower values of Attraction and Friction in Mass-Spring mode. |
| Attraction | The amount of attraction the cursor (your finger) has on the Fader |
| Friction | See Mass-Spring above |

| **Attributes** | | |
|---|---|---|
| capture | 0 or 1 | capture off/on |
| color | integer* | color of the object |
| cursor | 0 to 3 | cursor mode |
| grid | 0 or 1 | grid off/on |
| grid_steps | 1 to 33 | number of grid steps |
| label | 0 or 1 | label off/on |
| physic | 0 to 2 | physics mode |
| precision | 0 to 6 | decimal places of displayed value |
| name | text | Object name |
| rect | {X,Y,W,H} | object's position and dimensions |
| unit | text | unit of displayed value |
| value | 0 or 1 | value off/on |
| zoom | 1 to 50 | zoom in object |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.5 Knob

The Knob Object emulates two types of knobs: Classic knobs constrained between a min and max value (0 and 1 by default), and endless encoders that can reach any value through multiple successive turns.

| **Variables** | |
|---|---|
| x | The current value of the knob, based on absolute angle position in Classic mode, or rotation count in Endless mode. |
| **Properties** | |
| Name | The name of the Knob Object, also used as its default OSC address. |
| Endless Knob | If checked, the knob switches to endless mode. |
| Label | If checked, the Object's name is displayed above the knob. |
| Value | If checked, the current value of the Knob is displayed in the Interface. In addition, you can enter a formula for how the value is displayed. This does not affect the actual value sent by the Knob, which remains between 0 and 1. |
| Unit | This user-specified text is appended at the end of the value display. Use it to specify the type of value, as in dB or ms. |
| Precision | Specifies the number of decimal places for the value display. The default value is 3 and the maximum number is 6. This setting has no influence on the actual output of the Object. You have to scale the output using expressions or on the Target side. |
| Background | Pick the Knob's background color. |
| Foreground | Pick the Knob's foreground color. |

| **Behavior** | |
|---|---|
| Grid | If checked, the range of values produced by the Knob is quantized into [grid] steps. The maximum number of steps for the Fader is 33. |
| Capture | If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, the old school way from previous versions is restored, meaning an Object will react to whatever cursor is present at any moment in its area. |
| Cursor Mode | |
| • Limited | The Fader will respond to a new cursor only if the original one has been destroyed (i.e. finger is raised) |
| • Get Newer | Whenever a new cursor appears inside the Object's area, it gains full control of the Object |
| • Barycentric | Each cursor, old or new, has the same amount of influence on the Object. |
| • Cap Only | The Object acts like a conventional fader that doesn't react to cursors outside of the cap area |
| Physics | |
| • None | The Fader cap tracks one finger immediately. If other fingers touch the fader, they are ignored. |
| • Interpolate | The cap moves according to the value of Attraction from its current location to the location of your finger. Larger values for attraction (up to 1) cause the cap to move to the new finger location more quickly. When Attraction is set to 0, the cap cannot be moved by your finger. |
| • Mass-Spring | Attraction and Friction are both active. Friction ranges between 0 and 1. Lower values of friction mean that if the cap is moving it will tend to keep moving. With a value of 0, the cap will never stop moving. At a value of 1, the cap exactly follows your finger. Values of 1 for Attraction and Friction are essentially the same as if Physics is set to None.<br><br>Please consider that Lemur will send the Fader's position constantly with lower values of Attraction and Friction in Mass-Spring mode. |
| Attraction | The amount of attraction the cursor (your finger) has on the Fader |
| Friction | See Mass-Spring above |

| **Attributes** | | |
|---|---|---|
| color | {integer*,integer*} | color of background, color of foreground |
| cursor | 0 to 2 | cursor mode – Limited, Barycentric, Get Newer |
| grid | 0 or 1 | grid off/on |
| grid_steps | 1 to 33 | number of grid steps |
| label | 0 or 1 | label off/on |
| mode | 0 or 1 | polar or linear |
| name | text | Object name |
| physic | 0 to 2 | physics mode – None, Interpolate, Mass-Spring |
| precision | 0 to 6 | decimal places of displayed value |
| rect | {X,Y,W,H} | object's position and dimensions |
| unit | text | unit of displayed value |
| value | 0 or 1 | value off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.6 Leds

The Leds Object is a two-dimensional array of Leds. It can be a single led or a whole matrix of them, all with their individual light and value parameters. Remember that those properties can be set to vectors when dealing with several columns or rows of Leds.

The Leds are transparent to touch but not transparent in terms of display. This means you can implement switch or pad functionality by placing a hidden Switches or Pads Object underneath the Leds.

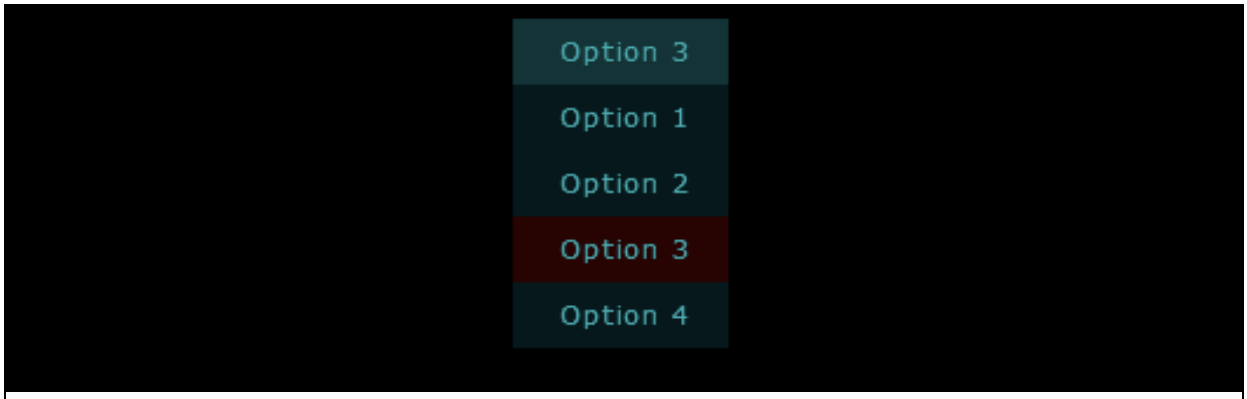
**Properties**

| | |
|---|---|
| Name | The name of the Leds Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed above the Leds. |
| Transparent | If checked, the Leds'background is transparent. |
| Value | In Bargraph mode this value represents the percentage of the Leds being switched on. If Bargraph mode is off, only 1 and 0 are accepted as values. If you send a single value (0,1) it will switch all Leds to their on or off colors. If you have multiple rows/columns, you can use a vector to individually address the Leds. |
| Columns | The number of columns of Leds contained in the Object. Only 16 columns of Leds can be set. |
| Rows | The number of rows of Leds contained in the Object. Only 16 rows of Leds can be set. |
| Multicolor | If checked, the 'colors' attribute of the Object can be set to a different value for each Led. |
| BarGraph | The Bar Graph mode makes a matrix or vector of Leds act as your typical bar graph: the Leds work together to graphically display the current value. |
| Color Off | Pick a color for the Leds' off-state. |
| Color On | Pick a color for the Leds' on-state. |
| Light | Can be a constant, a vector or any mathematical expression and controls the luminosity of your Objects. -2 means black, +2 means white, and you can choose any decimal number in-between. |

| **Attributes** | | |
|---|---|---|
| bargraph | 0 or 1 | bargraph off/on |
| color | {integer*,integer*} | color off, color on |
| colors | {integer*,integer*, . . } | vector of colors for multicolor use |
| column | 0 to 16 | number of columns |
| label | 0 or 1 | label off/on |
| name | text | Object name |
| rect | {X,Y,W,H} | object's position and dimensions |
| row | 0 to 16 | number of rows |
| transparent | 0 or 1 | transparent off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.7 Menu




The Menu Object is a "pop-up" menu selection tool. It can hide and display a list of up to 32 individual text items. By default the object outputs the index of the selection.

| **Variables** | |
|---|---|
| selection | The current index of the selection (1,2,3,4…) |

| **Properties** | |
|---|---|
| Name | The name of LemurMenu Object, also used as its default OSC address. |
| Scale Output | Scale Output to Midi values. |
| Transparent | If checked, the Object's background becomes transparent. |
| Font | Text content font size. |
| Alignment | A graphical menu for choosing the position of the text within the boundaries of LemurMenu's cells. |
| Color | Color of the the Object |
| Items | A list for editing the menu's content. |

| **Attributes** | | |
|---|---|---|
| color | integer* | object's color |
| items | {text,text…} | menu items |
| name | text | Object name |
| rect | {X,Y,W,H} | object's position and dimensions |
| scale | 0 or 1 | scale output off/on |
| transparent | 0 or 1 | transparent off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.8 Monitor


The Monitor produces no data when you touch it. Its purpose is to display information sent to Lemur by your computer.

**Variables**

| | |
|---|---|
| none | |

**Properties**

| | |
|---|---|
| Name | The name of LemurMenu Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed above the value. |
| Value | The checkbox is not functional. The text field next to Value represents the monitor's default value. Since any value can be sent to the monitor, there is no 0-1 limitation. |
| Unit | Appends arbitrary text to the end of the value display. |
| Precision | Number of floating-point digits that appear. A precision of 0 displays only the integer part of numbers the monitor receives. |
| Font | A menu for choosing the font size for the label and value of the Monitor. The font size ranges from 8pt to 24pt. |
| Alignment | A graphical menu for choosing the position of the text within the boundaries of the Object. You get a choice between 9 different positions. |
| Transparent | If checked, only the label and the value is displayed and the Monitor's background becomes transparent. This also makes it possible to use the Monitor as a text label anywhere in the Interface. |
| Color | Pick the Object's color. |

| **Attributes** | | |
|---|---|---|
| color | integer* | color of the object |
| label | 0 or 1 | label off/on |
| name | text | get Object name |
| precision | 0 to 6 | decimal places of displayed value |
| rect | {X,Y,W,H} | object's position and dimensions |
| transparent | 0 or 1 | transparent off/on |
| unit | text | unit of displayed value |
| value | 0 or 1 | value off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.9 MultiBall

The MultiBall Object assigns each finger to track one of a number of balls (up to 10) in a rectangular space. Balls can either always be visible or only appear when you touch the space; the latter is called ephemeral mode**.** The brightness of the balls is sent as the z variable in the Object.

| **Variables** | |
|---|---|
| x | A list of the horizontal positions of all the balls. |
| y | A list of the vertical positions of all the balls. |
| z | A list of the brightness values of all the balls. Brightness values change only when the MultiBall Object is in ephemeral mode. |
| **Properties** | |
| Name | The name of the MultiBall Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed in the Interface. |
| Numbers | If checked, a running number is displayed on each ball. |
| Multilabel | If checked, individual labels may be used for the Objects parts. |
| Multicolor | If checked, individual colors may be used for the Objects parts. |
| Balls | Number of balls (1 to 10) |
| Color | Pick a color for the MultiBall's frame |
| **Behavior** | |
| Grid | If checked, the range of values produced by the Balls is quantized into [grid] steps. The maximum number of steps for the MultiBall axis is 33. |
| Ephemeral | If checked, the MultiBall behaves in a mode where the balls disappear until you touch it with one or more fingers. The brightness of the balls becomes the value of the z variable of the Object, and the way the brightness changes over time is controlled by an ADSR envelope (described below). When |

| | |
|---|---|
| | Ephemeral is not checked, the balls are always visible and their z values are constantly 1. |
| Capture | If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, an Object will react to whatever cursor is present at any moment in its area. |
| Cursor Mode | |
| • Limited | The Balls will respond to a new cursor only if the original one has been destroyed (i.e. finger is raised) |
| • Barycentric | Each cursor has the same amount of influence on the Object. |
| • Get Oldest Ball | If all the balls are currently "possessed" by your fingers and a new cursor comes in, it will take control of the oldest ball (the one that lived the longest in Ephemeral, or the first ball in normal mode). |
| • Get Closest Ball | If all the balls are currently "possessed" by your fingers and a new cursor comes in, it will take control of the closest ball. |
| Physics | |
| • None | Balls move to finger positions immediately, and the settings of Attraction and Friction are ignored. |
| • Interpolate | Balls move toward finger positions according to the value of Attraction. Larger values for attraction (up to 1) cause a ball to move to a finger position more quickly. As the Attraction value is lowered, the balls take longer to arrive at the finger position. When Attraction is set to 0, your fingers can't move balls. |
| • Mass-Spring | Attraction and Friction are both active. Attraction works as described above under Interpolation. Friction ranges between 0 and 1. Lower values of friction mean that if a ball is moving it will tend to keep moving. With a value of 0, the balls will essentially never stop moving. At a value of 1, a ball will move only where you touch with your finger. Values of 1 for Attraction and Friction are essentially the same as if Physics is set to Interpolate. |
| Attraction | Amount of attraction the cursor (your fingers) have on the Balls. |
| Friction | See Mass-Spring above |
| Speed | This value multiplies the balls' speed after Physics computation by a user-defined expression. Input a singleton for the same effect on all balls, otherwise input a vector. Experiment with negative values for crazy effects. |
| Attack | Applies only when using ephemeral mode. The Attack value specifies the number of seconds over which the z variable (brightness of a ball) increases from its initial value of 0 to a maximum of 1 after you touch the screen. As an example, if the |

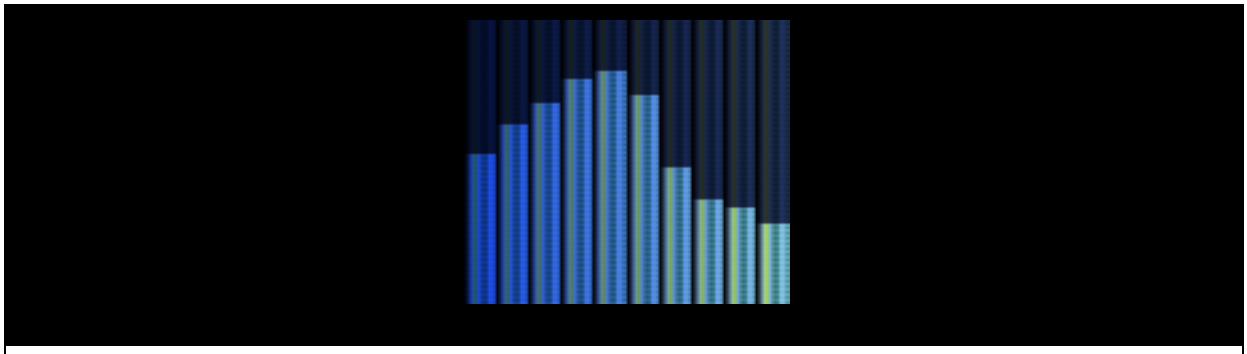
| | |
|---|---|
| | Attack value is 0, the ball will be at full brightness the moment you touch the screen. |
| Decay | Applies only when using ephemeral mode. The Decay value specifies the number of seconds over which the brightness will decrease after the initial Attack portion of the envelope has completed. During the Decay portion of the envelope, the z variable (brightness of a ball) will decrease from 1 to the level set by the Sustain value. |
| Sustain | Applies only when using ephemeral mode. The sustain value is the level (between 0 and 1) at which the brightness of the ball will remain as long the MultiBall Object is tracking your finger within its space. The Sustain level is reached after the Attack and Decay portion of the envelope have completed. If your finger lifts up from the touch surface before the completion of the Attack and/or Decay portion of the envelope, the Release portion of the envelope is triggered immediately after the Decay portion completes, and the brightness ultimately goes to 0. |
| Release | Applies only when using ephemeral mode. The Release value specifies the number of seconds over which the brightness of a ball will decrease from its Sustain level to 0, starting at the moment that you lift up ("release") your finger from the touch surface. |
| Hold | Its effect is similar to a sustain pedal, freezing the Object's state as long as its value is 1. When set to 0, if has no effect. This means this parameter should be used with a mathematical expression depending on other Objects. For instance, if you have a Switch Object named Sustain in your interface, you can set the hold parameter of a Pad to Sustain.x so the Switch gets the ability to freeze the current lightness. |
| HoldX | If 0, this has no effect. Any value greater than 0 freezes the respective ball on its current position on the x-axis. The y-axis remains active. Use a vector, if you want to affect specific balls. |
| HoldY | If 0, this has no effect. Any value greater than 0 freezes the respective ball on its current position on the y-axis. The x-axis remains active. Use a vector, if you want to affect specific balls. |

| **Attributes** | | |
|---|---|---|
| ball_enable | 0 or 1 | ball enable off/on |
| capture | 0 or 1 | capture off/on |
| color | integer* | color of the Objects frame |
| colors | {integer*,integer*, . . } | vector of colors for multicolor use |
| cursor | 0 or 1 | cursor off/on |

| | | |
|---|---|---|
| ephemere | 0 or 1 | ephemeral off/on |
| grid | 0 or 1 | grid off/on |
| grid_steps | {1 to 33, 1 to 33} | number of grid steps on the x and y axis |
| label | 0 or 1 | label off/on |
| labels | 0 or 1 | vector of labels for multilabel use |
| multicolor | 0 or 1 | label off/on |
| multilabel | 0 or 1 | multilabel off/on |
| nbr | 1 to 10 | number of balls |
| physic | 0 to 2 | physics mode – None, Interpolate, Mass-Spring |
| polyphony | 0 or 1 | polyphony off/on |
| rect | {X,Y,W,H} | object's position and dimensions |
| zoom | 1 to 50 | zoom in object |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.10 MultiSlider



The MultiSlider Object tracks movement across an array of sliders. You can "wipe" all the faders to a set value with one horizontal gesture. This is pretty hard to do with real—or virtual—faders.

| **Variables** | |
|---|---|
| x | A list of the vertical positions of all the individual sliders. |
| **Properties** | |
| Name | The name of the Object, also used as its default OSC address. |
| Horizontal | Swaps the MultiSlider's orientation from vertical to horizontal. |
| Bipolar | If checked, the Object's zero position is the slider's mid-length. |
| Label | If checked, the Object's name is displayed in the Interface. |
| Slider | Number of sliders (1 to 64) |
| Gradient | If checked, a gradient is applied on the Object's color. |
| Multicolor | If checked, the 'colors' attribute is active and can be used to change the colors of individual sliders. |
| Color | Drag the color bar to change the "thematic" color of the sliders. |
| Light | Can be a constant, a vector or any mathematical expression and controls the luminosity of your Objects. -2 means black, +2 means white, and you get to choose any decimal number in-between. |
| **Behavior** | |
| Grid | If checked, the range of values produced by the Sliders is quantized into [grid] steps. The maximum number of steps for the Fader is 33. |
| Capture | If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, the old school way from previous versions is restored, meaning an Object will react to whatever cursor is present at any moment in its area. |

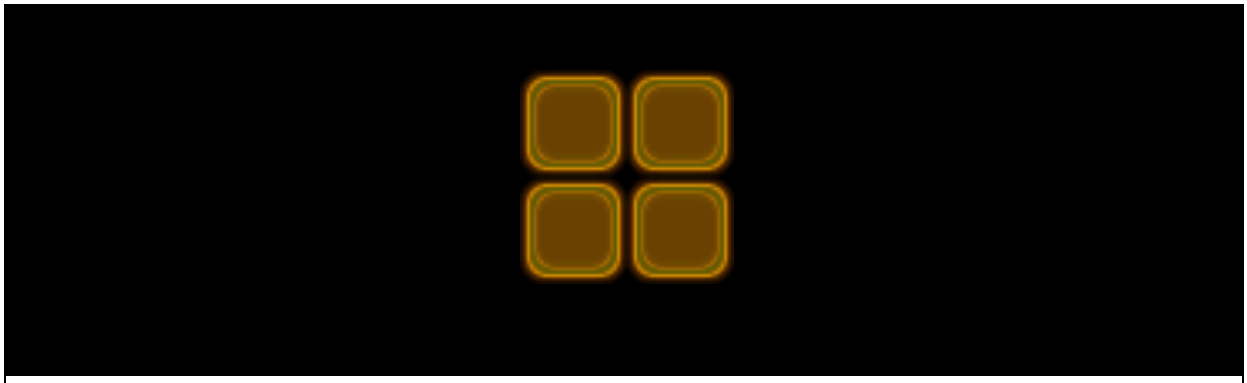
| | |
|---|---|
| Physic | If checked, the MultiSlider emulates the physics of an Object similar to a plucked string anchored at the left and right sides of the array of sliders. Your fingers "pluck" the string by lifting it up in one or more places. The values of the sliders ramp up to meet your fingers and track them as they move. Lifting your finger(s) from the surface releases the string, and its subsequent behavior is determined by the Tension, Friction, and Height values. |
| Tension | A value between 0 and 1 corresponding to the tension on a string. As tension increases, the frequency of oscillation of the string increases. Increasing the tension is something like turning the tuning peg of a guitar to raise the pitch of a string. |
| Friction | A value between 0 and 1 corresponding to the damping on a string. As friction increases, the damping on the oscillation increases. With large friction values, the string returns to its resting position quickly. With smaller friction values, the string may oscillate for a long time. |
| Height | When Tension is enabled, the height (0 to 1) is the value of the initial and resting position of the string. |

**Attributes**

| | | |
|---|---|---|
| bipolar | 0 or 1 | capture off/on |
| capture | 0 or 1 | capture off/on |
| color | integer* | color of the object |
| colors | {integer*,integer*,…} | vector of colors for individual sliders |
| grid | 0 or 1 | grid off/on |
| grid_steps | 1 to 33 | number of grid steps |
| horizontal | 0 or 1 | horizontal off/on |
| label | 0 or 1 | label off/on |
| multicolor | 0 or 1 | multicolor off/on |
| nbr | 1 to 64 | number of sliders |
| physic | 0 or 1 | physics off/on |
| rect | {X,Y,W,H} | object's position and dimensions |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.12 Pads



The Pads Object is a two-dimensional array of buttons that are triggered by touch. They are intended to trigger events instead of represent state, since they eventually return to an "off" value after you touch them.

| **Variables** | |
|---|---|
| x | A list of the envelope (brightness) values of the pads. |
| **Properties** | |
| Name | Name of the Pads Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed in the Interface. |
| Numbers | If checked, each pad is labelled with a number corresponding to the order in which its value is transmitted, starting with 0. |
| Columns | The number of columns of pads contained in the Object (max 16). |
| Rows | The number of rows of pads contained in the Object (max 16). |
| Multilabel | If checked, the 'labels' attribute is active, and each pad can be labelled individually. |
| Multicolor | If checked, the Object's 'colors' attribute is active, and each Pad can take its own color value. |
| Color Off | Color for the Object's off state. |
| Color On | Color for the Object's on state. |
| light | Can be a constant, a vector or any mathematical expression and controls the luminosity of your Objects. -2 means black, +2 means white, and you get to choose any decimal number in-between. |
| **Behavior** | |
| Capture | If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, an Object will react to whatever cursor is present at any |

| | |
|---|---|
| | moment in its area. |
| Attack | The Attack value specifies the number of seconds over which the x variable (pad brightness) increases from its initial value of 0 to a maximum of 1 after you touch the screen. As an example, if the Attack value is 0, the pad will be at full brightness the moment you touch the screen. An attack value of 10 means the pad will take 10 seconds to reach the full value. |
| Decay | The Decay value specifies the number of seconds over which the x variable (pad brightness) will decrease after the initial Attack portion of the envelope has completed. During the Decay portion of the envelope, the x variable (pad brightness) will decrease from 1 to the level set by the Sustain value. |
| Sustain | The Sustain value is the level (between 0 and 1) at which the x variable (pad brightness) will remain as long your finger is touching the pad. The Sustain level is reached after the Attack and Decay portion of the envelope have completed. If your finger lifts up from the touch surface before the completion of the Attack and/or Decay portion of the envelope, the Release portion of the envelope is triggered immediately after the Decay portion completes, and the brightness ultimately goes to 0. |
| Release | The Release value specifies the number of seconds over which the x variable (pad brightness) will decrease from its Sustain level to 0, starting at the moment that you lift up ("release") your finger from the touch surface. |
| Hold | Its effect is similar to a sustain pedal, freezing the Object's state as long as its value is 1. When set to 0, if has no effect. This means this parameter should be used with a mathematical expression depending on other Objects. For instance, if you have a Switch Object named Sustain in your interface, you can set the hold parameter of a Pad to Sustain.x so the Switch gets the ability to freeze the current lightness. |

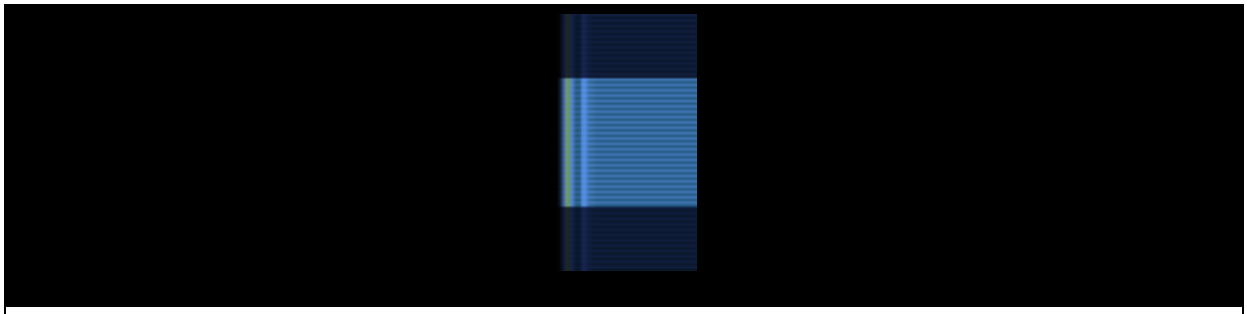
**Attributes**

| | | |
|---|---|---|
| capture | 0 or 1 | capture off/on |
| color | {integer*,integer*} | color off state, color on state |
| colors | {integer*,integer*} | Off state color values for each Pads |
| column | 1 to 16 | number of columns |
| label | 0 or 1 | grid off/on |
| labels | {Text,Text,..} | vector of labels for multilabel use |

| | | |
|---|---|---|
| multicolor | 0 or 1 | Multicolor off/on |
| multilabel | 0 or 1 | Multilabel off/on |
| rect | {X,Y,W,H} | object's position and dimensions |
| row | 1 to 16 | number of rows |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.13 Range



The Range Object is a slider with adjustable length. Touch either end to change the width/height of the range.

| **Variables** | |
|---|---|
| x | A vector containing the lower and higher positions. |
| **Properties** | |
| Name | The name of the Range Object, also used as its default OSC address. |
| Horizontal | Swaps the orientation of the Range from vertical to horizontal. |
| Label | If checked, the Object's name is displayed in the Interface. |
| Color | Pick the color of the Object |
| Light | Can be a constant, a vector or any mathematical expression and controls the luminosity of your Objects. -2 means black, +2 means white, and you get to choose any decimal number in-between. |
| **Behavior** | |
| Grid | If checked, the range of values produced by the Range sliders is quantized into [grid] steps. The maximum number of steps for the Fader is 33. |
| Capture | If Capture is checked, the Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off the Object will react to whatever cursor is present at any moment in its area. |
| Physics | If checked, the Range boundaries snap back to user-defined positions (that can evolve based on other variables) when you release them. Physics behavior is determined by the Tension, Friction, and Min_Height and Max_Height values. |
| Tension | A value between 0 and 1 corresponding to the tension of the string that links the boundaries to their snap-back positions. You can enter a vector to specify different tensions for the two extremities. |

| | |
|---|---|
| Friction | A value between 0 and 1 corresponding to the damping on a string. As friction increases, the damping on the oscillation increases. With large friction values, the string returns to its resting position quickly. With smaller friction values, the string may oscillate for a long time. You can enter a vector to specify different frictions for the two extremities. |
| Min_height | When Physics mode is enabled, the height (0 to 1) is the value of the initial and resting position of the mininum boundary of the Range. |
| Max_height | When Physics mode is enabled, the height (0 to 1) is the value of the initial and resting position of the maximum boundary of the Range. |
| Drag | If drag is not zero, the current range can be dragged with the finger without changing its limits. You may use this with a switch to toggle the behavior of your Range Object (put Switch.x into the drag field). The field can contain any expression. |

| **Attributes** | | |
|---|---|---|
| capture | 0 or 1 | capture off/on |
| color | integer* | color of the object |
| grid | 0 or 1 | grid off/on |
| grid_steps | 1 to 33 | number of grid steps |
| horizontal | 0 or 1 | horizontal off/on |
| label | 0 or 1 | label off/on |
| name | text | gets name of Object only – no set |
| physic | 0 or 1 | physics off/on |
| rect | {X,Y,W,H} | object's position and dimensions |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.14 RingArea

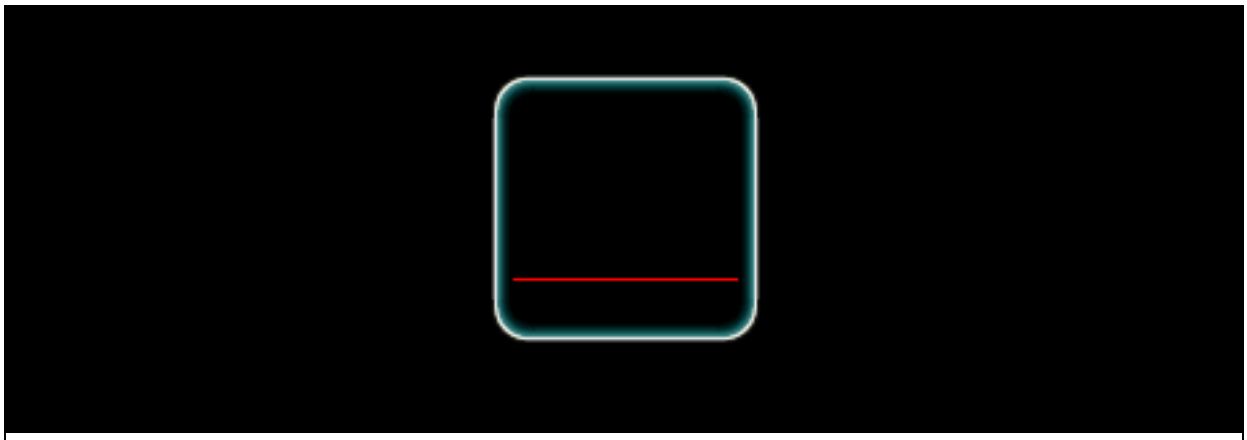
The RingArea tracks your finger inside a circular space. It reports the X and Y coordinates of a ball that can be programmed to have a variable degree of attraction toward a central point. You can specify the location of the attraction point within the circular space.

| **Variables** | |
|---|---|
| x | The horizontal position of the ball. |
| y | The vertical position of the ball. |
| **Properties** | |
| Name | The name of the RingArea Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed above the circular space. |
| Color | Pick a color for the RingArea. |
| **Behavior** | |
| Capture | If Capture is checked, the Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, the Object will react to whatever cursor is present at any moment in its area. |
| Attraction | A value from 0-1 representing the speed of the ball from its attraction point to your finger when you touch the Object, and from your finger to the attraction point when you release your finger. |
| Friction | A value from 0-1 representing the stickiness of the movement to either your finger or the attraction point. Lower values of friction make the ball overshoot the attraction point when it approaches, causing bouncing and/or oscillation. |
| Attractor X | A value from 0-1 representing the horizontal location of the attraction point within the Object's rectangle. A zero value of |

| | | |
|---|---|---|
| | Attractor X is at the left edge and a value of 1 is at the right edge. | |
| Attractor Y | A value from 0-1 representing the vertical location of the attraction point within the Object's rectangle. A zero value of Attractor Y is at the bottom edge and a value of 1 is at the top edge. | |
| **Attributes** | | |
| capture | 0 or 1 | capture off/on |
| color | integer* | color of the object |
| label | 0 or 1 | label off/on |
| name | text | gets name of Object only – no set |
| rect | {X,Y,W,H} | object's position and dimensions |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.15 SignalScope



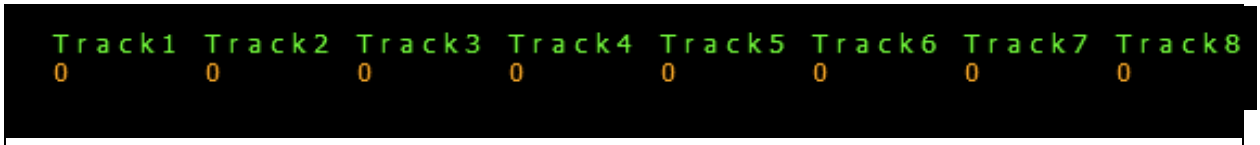
The SignalScope displays values of other Objects and variables on Lemur. The "trace" shows a recent history of the value of what you are monitoring. It is transparent to touch, meaning that you can place it on top of Objects and still interact with them.

| **Properties** | |
|---|---|
| Name | The name of the SignalScope Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed in the Scope area. |
| Mode XY | If checked, the signal scope shows both an X and Y value plotted against each other. If unchecked, the X value is time, shifting Y values to the left. |
| Transparent | If checked the Scope's frame is no longer visible. |
| Color | Pick a color for the Scope and its frame. |
| **Behavior** | |
| X | If Mode XY is enabled, X can be the value of a variable or a constant; otherwise it is assigned to time against which the Y value will be plotted. |
| Y | A variable or constant expression that will be periodically evaluated and plotted against X. For example, to plot the x variable of a Fader Object called chan1, enter chan1.x in the text field. |
| Time base | Corresponds to the time (in seconds) displayed on the scope. As the Time base increases, individual elements of the graph will decrease in width as more of the "past" is shown. |

| **Attributes** | | |
|---|---|---|
| color | integer* | color of the object |
| label | 0 or 1 | label off/on |
| mode | 0 or 1 | mode XY off/on |
| rect | {X,Y,W,H} | object's position and dimensions |
| transparent | 0 or 1 | transparent off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.16 SurfaceLCD




The SurfaceLCD Object emulates traditional control surfaces' LCD by making use of the MIDI input capabilities of Lemur to display track names, parameter changes, timecode, all updated in real-time by your Digital Audio Workstation. The SurfaceLCD is Mackie Control surface protocol compatible.

**Properties**

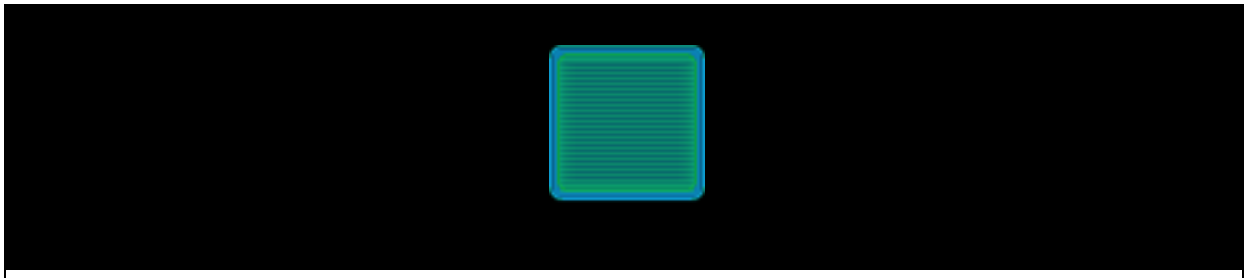
| | |
|---|---|
| Name | The name of the SurfaceLCD Object, also used as its default OSC address. |
| Transparent | If checked, the background of the SurfaceLCD disappears to obtain a see-through appearance. |
| Target | This lets you select the MIDI Target number for information display. First, setup a MIDI target with an input of your liking inside the Lemur Editor settings. Then, configure your DAW to send Mackie Control surface data to this MIDI input. Refer to the bundled examples for mappings of control Objects compatible with control surfaces protocols known by your DAW. |
| Display | |
| Main LCD | In this mode, the Object displays Track Information. |
| Timecode | In this mode, the Object displays up-to-date Timecode information. |
| Assignment | In this mode, the Object displays a three characters code describing the current assignment of other mapped control Objects. |

**Attributes**

| | | |
|---|---|---|
| color | integer* | color of the object |
| display | | |
| mode | 0 or 1 | mode XY off/on |
| rect | {X,Y,W,H} | object's position and dimensions |
| target | 0 or 7 | MIDI Target |
| transparent | 0 or 1 | transparent off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.17 Switches



The Switches Object is a two-dimensional array of toggle switches, representing and transmit one or more on-off states.

| **Variables** | |
|---|---|
| x | A list of the on-off values of the switches in the Object. The list starts with the top-left corner, and traverses the first row before starting at the beginning of the second row. |
| **Properties** | |
| Name | Name of the Object, also used as its default OSC address. |
| Label | If checked, the Object's name is displayed in the Interface. |
| Numbers | If checked, each switch is labelled with a number corresponding to the order in which its value is transmitted, starting with 0. |
| Radio | In Radio mode, only one switch can be turned on at any particular time. Turning on any switch turns all the others off. |
| Columns | Number of columns of switches contained in the Object (max 16). |
| Rows | Number of rows of switches contained in the Object (max 16). |
| Multicolor | If checked, the Switches 'colors' attribute is active, and color values can be set for each Switch individually. |
| Color Off | Pick a color for the Object's off-state |
| Color On | Pick a color for the Object's on-state |
| light | Can be a constant, a vector or any mathematical expression and controls the luminosity of your Objects. -2 means black, +2 means white, and you get to choose any decimal number in-between. |
| **Behavior** | |
| Capture | If Capture is checked, an Object will only react to cursors that were created inside its area. Even if the cursor later leaves the Object for another position, it will remain in control of the original Object, until it is destroyed eventually. When Capture is off, an Object will react to whatever cursor is present at any moment in its area. |

| | | |
|---|---|---|
| Paint | If this flag is active, you can "paint" on an array of switches by dragging your finger around. If paint is inactive, a touch only toggles the switch you hit first and dragging the finger around has no further effect. | |
| **Attributes** | | |
| capture | 0 or 1 | capture on/off |
| color | {integer*,integer*} | color off state, color on state |
| colors | {integer*,integer*, . . } | Off state color values for each Switches |
| column | 1 to 16 | number of columns |
| label | 0 or 1 | label off/on |
| labels | {Text,Text,..} | vector of labels for multilabel use |
| multicolor | 0 or 1 | Multicolor off/on |
| multilabel | 0 or 1 | Multilabel off/on |
| paint | 0 or 1 | paint off/on |
| radio | 0 or 1 | radio off/on |
| rect | {X,Y,W,H} | object's position and dimensions |
| row | 1 to 16 | number of rows |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

## 12.18 Text


The Text Object produces no data when you touch it. Its purpose is to display arbitrary text including whitespace characters typed in via the Lemur Editor.

| **Properties** | |
|---|---|
| Name | The name of the Text Object, also used as its default OSC address. |
| Transparent | If checked, only the text is displayed and the Object's background becomes transparent. |
| Text | Type in any text you want the Object to display for you. |
| Font | A menu for choosing the font size for the displayed text. The font size ranges from 8pt to 24pt. |
| Alignment | A graphical menu for choosing the position of the text within the boundaries of the Object. You get a choice between 9 different positions. |
| Color | Pick a color for the Object |
| light | Can be a constant, a vector or any mathematical expression and controls the luminosity of your Objects. -2 means black, +2 means white, and you get to choose any decimal number in-between. |

| **Attributes** | | |
|---|---|---|
| color | integer* | color of the object |
| content | text | displayed text |
| rect | {X,Y,W,H} | object's position and dimensions |
| transparent | 0 or 1 | transparent off/on |

*the value of the color attribute can range from 0 to 8355711: See the RGB(r,g,b) or HSV(h,s,v) internal functions to specify color component arguments in a 0 to 1 range eg RGB(0.5,0.5,0.5).

# Chapter 13 - Parser Reference

## 13.1 Vectors and Singletons

The Parser handles two kinds of variables: **vectors** and **singletons**. A vector (also known as an array or list), is a mathematical value that holds more than one element (number), while a singleton is a value containing a single value:

a **vector** : {0, 2, 0.5}

a **singleton** : 3

Lemur expression syntax allows operations on vectors and individual access to their elements. If you want to access the items of a vector separately, use brackets [].

a = {1,3,4}

a[0] = 1

a[2] = 4

This also works for multiple items:

a[0,2] = {1, 4}

Typically, the Variables of the Breakpoint, Leds, MuliSlider, MultiBall, Pads and Switches Objects are vectors when the Object features sub-objects (when they feature more than one ball, slider, pad etc..). For example, the x coordinates of the balls in a MultiBall object are accessed as

- myball.x[0]
- myball.x[1]
- etc.

Now let's look at what happens to a vector when we use it in an expression and combine it with operators, singletons or other vectors.

Assume we have a MultiSlider object called **moo** and a MultiBall object called **bar**. Below we have listed some expressions involving vectors and the various results that they would produce. Assume **moo.x** contains **{0.2 0.3 0.6}** and **bar.x** contains **{0.25 0.5}**

- moo.x * 3 Multiply all elements of moo.x by 3, returns {0.6 0.9 1.8}
- moo.x + 2 Add 2 to all elements of moo.x, returns {2.2 2.3 2.6}
- moo.x>0.5 Return a vector consisting of 1 or 0 depending on whether moo.x is greater than 0.5, returns {0. 0. 1.}
- moo.x + bar.x Add all elements of two vectors. The size of the result is the smaller of the two input vectors. Returns {0.45 0.8}
- moo.x[0] The first (index 0) element of moo.x, returns 0.2
- bar.x[1]The second element of bar.x, return 0.5
- moo.x[0.5] Interpolated value between bar.x[0] and bar.x[1], returns 0.3
- {moo.x, bar.x} Concatenates moo.x and bar.x, returns {0.2 0.3 0.6 0.25 0.5}
- {moo.x[0],bar.x[1.5]} Creates a new vector consisting of the first element of moo.x and the average of the second and third elements of bar.x, returns {0.2 0.625}

## 13.2 Lemur Internal clock

Lemur works with a clock speed of **60 ticks per second**. This means that every **16 ms** all states of Objects, Expressions and Functions are evaluated. These 16 ms constitute one Lemur **frame**.

There are some special considerations, though, as Lemur's brain tries to avoid unnecessary work. When it comes to deciding whether the value of a Variable is going to be transmitted via MIDI or OSC, Lemur first checks if the Variable has changed since the last frame. If it hasn't changed, it's not going to be transmitted.

## 13.3 Built-In Battery Variable

This variable will return the current charge of the battery as a float between 0.00 and 1.00. This can be used to give yourself a battery warning to avoid any chance of running out mid-gig.

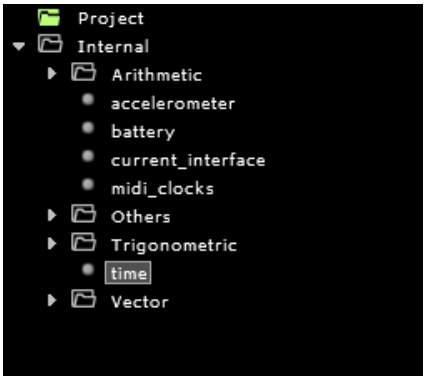
## 13.4 Built-In Accelerometer Variable

This variable will return the data from the device's built-in accelerometer as an x,y,z array. Try simply displaying accelerometer in a Monitor object then move the iPad or iPhone in all directions. Each axis is returned as a float between -1.00 and 1.00.

The accelerometer variable can be used for real-time control of Lemurs objects and expression allowing for 'non-touch control' of your Lemur. It retuns a vector of {x,y,z} values accessable directly as accelerometer[0], accelerometer[1] etc.

## 13.5 Built-In Time Variable

If you look at the Internal folder in the Project panel, you will see a pre-defined Variable called **time**.




The time Variable is a millisecond value you can use for creating time-varying behavior in Lemur. It represents the number of milliseconds since Lemur was turned on and it resets to 0 every hour. The time value is intended to be manipulated by mathematical Operators, particularly by multiplication (*), division (/), and by the modulo operator (%, used to calculate the remainder of a division)

By multiplying time by a value greater than 1, you produce a sequence of values that increase faster than clock time. By multiplying time by a value less than 1, you will produce a sequence of values that increase more slowly than clock time. You can use the modulo Operator (%) to create a repeating sequence of values that resembles a sawtooth wave. For example time % 1 produces a ramp from 0 to 999 that occurs over the course of a second.

Let 's use a Monitor Objects to display the values of the following expressions:

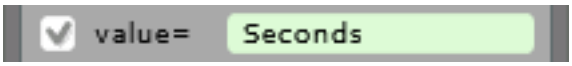
**Frames** = (time % 0.25) * 100

**Hours** = floor(time/3600)

**Minutes** = floor(time/60) % 60

**Seconds** = floor(time % 60)

Create global Variables for each of these Expressions, then use these Variables in other Lemur Objects. For example, here is the Properties Browser view for a monitor Object that would display a global Seconds variable.




Even if you are not connected to a Lemur, the resulting Monitor Object will begin changing immediately after you enter a time-based variable.


Another interesting use of the time variable would be the generation of an LFO:

**LFO** = sin(time * 2) *0.5 + 0.5

You can display this in a SignalScope to see what it does.

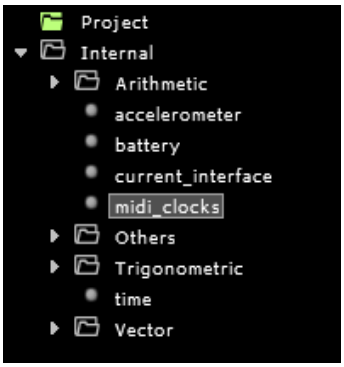
You can assign time-based variables to properties in Lemur Objects (such as Friction or the ADSR envelopes) to create Objects that change over time.

## 13.6 Built-in current_interface Variable

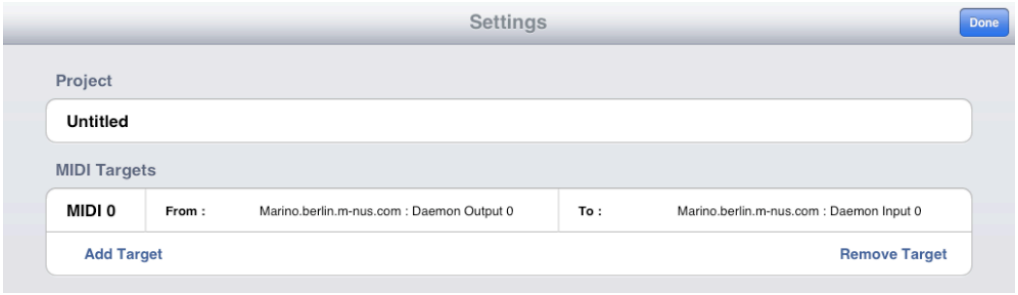
The internal folder also features the **current_interface** Variable, which simply gives you the Index of the currently displayed interface.

## 13.7 Built-in midi_clocks Variable

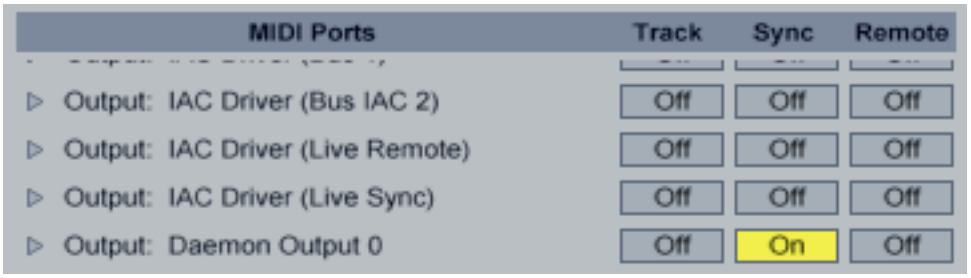
The Lemurs Internal folder also features the **midi_clocks** variable:



The midi_clocks variable can sync to incoming MIDI Clock signal on the 8 MIDI Targets ' inputs. Let's check it out with Ableton Live: make a connection between Lemur's MIDI Target 0 Input and

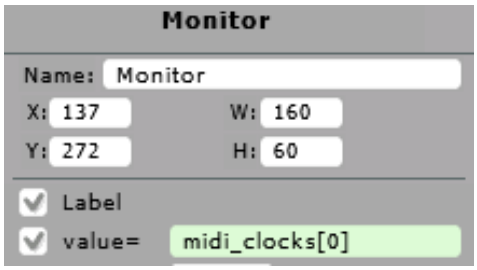



Live Sync Output:




MIDI ticks are sent 24 times per quarter note for synchronization purposes.

Now use a Monitor Object to display the MIDI clock ticks on Target 0:



## 13.8 Arithmetic Functions

**abs(a)** : absolute value
abs(-2) = 2

**ceil(a)** : rounded value to higher integer
ceil(1.2) = 2
ceil(1.8) = 2

**clamp(a, min, max)** : constrains a between min max
clamp(Fader.x, 0.2, 0.3) =
0.2 if Fader.x<=0.2
0.3 if Fader.x>=0.3
Fader.x otherwise

**floor(a)** : rounded value to the lower integer
floor(1.2) = 1
floor(1.8) = 1

**max(a,b)** : max of a and b
max(0.1, 0.8) = 0.8

**min(a,b)** : min of a and b
min(0.1, 0.8) = 0.1

**pow(a,b)** : a to the power of b
pow(2,4) = 16

**rand()** : returns a random value between 0 and 1 at each frame (a Lemur frame is 16ms)

**range(a, min, max)** : stretches a Variable value that normally goes from 0 to 1, to range [min, max]

range(Fader.x, 10, 100) returns 10 if Fader.x == 0,
returns 100 if Fader.x == 100 or returns value ranged between 10 and 100

**round(a)** : rounded value to the closest integer

round(1.2) = 1
round(1.8) = 2

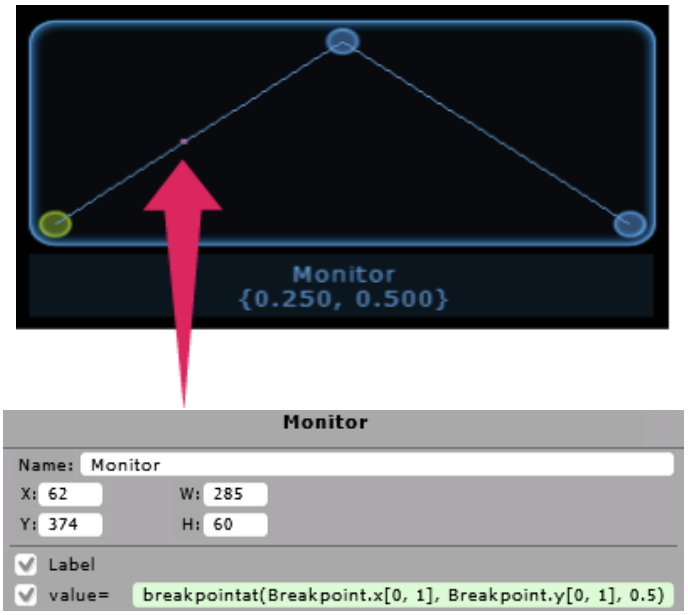
**sign(a)** : -1 if a<0, +1 if a>=0

sign(-12) = -1
sign(0) = +1
sign(12) = +1

**sqrt(a)** : square root of a
sqrt(64) = 8

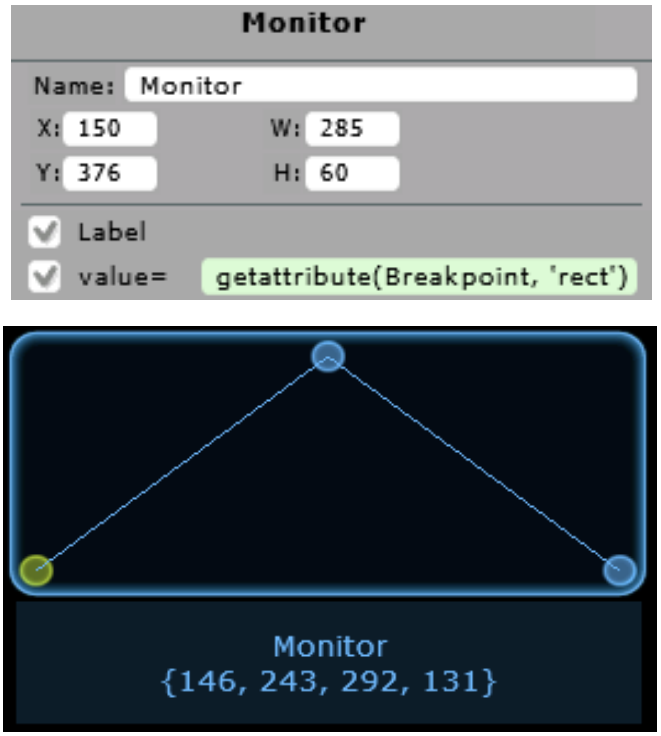
## 13.9 Object-related Functions

**breakpointat(x[],y[],phase)** : returns the position of a point along a Breakpoint's segment




**findobject(name)** : selects an Object within a tree structure (project, interface or container)

**getattribute(object,name)** : returns the value of the Object's attribute 'name'



**getattributelist(object)** : returns the Object's attribute list

**getexpression(object, name)** : returns the value of the expression 'name'

**getfirst(object)** : selects the first Object within a tree structure (project, interface or container)

**getnext(object)** : selects the next Object within a tree structure (project, interface or container)

**getobject()** : returns the object within which the script calling the function resides. This acts effectively as a 'self' reference.

**getobjectrect(object)** : returns the Object's position and dimensions as {X,Y,W,H}

**getparent(object)** : returns an Object's Parent Target (next higher hierarchy level)

**HSV(h,s,v)** : converts Hue, Saturation, Value indices (hue,saturation,value) to a single color value that the parser understands.

Lemur Color is represented by a single value from 0 to 8355711. The HSV(h,s,v) function calculates the appropriate single value, where H, V and V are values between 0.00 and 1.00.

Hue is the color, (usually in Degrees), Saturation is the density of color and Value is the brightness. Hue range is 0-Red through the Color Wheel to 1-Red, lower Saturation gives greyer color, higher Saturation gives vivid color and lower Value gives darker colors to black, higher Value are brighter.

The HSV(h,s,v) function may be used to obtain a value for when setting any Object's color attribute.

**RGB(r,g,b)** : converts 3 RGB primary color indices (Red, Green, Blue) into a single value that the parser understands

Lemur Color is represented by a single value from 0 to 8355711. The RGB(r,g,b) function calculates ((R*127 x 2^16) + (G*127 x 2^8) + B*127), where R, G and B are values between 0.00 and 1.00.

RGB color allows you to describe a color by it's component amounts of Red, Green and Blue.

The RGB(r,g,b) function is used to obtain a value for when setting any Object's color attribute .

| | |
|---|---|
| **selectinterface(index)** : | displays the selected Interface |
| **selecttab(object, index)** : | displays the selected tab (Container Object) |
| **setattribute(object, name, value)** : | sets an Object's attribute 'name' to the desired value |
| **setexpression(object, name, value)** : | sets an Objects's expression to the given value |
| **setobjectrect(object, rect[])** : | sets an Object's position and dimensions to {X,Y,W,H} |
| **show(object,state)** : | displays (state=1) or hides (state=0) an object |

## 13.10 Scripting Output Functions

**ctlout(target,ctrl,val,chan)** : outputs a Control Change MIDI message with specified Target, Controller number, value and channel.

**keycomboout(target,ctrl,alt,shift,key)** : outputs a simultaneous key strokes combination to the defined KbMouse Target.

**keyout(target,key,state)** : outputs a single key stroke to the defined KbMouse Target.

**midiout(target,msg[])** : outputs the defined MIDI message to the defined MIDI Target.

**mouselbutton(target,state)** : outputs a left mouse button click to the specified KbMouse Target.

**mousemove(target,x,y)** : outputs a x and y mouse movement to the specified KbMouse Target.

**noteout(target,note,vel,chan)** : outputs a Note On MIDI message with specified Target, Controller number, value and channel.

**oscout(target,address,args[])** : outputs the defined OSC command to the specified address and Target.

## 13.11 Vectorial Functions

**arraytostring(array[])** : converts an array to a string of values

**diff(a)** : returns the difference between the consecutive elements of an array

if a={w,x,y,z}

diff(a)={w, x-w, y-x, z-y}

Example:

diff({1,2,3,4,5,6,7,8})={1,1,1,1,1,1,1,1}

**fill(a,value,size)** : returns a vector with "size" items

the vector is filled from the left with n items = "value"

the number of items set to "value" depends on the a argument : a threshold between 0 and 1

when a = 0, there's no filling at all, and the vector is full of zeros

when a = 1, the vector is completely filled with items = value

when a = 0.5, the vector is half filled

Examples :

fill(1, 0.524, 4) = {0.524, 0.524, 0.524, 0.524}

fill(0, 0.524, 4) = {0, 0, 0, 0}

fill(0.5, 0.524, 4) = {0.524, 0.524, 0, 0}

**firstof(a)** : returns the position of the first non-null item in a vector

This is often used with switches in radio mode : firstof(x) returns the position of the enabled switch in the matrix

Examples:

firstof({0,0,1}) = 2

firstof({1,0,0}) = 0

note: if the vector only contains null items, the function returns the size of the vector

firstof({0,0,0}) = 3

**interlace(a,b)** : interlace 2 vectors to form a single vector

if a={1,2,3}, b={5,6,7}

interlace(a,b)={1,5,2,6,3,7}

**interlace3(a,b,c)** : interlace 3 vectors to form a single vector

if a={1,2,3}, b={5,6,7} and c={8,9,10}

returns {1,5,8,2,6,9,3,7,10}

**nonnull(array)** : returns the positions of non-null items in an array

nonnull({0,1,1,0})={1,2}

note: if the vector only contains null items, the function returns the size of the vector

nonnull({0,0,0}) = 3

**replace(a,b,position)** : takes an array, and replace the items starting at "position" with b ( b can be an array or a singleton )

Examples:

replace({0,0,0,0}, {12,2}, 0) = {12, 2, 0, 0}

replace({0,0,0,0}, 0.15, 3) = {0, 0, 0, 0.15}

If position is not an integer, the parser converts it using the floor function (next lower integer)

**set(a,value,position)** : takes an array, and change the item at "position" to "value" (position can be an array or a singleton)

Examples:

set({0,0,0,0}, 12, 0) = {12, 0, 0, 0}

set({0,0,0,0}, 12, {0,2}) = {12, 0, 12, 0}

If position is not an integer, the parser converts it using the floor function (next lower integer)

**sizeof(a)** : returns the size of a vector

**stretch(a,size)** : stretches a value or a range and returns a vector

If **a** is a single value (or a vector with a single item), the function returns a vector with "size" items containing that value

stretch(0, 12) = {0,0,0,0,0,0,0,0,0,0,0,0}

after that, you could do :

set(stretch(0, 12), 1, time%12), this returns:

{1,0,0,0,0,0,0,0,0,0,0,0} when time<1

{0,1,0,0,0,0,0,0,0,0,0,0} when time<2  0,0,1,0,0,0,0,0,0,0,0,0} when time<3 etc..

If **a** is a vector with more than one value, the function stretches the range over "size" items :

stretch({1,4}, 4} = {1,2,3,4}

stretch({0,10},11} = {0,1,2,3,4,5,6,7,8,9,10,11}

**subarray(array,start,length)** : chops an array into a defined sub-array

if a={1,2,3,4,5,6,7,8}

subarray(a,2,3)={3,4,5}

**sumof(a)** : sums all the items of an array

**wrap(a,size)** : creates a new array of length 'size', repeating the input array if necessary.

if a={1,2,3,4,5}, wrap(a,8)={1,2,3,4,5,1,2,3}

## 13.12 Trigonometric functions

**acos, asin, atan, cos, sin, tan,**

**log, log10, exp, pi** are the same as their mathematical counterparts.

**angle(x,y)** : returns the angle in radians formed by a vector of coordinates (x,y) to the positive x axis

Example:

Create a RingArea

Create a variable inside : a = angle(x-0.5, y-0.5)

this returns the angle position of the ball from the centre of the RingArea

**norm(x, y)** : length of a vector of coordinates (x,y)

Example :

create a Multiball with 2 balls

variable inside : distance = norm(x[1]-x[0], y[1]-y[0])

this returns the distance between ball 0 and ball 1

## 13.13 Operators

### Arithmetic

| | |
|---|---|
| **+** | add |
| **-** | subtract |
| ***** | multiply |
| **/** | divide |
| **%** | modulo |
| **++** | increment |
| **--** | decrement |

### Comparison

| | |
|---|---|
| **>** | greater than |
| **<** | smaller than |
| **>=** | greater or equal to |
| **<=** | smaller or equal to |
| **==** | equal to |

### Logic

When using logic and conditions in the parser, (**not 0**) is **true** and **0** is **false**.

| | |
|---|---|
| **!a** | NOT a |
| | 1 when a = 0 |
| | 0 when a!=0 |
| | a!=b returns 1 if a!=b (i.e a=2, b=4) or 0 if a==b |
| **a&&b** | logical **AND** |
| | 1 && 1 = 1 |
| | 1 && 0 = 0 |
| **a<b** | Less than : 1 if a < b |
| | 0 otherwise |
| **a>b** | Greater than : 1 if a>b |
| | 0 otherwise |

| | |
|---|---|
| **a==b** | Equal to : 1 if a==b |
| | 0 otherwise |
| **a>=b** | Greater than or equal to: 1 if a>=b |
| | 0 otherwise |
| **a<=b** | Less than or equal to: 1 if a<=b |
| | 0 otherwise |
| **a \|\| b** | logical **OR** |
| | 1 \|\| 0 = 1 |
| | 1 \|\| 1 = 1 |
| **a?b:c** | conditional statement, which translates as: “if (a is true) then b else c “ |

Example:

Creating an Expression "e" in a MultiBall and setting its value to *Pad.x?x:y* will mean that :

if Pad.x != 0 (i.e Pad is pressed), e=x

if Pad.x == 0 (i.e Pad is not pressed), e=y

You could for example map the e Expression to a Control Change mapped to the x axis when the pad is pressed, and y axis when th pad is released

## Assignment

Assignment operators allow the same variable name to contain different values at different times during the script execution.

| | |
|---|---|
| **=** | gets set to |
| **+=** | add and assign |
| **-=** | subtract and assign |
| **/=** | divide and assign |
| ***=** | multiply and assign |

## Bitwise

Bitwise operators allow operations on array patterns at the level of their individual elements.

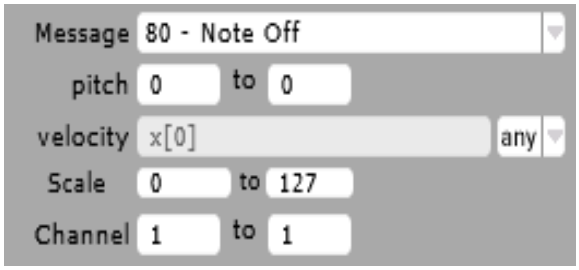
| | |
|---|---|
| **&** | binary **AND** |
| **\|** | binary **OR** |
| **<<** | left shift |
| **>>** | right shift |

# Chapter 14 - MIDI Mapping Message Reference

The MIDI section of the Lemur Editor Mapping Panel offers a wide range of MIDI messages your Objects' Variables can directly be assigned to. This chapter provide standard definitions of these messages as given by the MIDI specifications. Of course it is up to you to use the various messages to your liking. In a world of flexible and modular software, MIDI messages tend to lose their original meanings. In modular software like Reaktor or Max/MSP you can use any MIDI message to trigger or change anything you want.

## 14.1 Note Off

The Note Off message is sent when a key is released.
It is not widely implemented. Note On with a velocity of 0 is generally used to indicate a Note Off.




pitch: Dial in the MIDI note number. If the value is a vector the second field will be extrapolated automatically. You can override this setting by using your own value.

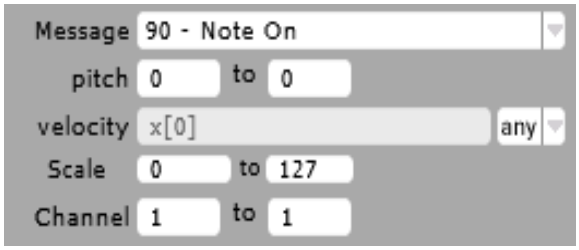
velocity: The Variable chosen from the Variable menu that is converted to velocity values is shown here.

Scale: Type in the desired target scaling for the MIDI messages. Possible values range from 0 to 127.

Channel: Dial in the MIDI channel. If the value is a vector the second field will be extrapolated automatically. You can always override this setting by putting in your own value. Possible values range from 1 to 16.

## 14.2 Note On

The Note On is sent when a key is depressed.
Note On with a velocity of 0 is generally sent when a key is released to indicate a Note Off.




pitch: Dial in the MIDI note number. If the value is a vector the second field will be extrapolated automatically. You can override this setting by using your own value.

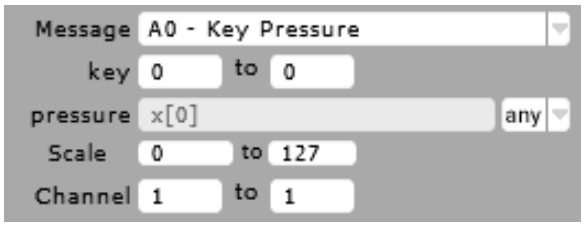
velocity: The Variable chosen from the Variable menu that is converted to velocity values is shown here.

Scale: Type in the desired target scaling for the MIDI messages. Possible values range from 0 to 127.

Channel: Dial in the MIDI channel. If the value is a vector the second field will be extrapolated automatically. You can always override this setting by putting in your own value. Possible values range from 1 to 16.

## 14.3 Key Pressure (Polyphonic Aftertouch)

Key Pressure, also called polyphonic aftertouch, gives the keyboarder a continuous controller for every key he had on his keyboard. It is used via pressing down on the different keys.
This works great for spreading out vector values to multiple key pressure values.




key: Dial in the MIDI note number. If the value is a vector the second field will be extrapolated automatically. You can override this setting by using your own value.

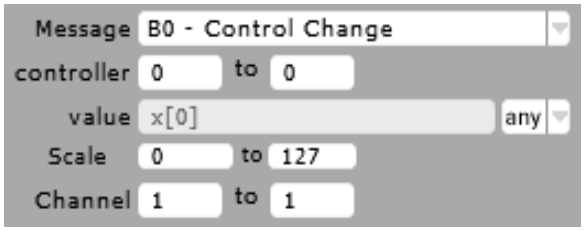
pressure: The Variable chosen from the Variable menu that is converted to key pressure values is shown here.

Scale: Type in the desired target scaling for the MIDI messages. Possible values range from 0 to 127.

Channel: Dial in the MIDI channel. If the value is a vector the second field will be extrapolated automatically. You can always override this setting by putting in your own value. Possible values range from 1 to 16.

## 14.4 Control Change

The Control Change data is used to produce pseudo-continuous data (ranging from 0 to 127).




controller: Dial in the MIDI controller number. If the value is a vector the second field will be extrapolated automatically. You can always override this setting by putting in your own value. Possible values range from 0 to 127

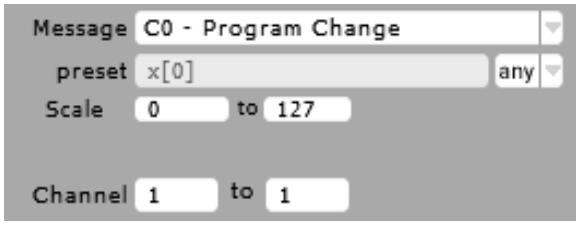
value: The Variable chosen from the Variable menu that is converted to the controller values is shown here.

Scale: Type in the desired target scaling for the MIDI messages. Possible values range from 0 to 127.

Channel: Dial in the MIDI channel. If the value is a vector the second field will be extrapolated automatically. Possible values range from 1 to 16.

## 14.5 Program Change

Used to change the patch number of synthesizers.



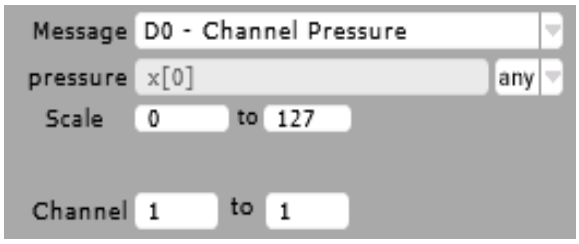

preset: The variable chosen from the Variable menu that is converted to program change values is shown here.

Scale: Type in the desired target scaling for the MIDI messages. Possible values range from 0 to 127.

Channel: Dial in the MIDI channel. If the value is a vector the second field will be extrapolated automatically. Possible values range from 1 to 16.

## 14.6 Channel Pressure

Another continuous controller used on a per MIDI channel basis.



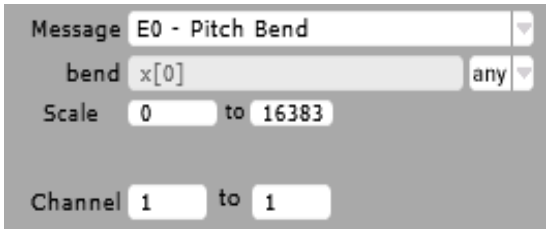

pressure: The Variable chosen from the Variable menu that is converted to channel pressure values is shown here.

Scale: Type in the desired target scaling for the MIDI messages. Possible values range from 0 to 127.

Channel: Dial in the MIDI channel. If the value is a vector the second field will be extrapolated automatically. Possible values range from 1 to 16.

## 14.7 Pitch Bend

PitchBend is used to indicate a change in the pitch wheel. It is a 14 bit value providing a higher resolution (0 - 16383) than normal 7 bit control change messages (0 – 127).



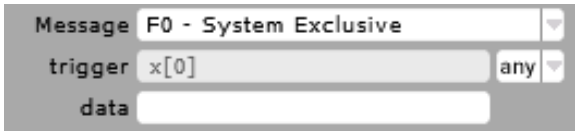

bend: The Variable chosen from the Variable menu that is converted to program change values is shown here.

Scale: Type in the desired target scaling for the MIDI messages. Note that pitch bend values range from 0 to 16383.

Channel: Dial in the MIDI channel. If the value is a vector the second field will be extrapolated automatically. Possible values range from 1 to 16.

## 14.8 System Exclusive

Used for custom data that is not covered by normal MIDI messages. The first part is usually a series of value representing an ID. If the device or software recognizes the ID as its own, it will listen to the rest of the message. Otherwise, the message will be ignored.

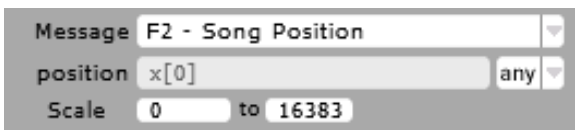



Trigger: The Variable chosen from the Variable menu that is converted to SysEx messages is shown here.

Data: Type in the desired SysEx string you want to send. Please have a look at the MIDI Target's documentation for details about possible SysEx messages.

## 14.9 Song Position

A 14 bit value that holds the number of MIDI beats (1 beat= six MIDI Timing Tick) since the start of the song.

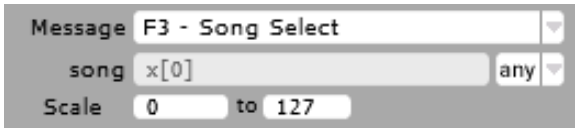



Position: The Variable chosen from the Variable menu that is converted to song position values is shown here.

Scale: Type in the desired target scaling for the position pointer. Note that song position pointer values range from 0 to 16383.

## 14.10 Song Select

The Song Select specifies which sequence or song is to be played.

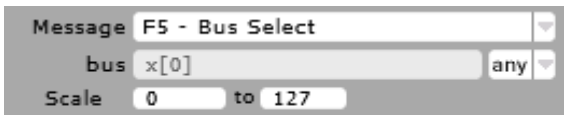



Song: The Variable chosen from the Variable menu that is converted to song number values is shown here.

Scale: Type in the desired target scaling for the song number. Possible values range from 0 to 127.

## 14.11 Bus Select

The Bus Select message specifies to which MIDI output further data should be sent.

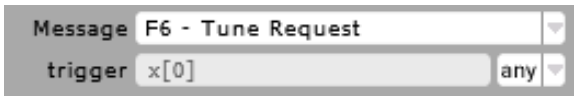



Bus: The Variable chosen from the Variable menu that is converted to bus number values is shown here.

Scale: Type in the desired target scaling for the bus number. Possible values range from 0 to 127.

## 14.12 Tune Request

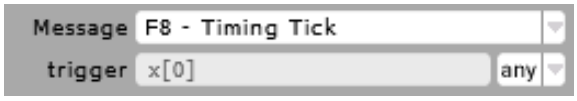
Used to trigger the tuning of oscillators in analog synthesizers.




Trigger:The Variable chosen from the Variable menu that is used to trigger tune request messages is shown here.

## 14.13 Timing Tick

Sent 24 times per quarter note for synchronization purposes. Build your own MIDI clock.




Trigger: The Variable chosen from the Variable menu that is used to trigger timing tick messages is shown here.

## 14.14 Start Song

Used to start the song or sequence from the beginning.


Trigger: The Variable chosen from the Variable menu that used to trigger start song messages is shown here.

## 14.15 Continue Song

Used to start the song or sequence from where it was stopped.


Trigger: The Variable chosen from the Variable menu that is used to trigger tune request messages is shown here.

## 14.16 Stop Song

Used to stop the song or sequence.


Trigger: The Variable chosen from the Variable menu that is used to trigger stop song messages is shown here.

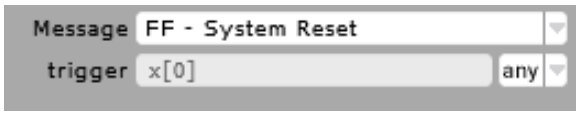
## 14.17 Active Sensing

When initially sent, the receiver will expect to receive another Active Sensing message each 300ms (max), or it will be assume that the connection has been terminated. At termination, the receiver will turn off all voices and return to normal (non-active sensing) operation.


Trigger: The Variable chosen from the Variable menu that is used to trigger active sensing messages is shown here.

## 14.18 System Reset

Resets all receivers in the MIDI chain to power-up status.




Trigger: The Variable chosen from the Variable menu that is used to trigger system reset messages is shown here.

# Appendix I – Keyboard Shortcuts

| | |
|---|---|
| New Project | Command/ Control+N |
| Open Project | Command/ Control+O |
| Save Project | Command/ Control+S |
| Save Project As | Command/ Control+ Shift+S |
| Import Module | Command/ Control+I |
| Export Module | Command/ Control+E |
| Lemur Settings | Alt+S |
| Lemur Synchro | Command/ Control+R |
| Lemur Connection | Alt+C |
| MIDI Map | Alt+M |
| Create Interface | Command/ Control+Shift+I |
| Create Object | Command/ Control+Shift+O |
| Create Expression | Command/ Control+Shift+E |
| Create Custom MIDI | Command/ Control+Shift+M |
| Create Executable Script | Command/ Control+Shift+X |
| Cut | Command/ Control+X |
| Copy | Command/ Control+C |
| Paste | Command/ Control+V |
| Undo | Command/ Control+Z |
| Redo | Command/ Control+Y |
| Toggle to Run mode – Execute. | Hold E |

# Appendix II – Object Attributes

Object Attributes are editable via the **setattribute(*object,'name',value*)** function or via an OSC command in the form ***/OSC_Addr/Object @attribute val*** *ie. /Container/Fader @grid 1*.

| | |
|---|---|
| Breakpoint | color, coordinates, editable, free, grid, grid_steps, label, name, nbr, physic, rect, zoom |
| Container | color, label, name, rect, tabbar, transparent |
| CustomButton | behavior, bitmap, capture, color, label_off, label_on, name, outline, rect |
| Fader | capture, color, cursor, grid, grid_steps, label, name, physic, precision, rect, unit, value, zoom |
| Knob | color, cursor, grid, grid_steps, label, mode, name, physic, precision, rect, type, unit, value |
| Leds | bargraph, color, colors, column, label, multicolor, name, rect, row, transparent |
| Menu | color, items, name, rect, scale, transparent |
| Monitor | color, label, name, precision, rect, transparent, unit, value |
| MultiBall | capture, color, colors, cursor, ephemere, grid, grid_steps, label, labels, multicolor, multilabel, name, nbr, physic, polyphony, pressure, rect, zoom |
| MultiSlider | bipolar, capture, color, colors, gradient, grid, grid_steps, horizontal, label, multicolor, name, nbr, physic, rect |
| Pads | capture, color, colors, column, label, labels, multicolor, multilabel, name, rect, row |
| Range | capture, color, grid, grid_steps, horizontal, label, name, physic, rect |
| RingArea | capture, color, label, name, rect |
| SignalScope | color, label, mode, name, rect, transparent |
| SurfaceLCD | color, display, name, rect, target, transparent |
| Switches | capture, color, colors, column, label, labels, multicolor, multilabel, name, paint, radio, rect, row |
| Text | color, content, name, rect, transparent |

These are the attributes reported by the getattributelist(Object) function.
The getattribute and setattribute functions are used to obtain or modify their values.

# Appendix II – Object Variables

Object Variables are accessable via dot notation ie if( Fader.z ==1) etc , and are set by assignment operators. ie. Fader.x =1

| | | |
|---|---|---|
| Breakpoint | x<br>y | Vector containing the points' horizontal positions<br>Vector containing the points' vertical positions. |
| Container | none | |
| CustomButton | x | The state of the CustomButton |
| Fader | x<br>z | Location of the cap. 0 is bottom/left -1 is top/right.<br>A 'flag' variable indicating whether the Fader is touched. |
| Knob | x | Current value of Knob – Absolute position or rotational count in Endless mode. |
| Leds | none | |
| Menu | selection | Current index of Menu selection |
| Monitor | none | |
| MultiBall | x<br>y<br>z | Vector containing the horizontal position of the balls.<br>Vector comtaining the vertical positions of the balls<br>Vector of brightness values of balls, in ephemeral mode. |
| MultiSlider | x | Value or Vector of vertical positions of the Sliders. |
| Pads | x | Value or Vector of the brightness value of the Pads. |
| Range | x | Vector containing the low and high positions of the Range |
| RingArea | x<br>y | The horizontal position of the RingArea ball<br>The vertical position of the RingArea ball |
| SignalScope | none | |
| SurfaceLCD | none | |
| Switches | x | Value or Vector of on-off values for Switches |
| Text | none | |

These are the inbuilt variables of the Lemur Objects
'Dot.notation' can be used to obtain or modify their values.

# Appendix III – Parser Quick Reference

## Operators

### Assignment

| | |
|---|---|
| = | to assign |
| += | add and assign |
| -= | subtract and assign |
| /= | divide and assign |
| *= | multiply and assign |

### Comparison

| | |
|---|---|
| == | equal to |
| > | greater than |
| < | smaller than |
| >= | greater than or equal to |
| <= | smaller than or equal to |

### Arithmetic

| | |
|---|---|
| + | add |
| - | subtract |
| * | multiply |
| / | divide |
| % | modulo |
| ++ | increment |
| -- | decrement |

### Bitwise

| | |
|---|---|
| & | binary AND |
| \| | binary OR |
| << | left SHIFT |
| >> | right SHIFT |

Bitwise operators allow operations on vector/array patterns at the level of their individual elements. Consult a programming reference for further details on their use.

### Logic

| | |
|---|---|
| &l& | logical AND |
| ! | logical NOT |

Logical Functions evaluate conditions to be 0 as False and 'not 0' as True.

| | |
|---|---|
| \|\| | logical OR |
| a?b:c | conditional expression |

Conditional expression is similar, but not identical to the if,then,else. statement

## Functions

### Arithmetic

| | | |
|---|---|---|
| abs(*a*) | Absolute value | |
| ceil(*a*) | Round-up to integer value | |
| clamp(a,min,max) | Constrain between values | |
| floor(*a*) | Round-down to integer value | |
| max(*a,b*) | Return highest value | |
| min(*a,b*) | Return lowest value | |
| pow(*a,b*) | Value *a* to the power of *b* | |
| rand() | Returns value 0 to 1 each frame | |
| range(a,min,max) | Maps a 0 to 1 value to *min,max* | |
| round(*a*) | Round to closest integer | |
| sign(*a*) | Returns -1 for negative, or + 1 | |
| sqrt(*a*) | Return the square root of *a* | |
| | | |
| **Internal** | | |
| accelerometer | iPad accelerometer {x,y,z} | |
| battery | iPad Battery level - 0 to 1 | |
| current_interface | Index of current interface | |
| midi_clocks | MIDI ticks variable – 24 ppqn | |
| time | mSec since turned on, resets hourly | |

| | | |
|---|---|---|
| **Others** | | |
| breakpoint([*x*],[*y*],*phase*) | Get position of point along segment | |
| findobject('*object*') | Select an Object in Project structure | |
| getattribute(*object,'name'*) | Get value of Object's 'attribute' | |
| getattributelist(*object*) | Get Objects list if attributes | |
| getexpression(*object,'name'*) | Get value of Object's 'expression' | |
| getfirst(*object*) | Get first Object in Project structure | |
| getnext(*object*) | Get next Object in Project structure | |
| getobject() | Get reference to self Object | |
| getobjectrect(*object*) | Get Object position and dimensions | |
| getparentobject(*object*) | Get Object's Parent | |
| HSV(*h,s,v*) | Lemur Color in HSV values (0 - 1) | |
| RGB(*r,g,b*) | Lemur Color in RGB values (0 - 1) | |
| selectinterface(*index*) | Display an Interface | |
| selecttab(*container,index*) | Display a tab of a Container | |
| setattribute(*object,'name,'value*) | Set Objects 'attribute' to value | |
| setexpression(*object,'name',value*) | Set an Objects expression to value | |
| setobjectrect(*object*,rect[]) | Set Object position and dimensions | |
| show(*object,state*) | Display or hide an Object | |
| | | |
| **Trigonometric** | | |
| acos, asin, atan, cos, sin tan, log, log10, exp, pi | Standard mathematical functions | |
| angle(x,y) | Angle in Radians | |
| norm(x,y) | Length of vector co-ordinates | |
| | | |
| **Vectorial** | | |
| arraytostring(*array*[]) | Converts the *array*[] to a String | |
| diff(*array*) | Difference between *array*{} elements | |
| fill(*array,value,size*) | Fills an *array* with *value* to *size* | |
| firstof(*array*) | Position of first non-null or size | |
| interlace(*a,b)* | Interlace 2 vectors | |
| interlace3(*a,b,c*) | Interlace 3 vectors | |
| nonnull(*array*) | Vector os positions of non-null items | |
| replace(*array,b,position*) | Replace items in *array* with *b* | |
| set(*array,value,position*) | Set *position* in *array* to *value* | |
| sizeof(*array*) | Returns of the *array* | |
| stretch(a,size) / (array,size) | Stretch value or range to a vector | |
| subarray(*array,start,length*) | Subarray of *array* of *length* from *start* | |
| sumof(*array*) | Sum of all items in *array* | |
| wrap(*array,size*) | Wrap an *array* to a *size* | |

# Appendix IV – MIDI Quick Reference

Lemur is MIDI compliant and is capable of sending and receiving all standard MIDI messages.

The following MIDI messages may be configured directly via the MIDI Mapping panel.

| | | | | |
|---|---|---|---|---|
| **MIDI MESSAGE** | **Hex** | **Data 1** | **Data 2** | **Remarks** |
| Note Off | 80 | Note: 0 - 127 | Velocity: 0 - 127 | Channel 1 - 16 |
| Note On | 90 | Note: 0 - 127 | Velocity: 0 - 127 | Channel 1 - 16 |
| Key Pressure | A0 | Key# 0 - 127 | Value: 0 - 127 | Channel 1 - 16 |
| Control Change | B0 | CC# 0 - 127 | Value: 0 - 127 | Channel 1 - 16 |
| Program change | C0 | Pgm# 0 - 127 | None | Channel 1 - 16 |
| Channel Pressure | D0 | Value: 0 - 127 | Value: 0 - 127 | Channel 1 - 16 |
| Pitch Bend | E0 | 0 – 16383 | | 14 bit : Channel 1 - 16 |
| System Exclusive | F0 | Device ID data | Sysex Data | |
| Song Position | F2 | 0 – 16383 | | 14 bit : 1 beat = 16 Timing Ticks |
| Song Select | F3 | 0 - 127 | None | |
| Bus Select | F5 | 0 - 127 | None | |
| Tune Request | F6 | None | None | Request Tuning Data |
| Timing Tick | F8 | None | None | 24 pulses per quarter note |
| Start Song | FA | None | None | Start song or sequence |
| Continue Somg | FB | None | None | Continue song or sequence |
| Stop Song | FC | None | None | Stop song or sequence |
| Active Sensing | FE | None | None | |
| System Reset | FF | None | None | Reset all MIDI devices |

It is also possible to construct more complex MIDI messages via Lemur's scripting environment. RPN/NRPN, 14bit CC's are all easily prgrammed. MIDI Out can also be directly programmed.

It is beyond the scope of this document to provide in-depth information for all these messages and their possible use. Please refer to the MIDI Manufacturers Association http://www.midi.org/ for the official MIDI specification, or the documentation of the software or hardware you are using for more specific information.

| **MIDI Implementation Chart**<br>Liine Lemur | | | |
|---|---|---|---|
| | **Transmit** | **Recognise** | **Remarks** |
| **1. Basic Information** | | | |
| MIDI channels | 1 - 16 | 1 - 16 | |
| Note numbers | 0 -127 | 0 - 127 | |
| Program change | 0 – 127 | 0 - 127 | * Response is scriptable |
| Bank Select | Yes | Yes | Via CC #0 - * Response is scriptable |
| Note-On Velocity | Yes | Yes | * Not Velocity sensitive - scriptable |
| Note-Off Velocity | Yes | Yes | * Not Velocity sensitive - scriptable |
| Channel Aftertouch | Yes | Yes | * Not Aftertouch sensitive - scriptable |
| Poly (Key) Aftertouch | Yes | Yes | * Not Aftertouch sensitive - scriptable |
| Pitch Bend | Yes | Yes | 14 bit value |
| Active Sensing | Yes | Yes | * Response is scriptable |
| System Reset | Yes | Yes | * Response is scriptable |
| Tune Request | Yes | Yes | * Response is scriptable |
| Universal System Exclusive: | Yes | Yes | Transmit via Custom MIDI or via scripting |
| NRPNs | Via Scripts | Via Scripts | MIDI Out is directly scriptable |
| | | | |
| **2. MIDI Timing and Synchronization** | | | |
| MIDI Clock / Timing Tick | Yes | Yes | 24 ppqn – pulse per quarter note |
| Song Position Pointer | Yes | Yes | 14 bit value – 1 beat=16 Timing Ticks |
| Song Select | Yes | Yes | 0 - 127 |
| Start<br>Continue<br>Stop | Yes<br>Yes<br>Yes | Yes<br>Yes<br>Yes | * Response is scriptable |
| MIDI Time Code | | | |
| MIDI Machine Control | | | |
| MIDI Show Control | | | |
| | | | |
| **3. Continuous Controllers** | | | |
| Control Number | All | All | 0 – 127 : Bidirectional control |
| | | | |
| * Not Velocity sensitive - scriptable | The Lemur Objects do not respond to velocity of finger taps,<br>but varying velocity values may be sent via scripting. | | |
| * Not Aftertouch sensitive - scriptable | The Lemur Objects do not respond to pressure of finger taps,<br>but varying aftertouch values may be sent via scripting. | | |
| * Response is scriptable | These messages are recognised and available for scripting.<br>They are not natively implemented but their response is scriptable.<br>ie. Lemur does not have 'Banks' and 'Programs' but this could be scripted. | | |

# Appendix V – Specifications

**General**

The Liine Lemur iOS app runs on iOS devices running a minimum recommended iOS version 4.2. This is required for CoreMIDI compatibility and support.

Lemur supports OSC and MIDI protocols over Wi-Fi. Lemur also supports CoreMIDI. You can use any standard MIDI interface through the Apple Camera Connection Kit, or an iOS specific audio interface such as iConnect MIDI. Some devices may not support all features such as sysex data.

Lemur supports up to 16 different and simultaneous MIDI and OSC port targets - 8 MIDI & 8 OSC.

**Specification**

Certain technical limitations exist within the Lemur environment. Some of these figures will be addressed in the near future, some later, some never.

- Menu object is currently set to maximum number of 32 items.
- Expression text length is currently set at maximum of 256 characters.
- Vector/List length in a script variable is currently set at a maximum of 256 elements.
- Multiline script length is currently set at a maximum of 4096 characters.
- Maximum number of entries in an array is currently set to 256 (maybe less when dealing with long entries);
- MIDI System Exclusive message maximum size is currently 256 bytes.
- Maximum size limits also apply to OSC messages due to underlying transport packet size.
- Lemur Editor file size limit is currently set to approximately 16MB per project. This file size limit is actually a different matter than the memory usage piechart.

  The resulting file size is just the number of characters necessary to describe the project in XML format. One project with many large MultiSliders will typically use more RAM, but that will not necessarily translate to a large file size when saved to disk.

Exceeding these limits may lead to unpredictable behavior, or unacceptable syntax errors.

Also note, there is no mechanism at the time for OnLoad scripts to figure out which external variables should be evaluated first. Scripts may run even though variables have not been filled yet, i.e its own tiny script hasn't yet executed. Use variable declared internal to the script instead.

Dependencies on variables that hold constant numerical values, or arrays of constant numerical values, pose no problem as they are recognized as such during script compilation. Unfortunately strings and arrays of strings are not recognized as constants. (That's why they don't display in blue in the project browser)

Bottom line is, OnLoad scripts relying on external numerical constants should all work fine, but there can be issues when depending on more complex variables, or other scripts even.

# Changelog

**Lemur 3.1/ Editor 3.1**

**2012/04/02**

Daemon: Added a warning on startup when port 8001 is already busy
Editor: Fixed a bug where objects get renamed with an infinitely long name
Editor: Added template stretching when switching resolution
Editor: Fixed a crash when setting a manual IP
Editor: Copy/Paste text from external application now works
Editor: Maximum JZML size raised to 16MB
Lemur: Fixed remembering OSC targets
Lemur: Added display of Interface name in Interface tabs
Lemur: Fixed a crash when sending a very long OSC list to Lemur
Lemur: Increased maximum length of SysEx messages to 256 bytes
Lemur: Increased maximum length of lists in scripts to 256 elements
Lemur: Fixed bug in Knob object when tapping center
Lemur: Fixed MultiBall bug where z variable was stuck is x and y variables were set from a script
Lemur: Improved the referencing of identically named objects exist in different scopes

**Editor 3.0.4**

**2012/02/29**

Editor: Fixed the 512kb limit for templates. Limit is now 2MB.

**Lemur 3.0.3 / Editor 3.0.3**

**2012/02/17**

Daemon: GUI not displaying if Daemon was launched before switching graphics cards on Lion.
Editor: Crash on drag-and-drop jzml from Finder
Editor: New default colors implemented
Editor: Color picker displays blank rectangle when reverting to 2D
Editor: Crash when duplicating an interface
Editor: Objects are deleted when template is using approximately more than 16% of memory
Lemur: Knobs could not be set to Linear mode
Lemur: Setting speed as vector for Breakpoint was broken.
Lemur: Allow tap-through of Liine ident video on app startup
Lemur: Default colors did not match in the iPad
Lemur: Breakpoint live edit mode seems was broken
Lemur: Parser HSV(x,y,z) function was broken