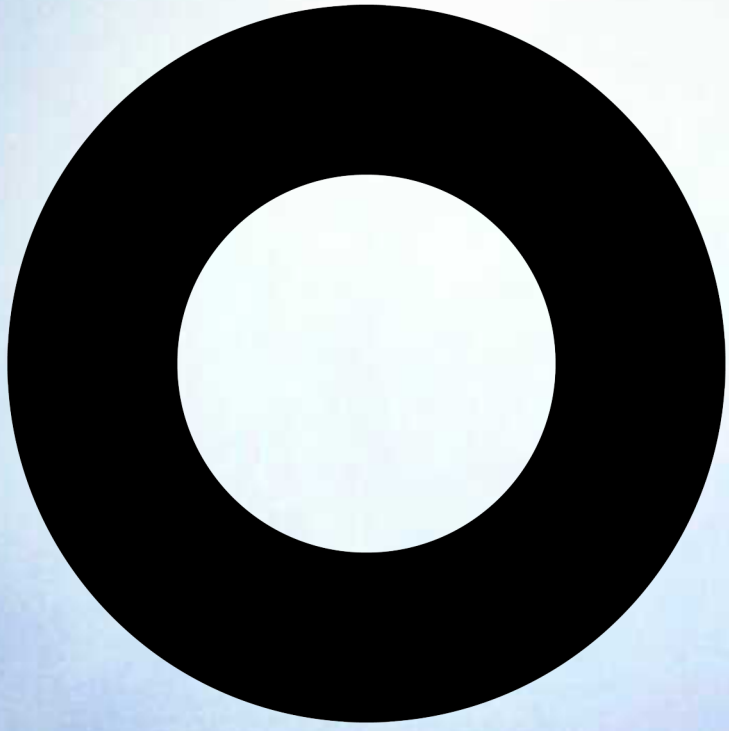
Max for Live
Ultimate Zen Guide



written by Julien Bayle
edited by Mark Towers

Ableton Certified Trainer

Max for Live Ultimate Zen Guide

Become a Max for Live Master and discover a new way of using Ableton Live

Julien Bayle

This book is for sale at http://leanpub.com/Max-for-Live-Ultimate-Zen-Guide

This version was published on 2013-12-05

ISBN 978-2-9546909-4-0

Tweet This Book!

Please help Julien Bayle by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Max for Live Ultimate Zen Guide written by @julienbayle & reviewed by @markmsb / #ableton #book #m4l

The suggested hashtag for this book is #m4lbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#m4lbook

Contents

Introduction

I'm **Julien Bayle**[1] and I have used Ableton Live Digital audio workstation since its first release. I was certified by Ableton[2] in 2010 and I teach not only **Ableton Live Suite**[3] but **Max6** and **Max for Live** both to individuals and groups. I have released **an eight and a half hour long training DVD about Ableton Live 9 & Push**[4]

Push & Live on the field

Max for Live was released in **2011**, and I have had lots of requests to write a book since then. Here we are! It is starting now and I hope that you will be totally happy and satisfied after reading this guide. I want you to be able to design your own Audio & MIDI FX but also your own sound generators and custom interfaces inside Ableton Live !

In the first part of this guide, I'm going to explain Max for Live basics: what **Max for Live** exactly is, what the differences with **Max6**, formerly named **Max/MSP**, are and also how it works with **Ableton Live** itself.

In the second part, we will look at something already known by some of you: the famous (but cryptic) **Live Object Model**, also named **LOM** within *Max for Live's Coding Mafia*. It is the inner structure of Live itself. We can reach, request and manipulate Live's parameters thru it by using two ways: **Max for Live** itself, but also the (also famous) **MIDI Remote Scripts**. We will explore Live's Application Programming Interface (also named Live API) and how to program it.

---

[1] My website: http://julienbayle.net

[2] ableton.com: https://www.ableton.com/en/education/certified-training/france/julien-bayle-marseille

[3] My courses: http://julienbayle.net/ableton-live/

[4] My DVD about Ableton Live 9 & Push: http://www.elephorm.com/audio-mao/formation-ableton-live/ableton-live-9.html#a_aid=julienbayle

In the third part, I will talk about **JavaScript** as another way to reach and work with Live's API. You'll like this part because it will push you into another field of programming within Max for Live.

In the fourth and final part, I will show you how we can easily design and build Max for Live devices like MIDI instruments, sound generators, MIDI Fx and Audio Fx and also how we can play and control Live using API from within the devices.

You can also follow me on twitter[5] & facebook[6]

Are you ready?

[5] https://twitter.com/julienbayle
[6] https://www.facebook.com/julien.bayle

About

Being published on leanpub.com[7], **this guide can easily be revised and updated. You will **automatically** be notified about any updates in the future.**

Stay Tuned.

[7] https://leanpub.com/Max-for-Live-Ultimate-Zen-Guide

About the review & edit of the book

This book has been entirely reviewed and edited by **Mark Towers** and I want especially thank him for such involvement, serious, expertise and kindness !

Mark Towers teaching Ableton Live in Leicester

Mark Towers is an Ableton Certified Trainer from the UK, he runs a Foundation Degree in Creative Sound Technology at Leicester College/De Montfort University, and teaches Ableton, Sound Design and Programming with Max 6 and Max for Live. Mark also specialises in Electronic Music Composition & Performance.

You can visit his website at marktowers.net[8]

[8] http://marktowers.net

FAQ

Version

The guide version is 1.05

1.05

- better quality for snapshots
- some typos fixed

This guide is not

- a whole course about Max6
- a whole course about Ableton Live
- a whole course (nothing can beat a course thru skype or face-to-face: ask my students!)

This guide is

- a guideline with the Max for Live world
- the guide you need to completely understand what Max for Live is and how it works with Ableton Live itself.
- THE tutorial which will drive you to create your own Max for Live devices perfectly fitting inside Ableton Live (presets' storage, parameters' automation, Live API etc)
- THE ONLY tutorial published & written by an Ableton Certified Trainer about Ableton Live

If you find a mistake or a typo in this guide

Please write to me using this page: http://julienbayle.net/contact[9]

If you would like to find out about courses on Ableton Live, Max6

Please write to me using this page: http://julienbayle.net/contact[10]

[9] http://julienbayle.net/contact
[10] http://julienbayle.net/contact

1 Max for Live Basics

Max for Live isn't a concept or a magical spell. There are lots of video tutorials out there that don't really explain it, so I will do that and I'm going to explain precisely what it is.

1.1 What ?

My own Max for Live definition is:

Max for Live is an **interactive visual programming environment** directly usable within Ableton Live for designing devices.

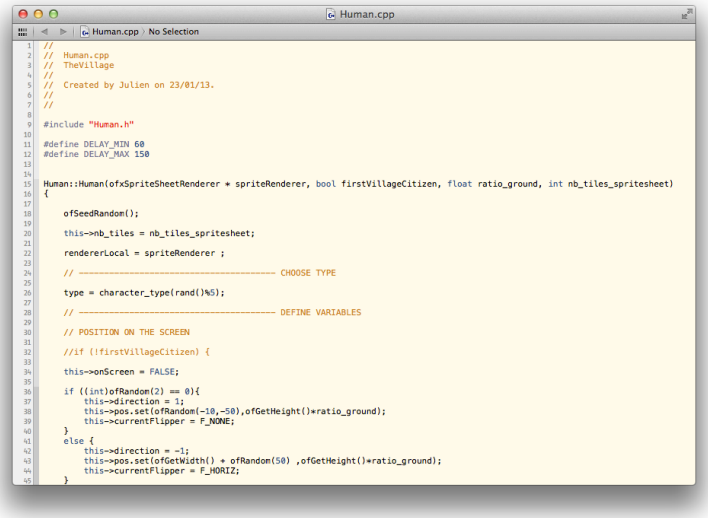
We are going to review Ableton Live & we will revise it in the following pages, but at the moment, let's begin by explaining what **interactive visual programming environment** means.

What is an interactive visual programming environment ?

There are a lots of programming languages. Each one of them have their own characteristics and paradigms which are basically the rules they are following and we have to follow in order to use them.

Here is, for instance, an example of something I coded in C++ language for a generative installation[1]:

[1]Generative Digital installation named The Village / Where2Now:http://julienbayle.net/works/collaboration/the-village



C++ language is based on text coding

This type of programming language requires a specific proramming workflow:

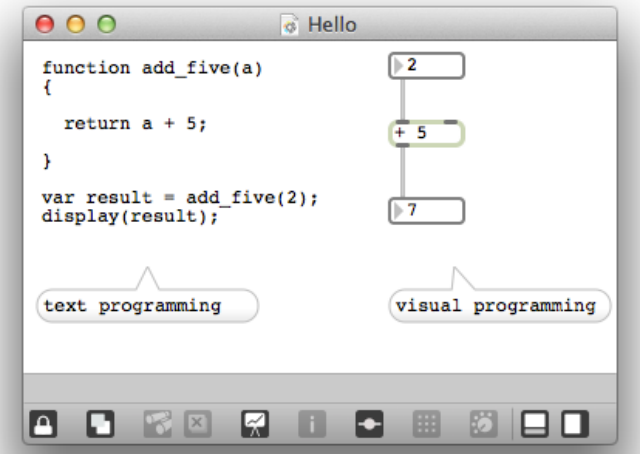
- we write the code
- we include some external programming libraries
- we compile
- we test

This series of steps is done cyclically: we modify, we compile, we modify again etc. There is something I like to call a **design disruption** that comes from this necessary compilation step which requires to stop coding and to compile & test before we begin to code again.

An **interactive visual programming environment** is interesting because it doesn't have any **design disruption**. As soon as we modify the program, we are seeing directly the consequences of our modification

A visual environment also provides a way to learn programming more easily and less abstractly. Indeed, we don't have to create complex abstract structures for which we would have to remember each part & detail but we are moving objects and linking them together.

Here is an example of what I mean with a picture:



A Javascript text *code* and its *visual* equivalent

On the left, there is a piece of JavaScript code. We are declaring a function named **add_five** which requires an argument. Then, this function is called with the number 2 as an argument and the **result** is stored inside the variable result which is displayed using the function **display()** that is not visible defined and declared here. **display()** is only here as an example showing that we have to call or use functions to display things in usual text-based programming language because the UI is often separated from the code itself (UI means User Interface).

On the right, the same thing but designed with Max6. There are two UI objects providing both features of displaying and modifying quantities. The object **[+ 5]** is the equivalent of the function **add_five** and adds 5 to any values incoming *from the top*. Then, **[+ 5]** emits the calculation result *to the bottom*.

BUT

I don't mean that one type is better than the other.

Neither one has less features than the other.

Interactive visual programming is just simpler to learn and less abstract than the other, especially for people with no programming background.

Max6 & Max for Live

Some of you will have already heard about **Max6**, formerly named **Max/MSP**.

Max6 is the latest version of what long-time users still call Max/MSP.

In the following pages, I will use the term *Max* to denote *Max6* in order to mean *Max6 environment*. But I will use *Max for Live* too, because it is very different from *Max* itself as we are going to find out.

Max vs Max for Live

I'm going to explain why Max for Live is not equivalent to Max6.

Max framework & Max runtime Max is a development environment available here: http://cycling74.com/shop[2]

Max is an autonomous framework that doesn't require you to own an Ableton Live license, and even if you had one, it wouldn't be necessary to launch Live to use Max.

When we program with Max, we create **patches** that we can edit & modify. A **patch** is a set of objects more or less linked together.

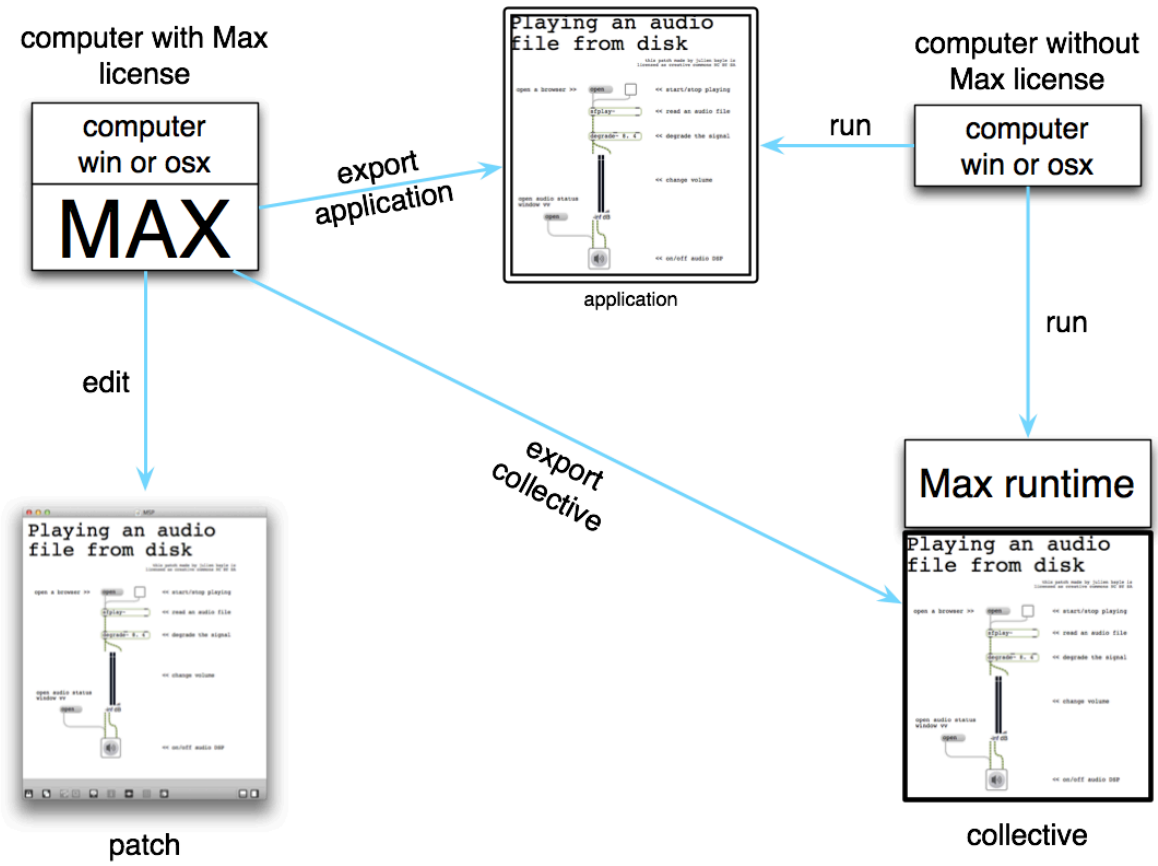
If we want to share a patch with another user who doesn't own a Max'license himself, we can create an autonomous Max **application**. This **application** is basically our patch, a set of dependencies (all objects required by the patch to be loaded) and the Max' **runtime**.

runtime is available for free here:

http://cycling74.com/downloads/runtime[3] in case we want to launch and use a **collective**. A **collective** is just the patch with all dependencies, but without the runtime.

Check the next figure, it illustrates how this works:

[2] http://cycling74.com/shop
[3] http://cycling74.com/downloads/runtime



Max, Max Runtime, Patch, Collective and Application

Ableton Live & Max for Live If you understand what the previous figure means, you won't have any problem to understand the next pages.

Max for Live doesn't provide any runtime.

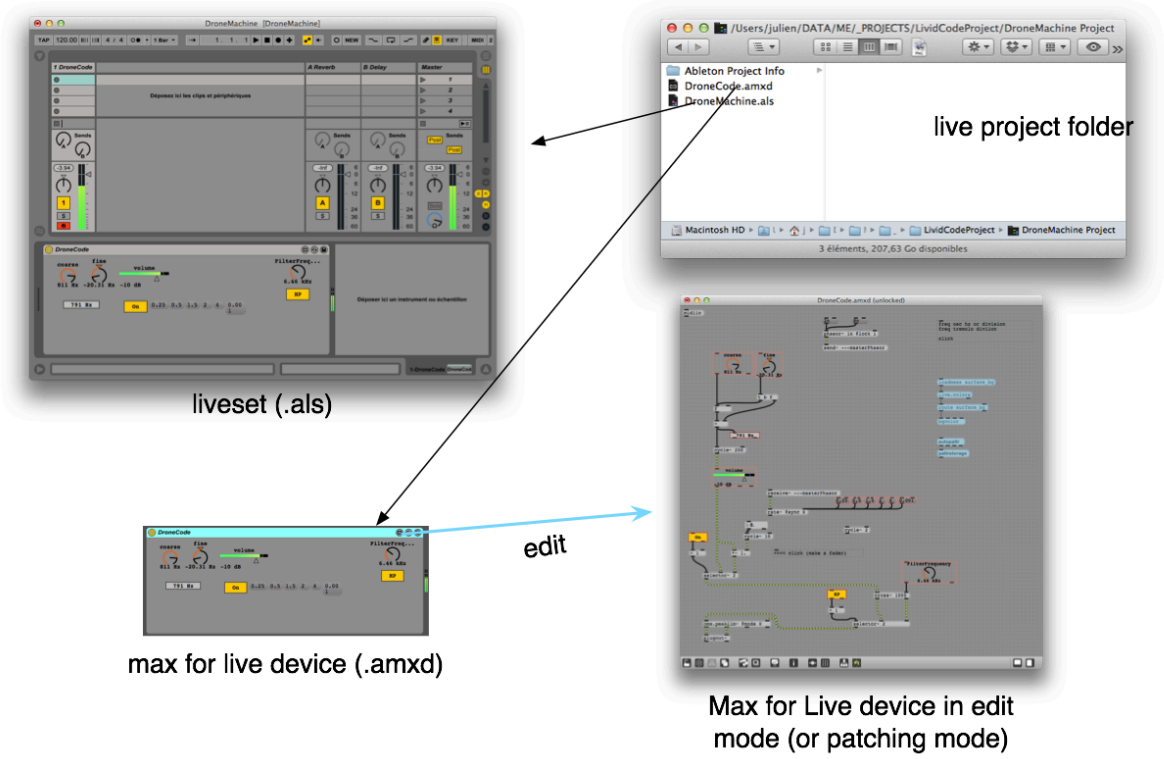
More precisely, the only software able to open, edit & run Max for Live patches is Ableton Live.

Actually, we refer to these patches as **Max for Live Devices** instead of just saying Max for Live patches because these latter sit within Max for Live devices.

These devices are stored as files containing patches required by them to work, and also all dependencies required too, especially in the case of those devices that have been frozen. We will take a closer look at this later in this book.

For now, we are going to explore another figure in order to understand clearly what these different entities are:

- **liveset** (file with the extension **.als** storing all liveset's data like clips' parameters, devices' parameters, MIDI mapping and much more)
- **liveset project** (folder containing one or more livesets, samples related to these livesets and much more)
- **device** (file with the extension .amxd containing all elements required by the patch to be loaded and used as a device in Live)
- **device in edit mode** inside a ... Max-style editing window



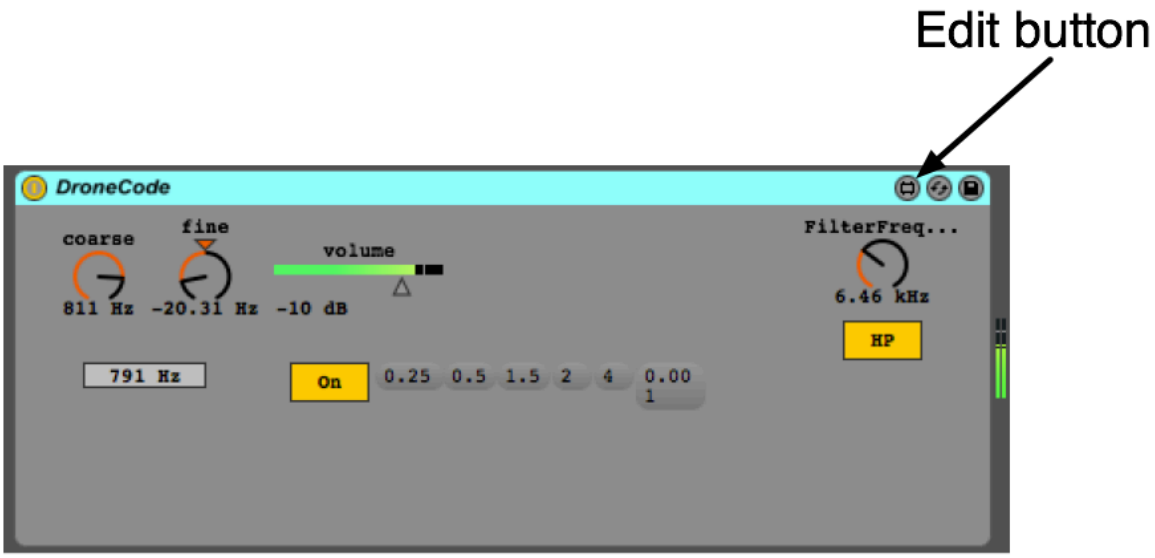
Liveset, liveset project, device and device in edit mode

You could think of Max for Live's runtime as being Ableton Live itself and Max for Live's editor as being Max run by Live.

One of the first direct consequence is :

> as soon as we load a Max for Live device, we can also edit it !

You can also click on the **edit** button of a Max for Live device and you'll see the underlying Max patcher open and provide you with the whole patch related to this device.



Max for Live's device EDIT button

Now we are a bit more aware of how Max for Live, and Max & Ableton Live are related, let's continue with licensing.

Use & licensing

In order to use Max for Live, we have to own an Ableton Live license, obviously. But not only, of course.

I only own Max for Live Then, I can use and edit Max for Live's devices.

But I cannot use Max outside of Ableton Live, I mean: as the pure Max framework.

I only own Max I CANNOT use Max for Live's devices, nor edit them, of course.

Obviously, I can use Max independently outside of Live or even connected to Live using MIDI or audio by using audio inter-application routing application like **Soundflower**[4] on OSX or **Jack**[5] on OSX or Windows.

I own Max & Max for Live I can use Max for Live's devices.

I can also use Max patches independently of Live and make them communicate with Max for Live's devices using OSC, for instance.

[4]Soundflower: http://cycling74.com/products/soundflower
[5]JACK: http://jackaudio.org

1.2 Ableton Live Basics

Ableton Live is an **audio & MIDI workstation**. We call this kind of software a **DAW** (standing for Digital Audio Workstation).

Let's explore what Ableton Live is a litte more.

Projects & livesets

A **project** is a **folder** inside of which we can store livesets.

It can contain multiple livesets and it is a convenient way to organize our music.

We can find imported audio sample files and also samples created within the liveset itself. When we use the feature *Collect All & Save*, all presets and elements required by the liveset are copied inside this project folder too in order to make it totally independent from any external files.

I'd suggest reading the Live manual if you want more details about these fundamentals.

Tracks

Live's livesets are structured around the **tracks**' concept, whatever the mode we use (session or arrangement modes)

A track can be an audio or a MIDI track.

Each track can contain a set of clips which are the most basic unit inside of which we can have MIDI sequences or audio samples, depending on the type of track. MIDI tracks can contain MIDI clips and audio tracks can contain audio clips.

In a track, only one clip can be playing at a time.

Each track also contains a mixer and a send FX section, and of course a devices' chain that can be seen at the bottom of the screen in both modes.

An **audio track** can contain:

- **audio clips** inside the clips area (clipslots in session mode or timeline in arrangement mode)
- **audio effects** inside the devices' chain

A **MIDI track** can contain:

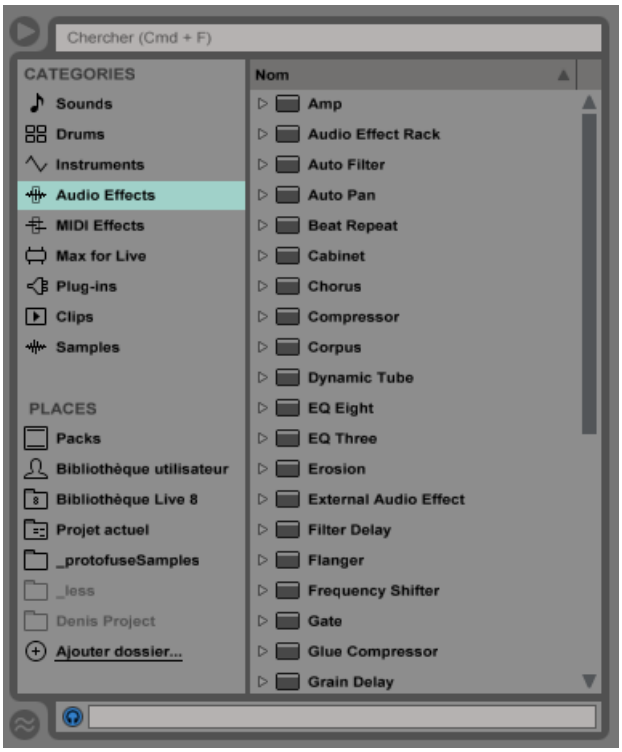
- **MIDI clips** in the clips area (clipslots in session mode or timeline in arrangement mode)
- **MIDI effects** inside the devices' chain, before the MIDI instrument
- **MIDI instrument** inside the devices' chain
- **audio effects** inside the devices' chain, after the MIDI instrument

Devices

We have refered to devices since the beginning of this book but we didn't define them…

Devices are those entities that we can drag from Live's browser and drop on to the tracks' device chain.

We can see in the browser (on the left) a big list of categories. Version 9 of Live can be a bit confusing at first sight.



Live's Browser

We find the different devices within each of the following categories:

- Instruments
- Audio Effects
- MIDI Effects
- Max for Live
- Plug-ins

Category names are quite self-explanatory.

In the **Instruments** category , we can find Live's native instrument devices like Operator, Analog or Sampler. They receive MIDI messages from Live itself through MIDI clips or from outside (from

another application within the same computer or from another computer or MIDI controller) and can produce audio depending on the MIDI messages received.

In the Audio Effects category: there are Live's native audio effects like delays, compressors or beat-repeat. They can process audio signals.

In the MIDI Effects category: there are Live's native MIDI effects like Pitch, Chord and some others. They can transform MIDI messages.

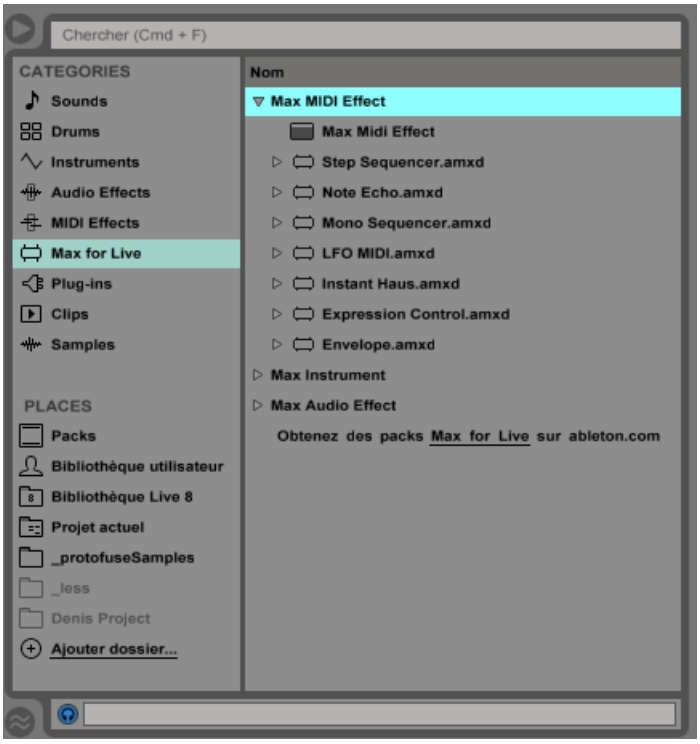
The Plug-ins category contains all non-native devices like VST for OSX and Windows or Audio-Units (AU) for OSX plugins. They are add-on devices usable in Live but that are not delivered with Live or even designed by Ableton.

Then, we have the Max for Live category.

There are 3 types of devices in this category:

- Max MIDI Effect
- Max Instrument
- Max Audio Effect

We have the same classification in 3 categories.

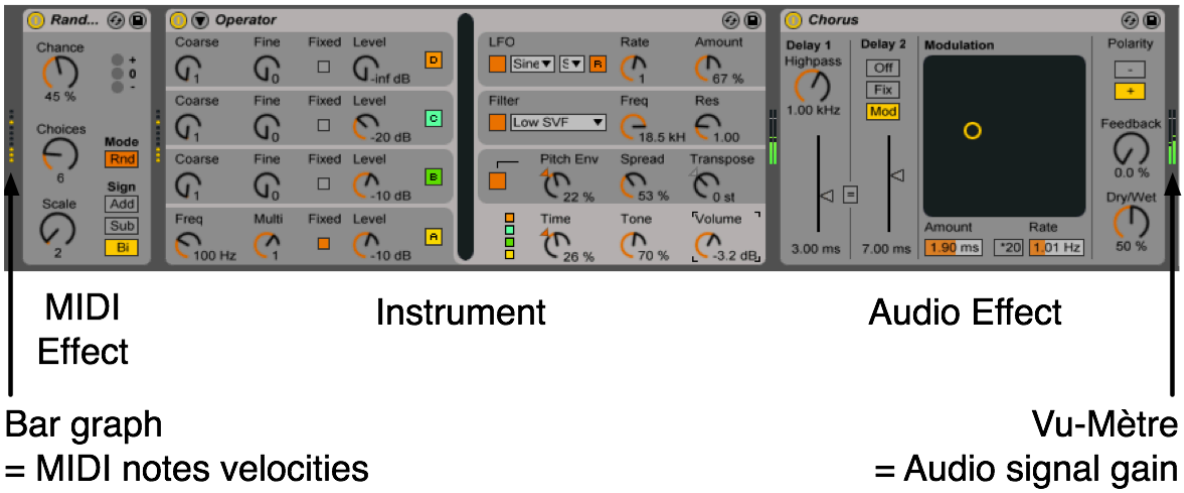


Max for Live's 3 different types

Why?

Because any device that we want to use has to be placed in a device's chain in Live. And, as we have just seen, a device's chain belongs to a track and a track can be either a MIDI or an audio track with all consequences raised.

When I'm teaching Ableton Live, I use the following image to illustrate the concept of MIDI effects, MIDI Instruments, and Audio Effects, look at this now:



MIDI & Audio in Live's devices chain

Bar graphs represent **MIDI** signals and **VU-meters** represent **audio signals**.

If we start from the left that is the entry point of a signal inside the devices' chain (the signal can come from inside Live itself or outside, as I said), we can see a **Bar graph**.

Because this device is included in-between two Bar graphs, that means we have a **MIDI Effect** here. It takes an incoming MIDI signal and outputs a MIDI signal: this is the basic definition of a MIDI Effect.

With the same reasoning, we can see that the next device is in-between a Bar graph on its left and a VU-Meter on its right. It receives a MIDI signal and produces an audio signal: this is a **MIDI Instrument**.

Then, the last device is in-between two VU-Meters. It is an **audio Effect**.

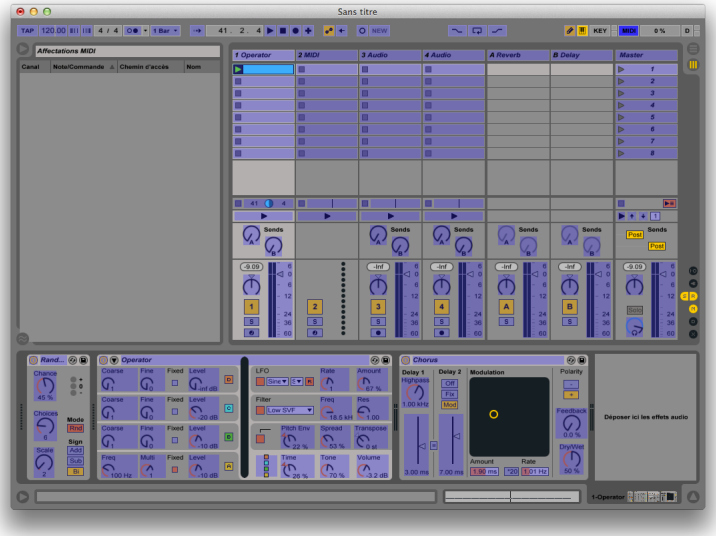
In the same way we are going to discover that Max for Live's devices are nothing but Live's devices, with specific features like edit mode of course, and this is why they have to be from one type or another such as MIDI effect or audio effect etc.

We are also going to see that in the case of a Max for Live device that doesn't have to process MIDI or audio but to be an interface between Live and a *Wiimote* controller for instance, we can use any of the 3 types. Indeed, this won't be important except for the place where we will want to place it in Live's device chain.

Let's continue with Live's basics.

MIDI Mapping

MIDI Mapping provides a way to associate MIDI messages like **Note** or **Control Change** with a livesets' parameters through **Live's GUI** (GUI stands for Graphical User Interface)



MIDI Mapping Edit Mode enabled

We can enable the **MIDI learn mode** by clicking on the MIDI text button at the top right in Live's GUI (or by pressing **CMD + M on OSX** & **CTRL + M on Windows**)

Once enabled, we can select either a parameter, elements like clipslots, but also scene selectors, we can then move a knob on our hardware controller. Then the mapping/association is made: the liveset parameter is linked to the hardware controller's knob.

PAY ATTENTION HERE

The liveset's MIDI mapping is stored inside the liveset itself and **nowhere else** !

Question: what can we do if we want a particular set of parameters to be mapped each time to the same set of knobs, whatever the liveset used ?

Indeed, because we may be playing our live performances quite often, we want to be able to retrieve our setup when we are working on a liveset, on stage etc. without having to remap everything everytime.

Of course, we can keep the same liveset and copy-paste it as a template. As we just saw, MIDI Mapping parameters will be copied with the liveset's file itself. It works fine. A lot of people do that. But we can do better by using **MIDI Remote Scripts**.

By the way, be a little bit more patient here: Max for Live is coming. But we have to keep all of that in mind before we dive in completely !

MIDI Remote Scripts

Some of you probably already know about **MIDI Remote Scripts**.

We are talking about **scripts** for programming languages that don't require a proper compilation step.

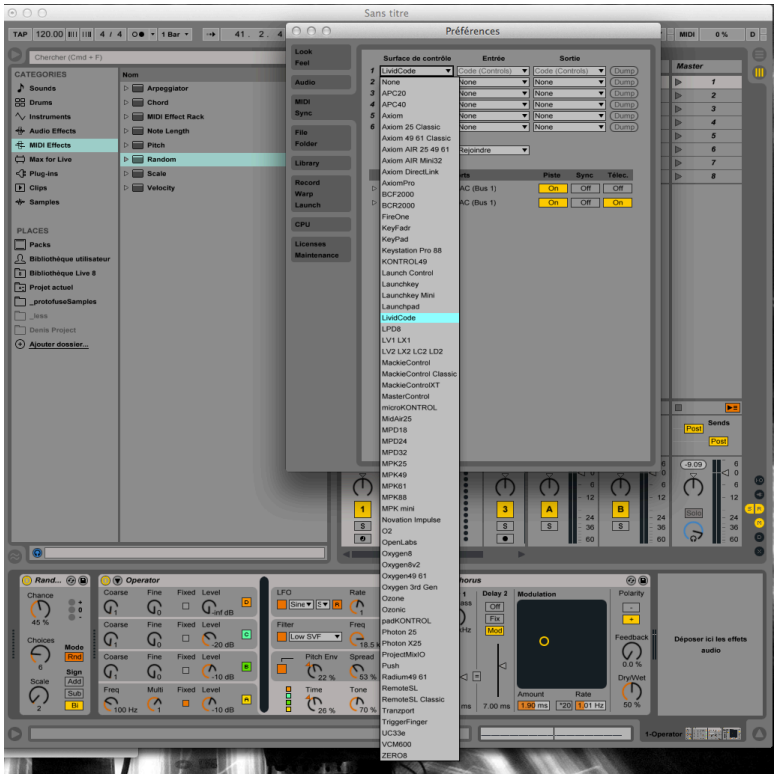
When we are typing JavaScript code, for instance, we don't need to compile it before to test it: an interpreter can run it directly. JavaScript is often used within web pages with HTML and in that case, the interpreter is just our web browser itself.

What are those scripts ?

MIDI Remote Scripts are Python-based scripts. Python is a programming language. They are directly interpreted (and run) by Ableton Live itself and provide ways for:

- controlling Live (act on Live)
- grab feedback from Live (observe Live)

By checking the **MIDI Sync tab** in **Live's preferences**, we can see in the top level part some parameters related to **Control Surfaces** (sometimes referenced as **Remote Surfaces**)



Enabling & Disabling MIDI Remote Scripts in Live

Each member of the list corresponds to a MIDI Remote Script available from Live's installation folders.

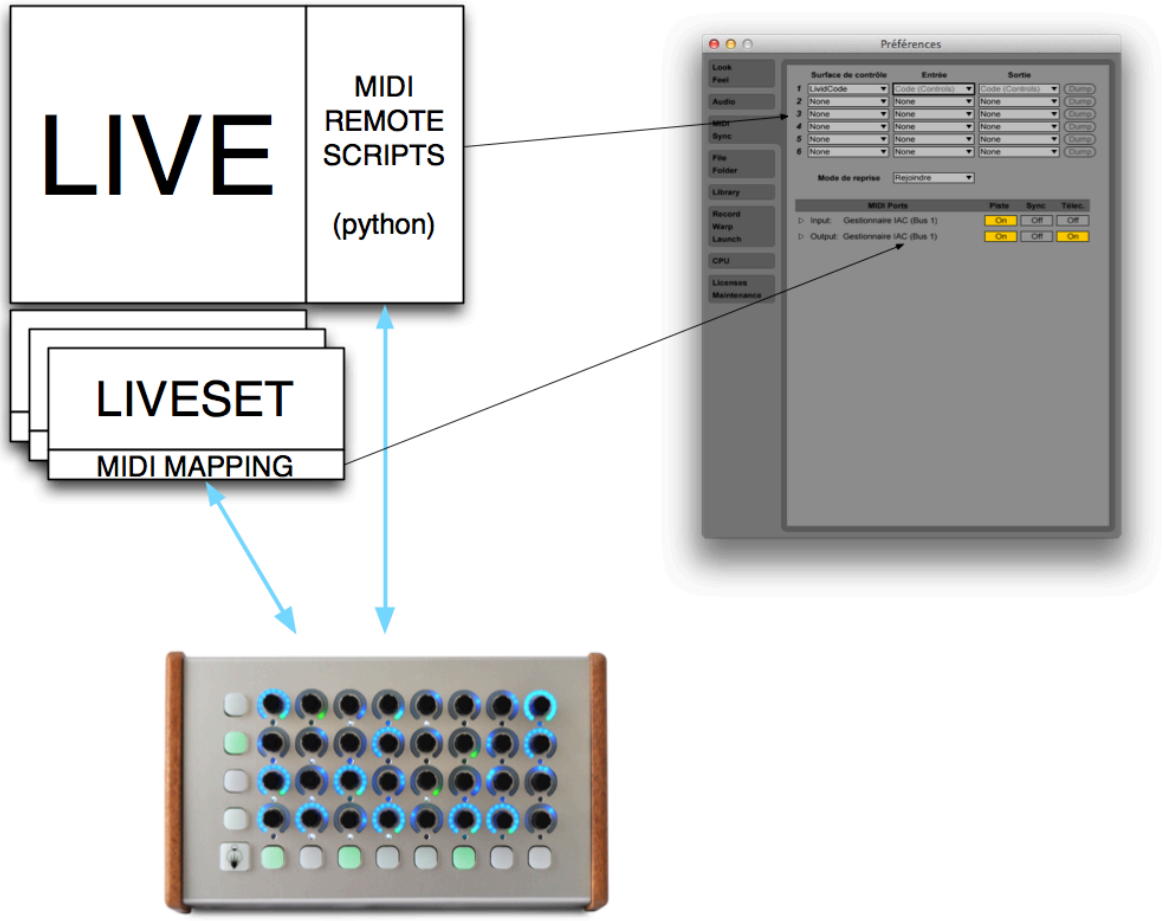
Each one is named after a controller. We can see APC20, APC40, Push and also LividCode, for instance. The latter involves the Livid Instruments' Code controller on which I am currently working on (a secret project, by the way)

Generally, these scripts are designed by the hardware's designers themselves for hardware handling in Live.

For instance, APC40's script provides a way to directly control the clip matrix using its buttons and also to grab all clips' states.

Basically, these scripts are an interface between these controllers and Live itself.

Here is a minimalistic schematic summarizing these concepts.



Livid Code controlling Live through MIDI Remote scripts OR classic MIDI mapping

Black arrows show the relationships between MIDI Mapping and MIDI Remote Scripts concepts AND Live's Preferences window.

In **blue**, we can see the control/feedback themselves.

In the **MIDI Remote Scripts** method, we can see that the configuration is depending on Live itself, but in the **MIDI Mapping** method, the configuration is based on the liveset itself. Of course, in this latter case, the MIDI interface configuration is depending on Live itself too, but please keep in mind that MIDI Mapping IS stored inside the liveset's file (the **.als** file)

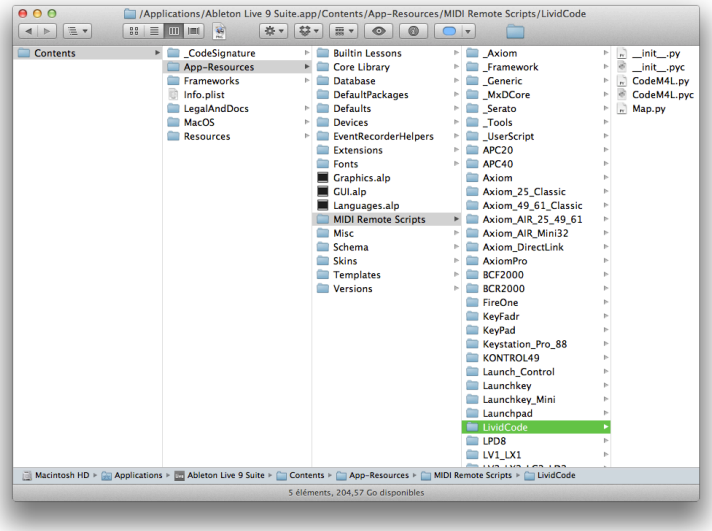
Where can we find the MIDI Remote Scripts on our computer?

These scripts are within Live's folder hierarchy.

On windows:

- C:ProgramDataAbletonLive xResourcesMIDI Remote Scripts\ on **Windows 7 & Vista**
- C:Documents and SettingsAll UsersApplication DataAbletonLiveX ResourcesMIDI Remote Scripts\ on **Windows XP**.

On OSX, we have to find the file **Live.app** in **Macintosh HD/Applications/Live x.x.x./** and right-click on it, then choose *Show Package Contents*. At last, if we look inside this folder: **Contents/App-Resources/MIDI Remote Scripts**, we can see this:



MIDI Remote Scripts on OSX

Wouldn't it look like the previous Remote Surfaces list we saw in Live's preferences ? Of course, it is directly related.

Hack and Script' edit

In the previous snapshot, we can see the opened folder **LividCode** and some files inside it with the extensions: **.py** and **.pyc**.

.py & .pyc file format The file extension **.py** is related to Python scripts' source codes. We can easily read and modify them. These are basic text files.

Actually, if we have a better look at the content of the folder, we can also see **.pyc** files. Those aren't editable. They are binary files.

These are bytecodes[6]. They can be considered as executables, binary files are directly executable by the Python interpreter, which is Live itself, as we already discovered.

If we modify a Python script source code, the next time we launch Live, and choose the corresponding Remote Surface in Live's Preferences, Live will take the **.py** source code and will pre-interprete it (this is a kind of pre-compilation) in order to optimize the execution and will create this **.pyc** bytecode file (one for each **.py** used)

Retrieve source codes from bytecodes Obviously, as we are more or less hackers, we want to read what is in these source code **.py** files. Indeed, we first need to understand how the Live API works and how we can design our own MIDI Remote Scripts.

As an early 2013 beta-tester for Ableton's PUSH controller, I especially wanted to read the source codes for this controller and I decompiled bytecodes early February 2013.

I published the whole sources and I'm doing my best to maintain and update them as soon as a new version of Live is released.

Everything is available here on Github:

https://github.com/gluon/AbletonLive9_RemoteScripts[7]

I know this helped and still helps a lot of you and especially application developers like those who made Conductr[8] for iPad.

Because nothing is officially documented, I tried to make some documentations:

- Push & hacks[9]
- MIDI Remote Scripts[10]
- Doc for python Live object[11]
- Docs for _Framework & _Tools objects[12]

I won't teach you how to program using Python language here, but how to do the same things by using Max for Live itself.

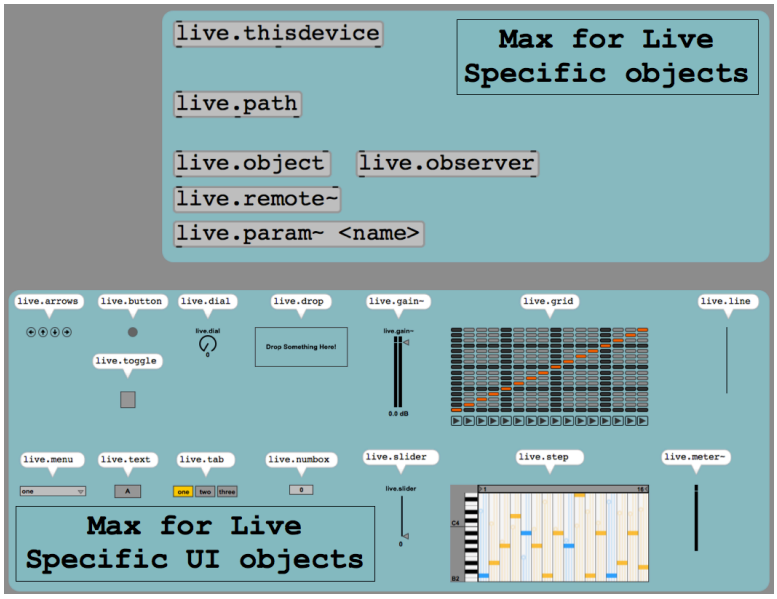
Let's dive now inside Max for Live.

[6] http://en.wikipedia.org/wiki/Bytecode
[7] https://github.com/gluon/AbletonLive9_RemoteScripts
[8] Conductr for iOS: http://www.conductr.net
[9] http://julienbayle.net/ableton-push
[10] http://julienbayle.net/ableton-live-9-midi-remote-scripts
[11] http://julienbayle.net/ableton-live-9-midi-remote-scripts/#liveAPI
[12] http://julienbayle.net/Ableton-Live-Scripts-Documentation

1.3 Max for Live specific objects

Max for Live contains a specific set of objects available while patching. Some of the objects are only usable within Max for Live, but others can be used in Max too, outside of Live, I mean.

Here is a global picture with specific objects of Max for Live.



Max for Live specific objects

We can see **two series of objects**:

- at the top, objects able to provide **control/feedback** from Live itself,
- at the bottom, Graphical User Interface's objects (**GUI**)

I will describe objects at the top in the next chapter.

live.arrows, **live.button**, **live.text** & **live.tab** are button type objects. We push, we select.

live.dial is basically a potentiometer, a knob.

live.number displays values but also provides a way to modify them.

live.menu presents a list to the user who can select an element in it.

live.drop makes the connection between a Max for Live device and the file system of the computer. We can drag'n'drop files on to this area and this object pops out the absolute path of those files. It is really interesting if we want to process files and to load them to memory [buffer~] objects in Max for Live for instance. There is no need to type a path, it finds it for you.

live.gain~ displays the audio signal level and provides a way to change the gain too.

live.meter~ only displays the audio signal level.

live.slider is a basic fader.

At last, **live.grid** & **live.step** are helpers and GUI to configure other objects.

live.grid works well with **chucker~** by providing a display and control to cut an incoming signal into pieces and to replay those pieces in another order.

live.step is a step sequencer.

We will talk about these objects in the next pages of the book because it is very important to know how to use each of them.

Because Live brings some features I want you to master, for instance, if you want to create a Max for Live device and be able to automate some of its parameters in Live, or to save them with the liveset, you have to apply specific actions.

Now, let's keep focused and explore the famous Live Object Model (LOM)

2 Live Object Model

This part of the book is **very important** and I'm going to try to explain and describe all of the following concepts as simply as possible in order to help you understand all that you'll need to build your own interfaces using Max for Live.

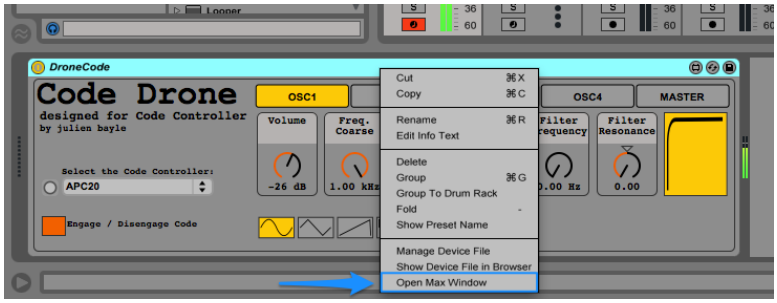
By the term interfaces, I mean devices that can interact with Live itself by **requesting data** from it (like grabbing the tempo in real-time and sending it as an OSC-transported value to another program for instance) but also by **controlling** Live itself (change the value of a parameter, change the transport state, etc)

At first, please keep in mind and in sight the Max window. We can display it in Max (and Max for Live) by typing:

- CMD + M on OSX
- CTRL + M on Windows

We can easily display data in the Max window like debugging information, but also log some data flows. It is a useful way to visualise data flows and learn how things work.

We can also display the Max window directly from Max for Live's devices **title bar**. By right-clicking on them and choosing the last option at the bottom **Open Max Window**, we can show it.



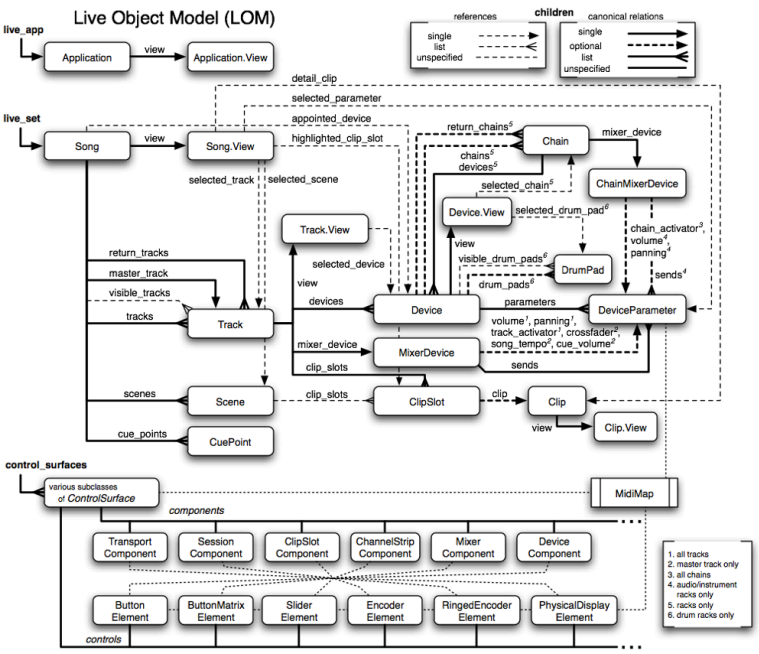
Opening the Max window from the device

2.1 What is LOM ?

The Live Object Model represents Live's internal structure. I mean, in the program, in the software itself. Actually, it would be easier to say: it is a map that guides us to everything accessible within the Live application through programming.

At first sight, we can see some elements which we already know about like **tracks**, **clips**, **devices**

Check this **big schematic** provided in the Max documentation in the specific Max for Live part:



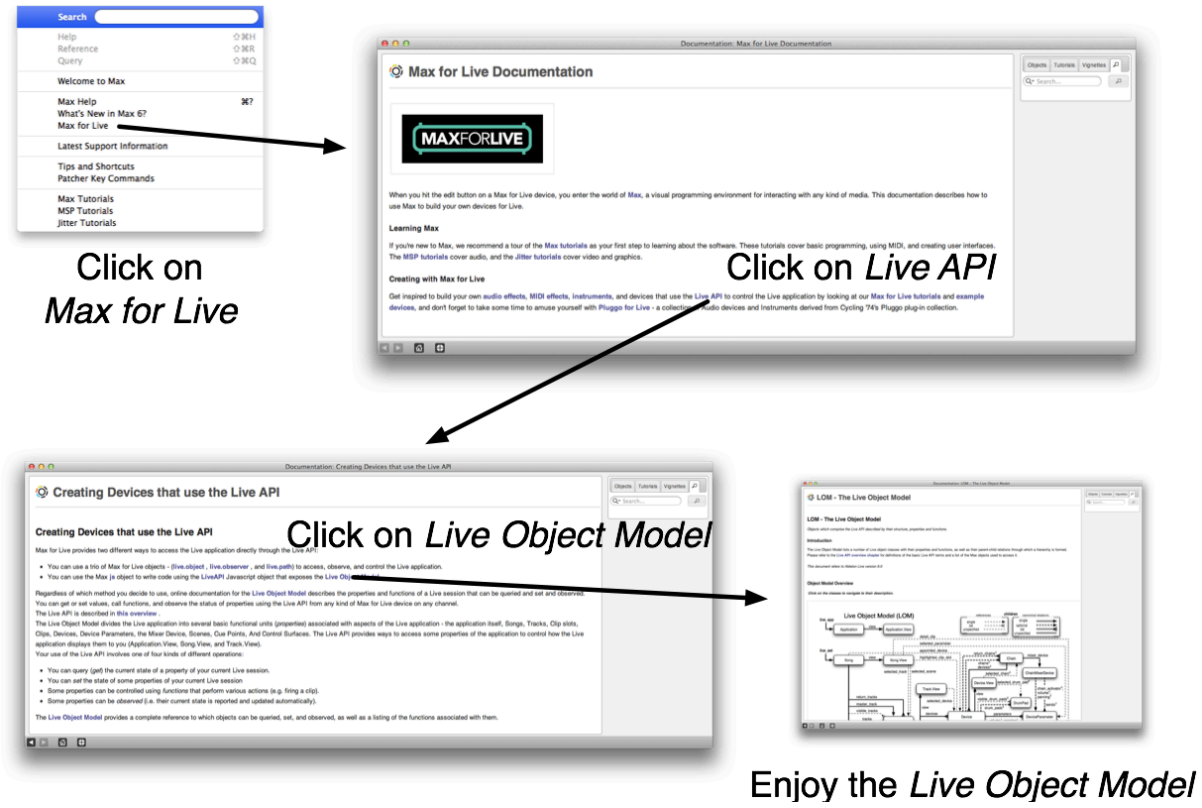
Live Object Model as available in the documentation by Ableton and Cycling'74

It looks cryptic and complex. It is a bit complex, but not that much!

We can also find this big picture on-line here[1]

If you want to work with this schematic under your eyes, you can easily reach it via the Max' help:

[1] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live_object_model



Reach the Live Object Model in the documentation in 3 clicks only

As well as being a big schematic overview, the Live Object Model also includes text on the same documentation page related to the schematic.

When we click over the **different rounded corner rectangles** in the schematic, we go directly to the text section corresponding to the rectangle.

Those rectangles are what we call **objects** (the *O* of LOM)

2.2 LOM's Objects

Objects are an elementary entity required by anyone who wants to use Max for Live and the LOM.

There are 19 objects available:

- Application & Application.view
- Song & Song.view
- Track & Track.view
- Clipslots
- Clip & Clip.view
- Device & Device.view

- DrumPad
- Chain
- ChainMixerDevice
- DeviceParameter
- MixerDevice
- Scene
- CuePoint
- ControlSurface

These objects provide access to specific elements of Live, visible or not, as we are going to find out. Each object has its own characteristics & properties. For instance, **tempo** is a property of the **Song** object. The looped or not looped state of a clip is defined as a property of the **Clip** object.

Application & Application.view

These objects are related to the Live application itself. For instance, if we want to display a user interface in a Max for Live device depending on the version of Live, we can request this property to the object **Application**.

The notation & suffix **.view** only means that the object is *a little bit more* related to elements of the graphical user interface of Live itself (and not exactly the liveset itself). ".view" suffixes also appear for other objects than Application.

Song & Song.view

Song is related to the liveset itself.

Track & Track.view

Track is related to the tracks themselves. Actually, it is related to one track only, each track being addressed by multiple instances of the object Track.

Clipslot

Clipslot is related to slots inside which we can put clips into, in session view.

Clip & Clip.view

These objects are related to **clips** themselves and how they can be displayed.

Device & Device.view

These objects are related to **devices** themselves and how we can display them.

DrumPad

DrumPad is related to the **drumpad of drumracks**. It is quite specific, but very useful.

Chain & ChainMixerDevice

These ones are about **Racks** and especially Racks' **chains**.

DeviceParameter

DeviceParameter Devices's parameters are all the parameters (dials, faders, buttons) available in Max for Live's UI devices.

MixerDevice

This one defines a **track's mixer**.

Scene

This one defines scenes. A scene is a row in the clips' grid in session mode.

CuePoint

This one defines arrangement mode's **markers**.

ControlSurface

ControlSurface is an interesting object providing a way to access MIDI controllers through MIDI Remote Scripts but from Max for Live. I especially worked on that for Livid Instruments and their Code Controller. You can check this page[2] for more information.

[2] http://julienbayle.net/livid-code

2.3 How to read the documentation

Each object is described and broken down in the same way.

Let's inspect the object **Scene** as an example:

Scene

This class represents a series of clip slots in Live"s session view matrix.

Canonical Path

live_set scenes N

Children

Name	*Type*	*Access*	*Description*
clip_slots	list of ClipSlot	get, observe	

Properties

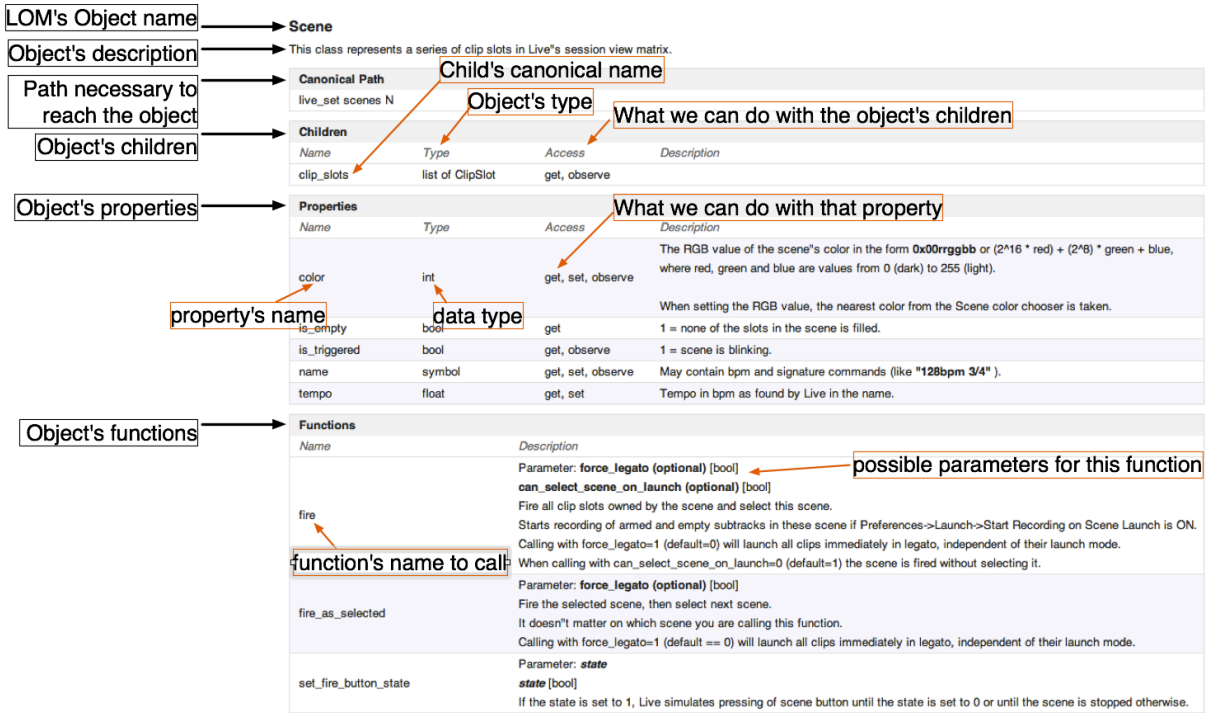
Name	*Type*	*Access*	*Description*
color	int	get, set, observe	The RGB value of the scene"s color in the form **0x00rrggbb** or (2^16 * red) + (2^8) * green + blue, where red, green and blue are values from 0 (dark) to 255 (light). When setting the RGB value, the nearest color from the Scene color chooser is taken.
is_empty	bool	get	1 = none of the slots in the scene is filled.
is_triggered	bool	get, observe	1 = scene is blinking.
name	symbol	get, set, observe	May contain bpm and signature commands (like **"128bpm 3/4"**).
tempo	float	get, set	Tempo in bpm as found by Live in the name.

Functions

Name	*Description*
fire	Parameter: **force_legato (optional)** [bool] **can_select_scene_on_launch (optional)** [bool] Fire all clip slots owned by the scene and select this scene. Starts recording of armed and empty subtracks in these scene if Preferences->Launch->Start Recording on Scene Launch is ON. Calling with force_legato=1 (default=0) will launch all clips immediately in legato, independent of their launch mode. When calling with can_select_scene_on_launch=0 (default=1) the scene is fired without selecting it.
fire_as_selected	Parameter: **force_legato (optional)** [bool] Fire the selected scene, then select next scene. It doesn"t matter on which scene you are calling this function. Calling with force_legato=1 (default == 0) will launch all clips immediately in legato, independent of their launch mode.
set_fire_button_state	Parameter: ***state*** ***state*** [bool] If the state is set to 1, Live simulates pressing of scene button until the state is set to 0 or until the scene is stopped otherwise.

Scene object's documentation

What does this all mean? Lets take another look:



Scene object's documentation explained

We can always see the shaded elements, even if there is no info/object listed.

2.4 Children, Properties & functions

Parent & Children objects' relationships

Children object concept is fairly straight forward: some objects have some children objects.

For instance, a liveset has tracks. Track instances are the children of the object Song (i.e. liveset). There is a hierarchical relationship between tracks & a song within the Live model.

Another example: tracks own clipslots and these latter can contain clips or not.

We can ask an object for a list of its children.

It is really important to grasp this concept in order to navigate inside the LOM.

Properties

Objects' properties are very varied and are intimately related to the object itself.

For instance, the object Scene owns a property defining its color and another one defining its name.

In the property part of the documentation, we can see something related to the *accessibility* of this property

There are 3 types of access to properties:

- get
- set
- observe

get is the way to request the current value of a property when we request it. This is a one-shot access.

set means we can change the value of the property.

observe means we can observe the property. It allows us to constantly monitor a value. By using this feature, as soon as a property's value changes, we are informed of its new value.

We are shortly going to learn how to use these 3 ways of accessing properties.

But before, we have to learn how to navigate within the LOM.

Functions

A function is an action launched through a considered object.

For instance, the function **fire** launched for a clip will trigger it with all consequences resulting. For instance, if the clip's launch mode is **toggle** and the clip is already playing, calling the **fire** function will stop it exactly as if we were to click on the small triangular button on the clip itself in Live's GUI.

2.5 Path, ID & navigation

In order for us to access and apply actions to LOM's objects such as property value modifications or function calls, we need to find the instance of the object that we want to work with.

Object / Instance & ID

An object is an abstract definition of a more concrete element of Live in the LOM.

An object *instance* is an effective materialization of a LOM *object*.

For instance, a *car* could be an object. It defines the global concept of what a car is. And *that specific green car* near my home is an instance of a car. It is a concrete and special materialization of the car concept, of the car object.

When we are playing with the LOM, we need to reach precise object instances.

Within Max for Live, the way to reach object instances can be done by using unique IDs which describe each instance of each object, said more simply: In Max for Live, we can reach objects' instances by using their internal IDs

These IDs are created and set as soon as we start to reach the corresponding instances. I mean, nothing is defined by default before this action.

For instance, if I'm trying to reach the Master's volume by requesting it through the LOM, the system is creating an ID, a way to reach it. For the whole Live session, this ID is used for that precise LOM object. It also can persist through sessions (i.e. between two different launches of the Live application)

How can I find the ID of the object I want to reach ?

We have just learned that we cannot predict IDs values before the first request to each LOM objects.

So we need a tool for this and it is the Max for Live object named **[live.path]**.

[live.path]

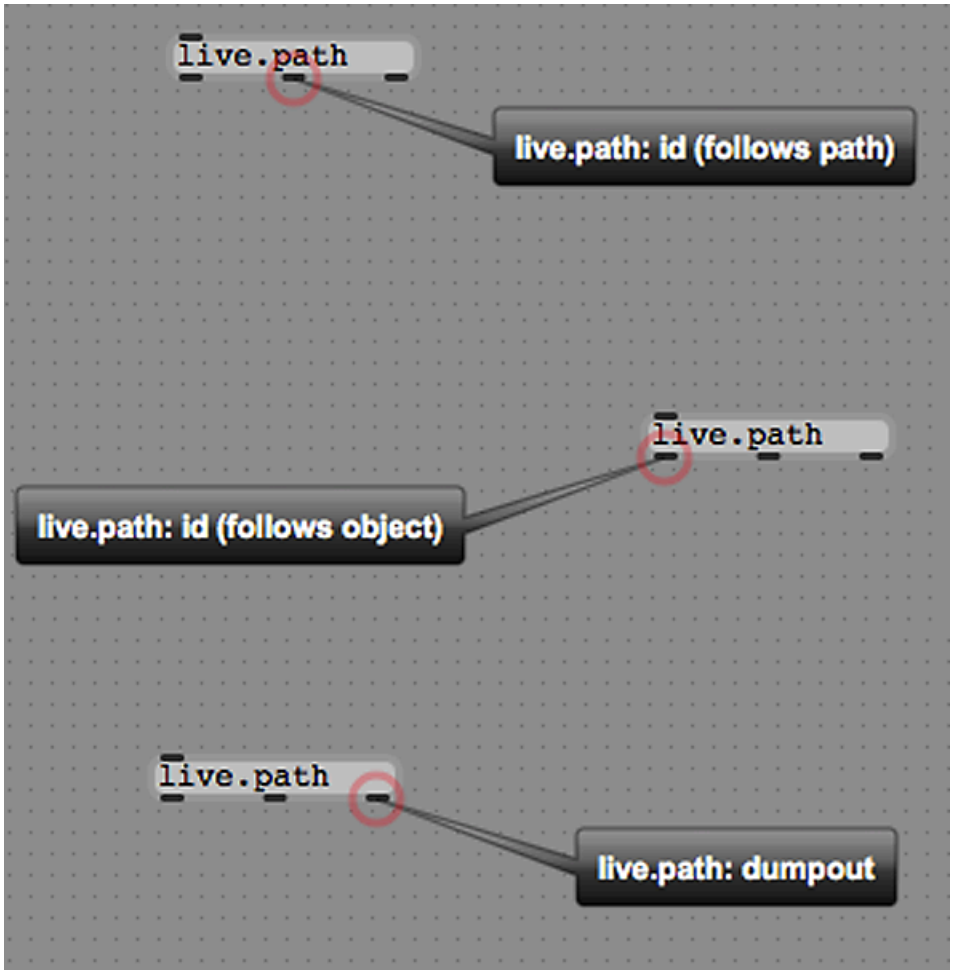
live.path translates an objects canonical, absolute or relative path into a corresponding ID for the object

We can follow another analogy here: on the Internet, we can reach Google's website by typing **google.com** directly into the address bar. But Internet communication protocols use IP address to make messages travel and indeed require IP addresses. Domain Name Servers (DNS) are here for that purposes: they convert names into IP address.

[live.path] is like the DNS for IDs in Max for Live.

Follow object or path ?

[live.path] provides **three outputs**.



live.path's three outputs

Help documentation tells us that:

- the first one provides IDs in *follow object* mode
- the second one provides IDs in *follow path* mode
- the last one is the **dumpout**

If the last one is the generic dumpout providing a way for objects to dump information while they receive particular messages, the role of the first two is a little bit more difficult to understand.

Let's imagine we are accessing a path like **live_set tracks 2** and we take the ID from the *follow object* output.

The ID is the one for the track which has the index 2 (it means the third track, as all indexes start from 0)

If I keep that ID and I send it to a **live_set tracks 2** object, this object will begin to handle the third track.

Ok.

But now, imagine that we move this third track to another place in the liveset (at the end of my tracks for instance)

The same ID we already have is STILL related to the freshly moved track (we chose **follow object** mode)

But imagine now another scenario. If the ID comes from the output *follow path* and I move the third track to the beginning of my tracks in the Liveset. We are okay to say the path itself has changed, right? Indeed, the track originally had the index 2 (as it was the third track) but because I moved it to the beginning of my tracks in Live, it now has the index 0. So if I play with the same ID I stored for instance, I won't play with the track I just moved, but with the new track that is in the position of track 3 in the Liveset, which now has the index 2 (it previously had the index 1, but has now moved up one as a result of my track rearranging)

So, be careful with which output you take the ID from.

2.6 One shot request with *get* and live.object

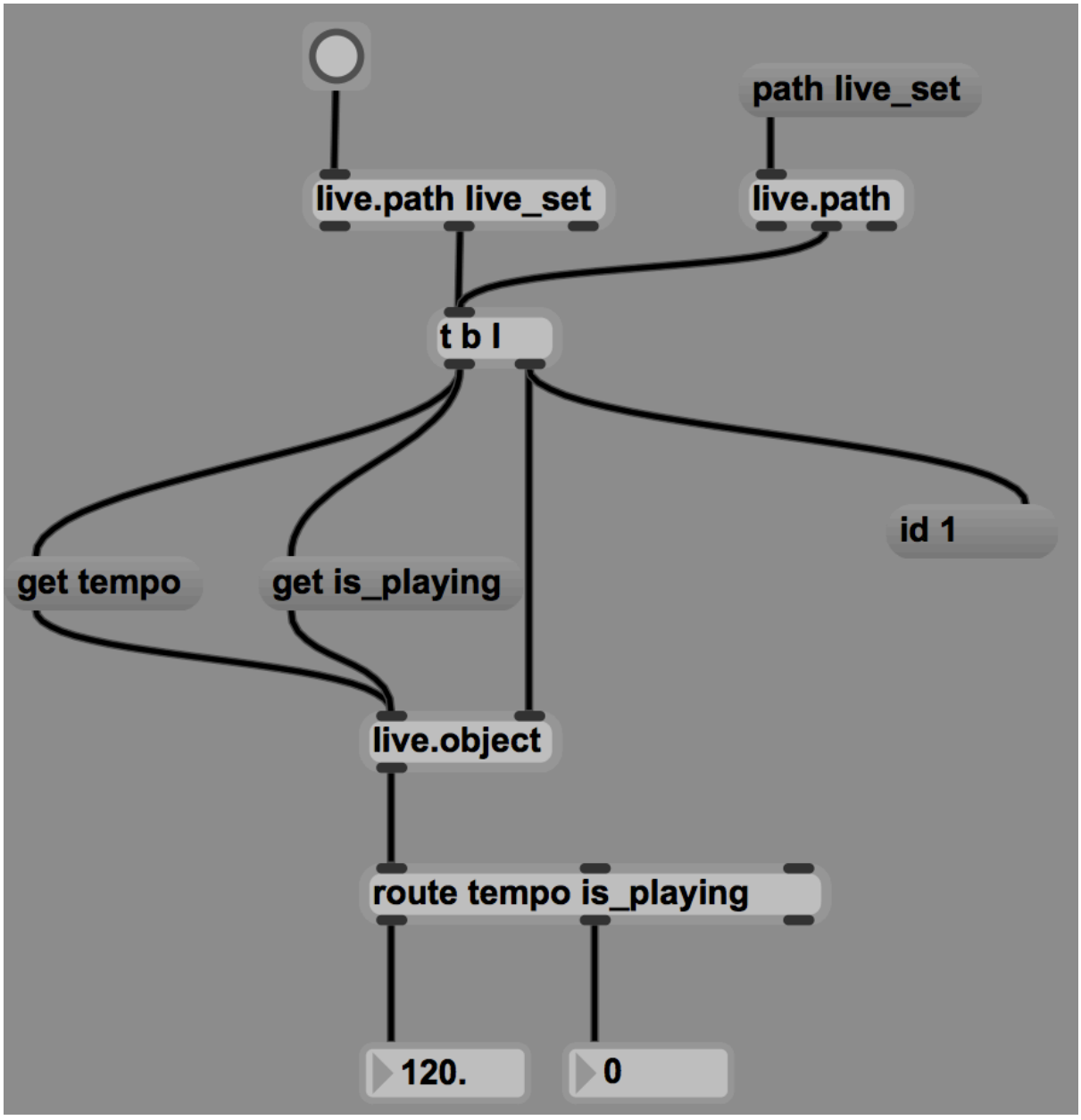
I like to experiment fast in Live using a basic setup with Max for Live. This one allows me to rapidly prototype things and to have it all in the same folder. Let's do that first.

Creating a Max for Live rapid testing setup

- Open Live, create an empty liveset and save it with a name related to your test.
- drag'n' drop a Max for Live device from the MIDI Effect type in a track.
- Click on the edit button of this device
- Save this device in the same folder as your Live project

This is a basic guideline that you can follow to quickly create a **rapid prototyping environment**.

Now, edit your device and create a patch as in the figure below:



LOM's testing

This structure provides a simple way to test and understand IDs, paths, and to make one-shot requests to Live by using **live.object**.

There are two ways of using **[live.path]**:

- setting the path as an argument. Arguments are assigned to objects by putting a space after their name, then the arguments itself.
- sending a message containing the keyword *path* followed by the path itself.

Both are syntactically equivalent, but the second is more interesting when we require **dynamic systems**. For instance, if we want the patch to be able to make requests about multiple tracks, we will need to build paths dynamically depending on the track we need information from. We cannot do that when the index of the track is set as an argument.

Here, from our small Max for Live device, we want Live to tell us what the tempo is and if the transport is playing or not.

If we check the LOM documentation, we can see that tempo and playing status are available as properties of the object Song. Indeed, these values depend on the liveset itself.

I made a small cut to show specifically the data we need to see within one table:

Song

This class represents a Live set. The current live set is reachable by the root path **live_set** .

Canonical Path			
live_set			

Children			
Name	*Type*	*Access*	*Description*
appointed_device	Device	get, observe	The appointed device is the one used by a control surface unless the control surface itself chooses which device to use. It is marked by a blue hand.
cue_points	list of CuePoint	get, observe	Cue points are the markers in the arranger to which you can jump.
master_track	Track	get	
return_tracks	list of Track	get, observe	
scenes	list of Scene	get, observe	
tracks	list of Track	get, observe	
view	Song.View	get	
visible_tracks	list of Track	get, observe	A track is visible if it is not a subtrack folded in. Hiding tracks be scrolling them out of view is something completely else.

Properties			
is_playing	bool	get, set, observe	1 = the Live transport is running. Can be used to stop or start the Live transport.
tempo	float	get, set, observe	Current tempo of the Live set in bpm, 20.0 ... 999.0. The tempo may be automated, so it can change depending on the current song time.

We need the object Song here

All that we need are these two properties :

- tempo
- is_playing

tempo data type is *float*, which basically means it is a number with a decimal point i.e 120.50. We see that we can also perform a get, a set and even an observe request on this property.

is_playing data type is a *bool* that stands for a boolean. Its value can be *true* or *false*. We can also perform a get, a set and an observe request on this property.

Now, we know which properties we can request and to which object we need to perform those requests.

Let's check the canonical path of the object Song: **live_set**

Let's pass it to **[live.path]** and this one will produce a result formatted like this: **ID <number>** where <number> is the numerical ID of the requested property.

BE CAREFUL OF ID 0

If ID 0 is the result of **[live.path]**

Then it means that no object fits with the path passed to the **[live.path]**

ID 0 is often a clue saying you made an error in the path.

Object **[trigger]**, abbreviated as **[t]**, provides a way to control the outgoing flow of messages. Here, **[t b l]** means that if the trigger object receives *something*, it will process it as a list that will be transmitted from the rightmost outlet then after this operation, it will send out a bang from its leftmost outlet.

The message element linked to its rightmost outlet is only here for control and display purposes, it shows us what ID is sent out of **[live.path]**. This value is transmitted to the **[live.object]** which will be *linked* to the object's instance related to the transmitted ID, here: the Song object.

From this moment, we can transmit multiple requests to **[live.object]** and it will process them accordingly to the LOM, relatively to the object Song, that means the liveset itself.

Then we send two messages to it:

- **get tempo**
- **get is_playing**

And **[live.object]** answers with the two responses :

- tempo 120.00
- is_playing 0

[route] object provides a way to route incoming messages to many outlets depending on a prefix (tag) in the received messages. It also un-tags these incoming messages (here it removes the *tempo* and *is_playing* prefixes)

Play a bit with this patch.

Launch some requests, observe the results, then change the tempo and activate the transport in Live.

It works fine but we need to relaunch the requests to get the updated values.

Requesting values occasionally

1/ find the property you want to know the value of

2/ find the object providing this property and remember the object's canonical path (the canonical path can be found under the object name in the LOM documentation)

3/ give this canonical path to live.object

4/ request the live.object with messages like : **get <property name>** and this one will answer with messages like **<property name> <current value of the property>**

This is good! But we have to make a request every time we change the value and if we change the value we know the new one anyway, so why would we have to request it… ?

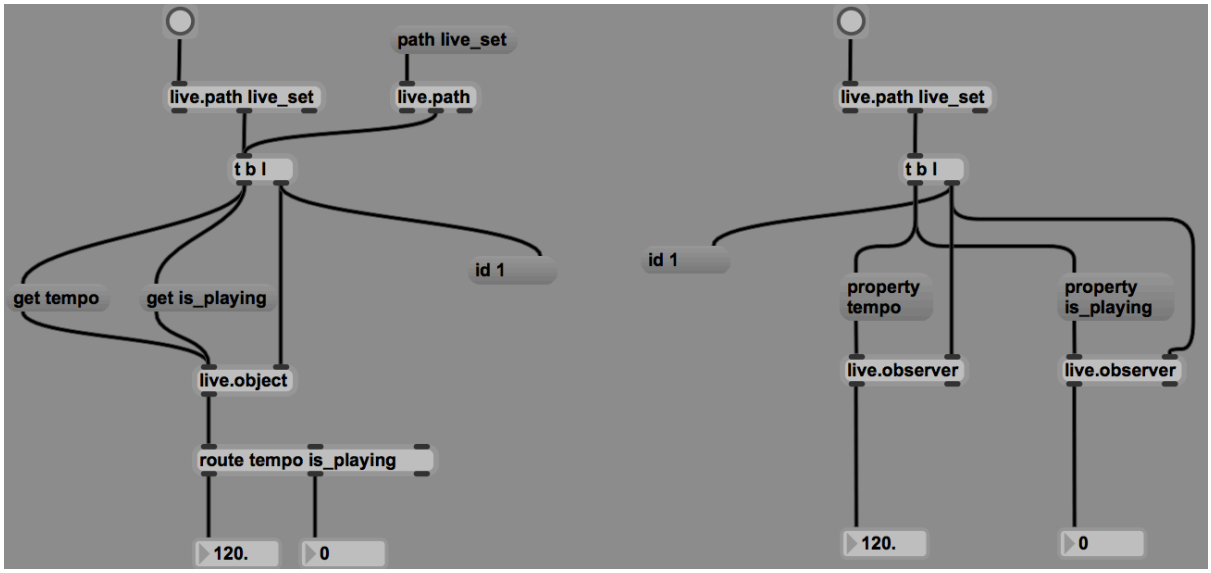
It doesn't make sense because *what we need is the updated tempo value at anytime.*

2.7 Observing properties in Live with live.observer

As mentioned previously, **[live.observer]** provides a way to be informed about a property's value change whilst the property is observed by the **[live.observer]** itself.

This means, we can give an ID to **[live.observer]**, we tell it what property we need it to follow, and it does the job.

Here is the corresponding example patch:



Observe tempo & transport's state

I have consciously left the previous patch to the left with the **get / [live.object]** technique.

We can see that the ID itself is the same in both cases. This is totally normal (and safe) as we can provide the same path which navigates to the Song object to both **[live.path]** objects.

Here, we have the **[t b l]** object again, and it transmits the ID to the **[live.observer]**.

Please, be aware of this:

[live.observer] can ONLY observe one property at a time !

If we need to observe two properties, we need two **[live.observer]** objects !

Then, to define which property the **[live.observer]** should follow, we send it this message:

property <property name>

From this moment, **[live.observer]** transmits from its outlet the value of the property. If this latter changes, a new value is sent out.

Play with it as you did with the previous patch.

Change the tempo, start & stop the transport while looking at the patch. You will see the **[live.observer]** objects update with the same correct values.

Observing a property

1/ find the property you want to know the value of

2/ find the object providing this property and remember the object's canonical path

3/ pass the canonical path to **[live.path]** which finds the ID for **[live.observer]**

4/ send the message **property <property name>** to the **[live.observer]** which will send out the current value of the property.

This is the way to always be informed with up to date properties' values.

2.8 Calling objects' functions with live.object

We are going to learn now how to call functions.

When we read LOM's documentation, we can see that each object owns a set of functions.

We are going to create a small patch that provides a way to trigger clips.

Seeking the object required

Let's scroll through the LOM documentation. We can see an interesting object, the **ClipSlot**.

Indeed, without worrying about whether there is a clip or not inside the clip slot, we can use the function **fire** and this function will, as read in the LOM, trigger the clip if there is one, or trigger the stop button of the clipslot if there is not.

This is exactly what we want to.

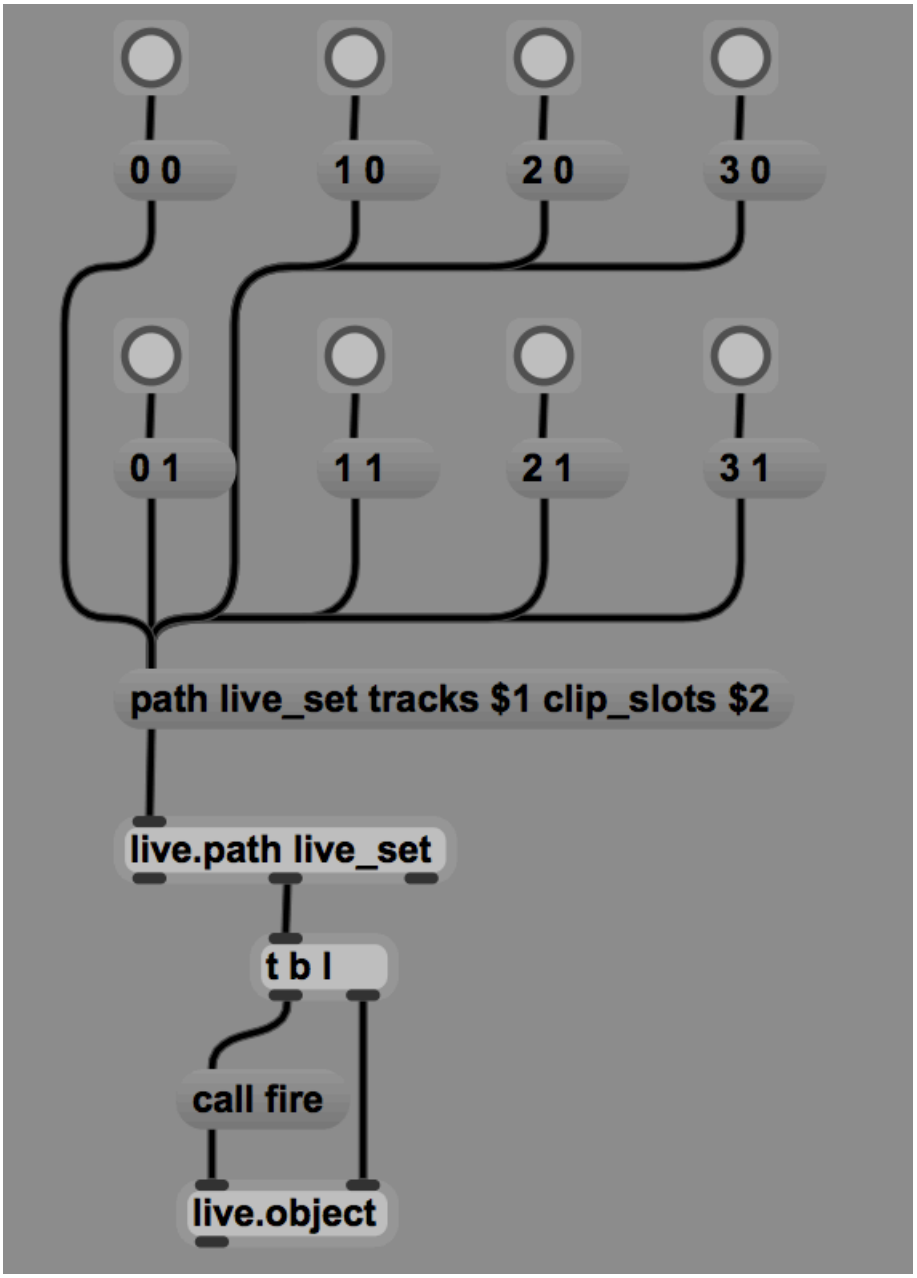
The canonical path to reach each clip slot in our clips' grid is:

live_set tracks N clip_slots M

with N being a track's index & M as scene's index (N and M are the coordinates of the clip slot, starting from zero in each dimension)

Building the patch

Here is a possible example patch.



Triggering clip slots

Here the canonical path has been built dynamically.

This means that when we push each button, the static message placed under the buttons will send out the following message:

path live_set tracks $1 clip_slots $2

$1 & $2 are what we use to call **placeholders**. These are variables (known as replaceable arguments) that get their values from incoming messages, in the same order of the incoming list.

I chose to put the index of the track first, then, the index of the scene (our clip slots coordinates)

For instance, if I send the message (2 1) to my message path, this will transmit **path live_set tracks 2 clip_slots 1** to **[live.path]**

And **[t b l]**, after having transmitted the ID received from the **[live.path]**, will trigger a bang sent to the message (call fire) and then, as soon as this message goes into the **[live.object]**, the function fire will be called and the corresponding action triggered.

Here, the clip slot is triggered *exactly as* if we were to push its stop button or if there was a clip the play/stop button of the clip.

Put some clips in the liveset between scene 0 and 1 and track 0 to 3 and play with the patch.

Calling objects' functions

1/ find the object and its function you want to call

2/ pass the canonical path of the object to **[live.path]** which finds the ID for **[live.object]**

3/ send the message **call <function name>** to **[live.object]**

Functions act on the liveset and thus modify, as a consequence, other properties.

For instance, triggering a clip with the transport stopped will also start the transport. The value of the is_playing property of the object Song will change to 1.

But we can also change some properties' values without using functions.

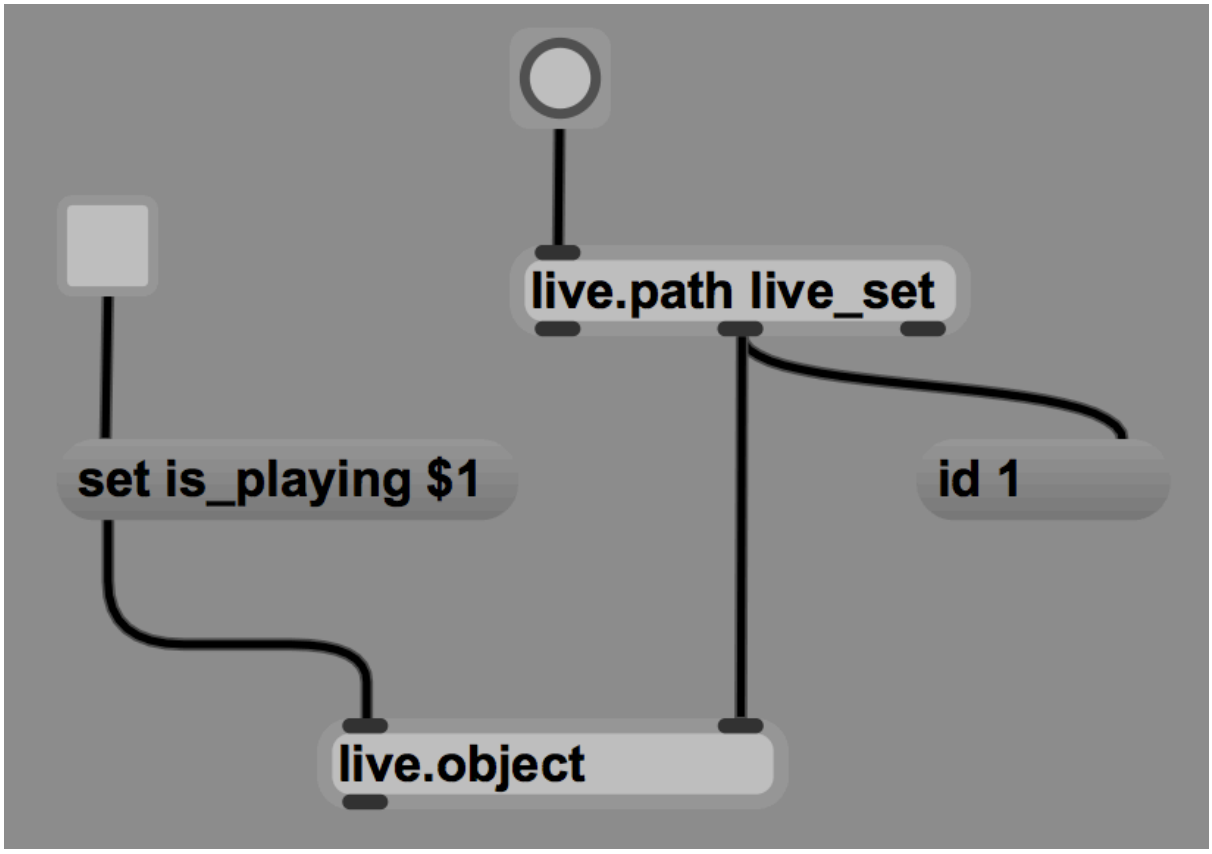
2.9 Changing properties' values by using *set* & live.object

We are going to create a button to start and stop the transport by changing the value of the property **is_playing**.

Don't forget to check what you can and cannot do with each property you want to use!

Indeed, **is_playing** allows us to **get**, **set** and **observe** and this is also why I'm often using it during my training sessions in order to illustrate how we can use all of a property's features, globally.

Look at that patch:



Apply a set on a Live object

It looks strangely like the first example we built with **get**.

Here, in place of **get is_playing**, we have:

set is_playing $1

This is a basic way to change a property's value.

[toggle] provides an easy way to change a value from 0 to 1 and from 1 to 0 with a nice and cheap user interface: a check button.

Push it several times, it will trigger Live's transport.

Changing a property's value

1/ find the property you want to modify

2/ find the related object and its canonical path

3/ send this canonical path to **[live.path]** which will find the ID for **live.object**

4/ send the message **set <property name> <new value>** to **[live.object]** and the property's value will be modified.

We can do this for continuous parameters too, I mean, a parameter that takes a range of values and not only on and off, or 0 and 1.

2.10 Controlling continuous parameters in Live with live.remote~

Imagine that you need to control the value of a track's mixer panning of track 3 and you want to make it follow a continuous sine wave, for instance.

We can totally do that with **[live.object]**.

But I must warn you about this, and would not encourage you to do it!

Why ?

Because when we do this with the **set / [live.object]** technique, all modifications are recorded in Live's UNDO history

By using **set / [live.object]** technique with continuous parameters, we would pollute very quickly the UNDO history.

BE AND WATCH CAREFULLY

You will find a lot of Max for Live devices programmed in this way.

Be careful, not all Max for Live device designers are very aware about this…

More, and this is apparently a consequence of this UNDO history fact, system performances aren't good in this case.

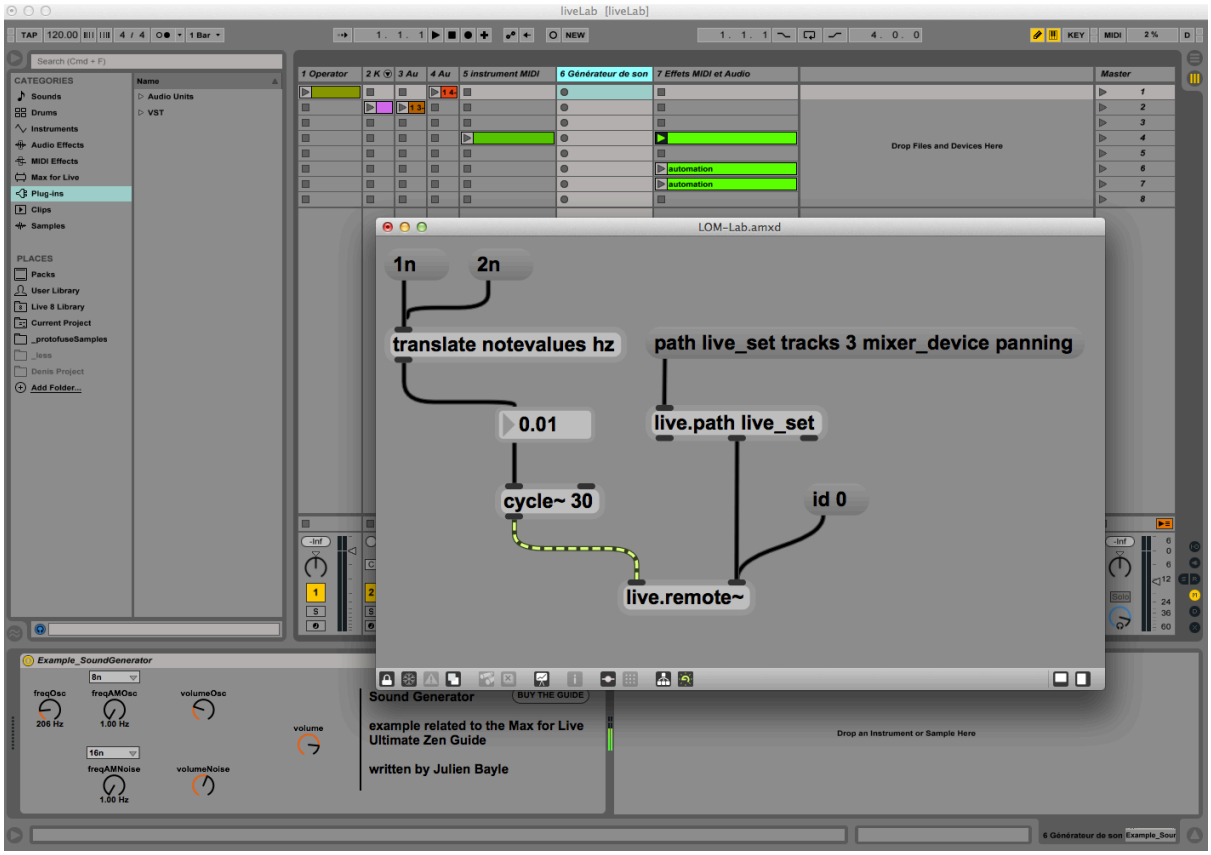
This is also why we have another tool available:

[live.remote~]

As we can see, there is the *famous tilde* ~ at the end of the name.

It means that this object is able to process signals generated by the MSP part of Max for Live, or at least entering into MSP.

Look at the next patch



Home-made LFO controlling a track's panning

Easy, isn't it?

But let's look at it closely, especially because I'm using a specific path.

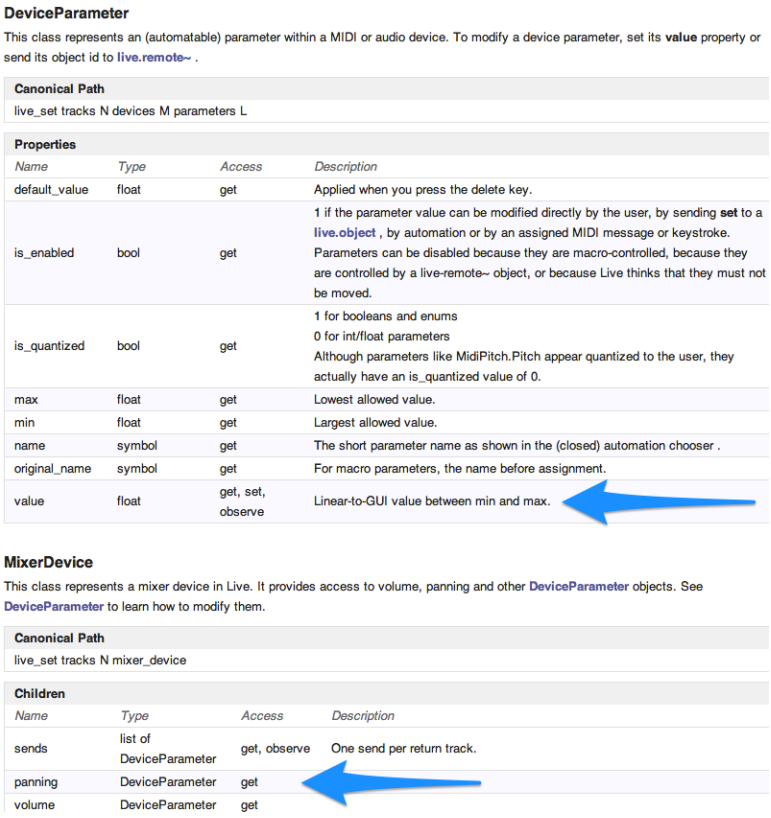
Look:

live_set tracks 3 mixer_device panning

And at this very moment, I have usually 4 or 5 people asking me the question:

"Hey! panning isn't an object! Because we want to modify it, this is a property and a property doesn't own an ID so live.path won't take it…"

"Very good question", I used to say: but look at the LOM a bit better.



DeviceParameter

This class represents an (automatable) parameter within a MIDI or audio device. To modify a device parameter, set its **value** property or send its object id to **live.remote~** .

Canonical Path

live_set tracks N devices M parameters L

Properties

Name	*Type*	*Access*	*Description*
default_value	float	get	Applied when you press the delete key.
is_enabled	bool	get	1 if the parameter value can be modified directly by the user, by sending **set** to a **live.object** , by automation or by an assigned MIDI message or keystroke. Parameters can be disabled because they are macro-controlled, because they are controlled by a live-remote~ object, or because Live thinks that they must not be moved.
is_quantized	bool	get	1 for booleans and enums 0 for int/float parameters Although parameters like MidiPitch.Pitch appear quantized to the user, they actually have an is_quantized value of 0.
max	float	get	Lowest allowed value.
min	float	get	Largest allowed value.
name	symbol	get	The short parameter name as shown in the (closed) automation chooser .
original_name	symbol	get	For macro parameters, the name before assignment.
value	float	get, set, observe	Linear-to-GUI value between min and max.

MixerDevice

This class represents a mixer device in Live. It provides access to volume, panning and other **DeviceParameter** objects. See **DeviceParameter** to learn how to modify them.

Canonical Path

live_set tracks N mixer_device

Children

Name	*Type*	*Access*	*Description*
sends	list of DeviceParameter	get, observe	One send per return track.
panning	DeviceParameter	get	
volume	DeviceParameter	get	

DeviceParameter & MixerDevice, two singular objects

Panning is a child of **MixerDevice**. And when we are checking its type, we can see that it is a **DeviceParameter**. Thus this IS an object.

If we are checking objects which are **DeviceParameter**, they don't have a lot of properties. We can see the property **value**, that is the current value of the considered parameter and min/max which are the minimum and maximum values allowed for this parameter in Live.

Access column in the table Children

BE CAREFUL HERE (too)!

On the previous snapshot, we can only see a 'get' in the MixerDevice's children table in front of panning.

It means that we can only get a list of the MixerDevice's panning children.

It doesn't mean that we can only use get for the underlying DeviceParameter object.

Said differently: In children tables, get, set, observe are only related to the relationship parent/children of the concerned object. In order to know what we can do with child objects, we need to look at the type of child object itself.

[translate] just translated a note value (this is the name of the relative time format in Max,

sometimes) to Hz value in order to set the sine wave frequency to modulate the panning of the track. This allows, for instance, to have one sine wave cycle per bar, or for 1/2 bar etc.

In Max for Live, the default transport is Live's transport.

Important notice:

live.remote~ holds the parameter and locks it restricting any manual modification while it controls it.

Indeed, in Live, the parameter linked to the live.remote~ is colored in **grey**.

It means we cannot automate it or modulate it.

In order to break the link between **[live.observer]**, **[live.object]** & **[live.remote~]** and an object, we have to send them a message **(ID 0)**.

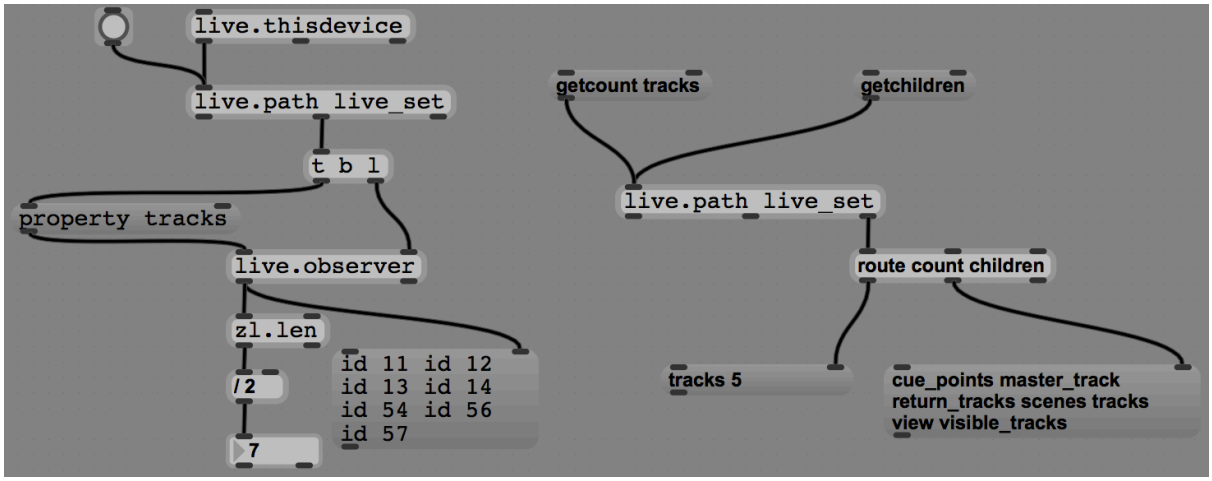
If we send another message (a real other ID) to **[live.remote~]** for instance, and if this ID was already associated with another **[live.remote~]**, we make another association and the first associated **[live.remote~]** is freed.

2.11 List and use objects' children

Each object has some children, or almost all.

We are going to try to list all tracks' IDs for instance from the Song object. Tracks are children of the Song object.

Look at that patch



Playing with children objects

On the left, we are focusing on the liveset itself and we are observing the property 'tracks'. Indeed, children in that case are also properties.

We can see that in the LOM:

Song

This class represents a Live set. The current live set is reachable by the root path **live_set** .

Canonical Path			
live_set			

Children			
Name	*Type*	*Access*	*Description*
appointed_device	Device	get, observe	The appointed device is the one used by a control surface unless the control surface itself chooses which device to use. It is marked by a blue hand.
cue_points	list of CuePoint	get, observe	Cue points are the markers in the arranger to which you can jump.
master_track	Track	get	
return_tracks	list of Track	get, observe	
scenes	list of Scene	get, observe	
tracks	list of Track	get, observe	
view	Song.View	get	
visible_tracks	list of Track	get, observe	A track is visible if it is not a subtrack folded in. Hiding tracks be scrolling them out of view is something completely else.

Observing child objects as properties

We can observe all Song's children of the type **tracks**. We could do the same with **scenes** or **return_tracks**.

This is a tip to keep in mind and is a not a very well known one.

It provides a way to:

- keep an up to date list of tracks' IDs
- know the exact current number of tracks in the liveset

[zl.len] measures the length of the list coming from **live.observer**. We know that this list contains ID's which use two list entries each **

Imagine for instance that you need to keep a trace of the states of all clip's in all tracks. If we were to create a new track or remove one, we would have to update our clips' grid. This is a nice way to trigger things when we add or remove a track!

Let's check the patch on the right.

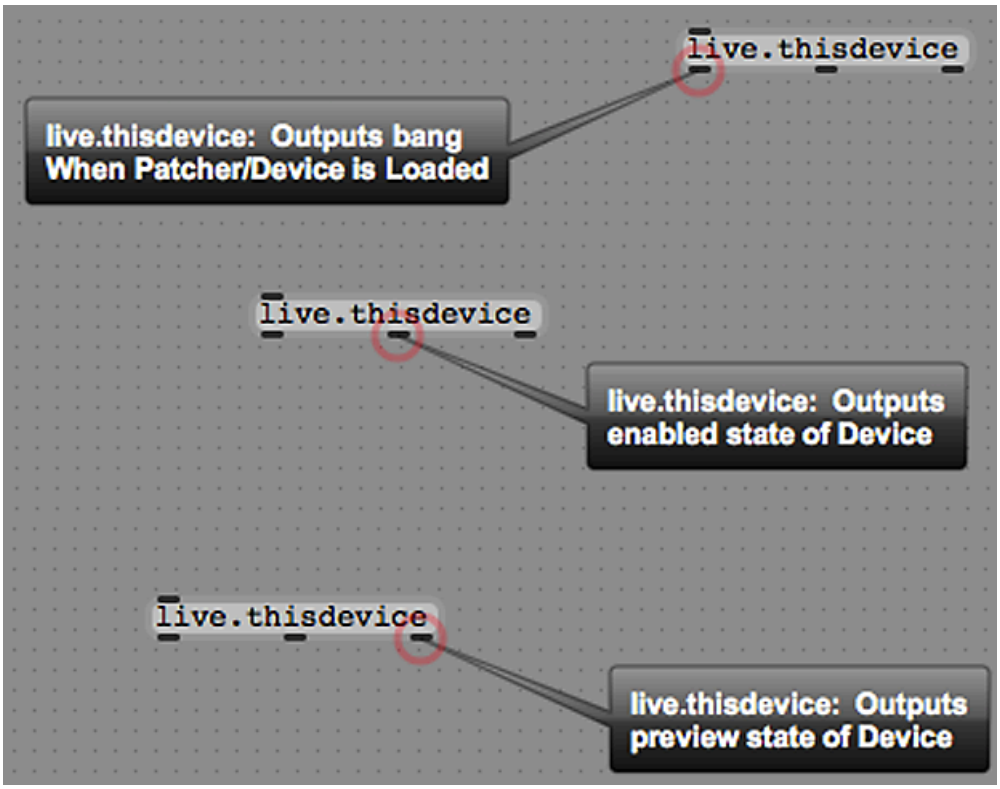
We can see that we can send other types of messages to **[live.path]**:

- **getcount <child>** provides the number of children of the type

All this information is sent out the right outlet of the **[live.path]** named dumpout .

2.12 live.thisdevice tool

[live.thisdevice] is another Max for Live specific object. It is really simple.



live.thisdevice

This object sends a bang from its leftmost outlet when the Max for Live device in which it is placed is loaded/initialised.

We use it in place of the classic [loadbang] object in Max for Live devices.

It provides also a way to observe the state of the device itself: it sends a 0 or a 1 from its middle outlet depending on whether the device is disabled or enabled.

This can be useful to eventually switch off or on a CPU expensive process when we don't need it, for instance.

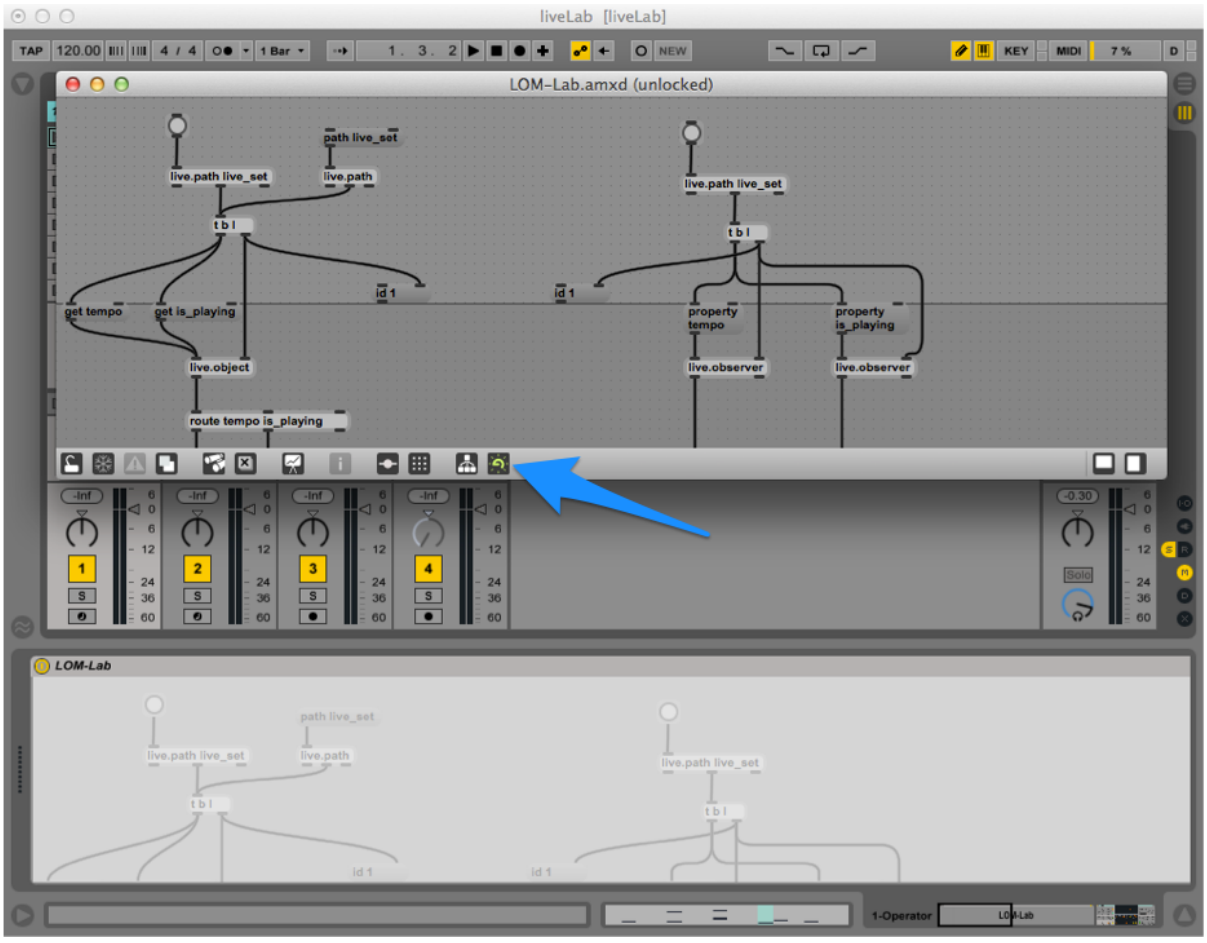
At last, the rightmost outlet provides the edition mode that can be:

- Preview mode
- Non-Preview mode

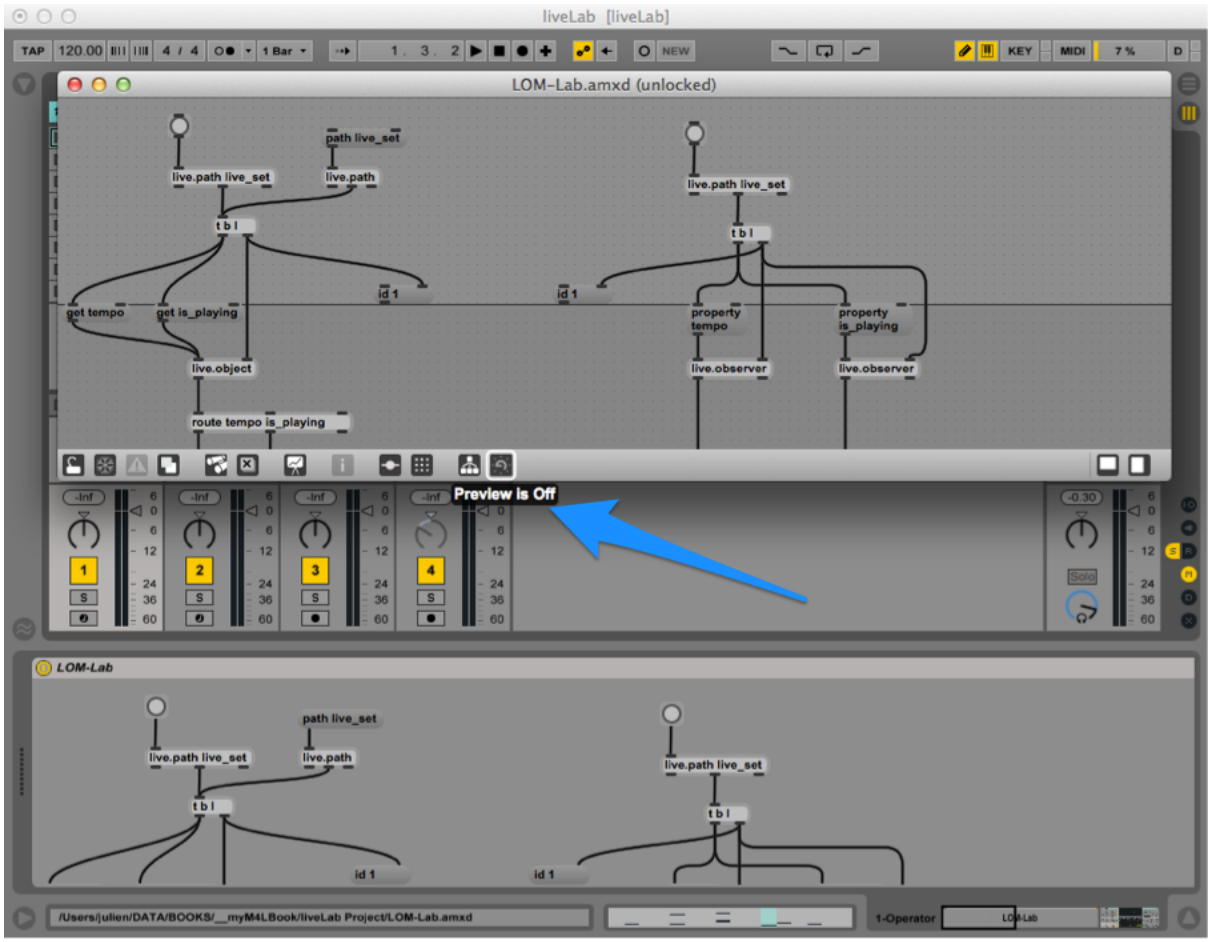
What is it all about ?

2.13 Mode Preview & performances

I was waiting for probably the end of the chapter about the LOM to talk about Preview mode and especially the computer performances with Max for Live.



Mode Preview On



Mode Preview Off

In Preview mode, we can edit and alter the Max for Live device exactly as if it was still connected to Live. If we act on the patch itself, this can have an effect on Live itself.

In that mode, the device is grayed in the devices' chain in Live and indeed, we cannot control the device using Live's UI.

In non-Preview mode (they call it Preview Off, anyway), this is totally reversed.

From a performance point of view and from the different experiences I have had and also some discussions with Max and Live's developers:

The best way to test our devices is to close the Max edit window and to test it in real conditions

Normally, Preview mode provides a way to avoid this step, and indeed for 90% of the time it is enough.

BUT, I often see some cases where the Live API was a bit confused and I had to close the Max edit window in order to test it in real conditions in Live.

Especially, if you need to test for performance, you need real conditions. That sounds obvious, but keep it in mind

Conclusions about the LOM

I strongly suggest that you dig and continue the road inside the LOM with these links and your documentation:

- LOM[3]
- Live API Overview[4]
- live.path help[5]
- live.object help[6]
- live.observer help[7]
- live.remote\~ help[8]

[3] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live_object_model
[4] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live_api_overview
[5] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live.path
[6] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live.object
[7] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live.observer
[8] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live.remote~

3 JavaScript in Max for Live

JavaScript[1] is an object-orientated script programming language.

JavaScript, also known as JS, is implemented inside Max which interprets it with a JS 1.6 engine.

Through the object **[js]**, we can directly write our own scripts and execute them in Max.

The interesting thing is that we can make calculations, store values and other pure programming wizardry, but especially we can manipulate a lot of Max' features directly with JS like MSP and Jitter stuff.

Indeed, lots of Max parts are exposed to JS and Max for Live parts too.

It means we can use JS to reach the LOM.

3.1 How to use Javascript in Max for Live ?

At first, we need to create a **[js]** object with the JS script filename as an argument.

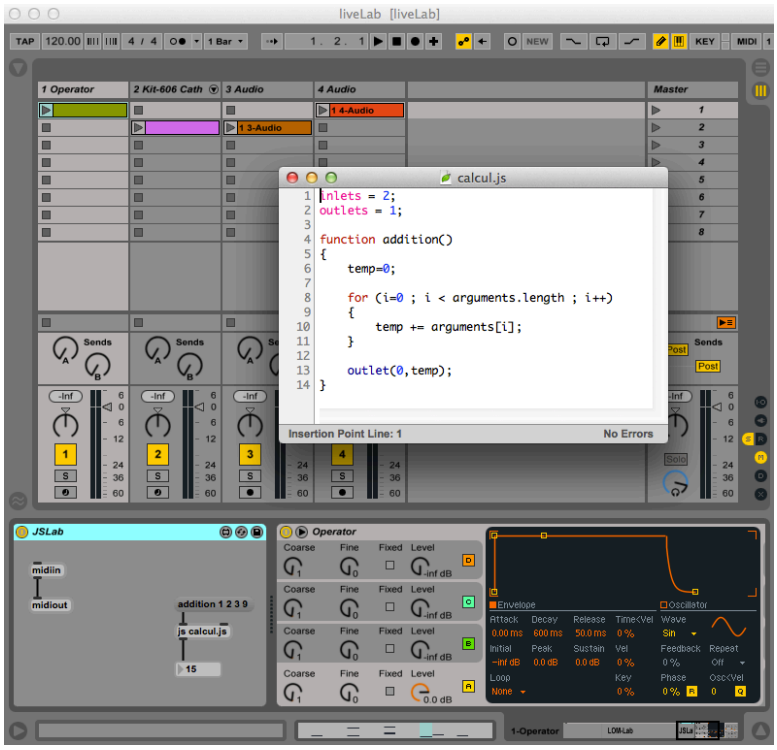
Usually, I do it like this:

1/ create the **[js]** object with the argument **<myScriptName.js>**

2/ double-click on the **[js]** object while pressing CMD on OSX or CTRL on Windows

3/ editing the script and save it with CMD + S on OSX or CTRL + S on Window

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript?redirectlocale=en-US&redirectslug=JavaScript



JS in Max for Live

We can see a script named **calcul.js** (which is basically a file on my hard disk) and this script can be edited in Max, and even in Live itself as soon as we can see the **[js]** object in the device's UI.

If we click on the message **(addition 1 2 3 9)**, JS takes the first element of the list and tests if there is a named function.

Here, it is the case and it passes the other elements of the list as arguments to the function, I mean: 1 2 3 & 9.

In the JS code, we can play with each argument passed to a JS script in Max by using the array arguments. In each array's slot, we have an argument available.

arguments.length is the length of the table, the number of elements contained by the Array.

In the code, I made a loop that checks all arguments and stores them and makes a sum 2 elements per 2 elements.

outlet(<outlet's index>,<value>) pops out the value **<value>** from the outlet with the index **<outlet's index>**.

This is a basic calculator.

I won't and unfortunately cannot make a whole JS course right here. Or fortunately for us, maybe.

I'd strongly suggest you to check these links :

- JavaScript's official documentation[2]
- The four JS's tutorials in Max' help
- JS in Max[3]

I'd really push you to study JS in Max because it can be a massive helper while patching.

Indeed, as soon as we have to handle repetitive tasks like loops in patch, we are often building a lot of spaghetti-styled patches. With JS, a basic ()for loop can be the only one statement needed, which can be much more simple.

3.2 Controlling Live API through JavaScript

The Live API is the programming interface that provides a way to access the LOM.

The different Max for Live objects we have already played with also provide a way to access the LOM.

Now, let's check how to do that with JS.

Live API & JS

Two important documents are :

http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#jsliveapi[4]

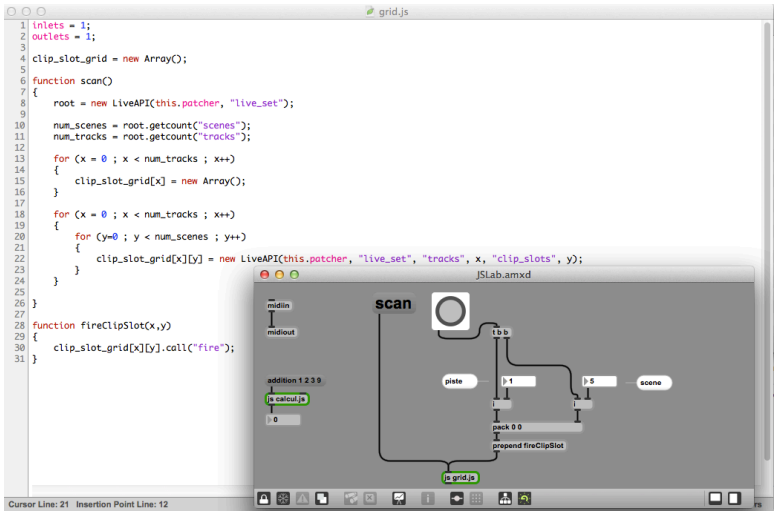
http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live_api_overview[5]

The principle is simple to understand: we can use all the LOM's objects with JS.

Act on the liveset

Let's look at a small example in which I will show you how to trigger clips using the clipslots object exactly how we did before but by using JS.

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript?redirectlocale=en-US&redirectslug=JavaScript
[3] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#javascriptinmax
[4] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#jsliveapi
[5] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#live_api_overview



JS triggering clips

Here is the code:

```
inlets = 1;
outlets = 1;

clip_slot_grid = new Array();

function scan()
{
        root = new LiveAPI(this.patcher, "live_set");

        num_scenes = root.getcount("scenes");
        num_tracks = root.getcount("tracks");

        for (x = 0 ; x < num_tracks ; x++)
    {
        clip_slot_grid[x] = new Array();
    }

        for (x = 0 ; x < num_tracks ; x++)
        {
                for (y=0 ; y < num_scenes ; y++)
                {
                        clip_slot_grid[x][y] = new LiveAPI(this.patcher,"live_set","tracks",x,"cli
ots",y);
                }
        }
```

```
}

function fireClipSlot(x,y)
{
        clip_slot_grid[x][y].call("fire");
}
```

In the JS code, I declared an Array **clip_slot_grid**. This one stores all the LOM objects I want to keep available like, here, the whole clips slots of my liveset's clips grid.

The function scan() scans the whole clips' grid and stores at the place **clip_slot_grid**[x][y] the clipslots object in the track x at the scene y (be careful, we start at zero each time)

Please keep in mind that it is the objects' instances that are stored!

If we do it like this, then we have access to each function, and each property of all objects directly by using the right element of our Array. Easy, isn't it?

Let's now describe a little bit more of our script:

```
root = new LiveAPI(this.patcher, "live_set");
```

Here, we are **initializing the API** and I decided to store the object Song directly in the variable named root. This is the highest object in my hierarchy. This basically represents the liveset as you already now know.

Then, as we know we can request the number of children of each type from the Song object. With Max for Live objects, we should use the message **getcount <child>** with **[live.path]**

Here, we call the method getcount of the Song object which is stored in root and here is how we do that:

```
        num_scenes = root.getcount("scenes");
        num_tracks = root.getcount("tracks");
```

We store those values in two variables: *numtracks* and *numscenes*

We have a double nested for () loop in which we store each clipslots object instance in **clip_slot_grid[x][y]** place of the Array, as we described before.

Here is the code:

```
clip_slot_grid[x][y] = new LiveAPI(this.patcher, "live_set", "tracks", x, "clip_s\
lots", y);
```

By doing this, we have an easy way to use a function named **fireClipSlot(x,y)** defined further with the argument x & y which are respectively track index and scene index, the coordinates of the clip slot we need to trigger. Thus, we have the opportunity to call fireClipSlot function (x, y) defined below.

This function basically calls the function fire of the ClipSlot object instance stored in **clip_slot_grid[x][y]**.

If we look at the patch itself, we have several elements.

We can firstly select the tracks' and scenes' indexes. These two integers are stored in **[i]** objects. These latter store the value they receive in their rightmost inlet without transmitting it from their outlets. Indeed, if we want them to send out stored values, we have to send a bang to the their leftmost inlet. This happens when we click on the **[button]** object.

[pack] object prepares a list with the track's index followed by the scene's index previously chosen and transmits the whole message to **[prepend]**. This latter adds a prefix according to the value it owns as an argument. Said differently, it adds the keyword **fireClipSlot** in front of the track's index and the scene's index , and this is the way to call the corresponding function in the JS object.

Aiming at LOM's object using JS

We have to use this kind of instantiation:

```
new LiveAPI([callback], <path or ID>);
```

[callback] is the name for callback functions and is optional. We'll see later what a callback function is.

This call with **new LiveAPI()** creates and returns an object. This object is stored inside a variable in order to keep it and to provide an easy way to retrieve it. With this, we can call functions related to these stored objects, change property's values: finally, this is what we did with **[live.object]**.

The **<path or ID>** part shows that we can also use an ID, if we already have it. We can also use the canonical path instead of the ID if we don't know the latter. The Live API will call internally and automatically the equivalent of [live.path] will convert the path into an ID.

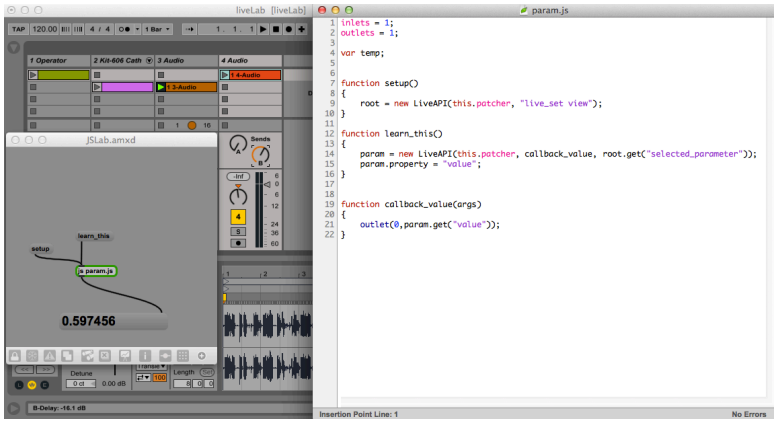
Properties' observing and callback functions

We just learned how to act on the liveset through a small example which showed how to store **objects' instances** in Arrays.

Let's check now how we can observe as we did with **[live.observer]** but within the JS.

I'm going to show you how we can observe a live parameter's value and transmit it to the patch (outside of the JS) by using Javascript.

Look at the next patch:



Callbacks in JS

Here is the code:

```
inlets = 1;
outlets = 1;
var temp;

function setup()
{
        root = new LiveAPI(this.patcher, "live_set view");
}

function learn_this()
{
        param = new LiveAPI(this.patcher, callback_value, root.get("selected_parameter")\
);
        param.property = "value";
}

function callback_value(args)
{
        outlet(0,param.get("value"));
}
```

We have two functions calls that we perform outside of the JS object:

- setup()
- learn_this()

setup() stores the instance of the **Song.view** object inside the variable **root**.

learn_this() provides a way to keep the parameter just selected with the mouse (or with a MIDI controller through the MIDI Mapping) and to observe its value.

In the JS, we have to change a bit of the syntax of the object's instantiation using **new LiveAPI()**. Indeed, we add the name of the callback function inside the call AND we define the property to observe like this: **<objcet>.property = <property name>**

How does it work?

Look at the function **learn_this()**

I instantiate that basically with the name of the callback (previously described as an optional argument):

```
param = new LiveAPI(this.patcher, callback_value, root.get("selected_parameter"));
```

Please, I didn't define a path or an ID… I could add the word: *WOW!*

However, please focus on the arguments passed to the object's constructor **LiveAPI()**:

```
root.get("selected_parameter")
```

root contains the object instance **Song.view** which provides a property named **selected_parameter**.

This one always contains the ID of the current selected parameter in Live. Interesting. Let's resume:

Then, I'm defining the property I want to follow.

```
param.property = "value";
```

A **DeviceParameter** provides always the property **value**. This latter stores, as its name describes, the value of the property itself.

If I summarise, we can observe the value of the parameter selected in Live and fixed by the call of the function **learn_this** in Max for Live. If this value changes, the callback is called.

The callback function is called each time the property's value changes.

It avoids making CPU expensive polling in which we would make a huge amount of request per second even if the property's value doesn't change. It would consume a lot of our CPU's cycles for nothing!

The listeners/callbacks strategy is widely used in the UI design field.

Unlink an observer with JS When we want to unlink an observer, we need to send "ID 0" to it.

However, even developers are a bit confused about this. And I don't say that without having discussed it with them.

JS uses a **garbage collector**. Everything that isn't used anymore, for instance outside the scope of a function, is progressively destroyed and the underlying memory is freed.

However, I (we) noticed that even by manually destroying objects using the function **delete**, even if we take care about emptying its value before by putting the value **null** inside of it, sometimes, the listener wasn't unlinked… I guess this will be improved in later versions.

Practically, this can be annoying in some specific cases.

You can maybe read more about that on my own blog and eventually comment too :

http://julienbayle.net/blog/2012/02/ability-to-cleanly-destroy-liveapis-callbacks[6]

We are now ready to create Max for Live devices.

[6] http://julienbayle.net/blog/2012/02/ability-to-cleanly-destroy-liveapis-callbacks/

4 Creating your own Max for Live devices

One of the most interesting opportunities Max for Live offers us, is the ability to create our own devices for Live. We are going to see that we can create very different types of devices.

Before this, we are going to explore some required concepts.

4.1 preset & pattr

When we are talking about *Max for Live devices*, we need to think of them as *Ableton Live devices*.

This means that our Max for Live devices have to be totally and smoothly integrated in Live's presets system.

Importantly, each device's parameters and settings must be:

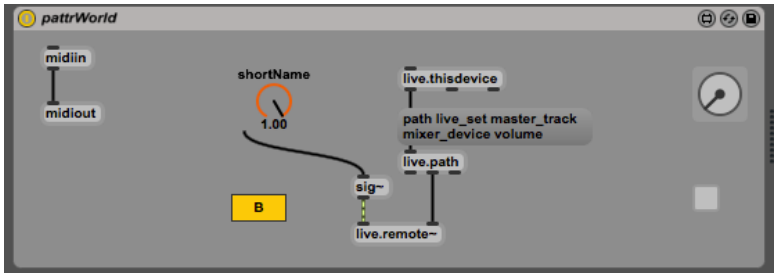
- storable within liveset files
- storable as device presets

For this purpose, it is required to use the pure Max' preset system combined with the Max for Live one, as we are going to see.

Max for Live specific objects VS the Native Max objects

Let's make a small experiment.

Look at this device:

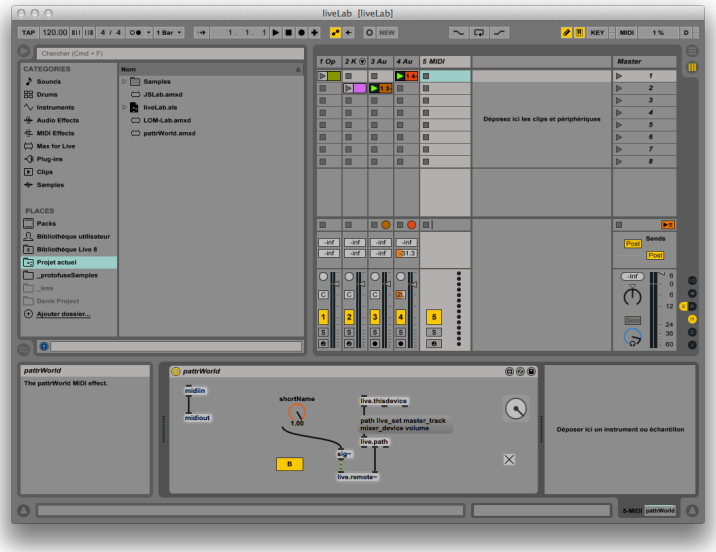


Presets and devices

There are Max for Live specific UI objects (**[live.dial]** & **[live.text]** at the left) and also two native Max objects (**[dial]** & **[toggle]**) at the right.

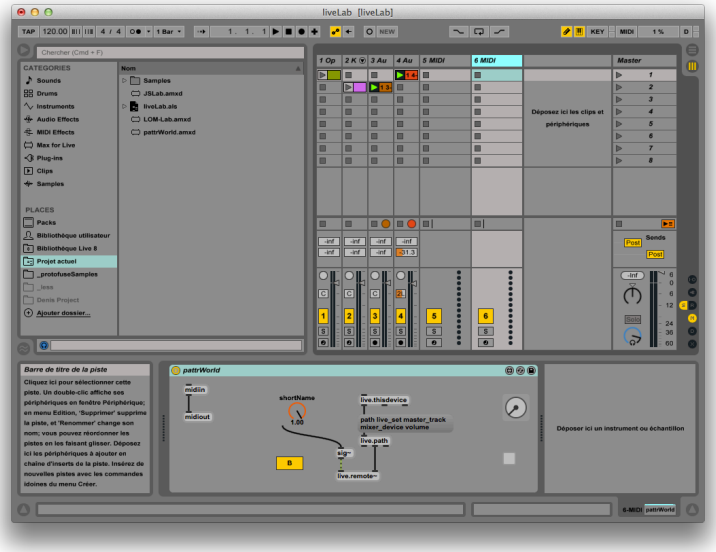
With Max for Live, ONLY specific Max for Live UI objects (those with a name beginning with "**live.**") are stored within livesets, as presets and when we are copying/pasting a device from one track to another.

I can prove this: I set the two *dials*' values to their maximum and I clicked on the **[live.text]** and the **[toggle]**.



All values have been modified

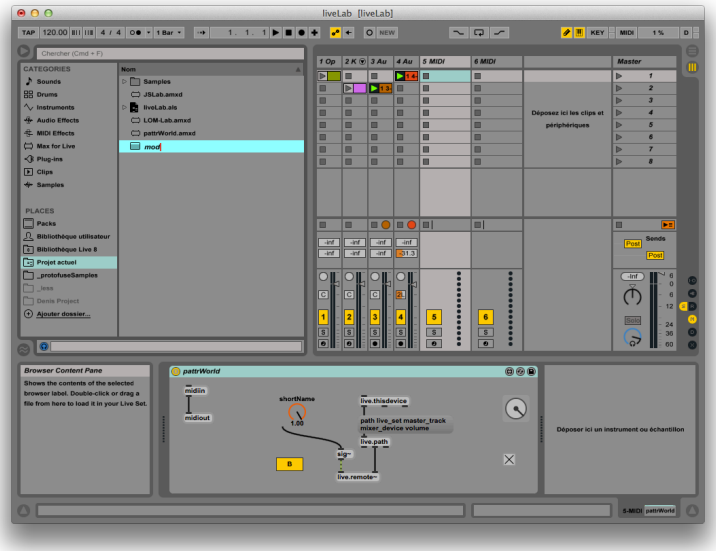
Then, I copy/paste the same device to another track named **6 MIDI**



Copy/paste only keeps the values and settings of Max for Live specific UI objects

The copy/paste operation only kept the values and settings of Max for Live specific UI objects. The native Max UI objects resorted to their initial/default values.

Let's check now how it works when we try to save the current settings of the device as a preset. I did that and named the preset mod.



mod preset saving

Then, I paste that preset in a new track named **7 MIDI**

Only Max for Live specific UI objects' states are stored in presets

Only Max for Live specific UI objects' states are stored in presets.

The same goes for the liveset: only specific UI objects' settings are stored with the liveset itself.

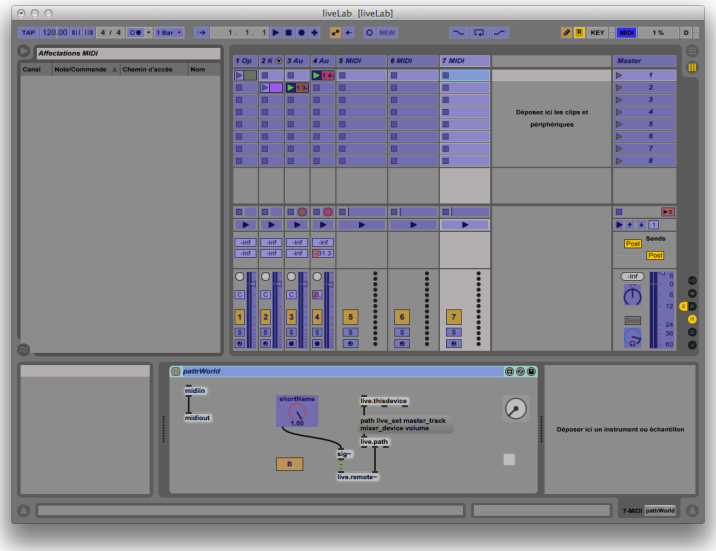
Automation and MIDI mapping works like that too.

Natively, we can ONLY apply automation in Live with Max for Live UI objects. As proof, check the next snapshot: we can see my own UI elements in the automation list but not the default ones (named default *dial* & *toggle*)

Only Max for Live UI objects are available for automation and modulation purposes

But there is a way to use all Max elements as we would like: **pattr** with the attributes **parameter mode enable** enabled.

MIDI mapping can only be used with Max for Live UI objects. It CANNOT be used with other pure Max UI objects excepted by using MIDI parsing, of course.



Only Max for Live UI Objects can be used for keyboard and MIDI Mapping

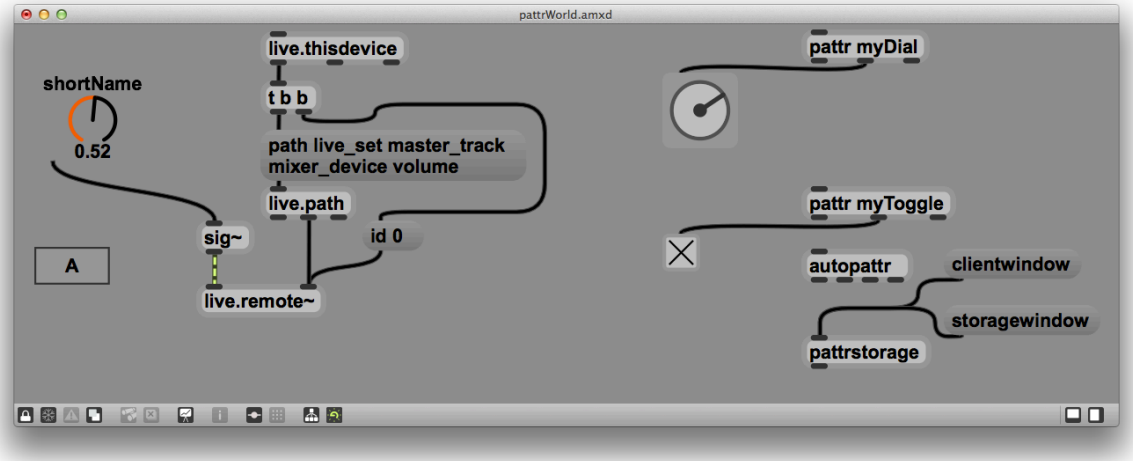
pattr & parameter mode enable

I'd suggest you to read the **pattr**'s documentation at this point. You can find it here[1] and some very useful tutorials too:

- tutorial 1[2]
- tutorial 2[3]

Look closely what I added to my patch:

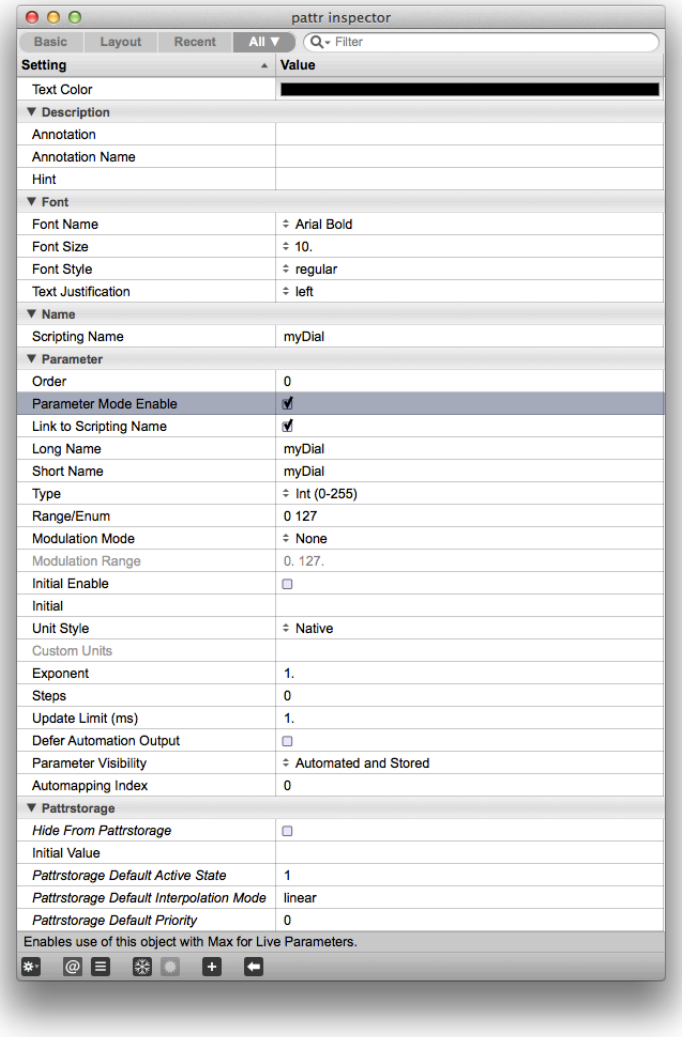
[1] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#pattr
[2] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#pattrchapter01
[3] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#pattrchapter02



pattr helps native Max UI Objects

I added a **[pattr]** object with a name as an argument for each native Max UI object I want to use as if they were Max for Live objects.

If I look at the inspector for each **[pattr]** object, here is what I can see:



Parameter Mode Enable enables some Max for Live features for Max objects

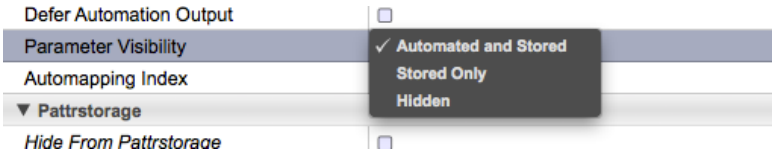
Importantly, we can see the **Parameter Mode Enable** attribute checked and enabled.

As soon as we check it, some new parameters appear below in the list within the inspector. We can recognize these parameters: these are the same ones visible in any Max for Live UI Objects.

As soon as we check it, we link the **[pattr]** object to Live's parameter's system.

I won't describe all attributes but only those which are very important to our study.

Parameter Visibility



Parameter Visibility defines how Live will handle this parameter

It sets up the parameter handles by the **[pattr]** object, i.e. the one linked by a patch cord to the middle outlet of the **[pattr]** (also called *bindto*). This parameter can be:

- automatable and storable
- only storable
- hidden

In the first case, the Max UI Object linked to the **[pattr]** will be potentially automatable and be stored with the liveset and the device's presets too.

In the second case, this parameter will only be stored, and not available for automation.

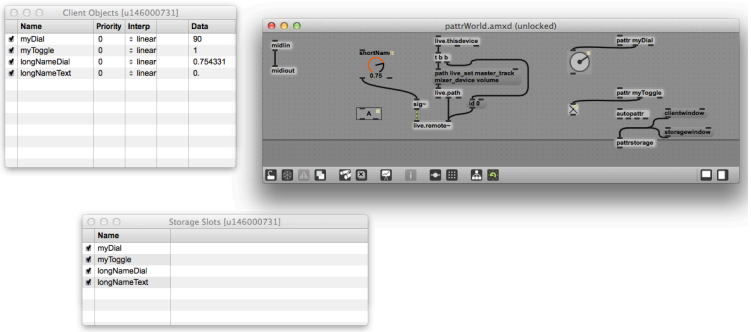
In the last case, it basically won't be handle by Live.

You may be thinking: why would we use hidden when we could simply not connect the Max UI object to a **[pattr]** ? The reason is easy to understand. We could build a preset bank within this device and handle its presets directly inside of the device and not in Live.

autopattr & pattrstorage

You probably noticed some objects like **[autopattr]** & **[pattrstorage]**.

[autopattr] links all objects that are not attached to **[pattr]** to the **[pattrstorage]** system, this latter is also a way to have an internal preset bank for all linked objects. For instance, this is also the way to attach all Max objects to **[pattrstorage]**.



The patch, the clientwindow & the storagewindow

If we send the message **clientwindow** to **[pattrstorage]**, we can see that a window appears: the **clientwindow** displays all parameters handled by the related **[pattrstorage]**.

If we send the message **storagewindow** to **[pattrstorage]**, we can see that another window appears: the **storagewindow** shows a set of slots in which we can store whole settings for all parameters.

One of the main interests of **[pattrstorage]** is the fact we can retrieve all parameters directly by recalling a **slot**.

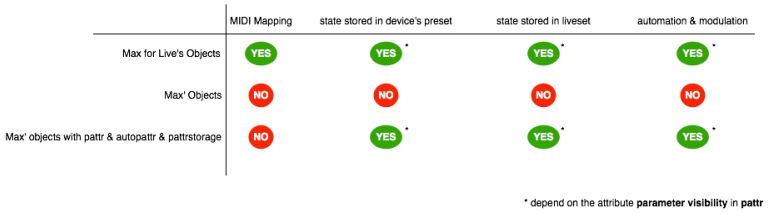
We can also morph between parameters stored in 2 separate slots. This is why we have a column named *interpolation* and we have a specific setting for each parameter. We can even choose the **interpolation's curve**

[pattrstorage]'s help can really help you to understand the full range of features:

http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#pattrstorage[4]

A table summarizing parameters in Max for Live & Live

Here is what you need to keep in mind about the handling of Max for Live & Live parameters.



Max for Live and Live parameters

4.2 Freezing Max for Live's devices

Max' patches can use native objects (those delivered when you buy the software) and subpatchers. In that case, our patch is autonomous and can be shared with other programmers directly, without taking special precautions, by supposing, of course, that we have the same version of Max.

But consider the following situations:

- we have external objects we coded ourselves with the Max6 SDK
- we use abstractions referring to other patches on our hard drive

Then, we'll have dependencies and if I send you the patch, your computer won't be able to solve them if I only send you the root patch

With Max for Live, we have two ways:

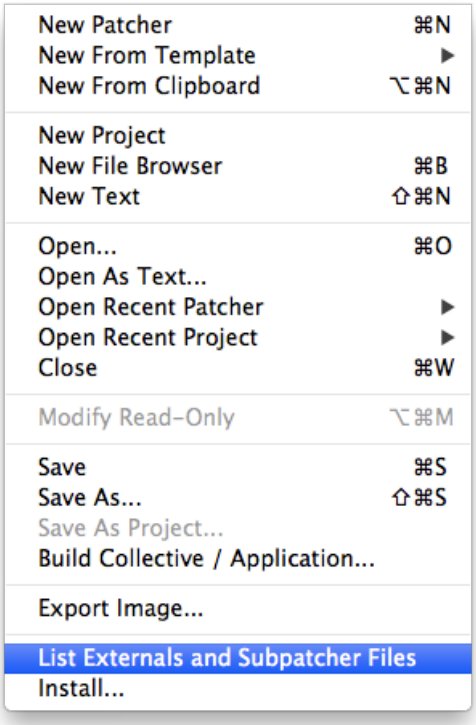
[4] http://www.cycling74.com/docs/max6/dynamic/c74_docs.html#pattrstorage

- we can handle dependencies ourselves by copying them and sending them alongside with the main patch
- we can just let the Freeze feature work for us

Obviously, the second solution is the one I would strongly suggest.

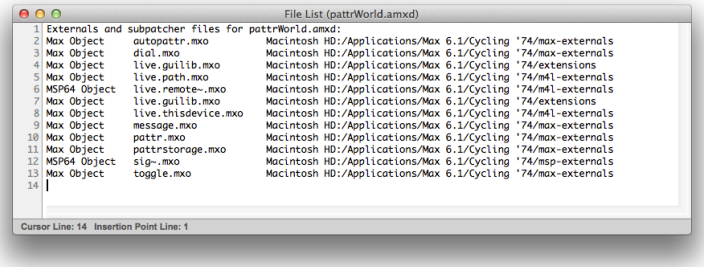
Handling dependencies manually

In the first case, we need to list the whole dependencies of our patch by clicking on **List Externals & Subpatcher Files** like this:



List all dependencies

Here is the result, in the example case :



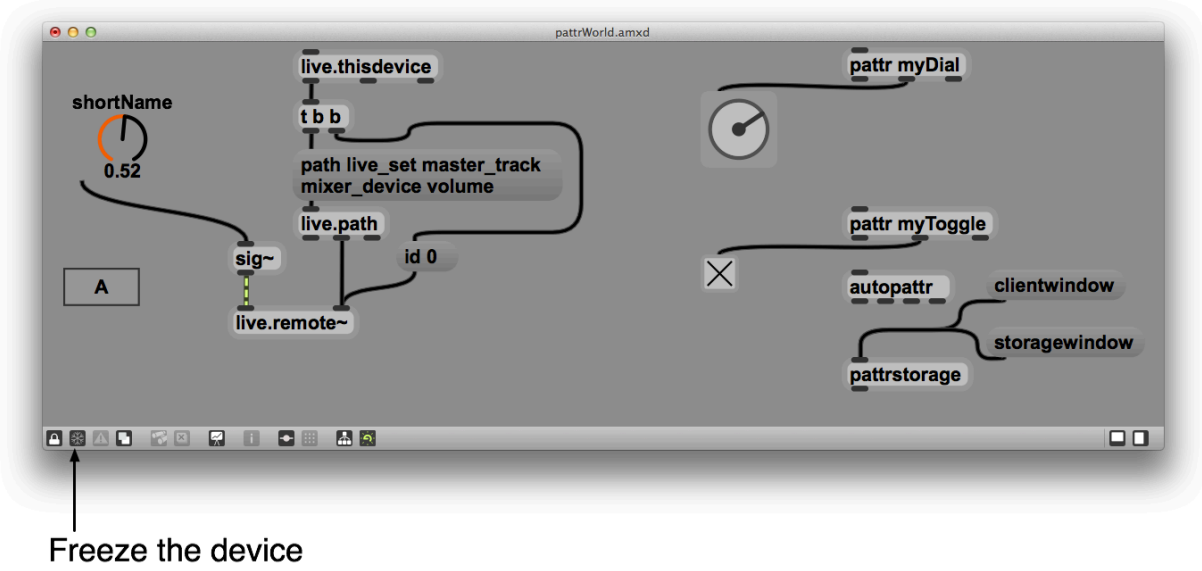
The dependencies list

We can see here that we don't have specific dependencies: the first column shows us only Max Objects.

Freezing devices

With Max, we can create collectives & standalone applications as explained at the very beginning of this guide. But we cannot do that with Max for Live because there is no runtime , except for Live itself.

But we have the device freezing feature:



Freeze button

As soon as we freeze it, it becomes non-editable. We have to unfreeze it in order to edit it again.

A frozen device can be saved, of course. And at this very moment, the .**amxd** file of the device will contain ALL dependencies of the main root patch. Usually, its size is increased a bit compared to the unfrozen file, especially if you are using a lot of dependencies.

Then, I can safely send you the new frozen device (the **.amxd** file) because all dependencies required to use it are included!

This is the main way to distribute devices.

4.3 Presentation mode

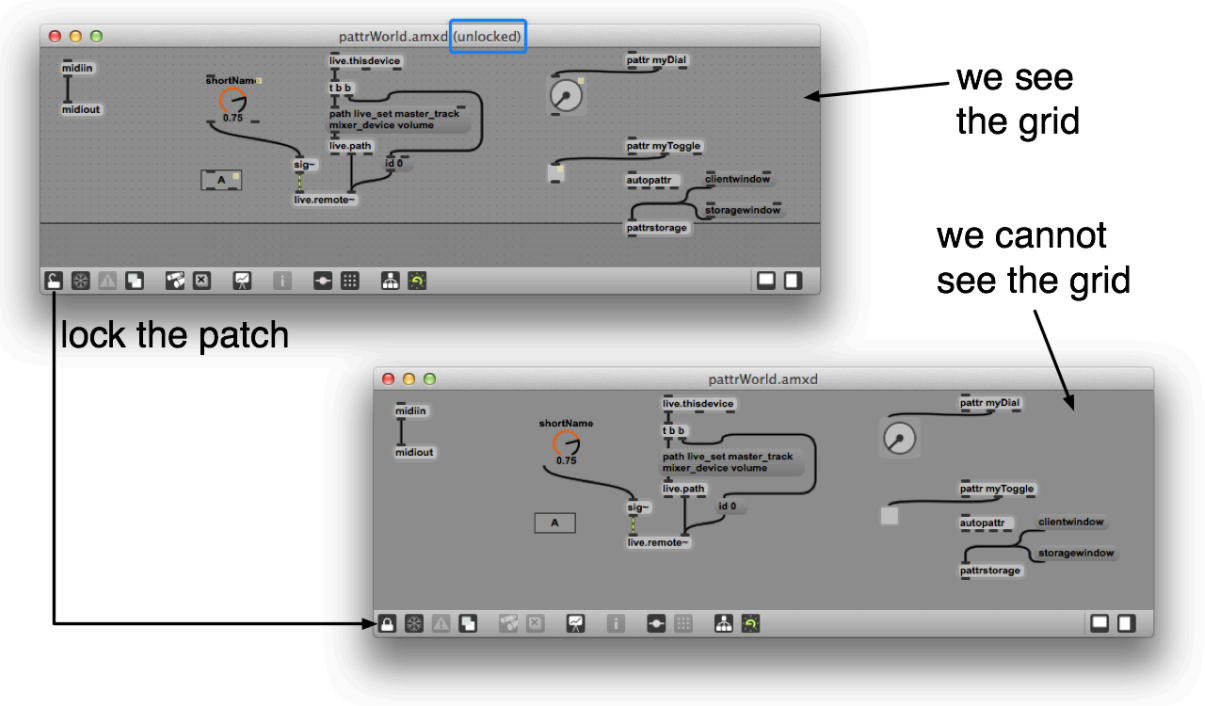
There are two display modes in Max, and also in Max for Live:

- Patching Mode (Editing Mode)
- Presentation Mode

Patching Mode

Patching mode can be locked or unlocked. If the patch is locked, we cannot edit it. This is *just* a way to play and test it without modifying it by mistake.

We can switch between locked and unlocked mode by pushing the small lock icon at the bottom left as we can see here:



Locking patches

When we have finished editing our patch, we may want to only allow some UI elements on the screen, but not the whole set of elements.

This is exactly the purpose of **Presentation mode**

Presentation mode

Every object in Max can be set up to be visible in Presentation mode or not. It is possible for all objects in Max and not only UI objects.

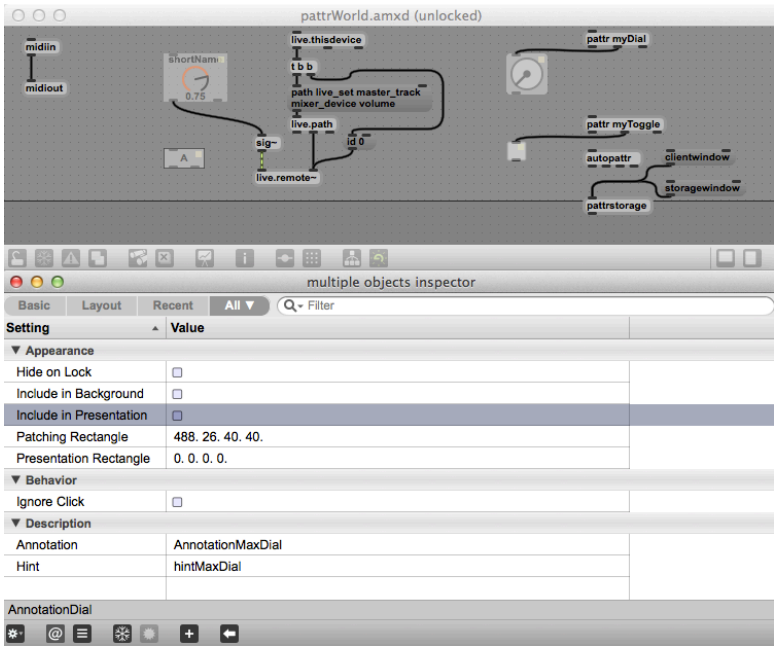
Build your own Presentation UI

There are three steps:

- choose and set some objects in Presentation mode
- place your UI elements on the display as you want to use them in live performance, for instance
- set up the patch itself in Presentation mode

We select objects that we want to see in Presentation mode. We can use multiple selection by selecting all objects while keeping the SHIFT key pressed.

Then we display the Inspector for our selection and we check the attribute **Include in Presentation**:

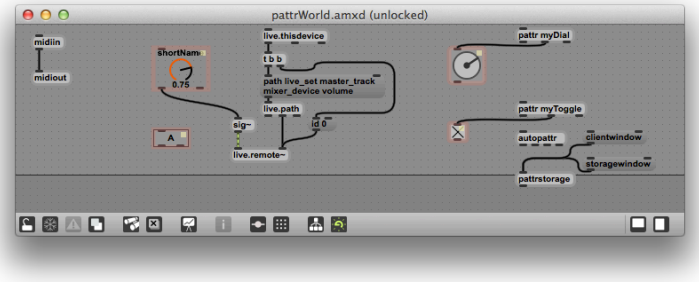


Include some objects in Presentation mode

We can also use some shortcuts:

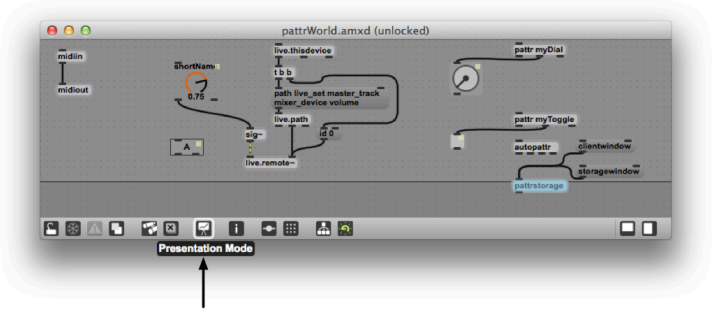
- CMD+SHIFT+P on OSX or CTRL+SHIFT+P on Windows to include objects in Presentation
- CMD+ALT+P on OSX or CTRL+ALT+P on Windows to remove them from Presentation

We can easily check if an object is set up to be in Presentation mode or not: if they are, there is a rose border **outlining them**:



We can easily see which objects will be in Presentation mode or not

Then, we can switch to the Presentation mode by clicking on the icon showing a small screen as shown here:

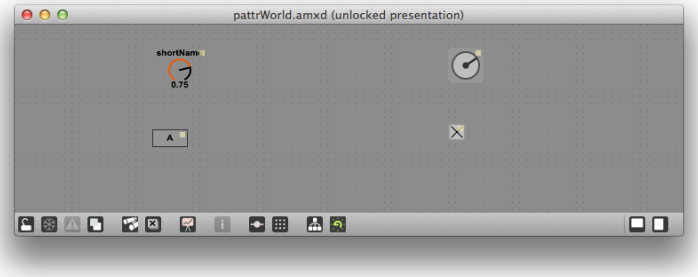


Enable/Disable the Presentation mode

We can also use the shortcut:

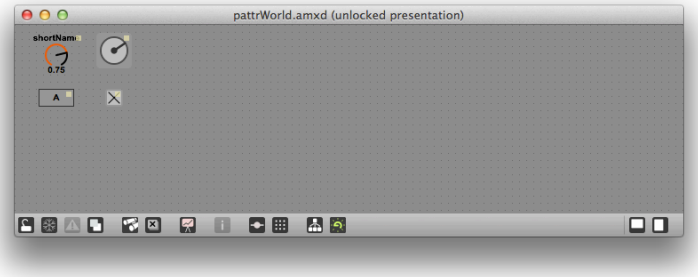
- CMD+ALT+E on OSX or CTRL+ALT+E on Windows for Presentation mode switching

In Presentation mode, we only see the objects we included in it.



Presentation mode

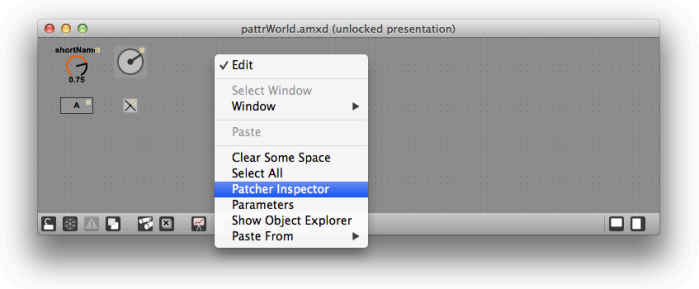
We can move them and place them exactly as we want to design our interface.



We design our final device's UI

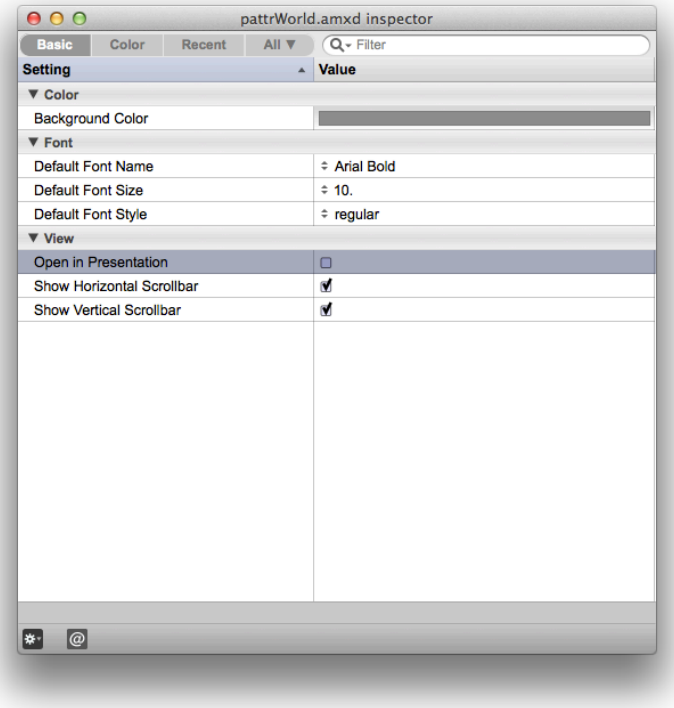
Then, we set up the patch in order to make it open directly in Presentation mode by default: this is the only way to have a Max for Live device UI in Live that fits with the Presentation mode.

Let's right click on the patch itself and choose **Patcher Inspector**:



Set up the patch iteself

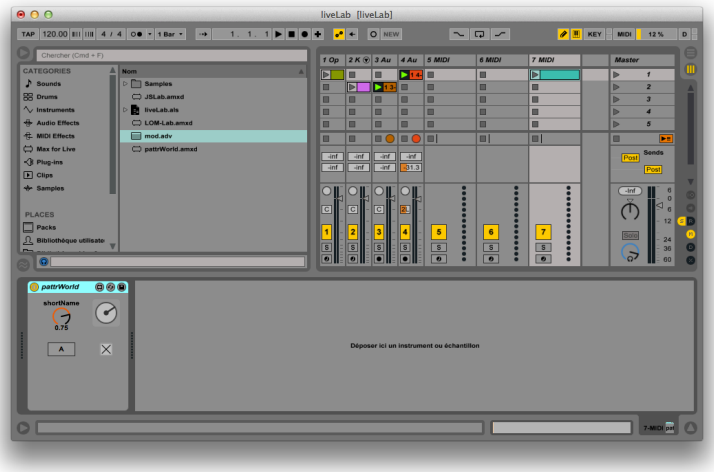
Then, we have to check **Open in Presentation**:



Check the Open in Présentation attribute

Save the device (CMD+S or CTRL+S) and close the edit Max window.

Check the render in Live:



Finished device in Live

We now have our own device UI.

Now, with all the knowledge we have, I'm about to give you some real device examples.

This guide isn't a proper Max course: probably, some concepts will be a bit tricky to understand but I'm going to describe things very simply.

4.4 Limitations of Max for Live compared to Max

Max for Live is Max operated by Live, which means we have some limitations coming from Live itself.

We have also identified in the first chapter: if you also own a Max license, you can run it outside of Live and Max won't be limited.

Max Limitations

It is possible to make data communications using **[send]** & **[receive]** between two devices in Live. BUT Cycling'74 developers warn us about latency/performances.

In real-life, it really depends on what kind of data flow we want to use between objects.

If we only send data sporadically, not in continuous flows, we won't feel any latency. I have tested and used to do that sometimes without any problems. Be careful if pure continuous data flows, anyway.

As we have also discovered, with Max for Live, we cannot create **collectives** or **standalone applications**.

Audio Limitations

Max for Live doesn't directly use Max audio drivers as if it were Max standalone.

Indeed, Max for Live's device's inputs & outputs are its proper inputs and outputs in the devices's chain in Live.

Max for Live devices are limited to 2 audio channels (left/right) and we also know that Max (more exactly MSP) can handle up to 512 channels.

Then, **[send~]** & **[receive~]** cannot send signals between devices: we have to use Live's own routing features.

MIDI limitations

Max for Live doesn't use its own MIDI drivers too.

When we are editing a Max for Live device that can send and receive MIDI data (MIDI FX or instrument) then it can be confusing as at this particular moment (when Max for Live is in edit mode, i.e. the Max edit patcher is opened but still operated by Live), Max uses its own MIDI drivers!

However, in normal use, Max for Live devices can receive MIDI information from the previous devices in the chain, or of course from the clip inside the same track. Of course, all of these depend on the routing and the monitor setting's of the considered track in Live

4.5 MIDI & audio specific objects for Live connection

4 objects are required:

- [midiin]
- [midiout]
- [plugin~]
- [plugout~]

midiin is the MIDI input

In a device that accepts MIDI information incoming (i.e. a MIDI FX or an instrument), we can grab all MIDI data sent to it inside the patch by using **[midiin]**.

midiout is the MIDI output

Following the same logic, if a MIDI device can send MIDI data (i.e. a MIDI FX), we have to send all data we want to go out of the device to the object **[midiout]**.

plugin~ is the audio input

If a device can receive an audio signal (i.e. an audio FX), we can grab this signal from within the patch by using this object **[plugin~]**.

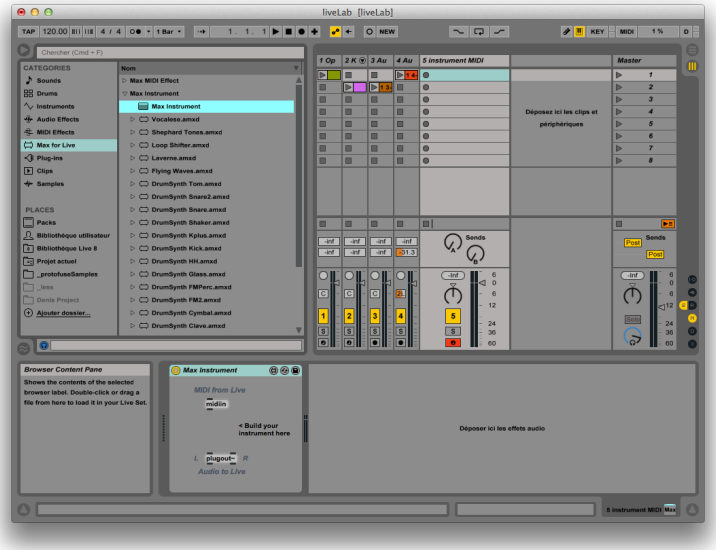
plugout~ is the audio output

If a device can send out an audio signal (i.e. and audio FX or an instrument), we have to send that signal inside the patch to the object **[plugout~]**.

4.6 An example of MIDI instrument

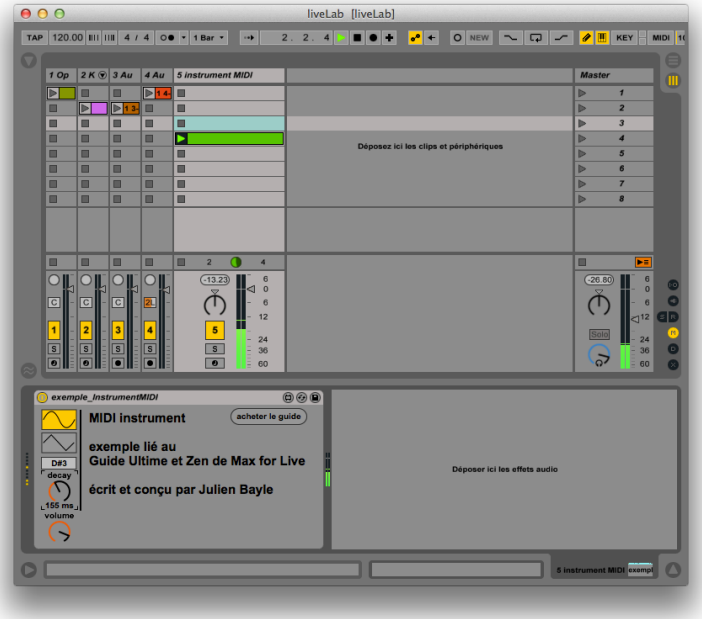
A MIDI instrument is a Max for Live device which owns a MIDI input object **[midiin]** & and audio output object **[plugout~]**.

By drag'n'dropping the default Max for Live Instrument into a track in Live, it creates a device already including **[midiin]** and **[plugout~]** objects:



A MIDI instrument ready to edit

Here is a very basic synthesizer I built.



Small Synthesizer

We can choose the waveform as sine (Max object **[cycle~]**) or saw wave (Max object **[saw~]**)

There are also two rotary knobs for **volume** and **decay**.

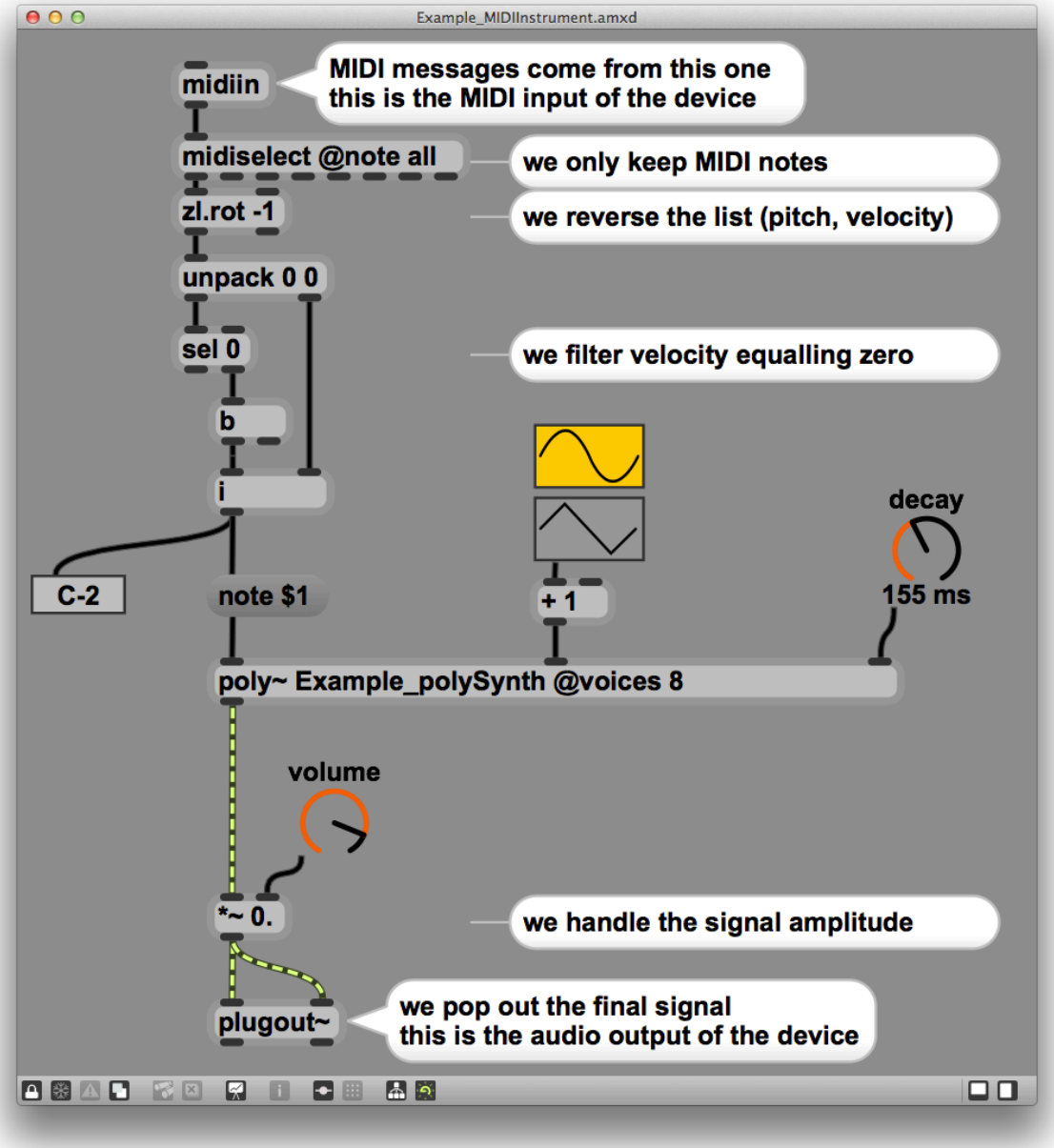
We grab all MIDI messages incoming to the device by using **[midiin]** and we filter them by using

[midiselect] in order to keep only MIDI notes.

Then, with a bit of Max' wizardry, we keep only MIDI notes with a velocity greater than 0. Meaning, we drop all MIDI notes off messages.

Then, we send the pitch number of all incoming MIDI notes to the message **note $1** which transmits the result to the proper synthesizer object.

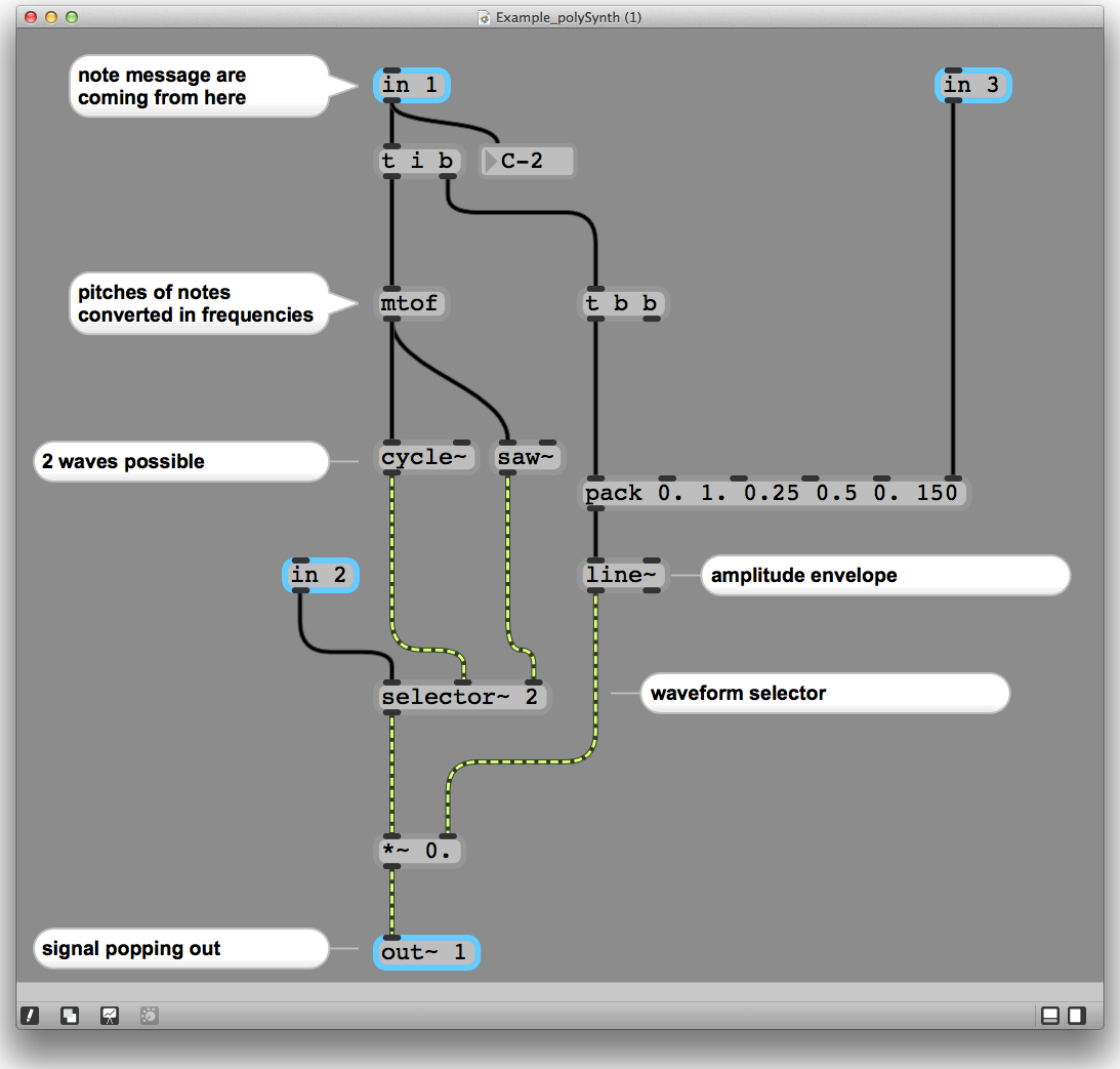
Its structure is very simple and based on the object **[poly~]**.



Cheap synthesizer

[poly~] is an object providing a way to build polyphonic synthesizers, i.e. synthesizers with multiple voices, i.e. able to play more than one voice at the same time, with a voice number limit of course. Beyond that limit, i.e. if we send another note to the synthesizer, we can set it up for instance to cut the previous notes off.

We have to build another patch. This one is saved as a proper **.maxpat** file and we can instantiate it inside **[poly~]** object.



The proper synth itself

I want to focus on three specific objects:

- [mtof]
- [line~]
- [selector~]

The first one converts a MIDI note pitch number into a frequency to tune correctly both oscillators **[cycle~]** & **[saw~]** in order to play the correct note.

The second one provides a simple way to apply an envelope to anything: here we use it to create an amplitude envelope to our signal.

The third is basically a signal selector. We can switch it by sending a command to the leftmost input and it will commute inputs to the output, or not. 0 means all off: no input signal is connected. 1 means the signal in the second input is connected to the output, 2 means the third input is connected etc.

Then, check how the implementation works: we just have to set the voice patch filename as the **[poly~]** object's argument.

We can also see that the output signal goes through another object that will multiply its value. It provides a basic way to control the signal amplitude, i.e. its volume.

At last, the signal is sent to both inputs of **[plugout~]** which is basically the plug to Live inside the devices' chain.

4.7 Example of a sound generator

The name synthesizer is used to describe a device that takes notes and then can produce sounds according to these received notes.

I'm using here the term sound generator to define a device that doesn't take any notes but will generate & produce continuous sounds. A bit like a drone machine would do.

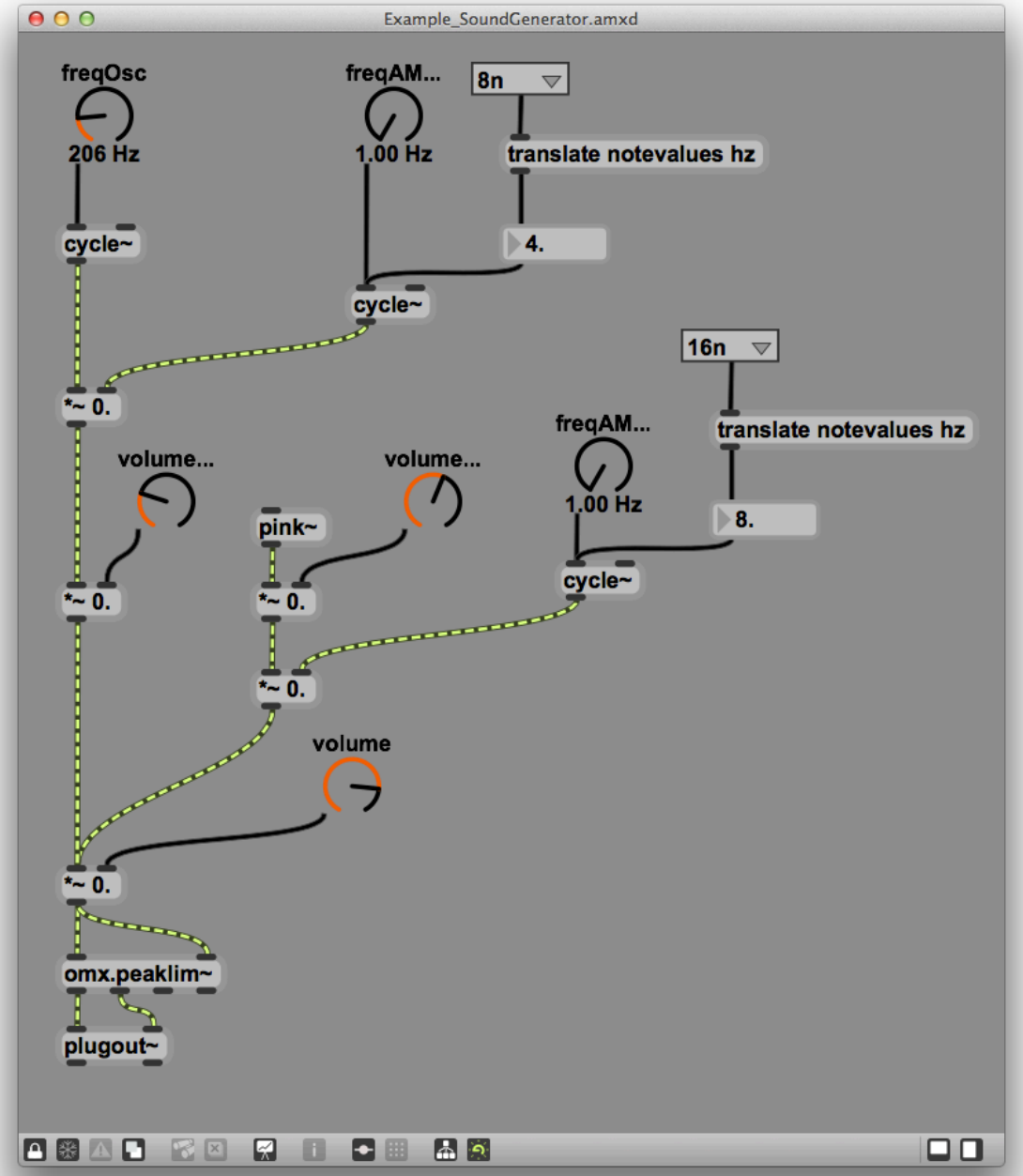
We could use a MIDI instrument type of Max for Live device as we could use an audio FX too. The main thing is: it has to own an audio output to produce sounds in Live.

Let's figure out how we can do that.

This basic sound generator shows some interesting concepts with few objects.

The sound generator integrated in a MIDI track

Let's check the patch itself.



Basic and cheap sound generator

At the top, we can see an object **[cycle~]** which is the basic sinusoidal oscillator in Max. We can choose its frequency by using its first inlet.

The signal produced enters into **[*~ 0.]** that multiplies its amplitude by the value of another signal.

This other signal is another **[cycle~]** for which I designed a basic system to choose its frequency

by using transport relative time values, thanks to the object **[translate]**[5] which converts relative time quantities (here in note values) into absolute time quantities (here Hertz). If Live's tempo is modified, **[translate]** generates the new frequency value. It works a bit like an observer, internally.

We have here quite a fast **amplitude modulation** also named **ring modulation**. Lower values would have made me name it *tremolo*.

The resulting signal is controlled by another volume control.

On the same logic, a noise generator is modulated by another sine wave controllable exactly as the one described before.

Then, both signals are aggregated into another **[*~]** which provides a way to control the global volume.

I'm aggregating here both signals also in order to produce some small digital distortions. I like this, just my cup of tea :)

The whole signal is separated into two equal signals entering in **[omx.peaklim~]**[6] that is a nice and native Max sound limiter.

4.8 MIDI FX example

Let's check another example: a MIDI Fx.

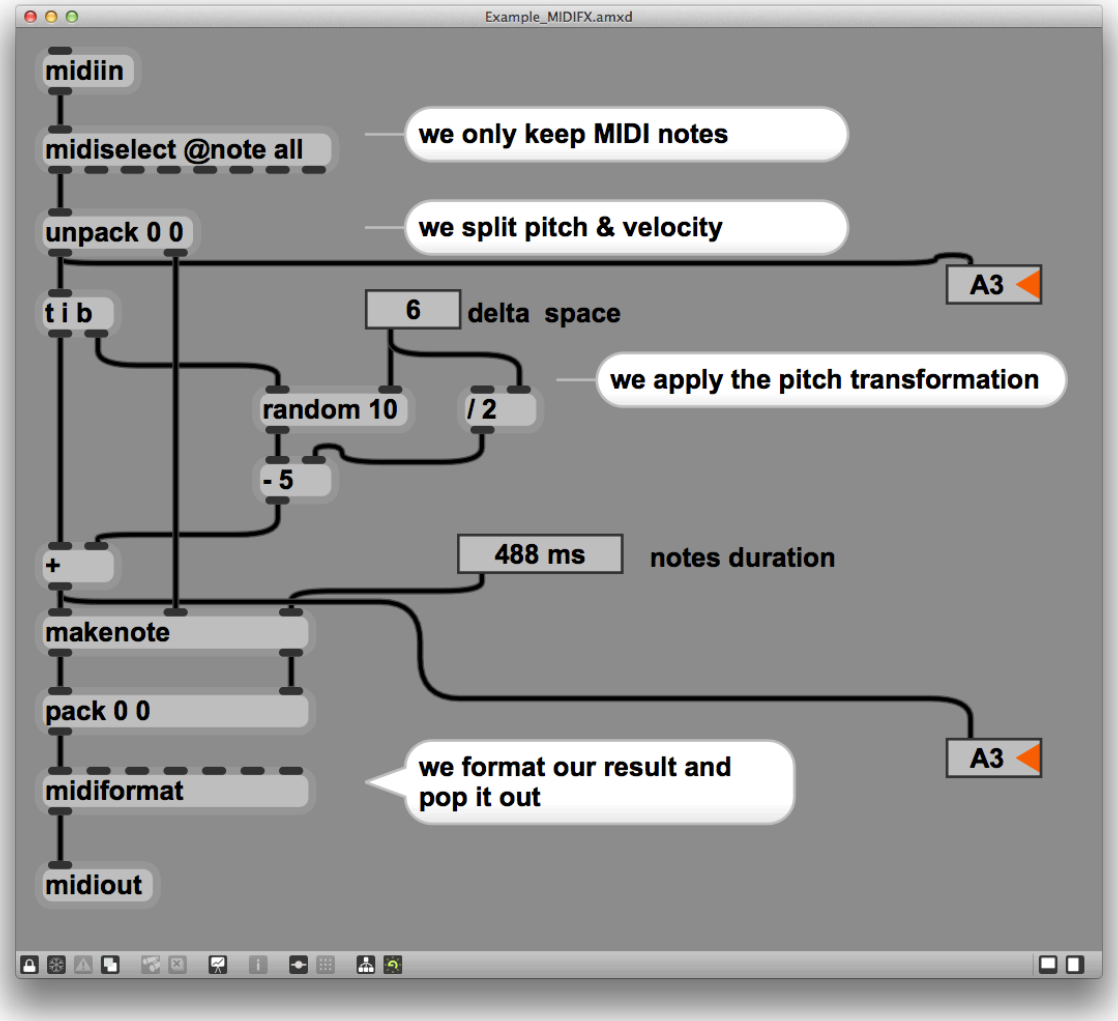
MIDI FX

We have to choose semi-tones interval (named delta space) and then we have to define a note duration in ms.

[5] The object translate

[6] OctiMax limiter object

Let's check the patch itself:



The MIDI Fx Patch

We grab only the notes in this devices and we unpack **pitch** and **velocity** into two separate ways.

We add a random number to the original **pitch**. This one can be positive or negative.

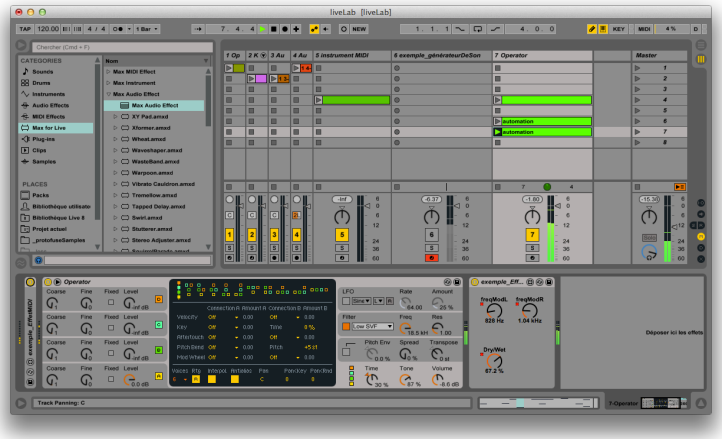
Then, we pass this new pitch calculated with the original velocity to **[makenote]** that will generate **note off** messages after a certain amount of time.

4.9 Audio FX example

Let's now create a basic and kind of **ring-modulator**, home-made!

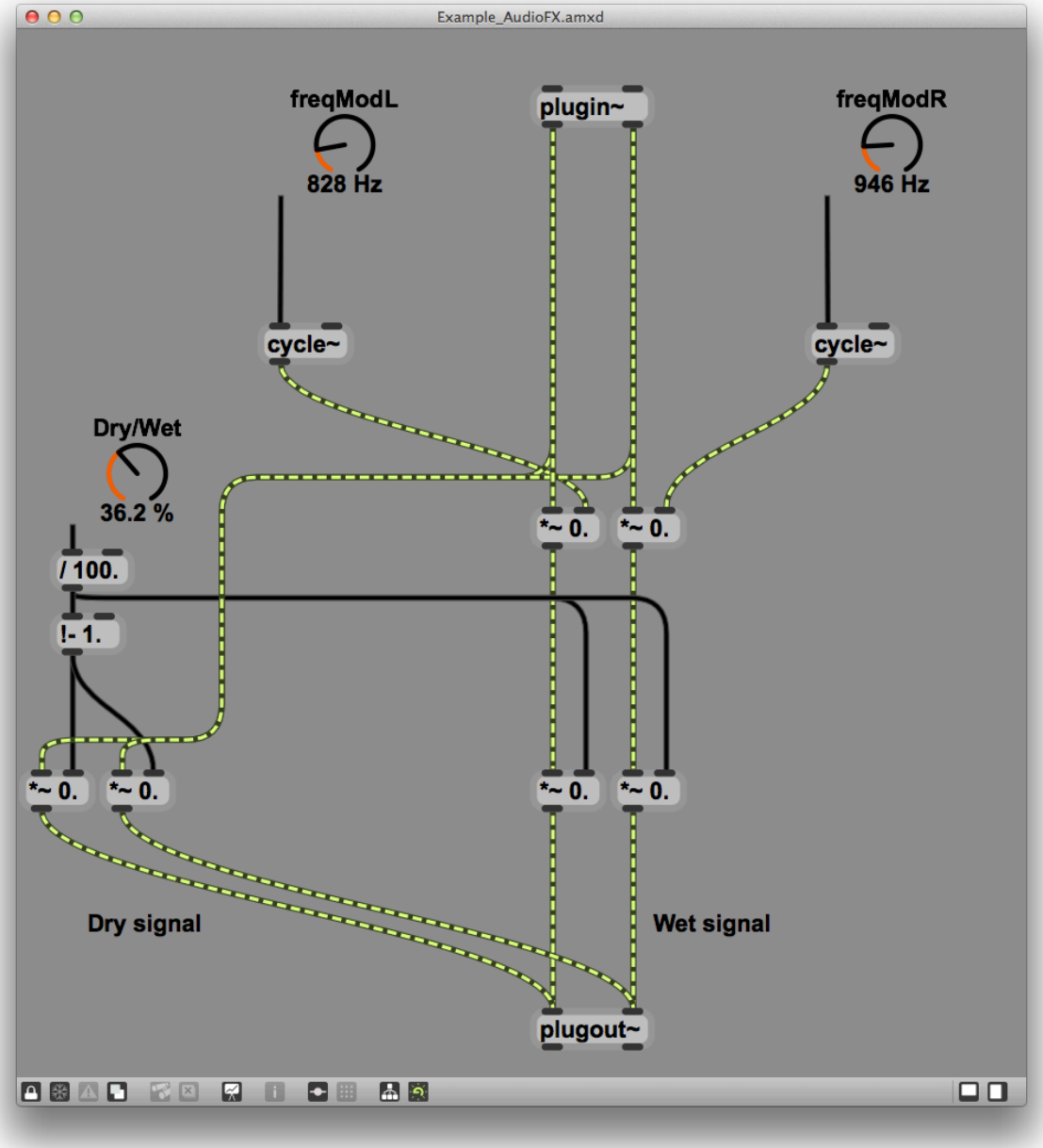
A **ring-modulator** is an FX well-known and available in Live's device **Frequency Shifter**.

Here is the device integrated in Live.



Audio FX

Here is the patch:



Audio FX patch

Here, we have the ring modulation, already described before and you also have a nice example of dry/wet circuits controlled only with one rotary knob.

Conclusions & perspectives

I hope that you learned a lot with this guide.

There are a lot of new concepts coming from Max for Live for both type of users: Max users & Live users.

We can notice, by seeing the big part of this guide focusing on the API & LOM, that Max for Live, besides and beyond MSP & Jitter, IS THE WAY to control Live and especially in live performances conditions.

We can for instance easily create a MIDI/OSC converter, hack scripts for our own custom controller, handle a clips' matrix with algorithms and all of this by using built-in and powerful Live features (UI, audio interface, clips management, presets etc)

This guide is not the end, but just the beginning !

Indeed, I will update it, add things, fix typos etc. If you bought it, you will be informed about each major update directly by email.

Take care and be creative,

Julien