

ReaLearn User Guide

Table of Contents

- [1. Quick start](#)
- [2. Introduction](#)
 - [2.1. What is ReaLearn?](#)
 - [2.2. Videos](#)
 - [2.3. Usage scenarios](#)
- [3. Basics](#)
 - [3.1. Control](#)
 - [3.2. Feedback](#)
 - [3.3. Editing a mapping](#)
 - [3.4. Controller setup](#)
 - [3.5. Automation and rendering](#)
- [4. Companion app](#)
 - [4.1. Connecting to ReaLearn](#)
 - [4.2. Viewing the controller projection](#)
 - [4.3. Editing the controller layout](#)
 - [Editing basics](#)
 - [Composite control elements](#)
 - [Batch-editing control elements](#)
 - [Dealing with the grid](#)
- [5. Tutorials](#)
 - [5.1. Using conditional activation to implement banks/pages](#)
 - [5.2. The same but with previous/next buttons](#)
 - [5.3. Using "Auto-load" to control whatever plug-in is currently in focus](#)
- [6. FAQ](#)
 - [6.1. How many instances and where to put them?](#)
 - [6.2. Should I use conditional activation or the "Enable/disable mapping" target?](#)
 - [More technical explanation](#)
- [7. Controller support](#)
 - [7.1. Explanation](#)
 - [7.2. General tips regarding controller setup and usage](#)
 - [7.3. List of tested controllers](#)
- [8. Available presets](#)
- [9. Reference](#)
 - [9.1. Main panel](#)
 - [Header panel](#)
 - [Input](#)
 - [Output](#)
 - [Menu](#)
 - [Copy listed mappings](#)
 - [Paste mappings \(replace all in group\)](#)
 - [Auto-name listed mappings](#)
 - [Name listed mappings after source](#)
 - [Make sources of all main mappings virtual](#)
 - [Make targets of listed mappings sticky](#)
 - [Move listed mappings to group](#)
 - [Advanced](#)
 - [Options](#)
 - [Server](#)
 - [Open preset folder](#)
 - [Reload all presets from disk](#)
 - [Open Pot Browser](#)
 - [OSC devices](#)
 - [Compartment parameters](#)
 - [Logging](#)
 - [Send feedback now](#)

- [Export to clipboard](#)
- [Import from clipboard](#)
 - ["?" \(Help\)](#)
- [Projection](#)
- [Let through](#)
- [Show](#)
- [Controller preset / Main preset](#)
- [Save as...](#)
- [Delete](#)
- [Notes](#)
- [Add one](#)
- [Learn many](#)
- [Search](#)
- [Filter source](#)
- [Filter target](#)
- [Bottom panel](#)
 - [Instance data...](#)
 - [Instance track](#)
 - [Instance FX](#)
- [Common compartment settings](#)
- [Controller compartment](#)
- [Main compartment](#)
 - [Auto-load](#)
 - [Global FX-to-preset links](#)
 - [Instance-wide FX-to-preset links](#)
- [Mapping row](#)
- [9.2. Mapping panel](#)
 - [General mapping properties](#)
 - [Conditional activation](#)
 - [Source](#)
 - [Category "MIDI"](#)
 - [CC value](#)
 - [Note velocity](#)
 - [Note number](#)
 - [Pitch wheel](#)
 - [Channel after touch](#)
 - [Program change](#)
 - [\(N\)RPN value](#)
 - [Polyphonic after touch](#)
 - [MIDI clock tempo](#)
 - [MIDI clock transport](#)
 - [Raw MIDI / SysEx](#)
 - [MIDI Script](#)
 - [Display](#)
 - [Specific program change](#)
 - [Category "OSC"](#)
 - [Address](#)
 - [Argument](#)
 - [Range](#)
 - [Is relative](#)
 - [Feedback arguments](#)
 - [Category "Keyboard"](#)
 - [MIDI device changes](#)
 - [Category "REAPER"](#)
 - [MIDI device changes](#)
 - [ReaLearn instance start](#)
 - [Timer](#)
 - [ReaLearn parameter](#)
 - [Speech](#)
 - [Category "Virtual"](#)

- [Multi](#)
 - [Button](#)
- [Target](#)
 - [Common target elements](#)
 - [Learn](#)
 - [Menu](#)
 - [Type](#)
 - [Value](#)
 - [Unit button](#)
 - [Common selectors](#)
 - [Selector "Instance"](#)
 - [Selector "Particular"](#)
 - [Selector "At position"](#)
 - [Selector "Named"](#)
 - [Selector "Dynamic"](#)
 - [Common elements and selectors for track targets](#)
 - [Track must be selected](#)
 - [Selection ganging](#)
 - [Respect grouping](#)
 - [Selector "<This>"](#)
 - [Selector "<Selected>"](#)
 - [Selector "<All selected>"](#)
 - [Selector "<Master>"](#)
 - [Selector "All named"](#)
 - [Selector "At TCP position"](#)
 - [Selector "At MCP position"](#)
 - [Selector "Dynamic \(TCP\)"](#)
 - [Selector "Dynamic \(MCP\)"](#)
 - [Selector "By ID or name \(legacy\)"](#)
 - [Common elements for on/off targets](#)
 - [Exclusive](#)
 - [Common elements for send targets](#)
 - [Kind](#)
 - [Send/Receive/Output](#)
 - [Common elements and selectors for FX targets](#)
 - [FX](#)
 - [Selector "<This>"](#)
 - [Selector "<Focused>"](#)
 - [Selector "Particular"](#)
 - [Selector "Named"](#)
 - [Selector "All named"](#)
 - [Selector "By ID or position \(legacy\)"](#)
 - [Input FX](#)
 - [Monitoring FX](#)
 - [FX must have focus](#)
 - [Common elements for pollable targets](#)
 - [Poll for feedback](#)
 - [Category "Real"](#)
 - [Global: Last touched](#)
 - [Global: Mouse](#)
 - [Global: Set automation mode override](#)
 - [Project: Any on \(solo/mute/...\)](#)
 - [Project: Invoke REAPER action](#)
 - [Project: Invoke transport action](#)
 - [Project: Browse tracks](#)
 - [Project: Seek](#)
 - [Project: Set playrate](#)
 - [Project: Set tempo](#)
 - [Marker/region: Go to](#)
 - [Track](#)

- [Track: Arm/disarm](#)
- [Track: Enable/disable all FX](#)
- [Track: Enable/disable parent send](#)
- [Track: Mute/unmute](#)
- [Track: Peak](#)
- [Track: Phase invert/normal](#)
- [Track: Select/unselect](#)
- [Track: Set automation mode](#)
- [Track: Set monitoring mode](#)
- [Track: Set automation touch state](#)
- [Track: Set pan](#)
- [Track: Set stereo pan width](#)
- [Track: Set volume](#)
- [Track: Show/hide](#)
- [Track: Solo/unsolo](#)
- [FX chain: Browse FXs](#)
- [FX](#)
- [FX: Enable/disable](#)
- [FX: Set online/offline](#)
- [FX: Load snapshot](#)
- [FX: Browse presets](#)
- [FX: Open/close](#)
- [FX parameter: Set automation touch state](#)
- [FX parameter: Set value](#)
- [Pot: Browse filter items](#)
- [Pot: Browse presets](#)
- [Pot: Preview preset](#)
- [Pot: Load preset](#)
- [Send: Automation mode](#)
- [Send: Mono/stereo](#)
- [Send: Mute/unmute](#)
- [Send: Phase invert/normal](#)
- [Send: Set automation touch state](#)
- [Send: Set pan](#)
- [Send: Set volume](#)
- [Clip: Invoke transport action](#)
- [Clip: Seek](#)
- [Clip: Volume](#)
- [MIDI: Send message](#)
- [OSC: Send message](#)
- [ReaLearn: Enable/disable instances](#)
- [ReaLearn: Dummy target](#)
- [ReaLearn: Enable/disable mappings](#)
- [ReaLearn: Load mapping snapshot](#)
- [ReaLearn: Modify mapping](#)
- [ReaLearn: Take mapping snapshot](#)
- [ReaLearn: Browse group mappings](#)
- [Category "Virtual"](#)
- [Glue](#)
 - [Reset to defaults](#)
 - [Reverse](#)
 - [Target Min/Max](#)
 - [Value sequence](#)
 - [Group interaction](#)
 - [Feedback type](#)
 - [Numeric feedback: EEL transformation](#)
 - [Textual feedback: Text expression](#)
 - [Dynamic feedback: Lua script](#)
 - [Feedback style](#)
 - [Source Min/Max](#)

- [Out-of-range behavior](#)
- [Mode \("Absolute mode"\)](#)
 - [Normal](#)
 - [Incremental button](#)
 - [Toggle button](#)
 - [Make relative](#)
 - [Performance control](#)
- [Takeover mode](#)
 - [Off](#)
 - [Pick up](#)
 - [Pick up \(tolerant\)](#)
 - [Long time no see](#)
 - [Parallel](#)
 - [Catch up](#)
- [Control transformation \(EEL\)](#)
- [Step size Min/Max](#)
- [Speed Min/Max](#)
- [Encoder filter \(dropdown\)](#)
- [Wrap](#)
- [Make absolute](#)
- [Fire mode](#)
 - [Fire on press \(or release if > 0 ms\)](#)
 - [Fire after timeout](#)
 - [Fire after timeout, keep firing \(turbo\)](#)
 - [Fire on double press](#)
 - [Fire after single press \(if hold < Max ms\)](#)
- [Button filter \(right dropdown\)](#)
- [Bottom section](#)
 - [Target control information](#)
 - [Help](#)
- [9.3. Provided REAPER actions](#)
- [9.4. Advanced settings](#)
 - [The YAML language](#)
 - [Supported configuration properties](#)
 - [Mapping lifecycle actions](#)
- [9.5. Configuration files](#)

Last update of text: 2023-06-05 (v2.16.0-pre.1)

Last update of relevant screenshots: 2021-04-27 (v2.8.0)

1. Quick start

Here's a step-by-step guide to help you get started with ReaLearn and a MIDI controller:

1. Start REAPER.
2. If you haven't already done it, [install ReaLearn via ReaPack](#).
3. Make sure your MIDI controller is enabled in *Options → Preferences... → Audio → MIDI Devices*
 - For the MIDI input device (control), tick *Enable input from this device* and untick *Enable input for control messages*.
 - For the MIDI output device (feedback), tick both *Enable output to this device* and *Do not send reset messages*.

Note

The option *Do not send reset messages* isn't available in older REAPER versions. If you don't use any external hardware synths, you can untick the global options *Reset on: Play* and *Reset on: Stop* instead!

4. Make sure the MIDI device is **not** in use as a REAPER control surface (in *Options → Preferences... → Control/OSC/web*).
5. Check if there's an existing controller preset for your MIDI controller (this is optional but can make things easier).
 - Extensions → ReaPack → Browse packages...
 - Type "realearn controller" in the *Filter* field.
 - You should see a list of ReaLearn controller presets.
 - If you find your controller in the list, right-click it, choose install and press OK.
6. Fire up an instance of ReaLearn
 - If you want your mappings to be specific to a particular project, create a new REAPER project or open an existing one. Right-click the track control panel and choose "Insert virtual instrument on new track...".
 - If you want your mappings to be automatically available in each of your projects, open REAPER's global monitoring FX chain (View → Monitoring FX) instead.
 - Then choose "VSTi: ReaLearn (Helgoboss)"
7. Configure the ReaLearn instance
 - Select your controller's MIDI device as *Input* and *Output* (if you have a controller that supports MIDI feedback).
 - If you have downloaded a controller preset:
 - Switch to *Controller compartment* and select the desired controller preset below.
 - This should fill the list below with so-called *controller mappings*.
 - When you are done, switch back to the *Main compartment*.
8. Add and learn your first mapping
 - Add a first mapping by pressing the *Add one* button.
 - A mapping appears that doesn't do anything yet because it just has a dummy target.
 - Press *Learn source* and move a control element on your MIDI controller.
 - Press *Learn target* and move e.g. the volume fader of a track.
 - Now your control element should control the track volume.

If you want to get the most out of your controller and learn about all of ReaLearn's cool features, please read on.

2. Introduction

2.1. What is ReaLearn?

Probably you know already that ReaLearn is a sort of improvement over REAPER's built-in MIDI/OSC learn. But what is it exactly? Let me put it this way:

ReaLearn is an instrument. It allows you to take whatever MIDI/OSC controller you have, be it a keyboard or some fader box, plug it in and play ... but instead of playing notes, you "play" REAPER itself!

And because ReaLearn supports MIDI/OSC feedback, you can also do the reverse: Let REAPER "play" your controller.

While this is still pretty vague, I think it captures the essence of ReaLearn. From a technical viewpoint it's a VSTi plug-in, so it is an instrument, quite literally. That's one thing that sets it immediately apart from the more conventional control surface feature in REAPER and 3rd-party efforts such as [CSI](#) or [DrivenByMoss](#). The goal of the latter-mentioned is to equip REAPER with support for specific controllers, typically dedicated DAW controllers such as [Mackie MCU](#) that are tailored to control a DAW just like a hardware mixer. And I suppose they do a pretty good job at that.

ReaLearn's approach is quite different: It gives you total control over which control element operates which REAPER parameter and provides you with a *learn* function that allows you build your own control mappings quickly and intuitively without writing configuration files. All of that on a *per-instance* basis. That's right, by default, the mappings are saved as part of the ReaLearn instance and therefore as part of your REAPER project. No need to pollute your global control mappings just for the needs of one project!

Nevertheless, since version 2, ReaLearn is also a great choice for setting up global mappings for usage across multiple projects. Just add ReaLearn to the monitoring FX chain of REAPER (View → Monitoring FX) and ReaLearn will be instantly available in all of your REAPER sessions without having to add it to a project first. In addition, ReaLearn provides a simple yet powerful preset system to make a set of mappings reusable in whatever project you want.

ReaLearn is designed to get the most out of general-purpose MIDI controllers, which - compared to the big and bulky DAW consoles - usually have the advantage of being small, USB-powered, more versatile and easier on the budget. ReaLearn doesn't impose many requirements on your controller. Thanks to features like [Conditional activation](#) and [Projection](#), it can turn even the cheapest MIDI controller into a powerhouse for controlling your DAW.

The usual ReaLearn workflow for a single mapping goes like this:

1. Add a mapping
2. Hit *Learn source* and touch some knob on your controller.
3. Hit *Learn target* and touch some target parameter.
4. Done.

If you want to learn multiple mappings in one go, this gets even easier via the *Learn many* button which can save you a lot of clicks.

The result are mappings that you can customize as you desire, for example by setting a target value range. All of that with MIDI/OSC feedback support, which was previously only available in the less dynamic, more global control surface world.

Summary: *ReaLearn is a sort of instrument for controlling REAPER.*

2.2. Videos

If you want to get a first impression of ReaLearn, a video is surely a good way.

Here's a list of official ReaLearn videos:

- [The ReaLearn Tutorials](#): A series of ReaLearn tutorials.
- [Introduction to ReaLearn 2](#): An in-depth introduction to ReaLearn 2, the sophisticated MIDI-learn plug-in for REAPER.

Here's a short, non-complete list of user-made videos. Please note that at the moment all of them relate to older ReaLearn versions and therefore might be partially outdated:

- [How To: ReaLearn and MIDI Controller for Track Sends in REAPER - Tutorial](#)
- [using ReaLearn to assign MIDI controllers to \(VST\) plugin parameters in Cockos Reaper](#)

2.3. Usage scenarios

Ultimately, ReaLearn gains whatever purpose you can come up with. Because it is a VSTi plug-in and provides many MIDI routing options, it's very flexible in how it can be used. You can "inject" it wherever you want or need it (limitation: using it in a take FX chain is not possible yet):

- **Input FX chain for live-only use:** Put it on a track's input FX chain in order to use it only for incoming "live" MIDI and let it control a parameter of an effect that's on the normal FX chain, right below a synthesizer. It will be active only if the track is armed for recording. All MIDI messages that are used for

parameter control will *automatically* be filtered by default and won't reach the controlled instrument, which is usually exactly what you need.

- **Grid controller for song switching:** Use some grid controller like the [AKAI APC Key 25](#) to arm/disarm various tracks (effectively enabling/disabling certain sound setups) by pressing the grid buttons - with the LEDs of the buttons indicating which setup is currently active.
- **Combination with other MIDI FX for interesting effects:** Slap it on a track FX chain, right between a MIDI arpeggiator and a synthesizer to arpeggiate the cutoff parameter of that synthesizer.
- **Monitoring FX for project-spanning setups:** Put it on the monitoring FX chain to have some control mappings available globally in all projects (similar to conventional control surface stuff).
- **Unusual settings for experimental stuff:** Create a track volume mapping with only feedback turned on. Choose "<FX output>" as MIDI output and play the synthesizer one position below in the FX chain by moving the track volume slider (whatever that might be good for ...).
- **Rotary encoders for avoiding parameter jumps:** How about a refreshingly "normal" use case? Let your rotary endless encoder control a track send volume without parameter jumps and restrict the value range to volumes below 0dB.
- **VST presets for easy reuse:** Save a bunch of commonly used mappings globally as FX presets.
- **Switching controller and main presets separately:** Maintain controller and main presets and switch between them as you like. Easily switch your controller without adjusting your FX presets.
- **Combination of multiple instances:** Use one ReaLearn instance to arm or disarm tracks that contain other ReaLearn instances to enable/disable different mapping groups. Group mappings and activate/deactivate them group-wise simply by instantiating multiple ReaLearn instances and enabling/disabling them as desired in the FX chain window.

... the possibilities are endless. It's all up to you! Use your creativity.

All of that makes ReaLearn especially well-suited for performers, people who use REAPER as a platform for live playing. It might be less interesting to people who are satisfied with a control surface setup off the shelf. But even so, as long as you have some general-purpose MIDI controller and you want a fine-tuned mapping to DAW parameters of all sorts, give ReaLearn a try. It might be just what you need. More so if the controller supports feedback (e.g. motorized faders, LEDs or LCDs).

Summary: *ReaLearn is tailored to usage scenarios typically desired by performers.*

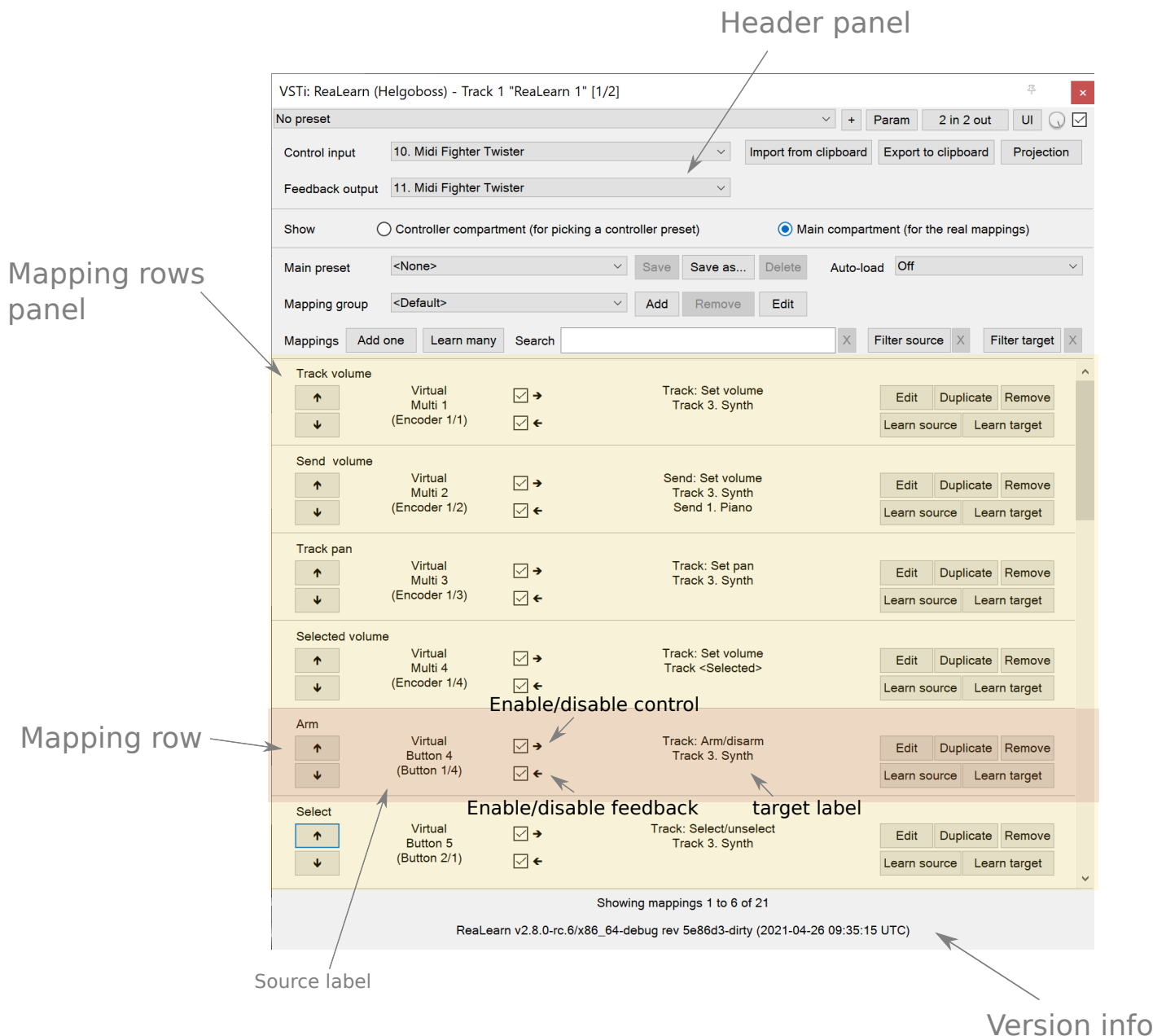
3. Basics

3.1. Control

After installing ReaLearn, you can fire it up just like any other VST instrument in REAPER: By adding it to an FX chain.

1. Right click in the track header area and choose "Insert virtual instrument on new track..."
2. Choose "VSTi: ReaLearn (Helgoboss)"

After that you should see ReaLearn's main panel (unlike this screenshot, it wouldn't contain any mappings yet):



On the very top you see the *header panel* for changing settings or doing things that affect this complete instance of ReaLearn. Below that there's the *mapping rows panel* which displays all main mappings in this instance of ReaLearn. There can be very many of them. On the very bottom you see some information about the version of ReaLearn that you are running.

It can be useful to route all keyboard input to ReaLearn, so you can enter spaces in the "Search" field:

1. Right click ReaLearn FX in the FX chain.
2. Enable "Send all keyboard input to plug-in".

Adding a mapping

Let's see how to add and use our first MIDI mapping:

1. Press the "Add one" button.
 - A new mapping called "1" should appear in the mapping rows panel.
 - For now, it doesn't have any effect. The default target is a [ReaLearn: Dummy target](#) target which basically does nothing.
2. Press the "Learn source" button of that new mapping.
 - Its label will change to "Stop".

3. Touch some control element on your MIDI controller (knob, encoder, fader, button, key, pitch bend, mod wheel, ...). For this example it's best to use something continuous, not a button or key.
 - If your MIDI is set up correctly, the button label should jump back to "Learn source" and the touched control element should appear in the *source label*. See below if this doesn't happen.
4. Press the "Learn target" button.
 - Its label will change to "Stop".
5. Touch the volume fader of your newly created REAPER track.
 - The button label should jump back to "Learn target" and "Track: Set volume" should appear in the *target label*.
6. Now you should be able to control the touched target with your control element.

Troubleshooting

ReaLearn doesn't appear in the list of plug-ins

- Make sure you look in the **VSTi** section (ReaLearn is an instrument).
- If REAPER crashes when scanning for plug-ins and the crash message shows something like `reaper_host64` or `reaper_host32`, you either have a 32/64-bit version mismatch or you have *Preferences → Plug-ins → Compatibility → VST bridging/firewalling* set to "In separate plug-in process" or "In dedicated process per plug-in". Please see the [installation instructions on the project website](#) for hints how to fix this.

ReaLearn doesn't learn MIDI messages

If the label remains at "Stop" at step 3, that means ReaLearn doesn't see the incoming MIDI messages. You need to have a look at your MIDI setup.

- Make sure the MIDI device is **not** installed as REAPER control surface (in *Preferences → Control/OSC/web*).
- Make sure **Enable input from this device** is checked for your controller MIDI input device in the REAPER preferences.

Enable input for control messages is totally irrelevant for ReaLearn. This is only used for REAPER's built-in MIDI learn, which uses the so-called *control MIDI path*.

Note With ReaLearn, you use the same MIDI path for controlling and playing, which is one reason why it is so flexible. It provides local and global MIDI message filtering, so you still don't need to worry about messages that are intended for control but suddenly cause your synthesizer to play MIDI notes.

- Make sure your audio hardware is not stuck (playback in REAPER should work).
- Make sure the track is armed for recording and has the appropriate MIDI device input.

Note This is necessary only if *Input* is set to *<FX input>*, which is the default. If you capture MIDI from a specific device, the track doesn't have to be armed.

- Some controllers, especially DAW controllers, are able to work with several protocols (MCU, HUI, MIDI, ...).
 - For this simple test, it's probably the best to make your controller enter a specific MIDI operation mode.
 - Although MCU and HUI is also just MIDI under the hood, these operation modes are more specialized and therefore need a bit of special attention. ReaLearn conveniently handles these modes when using the *Mackie Control* controller preset.
 - In any case, please consult the [Controller support](#) section, maybe you will find some information about your controller.

When you read this the first time, you might get the impression that this is a lot of work for setting up one simple control mapping. It's not. Learning mappings is a matter of a few secs after you got the hang of it.

ReaLearn also provides the "Learn many" button and a bunch of REAPER actions for one-click learning straight from a REAPER toolbar or triggered by a button on your controller. More about that later.

At this point: Congratulations! You have successfully made your first baby steps with ReaLearn.

Some words about MIDI routing

If you think that what we saw until now is not more than what REAPER's built-in MIDI learn already offers, I can't blame you. First, don't worry, there's more to come, this was just the beginning. Second, there *is* a difference. For some folks, this is an insignificant difference, for others it's a game changer, it depends on the usage scenario. The key to understand this difference is to understand the MIDI *routing*: In above example, *Input* was set to <FX input>. That means we used normal track MIDI messages to control a parameter in REAPER - let's call it *track MIDI path*. This is different from REAPER's built-in MIDI learn, which uses the totally separate *control MIDI path*.

Using the track MIDI path means it's completely up to you to decide what MIDI messages flow into ReaLearn. You decide that by using REAPER's powerful routing capabilities. For example, you can simply "disable" the mapping by disarming your track, a feature that is very desirable if you use REAPER as live instrument. Or you can preprocess incoming MIDI (although that should rarely be necessary given ReaLearn's mapping customization possibilities).

Instead of using <FX input>, you can also pick the MIDI device of your choice directly, in which case ReaLearn will ignore track MIDI messages and capture MIDI messages directly from the already open MIDI device.

Another thing worth to point out which is different from built-in MIDI learn is that we didn't use the action "Track: Set volume for track 01". Benefit: ReaLearn will let you control the volume of the track even if you move that track to another position. The track's position is irrelevant!

3.2. Feedback

In ReaLearn, every mapping has 2 directions: *control* (controller to REAPER) and *feedback* (REAPER to controller). So far we have talked about the *control* direction only: When you move a knob on your controller, something will happen in REAPER. But if your controller supports it, the other direction is possible, too!

Imagine you would use a MIDI-controllable motorized fader as control element to change the track volume. ReaLearn is capable of making that fader move whenever your track volume in REAPER changes - no matter if that change happens through automation or through dragging the fader with your mouse. Motorized faders are quite fancy. Another form of feedback visualisation are rotary encoders with LEDs that indicate the current parameter value.

How to set this up? Often it's just a matter of choosing the correct feedback device:

1. Make sure **Enable output to this device** and **Do not send reset messages** is checked for your controller MIDI output device in the REAPER preferences.

The option *Do not send reset messages* isn't available in some older REAPER versions. If you don't Note use any external hardware synths, you can untick the global options **Reset on: Play** and **Reset on: Stop** instead!

2. In ReaLearn's header panel, select your controller as *MIDI output*.

That should be it!

If it doesn't work and you have ruled out MIDI connection issues, here are some possible causes:

1. **Your controller is not capable of feedback via MIDI messages.**

- Some controllers *do* support feedback, but not via MIDI.
- If they support feedback via OSC, you are lucky because ReaLearn supports that, too. This is discussed in another section.
- If it's another protocol, you are out of luck. Reverse engineering proprietary protocols is out of ReaLearn's scope.
- Recommendation: Maybe you are able to find some bridge driver for your controller that is capable of translating generic MIDI messages to the proprietary protocol. Then it could work.
- Examples: Akai Advance keyboards, Native Instruments Kontrol keyboards

2. Your controller doesn't support feedback via generic MIDI messages but via MIDI SysEx.

- In this case, MIDI feedback is probably still achievable because since version 2.6.0 ReaLearn also supports feedback via MIDI system-exclusive messages. However, it's not going to be straightforward. Unless you find an existing controller preset for your controller, you'll have to read the MIDI specification of your controller (hopefully there is one) ... or you need to experiment a lot.
- Examples: Arturia MiniLab mkII (but we have a controller preset for this one!)

3. Your controller has multiple modes and currently is in the wrong one.

- Some controllers, especially DAW controllers, are able to work with several protocols.
- Recommendation: Consult your controller's manual and take the necessary steps to put it into something like a "generic MIDI" mode.
- Example: Presonus Faderport

4. Your controller expects feedback via messages that are different from the control MIDI messages.

- Usually, controllers with feedback support are kind of symmetric. Here's an example what I mean by that: Let's assume your motorized fader *emits* CC 18 MIDI messages when you move it. That same motorized fader starts to move when it *receives* CC 18 MIDI messages (messages of exactly the same type). That's what I call symmetric. E.g. it's not symmetric if it emits CC 18 but reacts when receiving CC 19.
- ReaLearn assumes that your controller is symmetric. If it's not, you will observe non-working or mixed-up feedback.
- Recommendation: Consult your controller's manual and try to find out which MIDI messages need to be sent to the controller to deliver feedback to the control element in question. Then, split your mapping into two, making the first one a control-only and the second one a feedback-only mapping. Adjust the source of the feedback-only mapping accordingly. In the next section you'll learn how to do that.
- Example: Presonus Faderport

Tip Have a look into the section [Tested controllers](#). Maybe your controller is listed there along with some tips.

3.3. Editing a mapping

When you press the *Edit* button of a mapping row, a so-called *mapping panel* appears, which lets you look at the corresponding mapping in detail and modify it:

Mapping

Name ☒ → Control enabled ☒ ← Feedback enabled

Active Modifier A ☒ Modifier B ☐

☐ Prevent echo feedback ☐ Send feedback after control

Source

Category

Type

ID

Target

Type

Track

FX

Parameter

☐ Input FX ☐ Track must be selected ☐ FX must have focus

Value 0 % 0.0 ms

Tuning

For all source characters

Source Min 0 Max 100

☐ Reverse

Out-of-range behavior

For knobs/faders and buttons (control only)

Mode

Control transformation (EEL, e.g. $y = x^8$ if you want a curve)

For buttons (control only)

Min 0 ms

Max 0 ms

Feedback transformation (EEL, for example $x = 1 - y$)

Help

This panel has 4 sections:

- **Mapping:** Allows to change the name and other general settings related to this mapping.
- **Source:** Allows to edit the *source* of the mapping. In most cases, a source represents a particular control element on your controller (e.g. a fader).
- **Target:** Allows to edit the *target* of the mapping and optionally some target-related activation conditions. A target essentially is the parameter in REAPER that should be controlled.

- **Glue:** Allows to change in detail how your source and target will be glued together. This defines *how* incoming control values from the source should be applied to the target (and vice versa, if feedback is used). This is where it gets interesting. Whereas REAPER's built-in MIDI learn provides just some basic modes like Absolute or Toggle, ReaLearn allows you to customize many more aspects of a mapping.

By design, source, glue and target are independent concepts in ReaLearn. They can be combined freely - although there are some combinations that don't make too much sense.

Changes in the mapping panel are applied immediately. Pressing the *OK* button just closes the panel.

Tip: It is possible to have up to 4 mapping panels open at the same time.

3.4. Controller setup

In order to get the most out of your controller in combination with ReaLearn, you should consider the general hints given in the section [Tested controllers](#).

3.5. Automation and rendering

Similarly to control surfaces, ReaLearn is primarily meant to be used for controlling targets "live". If you want to *persist* the resulting target value changes, you can do so by writing automation. Just as with any other automation, it will be included when you render your project.

It is possible to feed ReaLearn with track MIDI items instead of live MIDI data. This results in a kind of *pseudo automation*. Some users call this *MIDI CC based automation*. This feature can be quite interesting and appealing to MIDI fans.

Support for rendering pseudo automation is limited!

Let's say you finally want to render your project. If you don't watch out, your pseudo automation will simply be ignored!

Pseudo automation will only be rendered if you follow some very distinct rules:

Caution

- The target must be [FX parameter: Set value](#) (all other targets will most likely be ignored).
- The targeted FX must be on the same track as the ReaLearn instance itself.
- This only works in REAPER versions $\geq 6.52 + \text{dev0324}$.

I remember that **Online Render** used to respect all kinds of pseudo automation. However, this must have stopped working at some point (or it works only under particular circumstances or with certain settings, not sure). Anyway, now you need to follow the same rules as with offline rendering to make pseudo automation work.

4. Companion app

This section is about the *ReaLearn Companion* app, which is a separate software that powers ReaLearn's [Projection](#) feature.

At the moment it comes as [Android app](#) and [web app](#). The iOS app has not been published yet. The source code is available [here at GitHub](#).

4.1. Connecting to ReaLearn

The start screen lets you connect to a specific ReaLearn instance by scanning the QR code that pops up when pressing ReaLearn's [Projection button](#). It's also possible to enter the connection data manually, in case your device doesn't have a camera or in case you are using the web app (in which QR code scanning often doesn't work so well). If you are experiencing issues, follow the instructions given by the app and the setup guide which is displayed when pressing the [Projection](#) button!

Please note, if all you want is to open the web app on the computer that also runs REAPER/ReaLearn, you don't need to bother with QR codes or connection data at all. Just follow the link that is displayed in the setup guide.

ReaLearn allows many Companion apps to connect to it simultaneously, there's no artificial limit.

4.2. Viewing the controller projection

As soon as you have connected, you should be able to see the controller projection, which consists of both the controller layout and the current mapping of its control elements. If not, the app will give you a hint what's

missing. The control element labels will reflect the labels of your main mappings.

You can tap the screen to make the app bar disappear or reappear. There's a menu on the right side of the app bar which lets you change various aspects of the appearance. Just give it a try! Dark mode combined with high-contrast is especially nice on devices with OLED displays! All of these settings will be saved on your device, not in ReaLearn's controller preset.

Another thing you can do here is applying two-finger gestures in order to zoom/pinch.

4.3. Editing the controller layout

Pressing the pencil button in the app bar lets you enter edit mode. As soon as you do that, the control element labels will reflect the labels of your controller mappings and a palette will appear on the side of the screen.

Editing basics

You can drag the controls freely from the palette onto the scene and back. Pressing a control element opens a panel which lets you change its appearance. The two labels mentioned there are used in the following way:

1. If the control element is a composite control element (see below, e.g. push encoder), the first label represents the mapping of the first inner control element (e.g. the encoder) and the second label represents the mapping of the second inner control element (e.g. the button). See the *Midi Fighter Twister controller preset* for a real-world usage of this feature.
2. If the control element is just a normal control element, the second label is usually empty. Except this control element has more than one main mapping assigned: In that case the second label shows the second main mapping.

Whenever you press the save button (floppy disk) in the app bar, the layout is saved - not on your specific device but as part of ReaLearn's controller preset! So this same layout will automatically be available to all other connected Companion apps.

You can leave the edit mode by pressing the pencil button again. This gives you a preview of your current changes.

Attention: If you leave the controller projection view (e.g. by pressing the upper left arrow) or if you change your controller preset from within ReaLearn, all non-saved controller layout changes will be lost! So it's a good idea to save often. Once saved, there's no undo though. You can back up temporary states by copying the corresponding controller preset files (on the computer running ReaLearn) to a temporary backup location (see *Save as...* button documentation in the [Header panel](#) section).

Composite control elements

If you want one visual control element to contain 2 logical control elements (e.g. a push encoder = encoder + button), just move one control element onto another one - and they will merge into a composite control element. If you want to undo this merging, move the merged control element back on the palette - they will split up and you can drag them onto the scene again.

Batch-editing control elements

Sometimes it's a bit tedious to edit each control element separately. As soon as you long-press one control element, the Companion app will enter multi-edit mode and you can start adding/removing other control elements to/from the selection by just tapping them. When you move one element of the selection, all others will also be moved. You can open the control element appearance panel by long-pressing an element. All changes made in the panel will immediately be applied to all selected elements.

You can leave multi-edit mode either by unselecting all elements or by (temporarily) leaving the edit mode.

Known issue: In the web app, multi-edit mode currently doesn't work, there's a graphical glitch.

Dealing with the grid

You can hide the grid using the app bar menu. The grid will still have an effect though.

One way to get more fine-grained positioning is by decreasing the grid size. However, it doesn't go below a certain minimum and changing the grid size after already having positioned lots of elements might not be the best idea. Usually, the better way is to just expand the scene. Don't worry, your layout will always fit on the screen, no matter how large the scene actually is in terms of grid squares!

You can enlarge the scene by slightly moving a control element out of the scene. Do so in small steps and you will automatically have more space at your disposal. The scene will always be as big as the imaginary rectangle from the top-left control element to the bottom-right control element!

5. Tutorials

The screenshots in this section are slightly out of date. If you feel like contributing to the project, this is an area where you could help.

5.1. Using conditional activation to implement banks/pages

Users often ask if it's possible to do control surface bank-style mapping in order to switch to a completely different set of mappings with the press of a button. Yes, it is! It's done using the *conditional activation* feature with the activation mode "When bank selected".

Tip ReaLearn \geq 2.11.0 provides an alternative and probably more straightforward way to implement banks: The [ReaLearn: Enable/disable mappings](#) target.

I'll show you a minimal example but in great detail. Once you understand this example, you should be able to progress to bigger things. So let's assume you have 2 knobs and 2 buttons on your controller and you want to map some controls to parameters of the [Vital synth](#). Here's our goal:

- **Knob K1:** Controls decay of ENV X
- **Knob K2:** Controls frequency of LFO X
- **Button B1:** Sets X to 1
- **Button B2:** Sets X to 2

Step 1: Add all desired mappings

First, it's important to understand that conditional activation does one thing only: It switches mappings on or off. It doesn't magically change the target of a mapping or anything like that. Just on or off! Thus, the first thing you should do is adding all the knob mappings (for example by using "Learn many"). Here's the result:

VSTi: ReaLearn (Helgoboss) - Track 1 "ReaLearn"

No preset + Param 2 in 2 out UI

MIDI control input 10. Midi Fighter Twister Import from clipboard Export to clipboard Projection

MIDI feedback output 11. Midi Fighter Twister Let through: ☐ Matched events ☒ Unmatched events

Compartment Main mappings Preset <None> Save Save as... Delete Reload all

Mapping group <Default> Add Remove Edit Auto-load preset Off

Mappings Add one Learn many Search Filter source X Filter target X

ENV 1 Decay	Virtual Multi 1 (K1)	<input checked="" type="checkbox"/> → <input checked="" type="checkbox"/> ←	Track FX parameter Track 3. Vital FX 1. VSTi: Vital (Vital Audio) Param 51. Envelope 1 Decay	Edit Duplicate Remove Learn source Learn target
ENV 2 Decay	Virtual Multi 1 (K1)	<input checked="" type="checkbox"/> → <input checked="" type="checkbox"/> ←	Track FX parameter Track 3. Vital FX 1. VSTi: Vital (Vital Audio) Param 58. Envelope 2 Decay	Edit Duplicate Remove Learn source Learn target
LFO 1 Freq	Virtual Multi 2 (K2)	<input checked="" type="checkbox"/> → <input checked="" type="checkbox"/> ←	Track FX parameter Track 3. Vital FX 1. VSTi: Vital (Vital Audio) Param 158. LFO 1 Frequency	Edit Duplicate Remove Learn source Learn target
LFO 2 Freq	Virtual Multi 2 (K2)	<input checked="" type="checkbox"/> → <input checked="" type="checkbox"/> ←	Track FX parameter Track 3. Vital FX 1. VSTi: Vital (Vital Audio) Param 165. LFO 2 Frequency	Edit Duplicate Remove Learn source Learn target

Showing mappings 1 to 4 of 4

ReaLearn v1.12.0-pre12/x86_64 rev de942f (2021-01-20 14:50:42 UTC)

Note: As you can see, I gave the mappings friendly names, which is nice in general but really pays off once you use the projection feature. Also note that I used my Midi Fighter Twister preset and renamed the relevant encoders to K1 and K2.

At this point, all those mappings are always active, so moving K1 will affect both ENV 1 and ENV 2 decay whereas moving K2 will affect both LFO 1 and LFO 2 frequency! We need activation conditions to make sure that not all mappings are active at the same time.

Step 2: Assign mappings to groups

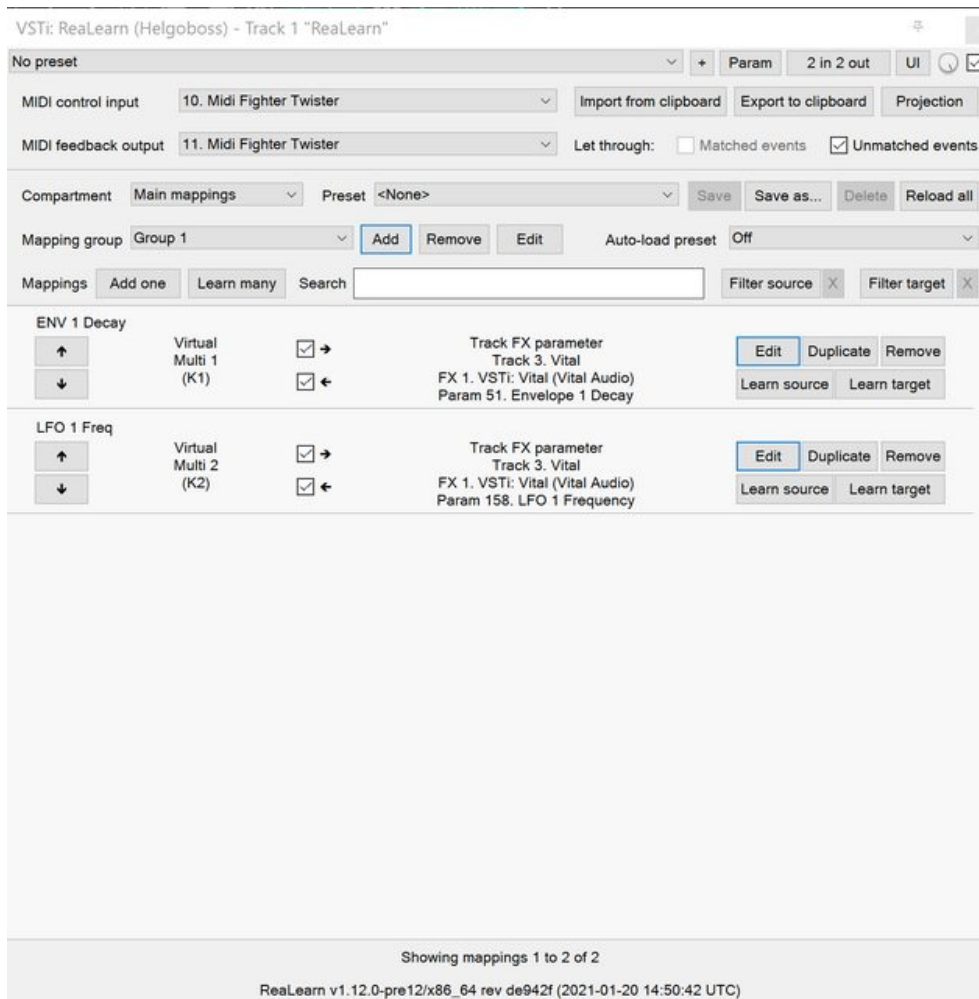
Now we could shoot ahead and directly set the activation condition of each mapping individually. **But** usually it's much better to activate/deactivate complete *groups* of mappings. When you press button B1, you want to have the "ENV 1 Decay" and "LFO 1 Freq" mappings active (= "Group 1"). When you press button B2, you want "ENV 2 Decay" and "LFO 2 Freq" to be active instead (= "Group 2"). And this is just a minimal example. You will probably have many more mappings in one group in the end.

Turns out, ReaLearn has something made exactly for that: Mapping groups. Using them will make your life way easier. We will create those 2 groups and distribute our knob mappings into both groups.

1. Right to "Mapping group", press "Add" and enter the name "Group 1". Repeat the same for "Group 2".
2. Select mapping group <Default> again.

- Now move every mapping to its corresponding group by right-clicking the mapping row and choosing the desired group.

Here's how "Group 1" looks like after this:



Please note that until now, this is purely cosmetic. It hasn't changed in any way how the mappings work.

Step 3: Set group activation conditions

Now let's set the activation conditions. First for "Group 1":

1. Select mapping group "Group 1".
2. Press "Edit".
3. In the "Active" dropdown, choose "When bank selected". Make sure that "Parameter" is set to "1. Parameter 1" and "Bank" to 0.

Repeat the same for "Group 2", but set "Bank" to 1. Should look like this:

Compartment
Main mappings
Preset
<None>
Save
Save as...
Delete
Reload all

Mapping group
Group 2
Add
Remove
Edit
Auto-load preset
Off

Mappings
Add one
Learn many
Search
Filter source
Filter target

ENV 2 Decay

↑
↓

Virtual Multi 1 (K1)

☒ →
☒ ←

Track FX parameter
Track 3. Vital
FX 1. VSTi: Vital (Vital Audio)
Param 58. Envelope 2 Decay

Edit
Duplicate
Remove
Learn source
Learn target

LFO 2 Freq

↑
↓

Virtual Multi 2 (K2)

☒ →
☒ ←

Track FX parameter
Track 3. Vital
FX 1. VSTi: Vital (Vital Audio)
Param 165. LFO 2 Frequency

Edit
Duplicate
Remove
Learn source
Learn target

Edit group

Name
Group 2
☒ → Control enabled
☒ ← Feedback enabled

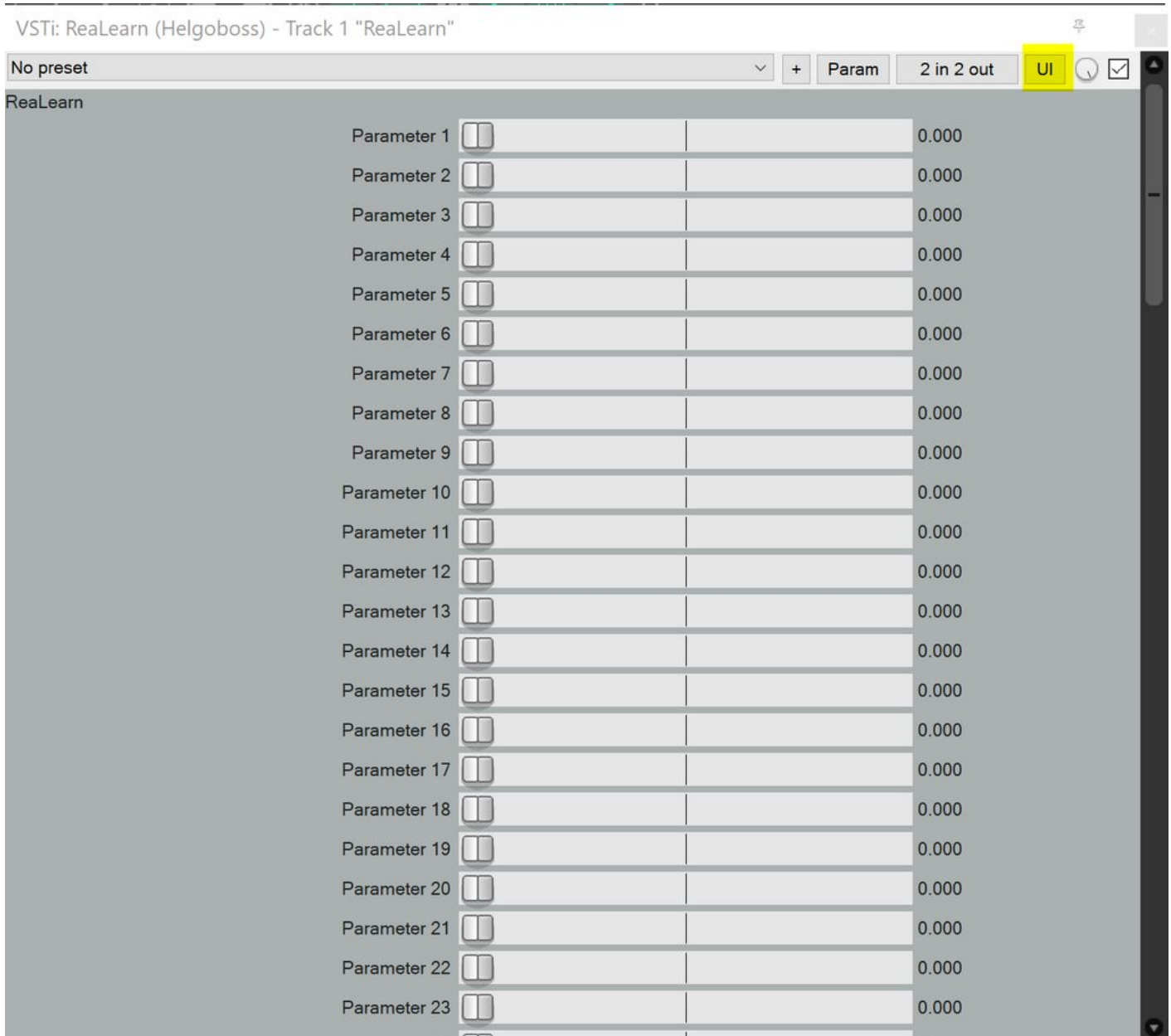
Active
When program selected
Bank
1. Parameter 1
Program
1

OK

Did you see how the mappings in "Group 2" turned grey? That means they became inactive! At this point, moving the knobs should affect ENV 1 and LFO 1 only.

Step 4: Understand "Parameter" and "Bank"

In the previous step, we have set "Parameter" to "Parameter 1". It's important to understand that we are talking about ReaLearn's own VST parameters. Each ReaLearn instance has 200 free parameters (100 per compartment) which don't do anything by default. One easy way to make them visible is by pressing the "UI" button at the top right of the FX window to switch to the parameter view:



See "Parameter 1" at the top? That's the one we used in our activation condition! Hence, once we change the value of this parameter, mappings will get activated or deactivated. You can try it! Move the parameter slider a bit to the right and you will observe that "Group 1" turned inactive. "Group 1" will be active when the slider is on the very left. "Group 2" will be active when the slider is pushed *slightly* more to the right. If you push it even more to the right, none of the mappings will be active. Enough! Press "UI" again to go back to the ReaLearn user interface.

Now that we know that the value of ReaLearn's internal "Parameter 1" is the key to activate/deactivate our mappings, the next step should be obvious: We need to map our buttons to it!

Step 5: Map buttons to bank parameter

We are going to map the buttons to "Parameter 1". Button B1 will set its value to 0 and button B2 will set its value to 1. Remember how we defined these two numbers in the activation conditions ... they are the "Bank" numbers!

1. Select mapping group <Default>.
2. Map the two buttons. The easiest way is to use "Learn many", switch to the parameter view once again and move the "Parameter 1" slider whenever ReaLearn asks you to touch the target.
 - Before you continue, make sure your screen looks similar to this (take note how I've given the mappings friendly names again):

VSTi: ReaLearn (Helgoboss) - Track 1 "ReaLearn"

No preset + Param 2 in 2 out UI ☒

MIDI control input 10. Midi Fighter Twister Import from clipboard Export to clipboard Projection

MIDI feedback output 11. Midi Fighter Twister Let through: ☐ Matched events ☒ Unmatched events

Compartment Main mappings Preset <None> Save Save as... Delete Reload all

Mapping group <Default> Add Remove Edit Auto-load preset Off

Mappings Add one Learn many Search Filter source Filter target

Activate group	Source	Target	Actions
Activate group 1	Virtual Button 13 (B1)	Track FX parameter Track <This> FX 1. VSTi: ReaLearn (Helgoboss) Param 1. Parameter 1	Edit Duplicate Remove Learn source Learn target
Activate group 2	Virtual Button 14 (B2)	Track FX parameter Track <This> FX 1. VSTi: ReaLearn (Helgoboss) Param 1. Parameter 1	Edit Duplicate Remove Learn source Learn target

Showing mappings 1 to 2 of 2

ReaLearn v1.12.0-pre12/x86_64 rev de942f (2021-01-20 14:50:42 UTC)

3. Edit the mapping for button B1 and set both Target Min/Max to 0 (this causes the button to always set the fixed value 0).
 - If you have a controller that is capable of feedback (button has LED), also set "Out-of-range behavior" to "Min". This makes sure that the LED lights up whenever this bank is selected but switches off otherwise.
4. Edit the mapping for button B2 and set both Target Min/Max to 1.

- Here's how the mapping panel for button B2 looks afterwards:

Mapping "Activate group 2"

Mapping

Name: Activate group 2 ☒ → Control enabled ☒ ← Feedback enabled

Active: Always

☐ Prevent echo feedback ☐ Send feedback after control Find in mapping list

Source

Learn

Category: Virtual

Type: Button

Number: 14 (B2)

Target

Learn Go there

Category: REAPER

Type: Track FX parameter

Track: <This>

FX: 1. VSTi: ReaLearn (Helgoboss) By ID or pc

Parameter: 1. Parameter 1

☐ Input FX ☐ Track must be selected ☐ FX must have focus

Value: 0 % 0.000

Tuning

Reset to defaults

For all source characters (but encoders feedback only)

Min: 0

Source

Max: 100

☐ Reverse

Out-of-range behavior: Min or max

For knobs/faders and buttons (control only)

Mode: Normal

Min: 0 % 0.000

Jump

Max: 100 % 100 %

☐ Slowly approach if jump too big

Control transformation (EEL, for example $y = 1 - x$)

For all source characters (control and feedback)

Min: 1 % 1 %

Target

Max: 1 % 1 %

Feedback transformation (EEL, for example $x = 1 - y$)

For encoders and incremental buttons (control only)

Step size

Min: 1 % 1

Max: 1 % 1

☐ Rotate

For buttons (control only)

Length

Min: 0 ms

Max: 0 ms

OK

- If feedback is desired, set "Out-of-range behavior" as described in the previous step.

That's it, the goal is achieved! Press the buttons and move the knobs to test it.

You might wonder why ReaLearn has been designed to use this particular mechanism for activating/deactivating mappings, in particular why it uses generic parameters to do the job. The answer is: This mechanism is insanely powerful. If you take the time and digest this for a while, you will realize that you can do almost anything with a clever combination of the "Mapping", "Parameter" and "Activation condition" concepts. This scenario is just one of many. Just see the next tutorial to understand why.

5.2. The same but with previous/next buttons

Now let's assume you don't want 2 buttons where each button should activate one particular bank but you want previous/next buttons to switch between the banks. Do everything as in tutorial 1 with the exception of step 5.

ReaLearn \geq 2.11.0 provides an alternative and probably more straightforward way to implement cycling Tip through banks with previous/next buttons: By combining multiple mappings with [ReaLearn: Enable/disable mappings](#) target (for defining the banks) with one [ReaLearn: Browse group mappings](#) targets (for cycling).

Step 5: Map buttons to bank parameter

1. As in tutorial 1.
2. As in tutorial 1.
3. Edit the mapping for button B2 ("Next group") and set mode to "Incremental button"
4. Edit the mapping for button B1 ("Previous group"), set mode to "Incremental button" *and* check the "Reverse" box (because you want to go in the other direction).

The "Previous group" mapping then looks like this:

The screenshot shows the 'Mapping Previous group' dialog box. The 'Mapping' section has 'Name' set to 'Previous group', 'Active' set to 'Always', and checkboxes for 'Control enabled' and 'Feedback enabled'. Below are checkboxes for 'Prevent echo feedback' and 'Send feedback after control', and a 'Find in mapping list' button. The 'Source' section has 'Learn', 'Category' set to 'Virtual', 'Type' set to 'Button', and 'Number' set to '13 (B1)'. The 'Target' section has 'Learn', 'Go there', 'Category' set to 'REAPER', 'Type' set to 'Track FX parameter', 'Track' set to '<This>', 'FX' set to '1. VSTi: ReaLearn (Helgoboss)', and 'Parameter' set to '1. Parameter 1'. There are checkboxes for 'Input FX', 'Track must be selected', and 'FX must have focus'. A 'Value' slider is set to 0. The 'Tuning' section has a 'Reset to defaults' button. It contains four sub-sections: 'For all source characters (but encoders feedback only)' with 'Min' at 0 and 'Max' at 100, and a checked 'Reverse' checkbox; 'For all source characters (control and feedback)' with 'Min' at 0 and 'Max' at 100; 'For knobs/faders and buttons (control only)' with 'Mode' set to 'Incremental buttons'; and 'For encoders and incremental buttons (control only)' with 'Step size' 'Min' and 'Max' both at 1, and a 'Rotate' checkbox. At the bottom, there is a 'For buttons (control only)' section with 'Length' 'Min' and 'Max' both at 0 ms. An 'OK' button is at the bottom center.

5.3. Using "Auto-load" to control whatever plug-in is currently in focus

This one seems to be a very popular use case: To create a dedicated set of mappings for a specific FX plug-in and load these mappings whenever focusing that plug-in on the screen. The easiest way to do this is to use the "Auto-load" feature.

To have a nice example, let's assume you want to build a first set of mappings for the VSTi plug-in [Vital](#). The procedure for other plug-ins is the same.

Step 1: Activate the correct controller preset

Before you start, I strongly recommend downloading a ReaLearn controller preset for your specific controller from ReaPack and activate it in the *controller compartment*. You will need to right-click the header panel and choose [Reload all presets from disk](#) to make a newly downloaded controller preset appear in the preset list. If there's no suitable preset for your controller available on ReaPack or in the [list of unofficial controller presets](#), build your own.

This step is completely optional, but it gives you many advantages, both in the short and long run. Please see section [Controller compartment](#) for details.

Step 2: Create mappings for your FX plug-in

In this step you will tell ReaLearn which control element on your controller should control which parameter of your FX plug-in:

1. Add Vital VSTi and a new *empty* ReaLearn instance, both as track FX (**not as monitoring FX**), preferably side-by-side so that you can see both.
 - It's also possible to use monitoring FX but for this particular tutorial it's important to use track FX (otherwise ReaLearn will not ask you if it should make your mappings project-independent in step 3).
2. In ReaLearn, press *Learn many*.
3. Move a control element on your controller, change a Vital parameter, move another control element, change another Vital parameter ... until you are finished!
4. Press *Stop*.

Step 3: Save mappings as main preset and link it to the FX type

Now let's save your newly created set of mappings as preset and link the preset to the Vital VSTi plug-in:

1. Make sure the *main* compartment is shown.
2. Press *Save as...* (next to *Preset*).
 - ReaLearn will ask you if it should make your mappings project-independent. Answer with *Yes* (important).
3. Enter a descriptive preset name, e.g. "Vital".
4. Right-click ReaLearn's header panel → Global FX-to-preset links → <Add link from FX "Vital.dll" to...> and choose the previously created "Vital" preset.
 - The name *Vital.dll* can vary, depending on your operating system.
 - If it doesn't mention *Vital* but another VST plug-in, focus your Vital VSTi plug-in instance for a moment and then go directly to ReaLearn and right-click the header panel.

Step 4: Activate "Auto-load"

Now you just have to set *Auto-load* to *Based on instance FX*. Since the [Instance FX](#) is by default the currently focused FX, ReaLearn will from now on activate your "Vital" preset whenever Vital VSTi plug-in has focus. If you want this in all projects without having to add ReaLearn to each project manually, add a dedicated ReaLearn instance to REAPER's monitoring FX chain (REAPER → View → Monitoring FX).

6. FAQ

6.1. How many instances and where to put them?

Since ReaLearn is a VST instrument, you can have many instances of it, not just one. A question that comes up pretty often is how many is right and where to put them?

There's no definitive answer to that, it all depends on what you want. Here are some basic rules of thumb:

1. You need at least one ReaLearn instance per controller.
2. For mappings that shall be available in any existing or new project automatically, create a ReaLearn instance on the monitoring FX chain
 - If you want to use multiple controllers, simply add multiple instances to the monitoring FX chain.
3. For in-project mappings which control arbitrary parameters on arbitrary tracks, you are totally free to choose where to put ReaLearn, there shouldn't be any differences in behavior.
 - Putting it on the master FX chain is a good choice if you want it out of your way and usually not be visible.
 - However, in order to be reminded that I use ReaLearn, I usually add some track at the very top and put all ReaLearn instances on its FX chain.
4. Let's say you have a VST instrument that you want to activate/deactivate for live playing by arming/disarming the corresponding track. And now you want to use ReaLearn as a kind of "Insert" effect for controlling parameters of that VST instrument (or other track-local parameters such as the track volume), only when it's active. Then put ReaLearn exactly on that track, somewhere *above* your VST instrument and select MIDI input <FX input>.
 - That way your VST instrument won't receive MIDI that's already processed by ReaLearn, which is cool.
 - If you want ReaLearn to only receive MIDI messages that originate from live playing and ignore MIDI that originates from track MIDI items, put it on the input FX chain of that track.

6.2. Should I use conditional activation or the "Enable/disable mapping" target?

ReaLearn 2.11.0 introduces an alternative to [Conditional activation](#): The ability to tag mappings and enable/disable them via the [ReaLearn: Enable/disable mappings](#) target. In general, one can say that conditional activation is slightly more powerful but that the enable/disable target is easier to use and enough in most common use cases.

You strictly need to use conditional activation if you ...

- ... want to activate/deactivate mappings using REAPER automation envelopes.
- ... want to sync the active/inactive state of a mapping with a [dynamic track selector](#) expression.
- ... want to activate/deactivate mappings in another ReaLearn instance.
- ... want to activate/deactivate mappings in another compartment within the same ReaLearn instance.

Note This particular limitation of the enable/disable target might disappear in the future.

You might prefer conditional activation if you ...

- ... have a modifier use case (not a bank switching use case).
 - It can be quite intuitive to think of modifiers as parameters that you define once and refer to them in the mapping (to be activated or deactivated) itself.
- ... have a modifier use case in which you want to combine multiple modifiers (e.g. activate a mapping only if the *Shift* and *Control* button is pressed at the same time)
 - This is really much easier to achieve using conditional activation, think about it.
- ... prefer that the mapping itself defines when it should be active or not (instead of dictating it via tags).

In all other circumstances the enable/disable target should be fine.

More technical explanation

Conditional activation introduces a level of indirection. It allows you to look at the act of enabling/disabling mappings as *two different concerns* by introducing an intermediate concept called a "modifier" (respectively a "bank").

In particular, it separates the following two concerns:

1. Switching a modifier on/off (respectively activating a bank)
2. Let other mappings follow the on/off state of the modifier (respectively the active bank)

You define these two concerns in different mappings:

1. "Modifier/bank-changing mappings"
2. "Modifier/bank-dependent mappings" (as activation condition)

If you use the alternative to conditional activation, [ReaLearn: Enable/disable mappings](#), you throw both of these concerns into one mapping!

The advantage of separating these two concerns is that you can change **1** (the modifier/bank-changing mappings, e.g. which button controls the modifier and how) without having to touch **2** (the dependent mappings)! And vice versa. That can make complex setups easier to understand and reason about!

The disadvantage is that it makes simple setups a bit harder to understand than necessary since you need at least two mappings instead of only one.

As always: Choose the right tool for the job and consider starting off with the easiest tool.

7. Controller support

7.1. Explanation

ReaLearn strives to support any general-purpose MIDI/OSC controller out there. However, there are some things you should know:

1. Not every controller works out of the box.
 - There are controllers which might need some initial preparation in order to work optimally with ReaLearn.
 - Don't fear this initial effort, it can pay off very quickly. ReaLearn is designed to get the most out of your controller and make it work the way *you* want it, not some company that wants to impose a certain type of workflow on you.
 - The versatility of a controller is a weakness but at the same time a strength. Taking some time to get to know and prepare your controller can make you end up with a tool that is much better suited for what you are trying to do than some bling-bling off-the-shelf solution.
2. Some controllers don't work perfectly, especially when it comes to the *feedback* direction.
 - Among those controllers that support MIDI feedback, not all of them handle the feedback messages flawlessly.
 - Depending on the nature of the particular problem, it might be possible to fix it in future ReaLearn versions. Therefore, if you encounter a problem in this area, feel free to [raise an issue](#).
3. Some controllers might have unique features that you can only use if you bring a bit of MIDI know-how and are ready to use advanced ReaLearn features.
 - Example: A controller might offer a way to change the appearance of an LED ring, but only via system-exclusive MIDI messages.
 - First, have a look if there's a controller preset already. Maybe it supports those advanced features already.
 - If not, ReaLearn offers the following features for such scenarios:
 - [Mapping lifecycle actions](#) (e.g. for sending MIDI sys-ex data on mapping activation)
 - [Raw MIDI source](#) (for sending MIDI sys-ex data in response to target value changes)
 - [MIDI script source](#) (same but for more complex scenarios)

- [MIDI: Send message](#) target (for sending MIDI sys-ex data triggered by a source)

So even ReaLearn is made for any controller, it's still useful to have a list of specific controllers and how they work in combination with ReaLearn. This list is available [here](#)

7.2. General tips regarding controller setup and usage

The following basic setup hints are usually valid, no matter the specific controller:

- Put your controller's buttons into momentary mode, *not* toggle mode.
- If you are in the lucky situation of owning a controller with endless rotary encoders, by all means, configure them to transmit relative values, not absolute ones!
 - Otherwise, you can't take advantage of ReaLearn's advanced features for sources emitting relative values, such as the "Step size" or "Speed" setting.
 - Also, preventing parameter jumps can never be as effective in absolute mode as in relative mode.
- If there are issues, consult the [Troubleshooting](#) section in the first part of this guide.

Consider the following general usage hints:

- If the device supports visual feedback and different LED colors, the LED color often depends on the target value and can be manually adjusted using "Source Min/Max" in the "Glue" section of the mapping.

Make sure to watch out for dedicated controller presets on the Helgoboss ReaPack repository and [list of unofficial controller presets](#)!

Important

Using an existing preset might save you a lot of mapping work (and possibly also layout work, if you want to use the projection feature).

7.3. List of tested controllers

The list of tested controllers is now available as [separate document](#).

8. Available presets

The lists of currently available controller and main presets are available here:

- [Controller presets](#)
- [Main presets](#)

9. Reference

So far we've covered the basics. Now let's look into everything in detail.

9.1. Main panel

Header panel

The header panel provides the following user interface elements, no matter if the *main* or *controller* compartment is shown:

Input

By default, ReaLearn captures MIDI events from *<FX input>*, which consists of all MIDI messages that flow into this ReaLearn VSTi FX instance (= track MIDI path). Alternatively, ReaLearn can capture events from a MIDI device directly, from an OSC device or from your computer keyboard. Be aware that MIDI will only work if *Enable input from this device* is checked for the selected MIDI input device in REAPER's MIDI preferences.

Output

Here you can choose if and where ReaLearn should send MIDI/OSC feedback. By default it's set to *<None>* for no feedback. If you want to enable feedback, pick a MIDI or OSC output device here. Keep in mind that for MIDI, *Enable output to this device* must be checked in REAPER's MIDI preferences. As an alternative, you can send feedback to *<FX output>*, which makes feedback MIDI events stream down to the next FX in the chain or to the track's hardware MIDI output.

Please note that sending MIDI feedback to the FX output has some drawbacks. First, it doesn't participate in ReaLearn's multi-instance feedback orchestration. That means you might experience LEDs/faders misbehaving when using multiple instances. Second, it doesn't work if ReaLearn FX is suspended, e.g. in the following cases:

Warning

- ReaLearn FX is disabled.
- Project is paused and ReaLearn track is not armed.
- ReaLearn FX is on input FX chain and track is not armed.

Menu

This opens ReaLearn's main menu. It's also accessible via right-click on Windows and Linux and control-click on macOS. It provides the following entries.

Copy listed mappings

Copies all mappings that are visible in the current mapping list to the clipboard (respecting group, search field and filters). You can insert them by opening the context menu in the row panel.

Paste mappings (replace all in group)

Replaces all mappings in the current group with the mappings in the clipboard.

Auto-name listed mappings

Clears the names of all listed mappings so ReaLearn's dynamic auto-naming mechanism can kick in.

Name listed mappings after source

Sets the names of each listed mapping to the first line of its source label.

Make sources of all main mappings virtual

Attempts to make the sources in the main compartment virtual by matching them with the sources in the controller compartment. This is useful if you already learned a bunch of sources in the main compartment, just to find out later that you would like to have used a controller preset.

Make targets of listed mappings sticky

Changes the targets of all currently listed mappings to use "sticky" object selectors by attempting to resolve the objects from non-sticky selectors. We call object selectors *sticky* if they refer to a very particular object (e.g. a track).

- Sticky selectors: <Master>, <This>, Particular
- Non-sticky selectors: <Dynamic>, <Focused>, <Selected>, <Instance>, <All selected>`, At position, Named, All named

Move listed mappings to group

Lets you move all currently listed mappings to the specified group. Perfect in combination with the textual search!

Advanced

Provides access to expert features.

- **Copy listed mappings as Lua:** Like *Copy listed mappings* but generates Lua code instead.
- **Copy listed mappings as Lua (include default values):** Generates Lua code that contains even those properties that correspond to ReaLearn's defaults.
- **Paste from Lua (replace all in group):** Like *Paste mappings (replace all in group)* but treats the clipboard content as Lua code.

- **Dry-run Lua script from clipboard:** Executes the Lua code in the clipboard and opens the returned data structure in a text editor.

The way Lua import works in ReaLearn is:

1. ReaLearn executes the Lua script (from clipboard).
2. ReaLearn attempts to interpret the return value as ReaLearn API object.
3. ReaLearn loads the API object

Note

If step 1 fails, ReaLearn displays an error messages that hopefully contains a line number.

If step 2 fails, ReaLearn shows a validation error message.

The command *Dry-run Lua script from clipboard* enables you to just execute step 1 and see the "expanded" result. This can help to make sense of a possible validation error message in step 2.

- **Freeze clip matrix:** Don't use this, this feature is not ready yet!

Options

- **Auto-correct settings:** By default, whenever you change something in ReaLearn, it tries to figure out if your combination of settings makes sense. If not, it makes an adjustment. This auto-correction is usually helpful. If for some reason you want to disable auto-correction, this is your checkbox.
- **Send feedback only if track armed:** If input is set to *<FX input>*, ReaLearn by default only sends feedback if the track is armed (unarming will naturally disable control, so disabling feedback is just consequent). However, if input is set to a MIDI or OSC device, *auto-correct settings* will take care of unchecking this option in order to allow feedback even when unarmed (same reasoning). You can override this behavior with this checkbox. At the moment, it can only be unchecked if ReaLearn is on the normal FX chain. If it's on the input FX chain, unarming naturally disables feedback because REAPER generally excludes input FX from audio/MIDI processing while a track is unarmed (**this is subject to change in future!**).
- **Reset feedback when releasing source:** When using ReaLearn the normal way, it's usually desired that feedback is reset when the corresponding sources are not in use anymore (e.g. lights are switch off, displays are cleared, motor faders are pulled down). You can prevent this ReaLearn instance from doing this by disabling this option. This can be useful e.g. when using REAPER/ReaLearn to control a hardware device (essentially using ReaLearn the other way around, "controlling from target to source").
- **Make instance superior:** If ticked, this instance is allowed to suspend other instances which share the same input and/or output device (hardware devices only, not FX input or output!). With this you can easily let your controller control e.g. the currently focused FX but fall back to your usual controls when it's closed. It's intended to be used primarily on instances that use [Auto-load](#).

Since ReaLearn 2.14.0, falling back to normal mappings when the FX loses focus in auto-load mode

Tip became much easier! One instance is enough. Your normal mappings will be memorized and reloaded once the FX loses focus. See [Auto-load](#).

- By default, ReaLearn instances are not superior, just normal. This is often okay because ReaLearn instances are friendly fellows and like sharing controllers with each other.
- For example, if 2 instances use the same input or output device and they use different control elements, they can peacefully coexist. And even if they share a control element for the *control direction*, they are still fine with it. The same control element will control 2 mappings, why not!
- Things start to get hairy as soon as 2 instances want to send *feedback* to the same control elements at the same time. You should avoid this. You should not even do this within one ReaLearn instance. This can't work.
- Sometimes you want one instance to suspend/cover/cancel/mute another one! You can do this by making this instance *superior*. Then, whenever this instance has at least one active mapping, all non-

superior instances with the same control and/or feedback device will be disabled for control and/or feedback.

- You can have multiple superior instances. Make sure they get along with each other :)
- **Use instance-wide FX-to-preset links only:** By default, instance-specific links are applied *in addition* to the global links and take precedence over the global ones. This checkbox makes sure that only instance-specific links are used.
- **Stay active when project in background:** Determines if and under which conditions this ReaLearn instance should stay active when the containing project tab is not the active one. Applies to in-project ReaLearn instances only, not to monitoring FX instances!
 - **Never:** Will only be active when its project tab is active.
 - **Only if background project is running:** Follows REAPER's project tab settings ("Run background projects" and "Run stopped background projects").
 - **Always (more or less):** Attempts to stay active no matter what. Please note that this is technically not always possible when using *<FX input>* or *<FX output>* when the background project is not running.

Server

ReaLearn features a built-in server which allows the [Companion app](#) (and in future also the Playtime app) to connect to ReaLearn. The server runs globally, not per instance!

- **Enable and start!:** This starts the server and makes sure it will automatically be started next time you use ReaLearn.
- **Disable and stop!:** This stops the server and makes sure it will not be started next time you use ReaLearn.
- **Add firewall rule:** Attempts to add a firewall rule for making the server accessible from other devices or displays instructions how to do it.

Open preset folder

Opens the ReaLearn preset folder in a file manager.

Reload all presets from disk

If you made direct changes to preset files or have downloaded presets via ReaPack, you should press this to reflect these changes in the preset lists of all open ReaLearn instances (reloads all preset files).

Note This **will not** yet apply an adjusted preset, it will just reload the list. If you want to apply a preset that has been changed on disk, you need to select it in the preset dropdown once again!

Open Pot Browser

This will open Pot Browser, a modern preset browser. It's recommended to use this from a ReaLearn instance on the monitoring FX chain, that way you have the browser accessible from any project.

Tip Add a toolbar button which triggers the REAPER action "ReaLearn: Open first Pot Browser" to get quick and convenient access to the browser.

Remarks:

- Pot Browser is in an experimental stage, it doesn't save any of your settings!
- Each ReaLearn instance has its own so-called *Pot Unit*. Each Pot Unit has its own filter and preset state. When you open the Pot Browser from an instance, it connects to the Pot Unit of that instance.
- ReaLearn's "Pot" targets such as [Pot: Browse presets](#) can be used to control the Pot Unit from any controller.

OSC devices

Allows one to display and modify the list of (globally) configured OSC devices.

- **<New>:** Opens a window for adding a new OSC devices.
 - **Name:** A descriptive name of the device, e.g. "TouchOSC on my Android phone".
 - **Local port:** Required for control. The UDP port on which ReaLearn should listen for OSC control messages.
 - **Important:** This port must be reserved exclusively for ReaLearn. If you already use this port in another application (e.g. in REAPER's own OSC control surface) it won't work and ReaLearn will bless you with an "unable to connect" message in the "Input" dropdown.
 - **Device host:** Required for feedback only. It's the IP address of the OSC device to which ReaLearn should send feedback messages. This address is usually displayed on your OSC device (e.g. as "Local IP address"). When targeting an OSC software that runs on the same computer as REAPER and ReaLearn, enter the special IP address 127.0.0.1 ("localhost").

When you configure your OSC device, you must provide a *host* as well. There you should enter the IP address of the computer which runs REAPER and ReaLearn.

Tip

You can easily find it by pressing the [Projection](#) button in ReaLearn and scrolling down a bit. It's the value next to **Host** and should start with 192.168..

- **Device port:** Required for feedback only. The UDP port on which the OSC device listens for OSC feedback messages.
- All OSC device configurations will be saved in the REAPER resource directory (REAPER → Actions → Show action list... → Show REAPER resource path in explorer/finder) in the JSON file `Helgoboss/ReaLearn/osc.json`.

• **Some device**

- **Edit:** Lets you edit an existing device (see *<New>*).
- **Remove:** Removes the device. This is a global action. As a consequence, all existing ReaLearn instances which use this device will point to a device that doesn't exist anymore.
- **Enabled for control:** If you disable this, ReaLearn will stop listening to this device. This can save resources, so you should do this with each device that is not in use (as an alternative for removing it forever).
- **Enabled for feedback:** If you disable this, ReaLearn won't connect to this device.
- **Can deal with bundles:** By default, ReaLearn aggregates multiple OSC messages into so-called OSC bundles. Some devices (e.g. from Behringer) can't deal with OSC bundles. Untick the checkbox in this case and ReaLearn will send single OSC messages.

Compartment parameters

This shows all parameters of the current compartment (you know, the ones that can be used for conditional activation and *<Dynamic>* selector expressions) and makes it possible to customize them. This is practical because it's completely up to you how to put these parameters to use.

Perfect for preset authors: Parameter settings are saved together with the compartment preset. Parameter values will be reset whenever you load a preset (just the ones in that compartment).

- **Param x Name:** Changes the name of this parameter.
- **Value count:** By default, ReaLearn parameter values are continuous in nature: They are arbitrary decimal numbers between 0.0 and 1.0. Although that's very flexible, it's often easier to work with a discrete value range. Entering a value count turns the parameter into a discrete parameter with the given number of

integer values. For example, a value count of 10 means that the parameter can represent exactly 10 values (0 to 9).

Choose the value count wisely and think twice before changing it to a different value at a later point in time!

Caution Reason: You probably want to refer to values of this parameter in certain parts of ReaLearn, e.g. in [Target Min/Max](#). If you do that and later change the value count, these value references will not be valid anymore. They will point to other integers than you intended to. So if you are not sure, better pick a large value count and stick to it!

Logging

- **Log debug info (now):** Logs some information about ReaLearn's internal state. Can be interesting for investigating bugs or understanding how this plug-in works.
- **Log real control messages:** When enabled, all incoming MIDI messages, OSC messages or key pressed will be logged to the console. Each log entry contains the following information:
 - Timestamp in seconds
 - ReaLearn instance ID (a randomly assigned ID that uniquely identifies a particular instance, will change after restart)
 - Message purpose
 - **Real control:** A message used for controlling targets.
 - **Real learn:** A message used for learning a source.
 - Actual message (MIDI messages will be shown as hexadecimal byte sequence, short MIDI messages also as decimal byte sequence and decoded)
 - Match result
 - **unmatched:** The message didn't match any mappings.
 - **matched:** The message matched at least one of the mappings.
 - **consumed:** Only for short MIDI messages. This short message is part of a (N)RPN or 14-bit CC message and there's at least one active mapping that has a (N)RPN or 14-bit CC source. That means it will not be processed. The complete (N)RPN or 14-bit CC message will be.
- **Log virtual control messages:** When enabled, all triggered virtual control elements and their values will be logged (see [Controller compartment](#)).
- **Log target control:** When enabled, all target invocations (parameter changes etc.) will be logged.
- **Log virtual feedback messages:** When enabled, all feedback events to virtual control elements will be logged (see [Controller compartment](#)).
- **Log real feedback messages:** When enabled, all outgoing MIDI or OSC messages will be logged to the console. The log entries look similar to the ones described above, with the following notable differences.
 - Message purpose
 - **Feedback output:** A message sent to your controller as response to target value changes.
 - **Lifecycle output:** A message sent to your controller as response to mapping activation/deactivation (see [Mapping lifecycle actions](#)).
 - **Target output:** A message sent because of either the [MIDI: Send message](#) or [OSC: Send message](#) target.

Usually ReaLearn sends feedback whenever something changed to keep the LEDs or motorized faders of your controller in sync with REAPER at all times. There might be situations where it doesn't work though. In this case you can send feedback manually using this button.

Export to clipboard

Pressing the export button allows you to copy ReaLearn's settings to the clipboard so you can import them in another instance or edit them in a text editor.

- **Export session as JSON:** Copies a *complete* dump of ReaLearn's current settings (including all mappings, even controller mappings) to the clipboard. The dump's data format is [JSON](#), a wide-spread data exchange format. It's a text format, so if you are familiar with the search & replace feature of your favorite text editor, this is one way to do batch editing. However, recent versions of ReaLearn provide a much better way of doing that: *ReaLearn Script*. Read about the other export options for learning more about it.

You can also use the export for some very basic A/B testing:

Tip

1. Choose *Export session as JSON*
2. Change some settings and test them
3. Restore the old settings by pressing *Import from clipboard*.

For the programmers and script junkies out there: It's perfectly possible to program ReaLearn from outside by passing it a snippet of JSON via [TrackFX_SetNamedConfigParm\(.\)](#). Parameter name is `set-state`. This mechanism is implemented on ReaLearn side using [REAPER's named parameter mechanism](#) (search for `named_parameter_name`).

Example that assumes that the first FX of the first track is a ReaLearn instance:

Tip

```
local track = reaper.GetTrack(0, 0)
local state = [[
{
  "controlDeviceId": "62",
  "feedbackDeviceId": "fx-output",
  "mappings": [
    {
      "name": "1",
      "source": {
        "type": 1,
        "channel": 0,
        "number": 64
      },
      "mode": {},
      "target": {
        "type": 2
      }
    }
  ]
}
]]
reaper.TrackFX_SetNamedConfigParm(track, 0, "set-state", state)
```

- **Export main/controller compartment as JSON:** Copies a dump of the currently visible compartment to the clipboard. It contains about the same data that a compartment preset would contain.
- **Export main/controller compartment as Lua:** Copies a dump of the currently visible compartment to the clipboard as Lua code (ReaLearn Script). This form of Lua export skips properties that correspond to ReaLearn's default values, resulting in a minimal result. Perfect for pasting in a forum or programming ReaLearn with focus on only those properties that matter to you.

- **Export main/controller compartment as Lua (include default values):** This Lua export includes even those properties that correspond to ReaLearn's default values, resulting in more text. This gives you the perfect starting point if you want to extensively modify the current compartment (using the Lua programming language) or build a compartment from scratch, using even properties that you haven't touched yet in the user interface!

Import from clipboard

Pressing the import button does the opposite: It restores whatever ReaLearn dump is currently in the clipboard.

"?" (Help)

Provides helpful links to the user guide and other stuff.

Projection

This is a quite unique feature that allows you to project a schematic representation of your currently active controller to a mobile device (e.g. a tablet computer). You can put this device close to your controller in order to see immediately which control element is mapped to which parameter. This is an attempt to solve an inherent problem with generic controllers: That it's easy to forget which control element is mapped to which target parameter. If you want to use this feature, just click this button and you will see detailed instructions on how to set this up. In order to use this feature, you need the *ReaLearn Companion* app, which has a [dedicated section](#) in this user guide.

Let through

ReaLearn by default "eats" incoming MIDI events for which there's at least one active mapping with that source. In other words, it doesn't forward MIDI events which are used to control a target parameter. However, unmatched MIDI events are forwarded!

You can change this using these checkboxes. E.g. you can tick **Matched events** if you want to forward even matched events. The exact behavior differs depending on what you choose as [Input](#):

- If input is set to <FX input>
 - MIDI events arrive from ReaLearn's FX input. If they get forwarded, they get forwarded to the FX output, usually to the plug-in which is located right below ReaLearn FX. The default setting often makes much sense here, especially if you put ReaLearn right above another instrument plug-in.
- If input is set to a MIDI hardware device
 - MIDI events arrive directly from the MIDI hardware device. If they get forwarded, they get forwarded to REAPER's tracks as they would usually do without ReaLearn. If they don't get forwarded, it means they get filtered and will never make it to the tracks. ReaLearn completely eats them, globally! That means, ReaLearn can act as global MIDI filter.
 - Please note, with input set to a real MIDI device, MIDI events coming from *FX input* are *always* forwarded to the FX output.
 - Also, MIDI events captured from a real MIDI device input are **never** forwarded to ReaLearn's FX output.

Tip This global MIDI filter feature is only available in REAPER v6.36+.

- If input is set to a OSC device
 - You won't see the checkboxes because they don't make sense for OSC.
- If input is set to computer keyboard
 - You can control whether key presses are forwarded to REAPER or not.
 - For example, unticking both checkboxes makes sure that only keyboard hotkeys defined in ReaLearn have an effect. This can be interesting for live scenarios in which you temporarily want to lower the risk of pressing the wrong key and messing up the performance. Just unlock the keys you absolutely need.

Show

This lets you choose which mapping compartment is displayed. A compartment is basically a list of mappings that can be saved as independent preset. Initially, ReaLearn shows the so-called "Main compartment" which contains the so-called "Main mappings" - the bread and butter of ReaLearn. However, there's another interesting compartment, the "Controller compartment". In a nutshell, this compartment lets you define which hardware controllers you have at your disposal and which control elements they have. Learn more about that feature in section [Controller compartment](#).

Controller preset / Main preset

This is the list of available presets for that compartment. By default, it's set to "<None>", which means that no particular preset is active. If you select a preset in this list, its corresponding mappings will be loaded and immediately get active. In the *controller* compartment, this list will essentially represent the list of available hardware controller presets. A few are shipped with ReaLearn itself (separately downloadable via ReaPack) but you can also define your own ones and add them to this list!

- **Save:** If you made changes to a preset, you can save them by pressing this button. This works for built-in presets as well but I would strongly recommend against changing them directly. Better use *Save as...* and choose a custom name.

Save as...

This allows you to save all currently visible mappings as a new preset. Please choose a descriptive name.

- Saving your mappings as a preset is optional. All controller mappings are saved together with your current ReaLearn instance anyway, no worries. But as soon as you want to reuse these mappings in other ReaLearn instances, it makes of course sense to save them as a preset!
- All of your presets end up in the REAPER resource directory (REAPER → Actions → Show action list... → Show REAPER resource path in explorer/finder) at `Data/helgoboss/realearn/presets`. They are JSON files and very similar to what you get when you press *Export to clipboard*.
- Those files are usually in the root of that `presets` directory but can also reside in sub directories (one level only, sub/sub directories are not supported). Please note that the sub directory name becomes a part of the preset ID, so better don't move existing presets around if you want preset references of existing ReaLearn instances to stay intact.
- JSON files can also contain custom data sections. For example, the ReaLearn Companion app adds a custom data section to controller presets in order to memorize the positions and shapes of all control elements.
- When pressing this button, ReaLearn might detect that your current mappings are referring to specific tracks and FX instances *within the current project*. This would somehow defeat the purpose of presets because what good are presets that are usable only within one project? That's why ReaLearn also offers you to automatically convert such mappings to project-independent mappings by applying the following transformations:
 - FX targets are changed to refer to *current instance FX** instead of a particular one. Their track is set to **<This>** because it doesn't matter anyway.
 - Track targets are changed to refer to a track via its position instead of its ID.
- If this is not what you want, you can choose to say no and make modifications yourself.

Delete

This permanently deletes the currently chosen preset. You can also delete built-in presets. However, if you use ReaPack for installation, it should restore them on next sync.

Notes

Allows you to save custom notes/comments for the current compartment. These notes are also included in compartment presets.

Add one

Adds a new mapping at the end of the current mapping list.

Learn many

Allows you to add and learn many new mappings in a convenient batch mode. Click this button and follow the on-screen instructions. Click *Stop* when you are finished with your bulk learning strike.

Search

Enter some text here in order to display just mappings whose name matches the text. The search expression also supports wildcards **and ? for doing blurry searches.** `*` stands for zero or more arbitrary characters and `?` stands for one arbitrary character.

Filter source

If you work with many mappings and you have problems memorizing them, you will love this feature. When you press this button, ReaLearn will start listening to incoming MIDI/OSC events and temporarily disable all target control. You can play around freely on your controller without having to worry about messing up target parameters. Whenever ReaLearn detects a valid source, it will filter the mapping list by showing only mappings which have that source. This is a great way to find out what a specific knob/fader/button etc. is mapped to. Please note that the list can end up empty (if no mapping has that source). As soon as you press *Stop*, the current filter setting will get locked. This in turn is useful for temporarily focusing on mappings with a particular source. When you are done and you want to see all mappings again, press the **X** button to the right. *Tip:* Before you freak out thinking that ReaLearn doesn't work anymore because it won't let you control targets, have a quick look at this button. ReaLearn might still be in "filter source" mode. Then just calm down and press *Stop*. It's easy to forget.

Filter target

If you want to find out what mappings exist for a particular target, press this button and touch something in REAPER. As soon as you have touched a valid target, the list will show all mappings with that target. Unlike *Filter source*, ReaLearn will automatically stop learning as soon as a target was touched. Press the **X** button to remove the filter and show all mappings again.

Bottom panel

At the bottom you can see:

- The current scroll position.
- The session ID of this this ReaLearn instance.
- Tags assigned to this ReaLearn instance.
- Information about the current instance track and instance FX.
- Information whether control and/or feedback is currently inactive instance-wide.
- Information about what version of ReaLearn you have.

Instance data...

Press this button to change various key-value data of this ReaLearn instance as a whole.

- **Session ID...:** This lets you customize the ID used to address this particular ReaLearn instance when using the [Projection](#) feature.
 - By default, the session ID is a random cryptic string which ensures that every instance is uniquely addressable. The result is that scanning the QR code of this ReaLearn instance will let your mobile device connect for sure with this unique instance, not with another one - remember, you can use many instances of ReaLearn in parallel. This is usually what you want.
 - But a side effect is that with every new ReaLearn instance that you create, you first have to point your mobile device to it in order to see its [Projection](#) (by scanning the QR code). Let's assume you have in many of your projects exactly one ReaLearn instance that lets your favorite MIDI controller control track volumes. By customizing the session ID, you basically can tell your mobile device that it should always show the [Projection](#) of this very ReaLearn instance - no matter in which REAPER project you are and even if they control the volumes of totally different tracks.
 - You can achieve this by setting the session ID of each volume-controlling ReaLearn instance to exactly the same value, in each project. Ideally it's a descriptive name without spaces, such as "track-volumes". You have to do the pairing only once et voilà, you have a dedicated device for monitoring your volume control ReaLearn instances in each project.

Caution Make sure to not have more than one ReaLearn instance with the same session ID active at the same time because then it's not clear to which your mobile device will connect!

- **At the moment, the session ID is part of the ReaLearn preset!** That means, opening a preset, copying/cutting a ReaLearn FX, importing from clipboard - all of that will overwrite the session ID. This might change in future in favor of a more nuanced approach!
- **Tags:** Lets you assign tags to this instance (a comma-separated list). They are important if you want to dynamically enable or disable instances using the [ReaLearn: Enable/disable instances](#) target.

Instance track

The second line of the bottom panel shows the current track chosen as **Instance track** for this instance of ReaLearn. This can be something like "Track 3" or "The currently selected track". Mappings in this ReaLearn instance can refer to this track by choosing the track selector [Selector "Instance"](#).

The instance track can be changed via target [Track](#).

Instance FX

The second line of the bottom panel also shows the current FX chosen as **Instance FX** for this instance of ReaLearn. This can be something like "FX 5 on track 3" or "The currently focused track". Mappings in this ReaLearn instance can refer to this FX by choosing the FX selector [Selector "Instance"](#).

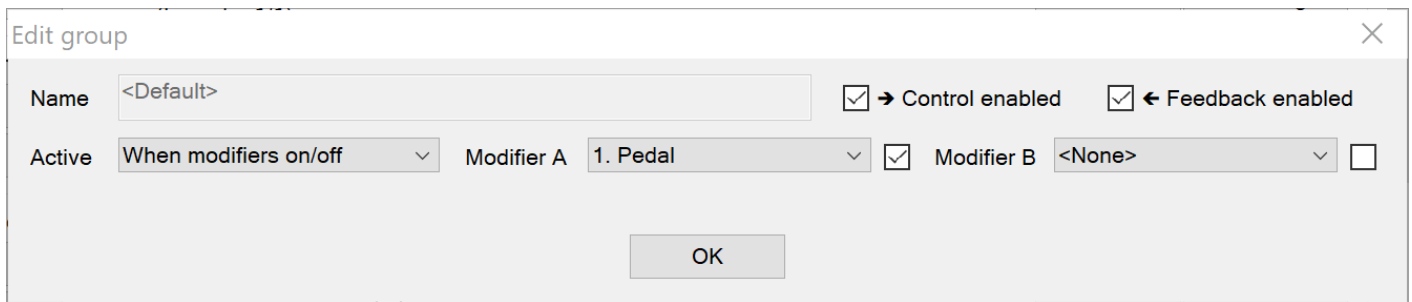
The instance FX can be changed via target [FX](#).

Common compartment settings

The header panel shows the following user interface elements, no matter if you are in the controller or main compartment:

- **Mapping group:** Mapping groups are part of the currently shown compartment and enable you to divide the list of mappings into multiple groups.
 - Groups can be useful ...
 - To apply an activation condition to multiple mappings at once.
 - To enable/disable control/feedback for multiple mappings at once.
 - To keep track of mappings if there are many of them.
 - This dropdown contains the following options:
 - **<All>:** Displays all mappings in the compartment, no matter to which group they belong. In this view, you will see the name of the group on the right side of a mapping row.
 - **<Default>:** Displays mappings that belong to the *default* group. This is where mappings end up if you don't care about grouping. This is a special group that can't be removed.
 - **Custom group:** Displays all mappings in your custom group.
 - You can move existing mappings between groups by opening the context menu (accessible via right-click on Windows and Linux, control-click on macOS) of the corresponding mapping row and choosing "Move to group".
 - Groups are saved as part of the project, VST plug-in preset and compartment preset.
- **Add:** Allows you to add a group and give it a specific name.
- **Remove:** Removes the currently displayed group. It will ask you if you want to remove all the mappings in that group as well. Alternatively they will automatically be moved to the default group.
- **Edit:** Opens the group panel. This allows you to change the group name and change things that affect all mappings in this groups: Assigning tags, enabling/disabling control and/or feedback, setting an activation condition. The activation condition that you provide here is combined with the one that you provide in the mapping. Only if both, the group activation conditions and the mapping activation condition are satisfied,

the corresponding mapping will be active. Read more about [conditional activation](#) below in the section about the [Mapping panel](#).



Since ReaLearn 2.10.0, mappings are processed from top to bottom, exactly in the order in which they are defined within the corresponding compartment. This matters if you want to map multiple targets to one button and the order of execution matters.

Important: There's an exception. ReaLearn's processing of its own VST parameters is always deferred.

- That means changing a ReaLearn parameter in one mapping and relying on it in the next one (in terms of conditional activation or in a `<Dynamic>` expression), will not work!
- You can work around that by delaying execution of the next mapping via [fire mode](#) but that's a dirty hack. ReaLearn's parameters are not supposed to be used that way!
- Imagine a railway: ReaLearn's targets can be considered as trains. Triggering a target means moving the train forward. ReaLearn's parameters can be considered as railway switches. Changing a parameter means setting a course. The course needs to be set in advance, at least one step before! Not at the same time as moving the train over the switch.

Controller compartment

By default, ReaLearn shows the list of main mappings. If you switch to the *controller* compartment, you will see the list of controller mappings instead. Each controller mapping represents a control element on your hardware controller, e.g. a button or fader. This view lets you describe your controller by - well - by adding mappings. Almost everything in ReaLearn is a mapping :)

Defining your own controllers can have a bunch of benefits:

- You can use the awesome [Projection](#) feature to project your controller mapping to a smartphone or tablet ([watch video](#)).
- You can use controller presets made by other users and thereby save precious setup time. Or you can contribute them yourself!
- You can make your main mappings independent of the actual controller that you use. This is done using so-called *virtual* sources and targets.
- It allows you to give your knobs, buttons etc. descriptive and friendly names instead of just e.g. "CC 15".
- You don't need to learn your control elements again and again. Although the process of learning an element is easy in ReaLearn, it can take some time in case the source character is not guessed correctly (e.g. absolute range element or relative encoder). Just do it once and be done with it!

If you want to make ReaLearn "learn" about your nice controller device, all you need to do is to create a suitable controller mapping for each of its control elements.

Let's first look at the "slow" way to do this - adding and editing each controller mapping one by one:

1. Press the "Add one" button.
2. Learn the source by pressing the "Learn source" button and touching the control element.
3. Press the "Edit" button.
4. Enter a descriptive name for the control element.
 - *Hint:* This name will appear in many places so you want it to be short, clear and unique!

5. Assign a unique virtual target.

- At this point we don't want to assign a *concrete* target yet. The point of controller presets is to make them as reusable as possible, that's why we choose a so-called *virtual* target.
- In the *Category* dropdown, choose *Virtual*.
- As *Type*, choose *Button* if your control element is a sort of button (something which you can press) and *Multi* in all other cases.
- Use for each control element a unique combination of *Type* and *ID*, starting with number 1 and counting.
 - Example: It's okay and desired to have one control element mapped to "Multi 1" and one to "Button 1".
- Just imagine the "8 generic knobs + 8 generic buttons" layout which is typical for lots of popular controllers. You can easily model that by assigning 8 multis and 8 buttons.
- Maybe you have realized that the *Glue* section is available for controller mappings as well! That opens up all kinds of possibilities. You could for example restrict the target range for a certain control element. Or make an encoder generally slower or faster. Or you could simulate a rotary encoder by making two buttons on your controller act as -/+ buttons emitting relative values. This is possible by mapping them to the same "Multi" in "Incremental button" mode.

Before you go ahead and do that for each control element, you might want to check out what this is good for: Navigate back to the *main* compartment, learn the source of some main mapping and touch the control element that you have just mapped: Take note how ReaLearn will assign a *virtual* source this time, not a MIDI source! It will also display the name of the control element as source label. Now, let's say at some point you swap your controller device with another one that has a similar layout, all you need to do is switch the controller preset and you are golden! You have decoupled your main mappings from the actual controller. Plus, as soon as you have saved your controller mappings as a preset, you can take full advantage of the [Projection](#) feature.

All of this might be a bit of an effort but it's well worth it! Plus, there's a way to do this *a lot* faster by using *batch learning*:

1. Press the "Learn many" button.
2. Choose whether you want to learn all the "Multis" on your controller or all the "Buttons".
3. Simply touch all control elements in the desired order.
 - ReaLearn will take care of automatically incrementing the virtual control element numbers.
4. Press "Stop".
5. Done!
 - At this point it's recommended to recheck the learned mappings.
 - ReaLearn's source character detection for MIDI CCs is naturally just a guess, so it can be wrong. If so, just adjust the character in the corresponding mapping panel.

You can share your preset with other users by sending them to info@helgoboss.org. I will add it to [this list](#).

Main compartment

The header panel for main mappings consists of a few more user interface elements:

Auto-load

If you switch this to *Based on instance FX*, ReaLearn will start to observe the [Instance FX](#) of this ReaLearn instance. By default, the instance FX is set to <Focused>, which means, it will reflect whatever FX is currently focused. Whenever the instance FX changes, it will check if you have linked a compartment preset to it and will automatically load it. Whenever the instance FX switches to an unlinked FX or the FX loses focus, ReaLearn falls back to the mapping list defined before activating auto-load. Of course this makes sense only if you actually have linked some presets. Read on! The header context menu (accessible via right-click on Windows and Linux, control-click on macOS) for the main compartment contains the missing piece of the puzzle:

Global FX-to-preset links

Manage a global list of links from plug-ins to ReaLearn main compartment presets.

- **Add link from last focused FX to preset:** This lets you link whatever FX window was focused before focusing ReaLearn, to an arbitrary main compartment preset. Needless to say, this only works if an FX has been focused before.
 - All links will be saved *globally*, not just within this project!
 - Location: REAPER resource directory (REAPER → Actions → Show action list... → Show REAPER resource path in explorer/finder) at Data/helgoboss/realearn/auto-load-configs/fx.json.
- **Arbitrary FX ID:** If you have added a link already, you will see them here in a list. What you see, is the so-called *FX ID*, which by default simply corresponds to the plug-in's original name (e.g. VSTi: ReaSynth (Cockos)).
 - **<Edit FX ID...>:** With this, you can edit the FX ID manually.
 - **All fields:** All the fields below support wildcards. E.g. instead on relying on the original plug-in name you could match plug-ins with similar file names (e.g. VST2 and VST3 at once): You can use * for matching zero or arbitrary many characters and ? for matching exactly one arbitrary character. E.g. Pianoteq 7 STAGE.* would match both Pianoteq 7 STAGE.dll (VST2) and Pianoteq 7 STAGE.vst3 (VST3).
 - **FX name:** Allows you to adjust the (original) plug-in name that triggers the preset change.
 - **FX file name:** Allows you to adjust the plug-in file name that triggers the preset change.
 - **FX preset name:** Maybe the FX name or file name is not enough for you to decide which preset you want to load. Good news: You can add a preset name as additional criteria! E.g. if you have use a sampler, you can load different ReaLearn presets depending on which sample library is loaded into your sampler. Just add two links with the same FX file name (e.g. Kontakt 5.dll) but different preset names. You can also use wildcards here!
 - **<Remove link>:** (Globally) this FX-to-preset link.
 - **Arbitrary main preset:** The checkbox tells you to which main preset the FX ID is linked. You can change the linked preset by clicking another one.

Instance-wide FX-to-preset links

This is like [Global FX-to-preset links](#) but saves the links as part of this ReaLearn instance. This is useful if you have 2 controllers (= and therefore 2 ReaLearn instances) and want them to auto-load different presets although the instance FX points to the same plug-in.

Mapping row

The mapping, source and target labels of a mapping row should be greyed out whenever the mapping is *off*. A mapping is considered as *on* only if the following is true:

1. The mapping is complete, that is, both source and target are completely specified.
2. The mapping is enabled as a whole.
3. The mapping has control and/or feedback enabled.
4. The mapping is active (see *conditional activation*).

If a mapping is *off*, it doesn't have any effect.

- ✓: This checkbox at the top left of the mapping row enables or disables the mapping as a whole.
- ●: This indicator at the very left of the mapping row lights on incoming control messages whenever they match the mapping source. Attention: This doesn't necessarily mean that the message will reach the target (although it often does). There are certain settings in the [Glue](#) section which allow you to filter messages even they matched the source (e.g. the *Source Min/Max*).

- **Up / Down:** Use these buttons to move this mapping up or down the list.
- **→ / ←:** Use these checkboxes to enable/disable control and/or feedback for this mapping. Disabling both has the same effect as disabling the mapping as a whole.
- **Edit:** Opens the mapping panel for this mapping.
- **Duplicate:** Creates a new mapping just like this one right below.
- **Remove:** Removes this mapping from the list.
- **Learn source:** Starts or stops learning the source of this mapping.
- **Learn target:** Starts or stops learning the target of this mapping.
 - *Tip:* Learning a target that is currently being automated is not possible at the moment because ReaLearn can't know if the value change notification is coming from the automation or your touch interaction.

Each mapping row provides a context menu (accessible via right-click on Windows and Linux, control-click on macOS), which allows you access to the following functionality:

- **Copy:** Copies this mapping to the clipboard.
- **Paste (replace):** Replaces this mapping with the mapping in the clipboard. If the clipboard contains just a part of a mapping (source, mode or target), then just this part gets replaced.
- **Paste (insert below):** Creates a new mapping that's like the mapping in the clipboard and places it below this mapping.
- **Copy part:** Copies just a part of the mapping (activation condition, source, mode or target).
- **Move to group:** Lets you move this mapping to another mapping group.
- **Advanced:** Provides access to expert features.
 - **Copy as Lua:** Copies this mapping as Lua code. This is an indispensable tool if you want to build your mappings in Lua because it gives you a readily executable code snippet that you can adjust as desired.
 - **Copy as Lua (include default values):** Includes even default values.
 - **Paste from Lua (replaces):** Like *Paste (replace)* but treats the clipboard content as Lua code.
 - **Paste from Lua (insert below):** Like *Paste (insert below)* but treats the clipboard content as Lua code.
 - **Log debug info (now):** Logs debug information about this particular mapping.

9.2. Mapping panel

At this point it's important to understand some basics about how ReaLearn processes incoming control events. When there's an incoming control event that matches a particular source, one of the first things ReaLearn does is to normalize it to a so-called *control value*.

A control value can be either absolute or relative, depending on the source character:

- **Source emits absolute values (e.g. faders):** The control value will be absolute, which means it's a 64-bit decimal number between 0.0 and 1.0. You can also think of it in terms of percentages: Something between 0% and 100%. 0% means the minimum possible value of the source has been emitted whereas 100% means the maximum.
- **Source emits relative values (e.g. rotary encoders):** The control value will be relative, which means it's a positive or negative integer that reflects the amount of the increment or decrement. E.g. -2 means a decrement of 2.

After having translated the incoming event to a control value, ReaLearn feeds it to the mapping's glue section. The glue section is responsible for transforming control values before they reach the *target*. This transformation

can change the type of the control value, e.g. from relative to absolute - it depends on the mapping's target character. The glue section can even "eat" control values so that they don't arrive at the target at all.

Finally, ReaLearn converts the transformed control value into some target instruction (e.g. "set volume to -6.0 dB") and executes it.

Feedback (from REAPER to controller) works in a similar fashion but is restricted to absolute control values. Even if the source is relative (e.g. an encoder), ReaLearn will always emit absolute feedback, because relative feedback doesn't make sense.

General mapping properties

This section provides the following mapping-related settings and functions:

- **Name:** Here you can enter a descriptive name for the mapping. This is especially useful in combination with the search function if there are many mappings to keep track of. If you clear the name, ReaLearn will name the mapping automatically based on its target.
- **Tags:** Use this to assign arbitrary tags to this mapping (comma-separated). These tags can be used to organize mappings in a way that is much more flexible than groups.
 - Mapping tags are also displayed in mapping rows, including the ones inherited by groups.
 - In the header panel [search field](#), you can search for mappings that have a certain tag by entering the tag name prefixed with the hash character #. For example, you can search for all mappings tagged with the tag `mixing` by entering `#mixing`.
 - Tags are not just something for people that love to keep things tidy! They also get meaning in combination with certain ReaLearn targets such as [ReaLearn: Enable/disable mappings](#).
- **Control enabled / Feedback enabled:** Use these checkboxes to enable/disable control and/or feedback for this mapping.
- **Active:** This dropdown controls so-called conditional activation of mappings. See the [Conditional activation](#) section below.
- **Feedback:**
 - **Normal:** Makes ReaLearn send feedback whenever the target value changes. This is the recommended option in most cases.
 - **Prevent echo feedback:** This option mainly exists for motorized faders that don't like getting feedback while being moved. If checked, ReaLearn won't send feedback if the target value change was caused by incoming source events of this mapping. However, it will still send feedback if the target value change was caused by something else, e.g. a mouse action within REAPER itself.
 - **Send feedback after control:** This checkbox mainly exists for "fixing" controllers which allow their LEDs to be controlled via incoming MIDI/OSC *but at the same time* insist on controlling these LEDs themselves. For example, some Behringer X-Touch Compact buttons exhibit this behavior in MIDI mode. This can lead to wrong LED states which don't reflect the actual state in REAPER. If this option is not selected (the normal case and recommended for most controllers), ReaLearn will send feedback to the controller *only* if the target value has changed. For example, if you use a button to toggle a target value on and off, the target value will change only when pressing the button, not when releasing it. As a consequence, feedback will be sent only when pressing the button, not when releasing it. However, if this option is selected, ReaLearn will send feedback even after releasing the button - although the target value has not been changed by it. Another case where this option comes in handy is if you use a target which doesn't support proper feedback because REAPER doesn't notify ReaLearn about value changes (e.g. "Track FX all enable") and you have "Poll for feedback" disabled. By choosing this option, ReaLearn will send feedback whenever the target value change was caused by ReaLearn itself, which improves the situation at least a bit.
- **Show in projection:** When unticked, this mapping will not show up in the [Projection](#). Useful e.g. for feedback-only mappings or buttons with multiple assignments.
- **Advanced settings:** This button is for experts. There are some advanced mapping-related settings in ReaLearn that are not adjustable via its graphical user interface but only by writing text-based

configuration. Pressing this button should open a small window in which you can write the configuration for this mapping. If the button label ends with a number, that number denotes the number of top-level configuration properties set for that mapping. That way you can immediately see if a mapping has advanced settings or not. You can learn more about the available properties in the section [Advanced settings](#).

- **Open in text editor (Windows and Linux only):** Opens the settings in the system text editor or whatever program is associated with YAML files. It depends on your system setup if this works or not. If it does and if your text editor is good, this can make editing larger YAML snippets more convenient (e.g. by providing syntax highlighting). As soon as you save the file and close the editor, the text will automatically appear in the "Advanced settings" text area.
- **Help:** Will open an online version of the user guide section that describes the available configuration properties.
- **Find in mapping list:** Scrolls the mapping rows panel so that the corresponding mapping row for this mapping gets visible.
- **Beep on success (checkbox on the bottom-left):** Makes the mapping play a sound whenever the target has been invoked successfully. Nice for trigger-like targets such as [ReaLearn: Take mapping snapshot](#) for which there's no other good way to know if it worked.
- **Previous/next buttons:** Allows you to jump to the previous or next mapping. Considers only mappings that are currently visible in the mapping rows panel.
- **Enabled (checkbox on the bottom-right):** Enables or disables the mapping as a whole.

Conditional activation

Conditional activation allows you to dynamically enable or disable this mapping based on the state of ReaLearn's own plug-in parameters and since recently even on the state of arbitrary targets. This is a powerful feature. It is especially practical if your controller has a limited amount of control elements and you want to give control elements several responsibilities. It let's you easily implement use cases such as:

- "This knob should control the track pan, but only when my sustain pedal is pressed, otherwise it should control track volume!" (modifier use cases)
- "I want to have two buttons for switching between different banks where each bank represents a group of mappings." (bank use cases)
- "I want to control the volume of this track only if it's not muted." (target-state based use cases)

Since ReaLearn 2.11.0, [ReaLearn: Enable/disable mappings](#) provides a slightly less powerful but more Tip straightforward way to implement use cases that were before only achievable with parameter-based conditional activation.

There are 6 different activation modes:

- **Always:** Mapping is always active (the default)
- **When modifiers on/off:** Mapping becomes active only if something is pressed / not pressed
- **When bank selected:** Allows you to step through different groups of mappings (sometimes also called "pages")
- **When EEL met** Let an EEL formula decide (total freedom)
- **When expression met:** Let an expression decide (total freedom)
- **When target value met:** Let the current value of the target of another mapping decide

Note

At this occasion, some words about ReaLearn's own freely assignable FX parameters.

ReaLearn itself isn't just able to control parameters of other FX, it also offers FX parameters itself. At the moment it offers 200 FX parameters, 100 for the main compartment and 100 for the controller compartment. You can control them just like parameters in other FX:

- Via automation envelopes,
- via track controls,
- via REAPER's own MIDI/OSC learn
- ... and of course via ReaLearn itself.

Initially, they don't do anything at all. First, you need to give meaning to them by referring to them in activation conditions or `<Dynamic>` selector expressions.

When modifiers on/off

This mode is comparable to modifier keys on a computer keyboard. For example, when you press `Ctrl+V` for pasting text, `Ctrl` is a modifier because it modifies the meaning of the `V` key. When this modifier is "on" (= pressed), it activates the "paste text" and deactivates the "write the letter V" functionality of the `V` key.

In ReaLearn, the modifier is one of the FX parameters. It's considered to be "on" if the parameter has a value greater than 0 and "off" if the value is 0.

You can choose up to 2 modifier parameters, "Modifier A" and "Modifier B". If you select "<None>", the modifier gets disabled (it won't have any effect on activation). The checkbox to the right of the dropdown lets you decide if the modifier must be "on" for the mapping to become active or "off".

Example: The following setting means that this mapping becomes active *only* if both "Parameter 1" and "Parameter 2" are "on".

- **Modifier A:** "Parameter 1"
- **Checkbox A:** Checked
- **Modifier B:** "Parameter 2"
- **Checkbox B:** Checked

Now you just have to map 2 controller buttons to "Parameter 1" and "Parameter 2" via ReaLearn (by creating 2 additional mappings - in the same ReaLearn instance or another one, up to you) et voilà, it works. The beauty of this solution lies in how you can compose different ReaLearn features to obtain exactly the result you want. For example, the *absolute mode* of the mapping that controls the modifier parameter decides if the modifier button is momentary (has to be pressed all the time) or toggled (switches between on and off everytime you press it). You can also be more adventurous and let the modifier on/off state change over time, using REAPER's automation envelopes.

When bank selected

This is the correct activation mode if you want control surface "bank-style" mapping. An in-depth tutorial how to implement this can be found in the [Tutorials](#) section, tutorial number 1.

Tip For this kind of use cases you should consider the new [ReaLearn: Enable/disable mappings](#) target, which is available since ReaLearn 2.11.0 as an alternative. It's slightly less powerful than conditional activation but probably easier to use, partly because you can dictate which mappings should be active "from outside", not from the perspective of the mapping itself.

You can tell ReaLearn to only activate your mapping if a certain parameter has a particular value. The particular value is called "Bank". Why? Let's assume you mapped 2 buttons "Previous" and "Next" to increase/decrease the value of the parameter (by using "Incremental button" mode, you will learn how to do that further below). And you have multiple mappings where each one uses "When bank selected" with the same parameter but a different "Bank". Then the result is that you can press "Previous" and "Next" and it will switch between different mappings within that parameter. If you assign the same "Bank" to multiple mappings, it's like putting those mappings into one group which can be activated/deactivated as a whole.

Switching between different programs via "Previous" and "Next" buttons is just one possibility. Here are some other ones:

- **Browse banks using a rotary encoder:** Just map the rotary encoder to the "Bank" parameter and restrict the target range as desired.
- **Activate each bank with a separate button:** Map each button to the "Bank" parameter (with absolute mode "Normal") and set "Target Min/Max" to a distinct value. E.g. set button 1 min/max both to 0% and

button 2 min/max both to 1%. Then pressing button 1 will activate bank 0 and pressing button 2 will activate bank 1.

In previous versions of ReaLearn you could use other methods to achieve a similar behavior, but it always involved using multiple ReaLearn instances:

- **By enabling/disabling other ReaLearn instances:** You can use one main ReaLearn instance containing a bunch of mappings with [FX: Enable/disable](#) target in order to enable/disable other ReaLearn FX instances. Then each of the other ReaLearn instances acts as one mapping bank/group.
- **By switching between presets of another ReaLearn instance:** You can use one main ReaLearn instance containing a mapping with [FX: Browse presets](#) target in order to browse presets of another ReaLearn FX instance. Then each preset in the other ReaLearn instance acts as one mapping bank/group. However, that method is pretty limited and hard to maintain because presets are something global (not saved together with your REAPER project).

With *Conditional activation* you can do the same (and more) within just one ReaLearn instance.

Tip If you want to adjust the number of banks and improve bank handling in general, set a discrete value count for the corresponding bank parameter (see [Compartment parameters](#)).

When EEL met

This is for experts. It allows you to write a formula in [EEL2](#) language that determines if the mapping becomes active or not, based on potentially all parameter values. This is the most flexible of all parameter-based activation modes. The other modes can be easily simulated. The example modifier condition scenario mentioned above written as formula would be:

```
y = p1 > 0 && p2 > 0
```

y represents the result. If y is greater than zero, the mapping will become active, otherwise it will become inactive. p1 to p100 contain the current parameter values. Each of them has a value between 0.0 (= 0%) and 1.0 (= 100%).

This activation mode accounts for ReaLearn's philosophy to allow for great flexibility instead of just implementing one particular use case. If you feel limited by the other activation modes, just use EEL.

Tip For most activation conditions which need this amount of freedom, the newer activation mode [When expression met](#) is a slightly better choice because it's easier to use and generally performs a bit better.

When expression met

This is very similar to the previous EEL activation mode. But instead of EEL, it lets you use the same expression language as used in [dynamic selectors](#) to express the activation condition.

The equivalent expression to above EEL example is:

```
p[0] > 0 && p[1] > 0
```

When target value met

This is different from all the other activation condition types in that it doesn't look at ReaLearn's internal parameter values. Instead, it looks at the target of another mapping (the so-called "lead mapping") and switches our mapping (the so-called "follow mapping") on or off depending on the target value of the lead mapping.

It works like this:

1. Create the lead mapping and give it a target, e.g. [Track: Select/unselect](#).
 - This lead mapping doesn't need to have a source. It can even be completely disabled!
2. In the **Mapping** dropdown, pick this newly created mapping.
3. In the **Expression** text field to the right, enter `y > 0`.
 - This means you want the follow mapping to be active whenever the target value of the lead mapping is greater than 0.0. Or in other words, when it's "switched on".

You can detect an inactive target by using `y == none`.

Custom parameter names

Because ReaLearn's parameters are freely assignable, they have very generic names by default. However, as soon as you give them meaning by using them in a specific way, it can be helpful to give them a meaningful name. You can do that:

1. Switch to the compartment whose parameter names you want to change.
2. Open the header panel context menu (accessible via right-click on Windows and Linux, control-click on macOS) and open the *Compartment parameters* submenu.
3. Here you will find each of the 100 compartment parameters with their current names. Simply click the name to change it.

Parameter names are not global, they are always saved together with the REAPER project / FX preset / track template etc. They will also be saved/restored as part of the compartment preset.

Use case: Control A when a button is not pressed, control B when it is

Here's how you would implement a typical use case. You want your rotary encoder to control target A when the button is not pressed and control target B when it's pressed.

1. Create a mapping for the button
 - As "Target", you need to choose ReaLearn itself (Type: [FX parameter: Set value](#), Track: <This>, FX: "... VSTi: ReaLearn (Helgoboss)"). As "Parameter", choose an arbitrary ReaLearn parameter, e.g. "Parameter 1".
 - As "Mode", choose either "Absolute" (if you want to switch the encoder function just momentarily) or "Toggle" (if you want the button to toggle between the two encoder functions).
2. Create a mapping with target A
 - Set "Active" to "When modifiers on/off", "Modifier A" to "Parameter 1" and disable the checkbox beside it. Set "Modifier B" to <None>.
 - This basically means "Hey, ReaLearn! Please activate this mapping only if ReaLearn Parameter 1 is **off**!" (remember, we control ReaLearn Parameter 1 using the button).
 - At this point, turning your encoder should control target A, but only if you don't press the button!
3. Create a mapping with target B
 - Just as in step 2, set "Active" to "When modifiers on/off" and "Modifier A" to "Parameter 1". **But:** Now **enable** the checkbox beside it. Set "Modifier B" to <None>.
 - This basically means "Hey, ReaLearn! Please activate this mapping only if ReaLearn Parameter 1 is **on**!"
 - At this point, turning your encoder should control target A if you don't press the button and control target B if you press the button.

Source

As mentioned before, a source usually represents a single control element on your controller. Sources share the following common settings and functions:

- **Learn:** Starts or stops learning the source of this mapping.
- **Category:** Lets you choose the source category.
 - **None:** A special kind of source that will never emit any events. It's intended to be used on mappings which are not supposed to be controlled directly but only via [Group interaction](#).
 - **MIDI:** Incoming MIDI events.
 - **OSC:** Incoming OSC events.
 - **REAPER:** Events that can occur within REAPER.

- **Virtual:** Invocations of virtual control elements (coming from virtual controller mappings). This source category is available for main mappings only.

- **Type:** Let's you choose the source type. Available types depend on the selected category.

All other UI elements in this section depend on the chosen category.

Category "MIDI"

All types in the MIDI category have the following UI elements in common:

- **Channel:** Optionally restricts this source to messages from a certain MIDI channel. Only available for sources that emit MIDI channel messages.

The remaining UI elements in this section depend on the chosen source type.

CC value

This source reacts to incoming MIDI control-change messages.

- **CC:** Optionally restricts this source to messages with a certain MIDI control-change controller number.
- **Character:** MIDI control-change messages serve a very wide spectrum of MIDI control use cases. Even though some control-change controller numbers have a special purpose according to the MIDI specification (e.g. CC 7 = channel volume), nothing prevents one from using them for totally different purposes. In practice that happens quite often, especially when using general-purpose controllers. Also, there's no strict standard whatsoever that specifies how relative values (increments/decrements) shall be emitted and which controller numbers emit them. Therefore you explicitly need to tell ReaLearn about it by setting the *source character*. The good news is: If you use "Learn source", ReaLearn will try to guess the source character for you by looking at the emitted values. Naturally, the result is not always correct. The best guessing result can be achieved by turning the knob or encoder quickly and "passionately" into clockwise direction. Please note that guessing doesn't support encoder type 3. The possible values are:
- **Range element (knob, fader, etc.):** A control element that emits continuous absolute values. Examples: Fader, knob, modulation wheel, pitch bend, ribbon controller. Would also include a endless rotary encoder which is (maybe unknowingly) configured to transmit absolute values.
- **Button (momentary):** A control element that can be pressed and emits absolute values. It emits a > 0% value when pressing it and optionally a 0% value when releasing it. Examples: Damper pedal.
- **Encoder (relative type x):** A control element that emits relative values, usually an endless rotary encoder. The *x* specifies *how* the relative values are sent. This 1:1 corresponds to the relative modes in REAPER's built-in MIDI learn:
 - **Type 1:**
 - 127 = decrement; 0 = none; 1 = increment
 - 127 > value > 63 results in higher decrements (64 possible decrement amounts)
 - 1 < value <= 63 results in higher increments (63 possible increment amounts)
 - **Type 2:**
 - 63 = decrement; 64 = none; 65 = increment
 - 63 > value >= 0 results in higher decrements (64 possible decrement amounts)
 - 65 < value <= 127 results in higher increments (63 possible increment amounts)
 - **Type 3:**
 - 65 = decrement; 0 = none; 1 = increment
 - 65 < value <= 127 results in higher decrements (63 possible decrement amounts)
 - 1 < value <= 64 results in higher increments (64 possible increment amounts)

- **Toggle-only button (avoid!):** A control element that can be pressed and emits absolute values. It emits a > 0% value when pressing it, no value when releasing it and a 0% value when pressing it again.
 - Hint: This is a workaround for controllers that don't have momentary buttons! You should only use this character if there's absolutely no way to configure this control element as a momentary button.
 - Background: ReaLearn can make a momentary hardware button work like a full-blown toggle button (ReaLearn's toggle mode is inherently more powerful than your controller's built-in toggle mode!). However, the opposite is not true. It can't make a toggle hardware button act like a momentary button.
 - The way this character works: ReaLearn will simply emit 100%, no matter what the hardware sends.
 - **Attention:** If you use the toggle-only source character in combination with mode "Incremental button", you must leave source max at the (default) theoretical maximum value for that source (e.g. 127 for MIDI CC). Even if your controller device only sends 0 and 1 and in all other mappings you would enter the controller's concrete (instead of theoretically possible) maximum value. Otherwise, for this special case, a fixed out-of-range-behavior will set in that will just ignore all button presses.
- **14-bit values:** If unchecked, this source reacts to MIDI control-change messages with 7-bit resolution (usually the case). If checked, it reacts to MIDI control-change messages with 14-bit resolution. This is not so common but sometimes used by controllers with high-precision faders.

Note velocity

This source reacts to incoming MIDI note-on and note-off messages. The higher the velocity of the incoming note-on message, the higher the absolute control value. Note-off messages are always translated to 0%, even if there's a note-off velocity.

- **Note:** Optionally restricts this source to messages with a certain note number (note numbers represent keys on the MIDI keyboard, e.g. 60 corresponds to C4).

Note number

This source reacts to incoming MIDI note-on messages. The higher the note number (= key on a MIDI keyboard), the higher the absolute control value.

This essentially turns your MIDI keyboard into a "huge fader" with the advantage that you can jump to any value at any time.

Pitch wheel

This source reacts to incoming MIDI pitch-bend change messages. The higher the pitch-wheel position, the higher the absolute control value. The center position corresponds to an absolute control value of 50%.

Channel after touch

This source reacts to incoming MIDI channel-pressure messages. The higher the pressure, the higher the absolute control value.

Program change

This source reacts to a range of incoming MIDI program-change messages. The higher the program number, the higher the absolute control value.

(N)RPN value

This source reacts to incoming non-registered (NRPN) or registered (RPN) MIDI parameter-number messages. The higher the emitted value, the higher the absolute control value.

(N)RPN messages are not widely used. If they are, then mostly to take advantage of their ability to transmit 14-bit values (up to 16384 different values instead of only 128), resulting in a higher resolution.

- **Number:** The number of the registered or unregistered parameter-number message. This is a value between 0 and 16383.

- **RPN:** If unchecked, this source reacts to unregistered parameter-number messages (NRPN). If checked, it reacts to registered ones (RPN).
- **14-bit values:** If unchecked, this source reacts to (N)RPN messages with 7-bit resolution, including increment/decrement messages. If checked, it reacts to those with 14-bit resolution. In practice, this is often checked.
- **Character:** See [CC value source](#).

Polyphonic after touch

This source reacts to incoming MIDI polyphonic-key-pressure messages. The higher the pressure, the higher the absolute control value.

- **Note:** Optionally restricts this source to messages with a certain note number.

MIDI clock tempo

This source reacts to incoming MIDI clock (MTC) tempo messages. These are metronome-beat-like messages which can be regularly transmitted by some DAWs and MIDI devices. The frequency with which this message is sent dictates the tempo.

The higher the calculated tempo, the higher the absolute control value. A tempo of 1 bpm will be translated to a control value of 0%, a tempo of 960 bpm to 100% (this corresponds to REAPER's supported tempo range).

This source can be used in combination with the [Project: Set tempo](#) target to obtain a "poor man's" tempo synchronization. Be aware: MIDI clock naturally suffers from certain inaccuracies and latencies - that's an issue inherent to the nature of the MIDI clock protocol itself. E.g. it's not really suitable if you need super accurate and instant tempo synchronization. Additionally, ReaLearn's algorithm for calculating the tempo could probably be improved (that's why this source is marked as experimental).

MIDI clock transport

This source reacts to incoming MIDI clock (MTC) transport messages. These are simple start, continue and stop messages which can be sent by some DAWs and MIDI devices.

- **Message:** The specific transport message to which this source should react.

Raw MIDI / SysEx

This source primarily deals with system-exclusive MIDI messages. Since ReaLearn v2.11.0, it supports both control and feedback direction!

- **Pattern:** Pattern describing the raw MIDI message.

Pattern basics

In its most basic form, the pattern is a sequence of bytes notated as hexadecimal numbers. This is typical notation, especially for system-exclusive MIDI messages.

Example:

```
F0 00 20 6B 7F 42 02 00 10 77 00 F7
```

If you enter this and ReaLearn receives this system-exclusive message from the input, it will fire a 100% value. If feedback is set up correctly, this message will be sent to the device whenever the target value changes.

Remarks:

- You can check if the correct feedback messages are sent to the device by [logging feedback messages](#).
- Each byte is written using 2 hexadecimal digits.
- Spaces between the bytes can be omitted.
- You can express all types of MIDI messages using this raw notation (e.g. pitch wheel), not just system-exclusive ones. If you do this, it will work as expected for the *feedback* direction. Please note that it will not work for the *control* direction at the moment (I don't think this is needed).

- If you want a system-exclusive MIDI message, you *must* include its start (F0) and end status byte (F7)!

Binary notation

ReaLearn also supports binary notation of a byte. You need to enclose the binary digits of one byte in brackets.

Example:

```
F0 00 20 [0110 1011] 7F 42 02 00 10 77 00 F7
```

This is equivalent to the first example (6B in hexadecimal notation is the same as 0110 1011 in binary notation).

Remarks:

- Between the brackets, each digit represents one bit. The left bit is the most significant one.
- Spaces between the two nibbles (4 bits) can be omitted.

Extracting and encoding a value

For the *feedback* direction, the examples I've shown you so far aren't real-world examples, because there's no point in sending the same MIDI message to the device over and over again! If you really would want to send a constant MIDI message to the device, you would be much better off using a [Mapping lifecycle action](#), which allow you to send raw MIDI messages once when a mapping is initialized, not on every target value change.

But even for the *control* direction, you might want to react to a whole *range* of system-exclusive messages, not just a fixed one. One part of your message might represent a variable value. You might want to extract it and control the target with it.

Fortunately, ReaLearn offers a uniform way to extract a variable value from the raw MIDI message (control) or encode the current target value as part of it (feedback). Bytes which contain a variable value (or a part of it) *must* be expressed using binary notation.

Example:

```
F0 00 20 6B 7F 42 02 00 10 77 [0000 dcba] F7
```

The second nibble of the second last byte contains the lowercase letters dcba. This is the portion of the byte that denotes the variable value.

Each letter represents one bit of the variable value:

- a - Bit 1 (least significant bit of the variable value)
- b - Bit 2
- c - Bit 3
- d - Bit 4
- ...
- m - Bit 13
- n - Bit 14
- o - Bit 15
- p - Bit 16 (most significant bit of the variable value)

The resolution of the variable value always corresponds to the letter in the whole pattern which represents the highest bit number. In the example above, the resolution is 4 bit because there's no letter greater than d in the pattern.

In the following example, the resolution is 7 bit because n is the greatest letter in the whole pattern.

```
F0 00 20 6B 7F 42 02 00 10 [00nm lkji] [hgfe dcba] F7
```

Remarks:

- The highest resolution currently supported is 16 bit (= 65536 different values).
- You can put these letter bits anywhere in the pattern (but only within bytes that use binary notation).

Byte order

This form of notation is slightly unconventional but I think it's very flexible because it gives you much control over the resulting MIDI message. This amount of control seems appropriate considering the many different ways hardware manufacturers used and still use to encode their MIDI data. When a number is expressed within more than one byte, manufacturers sometimes put the most significant byte first and sometimes the least significant one, there's no rule. This notation supports both because you decide where the bits end up:

Example for "most significant byte first":

```
F0 00 20 6B 7F 42 02 00 10 [ponm lkji] [hgfe dcba] F7
```

Example for "least significant byte first":

```
F0 00 20 6B 7F 42 02 00 10 [hgfe dcba] [ponm lkji] F7
```

More examples

"Romeo and Juliet" bits (separated by 2 bytes):

```
F0 [1111 000b] [a101 0100] F7
```

Simple on/off value (1 bit only):

```
F0 A0 [1111 010a] F7
```

This behaves like pitch wheel (because the pattern describes exactly the way how pitch wheel messages are encoded):

```
E0 [0gfe dcba] [0nml kjih]
```

MIDI Script

This source is feedback-only and exists for enabling more complex feedback use cases such as controlling LCDs that are not yet supported by the [Display](#) source. It lets you write an EEL or Lua script that will be executed whenever ReaLearn "feels" like it needs to send some feedback to the MIDI device.

- **Kind:** Whether to use the EEL or Lua language.
- **Script:** The script. Is disabled if the script contains more than one line.
- **...:** Opens the script in a separate window (for multi-line scripts).

Tip Prefer the [Display](#) source over this one whenever possible. It's easier to use.

General mechanics

- The script receives an input and must produce an output.
- **Script input**
 - The main input is the current feedback value, which the script can access as a variable.
- **Script output**
 - The main output that the script is supposed to return is the MIDI message to be sent to the MIDI device.
 - Additionally, the script can provide a so-called *feedback address*, which is supposed to uniquely identify the LED, motor fader or display. It's important to provide an address if you want ReaLearn to handle feedback relay correctly, e.g. that it switches off the LED when not in use anymore and doesn't switch it off if another mapping "takes over" the same LED. By convention, the constant (non-variable) bytes of the MIDI message should be used as address. The examples below might help to understand.

EEL script specifics

- **Script input**
 - EEL scripts can access numeric feedback values only. The current numeric feedback value is available as variable *y*, a floating point number between 0.0 and 1.0. This is essentially the current normalized target value after being processed by the "Glue" section of the mapping.

- **Script output**

- In order to provide the output MIDI message, you must assign the raw bytes of that message to subsequent slots of the EEL script's virtual local address space (by indexing via brackets) **and** set the variable `msg_size` to the number of bytes to be sent. If you forget the latter step, nothing will be sent because that variable defaults to zero!
- In order to provide the address, simply assign an appropriate number to the `address` variable.

- **Examples**

- The following example creates a 3-byte MIDI message.

```
address = 0x4bb0;
msg_size = 3;
0[] = 0xb0;
1[] = 0x4b;
2[] = y * 64;
```

Lua script specifics

- **Script input**

- Lua scripts can access numeric, text and dynamic feedback values.
- Here's the list of input variables:
 - `y`: The feedback value, either numeric (`type(y) == "number"`) or text (`type(y) == "string"`).
 - `context.feedback_event.color`: The color as set in the [Blue](#) section. Either the default color (`== nil`) or an RGB color (table with properties `r`, `g` and `b`).
 - `context.feedback_event.background_color`: The background color.

- **Script output**

- A Lua script can even generate multiple output messages.
- You need to return a table which contains the following keys:
 - `address`: The feedback address.
 - `messages`: An array containing all the messages, where each message itself is an array containing the message bytes.

- **Examples**

- Creates a 3-byte MIDI message, assuming that `y` is a numeric normalized value.

```
return {
  address = 0x4bb0,
  messages = {
    { 0xb0, 0x4b, math.floor(y * 10) }
  }
}
```

- Creates a MIDI sys-ex message that changes the color of some fictional device LED/display:

```
local color = context.feedback_event.color
if color == nil then
  -- This means no specific color is set. Choose whatever you need.
  color = { r = 0, g = 0, b = 0 }
end
return {
  address = 0x4b,
  -- Whatever messages your device needs to set that color.
```

```

    messages = {
        { 0xf0, 0x02, 0x4b, color.r, color.g, color.b, 0xf7 }
    }
}

```

- Creates a 3-byte MIDI message, assuming that *y* is a text value.

```

local lookup_table = {
    playing = 5,
    stopped = 6,
    paused = 7,
}
return {
    messages = {
        { 0xb0, 0x4b, lookup_table[y] or 0 }
    }
}

```

Tip Please note that this kind of simple mapping from text values to integer numbers doesn't need a script. You can use the `feedback_value_table` [Glue](#) property instead, which can only be set via API though. Do a full-text search for `feedback_value_table` in directory `resources/controller-presets` of the [ReaLearn source code](#) to find usage examples.

Display

This is a feedback-only source used to display text on MIDI-controllable hardware displays (LCDs, OLED displays, 7-segment displays, etc.).

- **Protocol:** Lets you choose the display protocol, which tells ReaLearn how it should communicate with the hardware display and which options it supports.
 - **Mackie LCD:** Use this for MCU-compatible LCDs. Depending on your particular control surface, there can be up to 8 LCDs, each of which has up to 2 lines.
 - **Mackie XT LCD:** Use this to control the displays of MCU XT devices (= control surface extenders, which provide additional faders and displays).
 - **X-Touch Mackie LCD:** Like *Mackie LCD* but also supports colors on certain X-Touch devices.
 - **X-Touch Mackie XT LCD:** Like *Mackie LCD XT* but also supports colors on certain X-Touch devices.
 - **Mackie 7-segment display:** Use this for MCU-compatible 7-segment displays (you know, the ones which only show digits). There's usually one small assignment display and a larger one for showing the time code.
 - **SiniCon E24:** Use this with the [SiniCon E24 controller](#).
 - **Launchpad Pro - Scrolling text:** Displays looped scrolling text on a Novation Launchpad Pro. Only seems to work if you set *Output* to `MIDIOUT2` (Launchpad Pro).
 - **Studiologic SL Keyboard display:** Displays text on the display of Studiologic SL keyboards (tested with SL88).
- **Display:** Choose the particular display or display portion to which you want to send text.
- **Line:** Choose the line number.

Caution For controllers with multiple displays and lines, ReaLearn allows you to spread your text over all available displays and lines. This is great if you need to display a lot of text but one display doesn't provide enough space. But be aware: Replacing feedback with other feedback ("feedback relay") doesn't work so nicely anymore if you make use of this feature.

If you want to know how to define which text shall be sent to the displays, please see [textual feedback](#) in the **Glue** section.

Specific program change

This source reacts to MIDI program-change messages with a specific program. This is a trigger-only source, that means it always fires 100% (whenever the program number corresponds to the configured one).

Category "OSC"

OSC sources allow configuration of the following aspects:

Address

This needs to correspond exactly to the address of the corresponding control element on your OSC device. Example: `/1/fader1`. You don't need to figure that out yourself, just use the *Learn* function.

Argument

Each OSC message consists of an arbitrary number of arguments. In most cases, e.g. with faders, knobs or buttons, it's just one argument. X/Y controls often send 2 arguments, one for each axis. There are rare cases in which messages have even more arguments.

The first dropdown menu allows you to choose the number of the argument that ReaLearn should look at and process. 1 denotes the first argument, 2 the second one, and so on.

The second dropdown menu lets you choose the argument type which ReaLearn should use to construct a proper feedback message.

- This is usually the same type as the one used for control direction. For control direction, choosing an explicit type is irrelevant because ReaLearn handles whatever type arrives automatically in the best possible way.
- If you use *Learn*, the type is filled automatically.
- The value to be sent will be derived from the type (see [Feedback arguments](#)):

Type	Property
Float	<code>value.float</code>
Double	<code>value.double</code>
Int	<code>value.int</code>
Long	<code>value.long</code>
Bool	<code>value.bool</code>
Nil	<code>nil</code>
Inf	<code>inf</code>
String	<code>value.string</code>
Color	<code>style.color</code>

- If you want more control over what feedback values are sent, use the [Feedback arguments](#) field.

Range

Values of argument types *Float* and *Double* are by default interpreted as decimal values between 0.0 and 1.0. You can change that by entering a different value range here. Even negative numbers are allowed.

Customizing the value range is especially important for argument types *Int* and *Long* because they don't have a standard value range.

Is relative

Some messages transmitted by OSC devices are meant to be interpreted as relative increments/decrements instead of absolute values, e.g. jog wheels. When you enable this checkbox, ReaLearn will treat each received *1* value as an increment and *0* value a decrement.

Feedback arguments

Allows you to define exactly which feedback value is sent at which argument position. If this field is non-empty, the *Type* dropdown will be ignored.

The format of this field is very simple: You enter feedback value property keys separated by spaces. Each entered property key corresponds to one argument position. E.g. if you want ReaLearn to send the current feedback value in text form at argument 1 and the color (see [Feedback style](#)) as RRGGBB string at argument 2, you would enter:

```
value.string style.color.rrgbbb
```

The following properties are available:

Property	Type	Description
value.float	Float	Numeric feedback value interpreted as float
value.double	Double	Numeric feedback value interpreted as double
value.bool	Bool	Numeric feedback interpreted as bool (on/off only)
value.string	String	Numeric or textual feedback value formatted as string
style.color.rrgbbb	String	Feedback value color formatted as RRGGBB string
style.background_color.rrgbbb	String	Feedback value background color formatted as RRGGBB string
style.color	Color	Feedback value color as native OSC color
style.background_color	Color	Feedback value background color as native OSC color
nil	Nil	Nil value
inf	Infinity	Infinity value

Category "Keyboard"

This source reacts to pressing or releasing a key on your computer keyboard. It emits a value of 100% when the key is pressed and 0% when released.

Usage:

- In order to set the key, simply click the **Learn** button and press the key of your choice.

- In addition to the key label, ReaLearn might show some warnings regarding the portability of your keystroke.
 - This helps you to avoid keyboard shortcuts that don't reliably work cross-platform (in other operating systems) or on other keyboard layouts.
 - You can ignore portability warnings if you use just this operating system and don't plan to share your keyboard presets with other users.

Tips:

- This only works if [Input](#) is set to **Computer keyboard**.
- If you hold a key, it will not keep firing. This is by design! Use [Fire after timeout, keep firing \(turbo\)](#) instead.
- Key combinations are not supported. This is by design! Use [Conditional activation](#) instead.

MIDI device changes

Category "REAPER"

MIDI device changes

This source emits a value of 100% whenever any MIDI device is connected and 0% whenever any MIDI device is disconnected. You can map this to the REAPER action "Reset all MIDI devices" to achieve true plug and play of MIDI devices (provided the corresponding device has been enabled at least once in REAPER's MIDI device preferences).

ReaLearn instance start

This source fires (emits a value of 100%) when ReaLearn starts. It can be used to execute an actions or restore certain states on REAPER startup or project load.

Timer

This source fires (emits a value of 100%) repeatedly every n milliseconds.

ReaLearn parameter

This source fires whenever one of ReaLearn's [Compartment parameters](#) is changed.

One of many ways to use this is to create macro parameters which control multiple parameters of multiple other plug-ins.

At the moment, mappings with this source can't participate in rendering. So it's important to write Warning down automation **before** rendering. This applies **in addition** the things pointed out in [Automation and rendering](#).

Speech

This source works for feedback only. It uses the native Windows or macOS text-to-speech engine to speak out any feedback value.

Category "Virtual"

As pointed out before, *virtual* sources exist in order to decouple your mappings from the actual MIDI/OSC source.

If you want to define a virtual source, you first need to choose among two types of virtual control elements: "Multi" (control elements that support more than 2 values) and "Button" (simple on/off controls). It's sort of the lowest common denominator among all possible control element types. This distinction is used by ReaLearn to optimize its user interface. In future, it might be used for additional improvements.

Both types are explained in detail below. They support the following settings:

- **ID:** A number or name for uniquely identifying the control element.

- Numbers are especially suited for the 8-knobs/8-buttons layouts. In a row of 8 knobs one would typically assign number 1 to the leftmost and number 8 to the rightmost one. It's your choice.
- For more advanced virtual control scenarios it can be useful to think in names instead of numbers. That's why the IDs of virtual control elements are not limited to numbers only. You can use up to 32 alphanumeric and punctuation characters (no exotic characters, e.g. no umlauts).
- **Pick:** Lets you conveniently pick out of predefined numbers and names. If you want your main preset to be compatible with as many controller presets as possible, try to use predefined names instead of inventing your own naming scheme.
 - **DAW control:** The names you see here are heavily inspired by the wording used with Mackie Control devices.
 - **Numbered:** Simply lets you pick among any number between 1 and 100. Wow, you can save up to 3 key presses!!!

Multi

Represents a control element that you can "move", that is, something that allows you to choose between more than 2 values. Usually everything which is *not* a simple on/off button :) Here's a list of typical *multis*:

- Fader
- Knob
- Pitch wheel
- Mod wheel
- Endless encoder
- XY pad (1 axis)
- Touch strip
- (Endless) rotary encoder
- Velocity-sensitive pads or keys

Button

Represents a control element that distinguishes between two possible states only (e.g. on/off), or even just one ("trigger"). Usually it has the form factor of a button that you can "press". Here's a list of typical *buttons*:

- Play button
- Switch
- Sustain pedal

Please note that velocity-sensitive keys should be exposed as "Multi", not as "Button" - unless you know for sure that you are not interested in the velocity sensitivity.

Target

A target is a thing that is supposed to be controlled.

Common target elements

Learn

Starts or stops learning the target of this mapping.

Menu

Opens a small menu related to the target section:

- **Pick recently touched target (by type):** Gives you a list of recently touched parameters or executed actions in REAPER. When you click one of it, the target will be populated accordingly. It's an alternative to "Learn". Please note that not all targets can be picked that way, some have to be configured manually.
- **Go there (if supported):** If applicable, this makes the target of this mapping visible in REAPER. E.g. if the target is a track FX parameter, the corresponding track FX window will be displayed.

Type

- **Left dropdown:** Lets you choose the target category.
 - **Real:** Targets that are about actually changing something "real", e.g. in REAPER or ReaLearn itself.
 - **Virtual:** Targets that invoke virtual control elements. This source category is available for controller mappings only.
- **Right dropdown:** Lets you choose a target type within that category.

Value

Reflects the current value of this mapping target and lets you change it (either via slider and text field or via buttons, depending on the target character).

- If the target can't be resolved at the moment, it will show "Target currently inactive!".

Unit button

On the right side of the current value you will see a button with a label such as 1. dB (%). This button displays the currently selected unit which is used for displaying and entering target values. The number in the parentheses denotes the unit which is used for displaying and entering target step sizes. Clicking the button switches between the units. Currently there are two options:

- **1. Use native target units:** Uses the target-specific unit, e.g. dB for volume targets. If the target doesn't have any specific units, it will displayed as 1. - (-).
- **2. Use percentages:** Uses percentages for everything, which can be nice to get a uniform way of displaying/entering values instead of having to deal with the sometimes clunky target-specific units.

Common selectors

Targets that need a track, FX, FX parameter or send/receive have dropdowns that let you choose how you want to *address* these objects. Let's call them *object selectors*. Here's an explanation of commonly available object selectors.

Note The descriptions below are sometimes a bit tailored to *track* objects but the same applies to all other objects that support it.

Selector "Instance"

This selector makes the target work on the current [Instance track](#) or current [Instance FX](#) of this particular ReaLearn instance.

Selector "Particular"

Lets you pick a specific object (e.g. track) and refer to it by its unique ID. This is the default and in most cases what you want. Choose this if you want ReaLearn to always control that very particular track even in case you move it somewhere else or rename it.

Please note that it's *not possible* with this setting to create a ReaLearn preset that is reusable among different projects. Because a track ID is globally unique, even across projects. That also means it doesn't make sense to use this setting in a ReaLearn monitoring FX instance.

Selector "At position"

This is the most straightforward selector. It lets you refer to a track by its position in the track list. This is great if you want to build a preset that you are going to reuse among multiple projects.

However, this selector has the disadvantage that things fall apart if you reorder, insert or delete tracks. This is why it's not the default.

Next to the dropdown you will find a text field. Here you should enter the position as number, starting with number 1.

Selector "Named"

Allows you to choose a track depending on its name. In case there are multiple tracks with the same name, it will always prefer the first one. This will allow you to use one ReaLearn preset across multiple projects that have similar naming schemes, e.g. as monitoring FX.

In the name field next to the dropdown, you can enter a name. If you don't want exact matching, you can use wildcards:

- * for matching zero or arbitrary many characters
- ? for matching exactly one arbitrary character.
- Example: Violin * would match Violin 1 or Violin 12 but not 12th Violin.

Selector "Dynamic"

This selector allows you to *calculate* which object (e.g. track) you want to use.

When you choose this option, a text field will appear next to it. This lets you enter a mathematical expression whose result should be the object's *index*. E.g. the first track in the project has index 0, the master track -1. For your convenience, you will find a small text label next to the expression text field that always shows the current result of your formula (clamped to the target value range).

Note Please note that the expression language is *not EEL* - this is a notable difference to ReaLearn's control/feedback transformation text fields! The expression language used here just provides very basic mathematical operations like addition (+/-), multiplication (*) etc. and it also doesn't allow or need any assignment to an output variable.

The dynamic selector is a very powerful tool because you can use some special variables:

Variable	Type	Applicable to	Description
none	-	All selectors	Special value that represents a "not set" value. See below for examples.
p	Array of floating-point numbers	All selectors	Allows you to access the values of ReaLearn's internal parameters. E.g. you can get the number of the first parameter by writing <code>p[0]</code> . By default, parameter values are normalized floating point values, that means they are decimal numbers between 0.0 and 1.0. This can be changed by customizing the parameter with a specific integer value count (see Compartment parameters).
p1 ... p100	Floating-point numbers	All selectors	Values of ReaLearn's internal parameters as single variables. <i>Deprecated:</i> Recent ReaLearn versions offer the <code>p</code> array instead. Better use that one.
selected_track_index	Integer ≥ -1	Track selectors	Resolves to the zero-based index of the first currently selected track within the containing project. If no track is selected, this resolves to <code>none</code> . If the master track is selected, -1.
selected_track_tcp_index	Integer ≥ -1	Track selectors	Like <code>selected_track_index</code> but counts only tracks that are visible in the track control panel.

Variable	Type	Applicable to	Description
<code>selected_track_mcp_index</code>	Integer ≥ -1	Track selectors	Like <code>selected_track_index</code> but counts only tracks that are visible in the mixer control panel. Lets you access the indexes of multiple selected tracks.
<code>selected_track_indexes</code>	Array of integers ≥ -1	Track selectors	E.g. if 2 tracks are selected, <code>selected_track_indexes[0]</code> gives you the index of the first selected track whereas <code>selected_track_indexes[1]</code> gives you the index of the second selected track. <code>selected_track_indexes[2]</code> would resolve to <code>none</code> .
<code>this_track_index</code>	Integer ≥ -1	Track selectors	Resolves to the zero-based index of the track on which this ReaLearn instance is located.
<code>instance_track_index</code>	Integer ≥ -1	Track selectors	Resolves to the zero-based index of the instance track of this ReaLearn instance.
<code>instance_track_tcp_index</code>	Integer ≥ -1	Track selectors	Like <code>instance_track_index</code> but counts only tracks that are visible in the track control panel.
<code>instance_track_mcp_index</code>	Integer ≥ -1	Track selectors	Like <code>instance_track_index</code> but counts only tracks that are visible in the mixer control panel.
<code>this_fx_index</code>	Integer ≥ 0	FX selectors	Resolves to the zero-based index of this ReaLearn FX instance.
<code>instance_fx_index</code>	Integer ≥ 0	FX selectors	Resolves to the zero-based index of the instance FX of this ReaLearn instance.
<code>tcp_fx_indexes</code>	Array of integers ≥ 0	FX selectors	Lets you access the indexes of FXs in a track control panel. E.g. <code>tcp_fx_indexes[2]</code> will resolve to the index of the third FX displayed in the track control panel.
<code>tcp_fx_parameter_indexes</code>	Array of integers ≥ 0	FX parameter selectors	Lets you access the indexes of FX parameters in a track control panel. E.g. <code>selected_fx_parameter_indexes[2]</code> will resolve to the index of the third FX parameter displayed in the track control panel. This only makes sense if used in conjunction with <code>tcp_fx_indexes</code> .

Examples of dynamic track expressions:

- `p1 * 99`
 - Will point to track with index 0 (first track) if "Parameter 1" is set to the minimum and to track with index 99 (= track number 100) if it's set to the maximum.
 - If you use a formula like that, you should make sure that "Parameter 1" is controlled with a step size that allows for exactly 100 different values. This conforms to ReaLearn's default step size $0.01 = 1\%$.
 - Since ReaLearn 2.13, this is easier because it adds support for integer parameters:

- Set the value count of "Parameter 1" to 100 (see [Compartment parameters](#))
 - You can now treat the parameter in the formula as an integer (just p1 instead of p1 * 99).
 - Most importantly, ReaLearn will take care of using the correct step size automatically when setting up a mapping for controlling that parameter.
- $p1 * 3 * 100 + p2 * 99$
 - This will treat "Parameter 1" as a kind of bank selector that allows you to choose between exactly 4 banks (0, 1, 2, 3) of 100 tracks each. "Parameter 2" will select the track number within the bank. You see, this is very flexible.

Common elements and selectors for track targets

When choosing a track, the following additional elements and selectors are available:

Track must be selected

If checked, this mapping will be active only if the track set in *Track* is currently selected.

Selection ganging

If checked and if the track in question is selected, all other selected tracks will be adjusted as well. This uses REAPER's built-in selection-ganging feature and therefore should behave exactly like it.

Respect grouping

If checked, track grouping will be taken into account when adjusting the value. This uses REAPER's built-in track grouping feature and therefore should behave exactly like it.

Note In older REAPER versions (< 6.69+dev1102), this can only be enabled together with selection ganging when using it on volume, pan or width targets.

Selector "<This>"

Track which hosts this ReaLearn instance. If ReaLearn is on the monitoring FX chain, this resolves to the master track of the current project.

Selector "<Selected>"

Currently selected track. If multiple tracks are selected, refers only to the first one.

Selector "<All selected>"

All currently selected tracks. This makes track targets (not FX target and not send targets) do their job on *all* selected tracks. The feedback value always corresponds to the highest value among all selected tracks.

Caution If you select many tracks, things can become quite slow!

Selector "<Master>"

Master track of the project which hosts this ReaLearn instance.

- If ReaLearn is on the monitoring FX chain, this resolves to the master track of the current project.
- If you don't have ReaLearn on the monitoring FX chain but you want to control an FX on the monitoring FX chain, this option is the right choice as well. Make sure to enable the "Monitoring FX" checkbox.

Selector "All named"

Allows you to use wildcards (see [Selector "Named"](#)) to make track targets do their thing on all matching tracks instead of only the first one.

Selector "At TCP position"

Like [Selector "At position"](#) but just considers tracks that are visible in the track control panel.

Selector "At MCP position"

Like [Selector "At position"](#) but just considers tracks that are visible in the mixer control panel.

Selector "Dynamic (TCP)"

Like [Selector "Dynamic"](#) but the result should be an index counting only tracks visible in the track control panel.

Selector "Dynamic (MCP)"

Like [Selector "Dynamic"](#) but the result should be an index counting only tracks visible in the mixer control panel.

Selector "By ID or name (legacy)"

This lets you refer to a track by its unique ID and name as fallback. This was the default behavior for ReaLearn versions up to 1.11.0 and is just kept for compatibility reasons.

Deprecated: You shouldn't use this selector anymore.

Common elements for on/off targets

Targets which control an on/off-style property of tracks (e.g. [Track: Solo/unsolo](#)) additionally provide the following elements.

Exclusive

By default, this option is set to "No".

- **No:** Makes the track target affect just this track.
- **Within project:** Switches the property on (off) for this track and off (on) for all other tracks in the project.
- **Within folder:** Switches the property on (off) for this track and off (on) for all other tracks in the same folder and same level.
- **Within project (on only):** Variation of *Within project* that applies exclusivity only when switching the property on for this track. In other words, it never switches the property on for other tracks.
- **Within folder (on only):** Variation of *Within folder* that applies exclusivity only when switching the property on for this track. In other words, it never switches the property on for other tracks.

Common elements for send targets

Only available for targets that work on a send/receive:

Kind

The kind of send/receive that you want to control.

- **Send:** Send from the track above to another track of your choice. If you choose [Selector "Particular"](#), ReaLearn will memorize the ID of the destination track. That way you will still control the correct send even if you delete another send in that track.
- **Receive:** Receive from another track of your choice to the track above (opposite direction of send). If you choose the [Selector "Particular"](#) selector, ReaLearn will memorize the ID of the source track.
- **Output:** Send from the track above to a hardware output. Please note that with hardware outputs, [Selector "Particular"](#) is the same as [Selector "At position"](#) because hardware outputs don't have unique IDs.

Send/Receive/Output

This lets you choose the actual send/receive/output.

Common elements and selectors for FX targets

The following elements and selectors are available for targets associated with a particular FX instance.

FX

The FX instance associated with this target. ReaLearn will search for the FX in the output or input FX chain of the above selected track.

Selector "<This>"

Always points to the own ReaLearn instance. Perfect for changing own parameters, e.g. for usage of the conditional activation or <Dynamic> features (especially important if you want to create reusable presets that make use of these features).

Selector "<Focused>"

Currently focused FX. *Track* and *Input FX* settings are ignored.

Selector "Particular"

Lets you pick a specific FX in the FX chain. Renaming the FX or moving it within the FX chain is fine - ReaLearn will still keep controlling exactly this FX. Please note that this only makes sense if you address the containing track using [Selector "Particular"](#) as well.

Selector "Named"

Lets you address the FX by its name in the FX chain. Just as with tracks, you can use wildcards to have a blurry search.

Selector "All named"

Allows you to use wildcard3s (see [Selector "Named"](#)) to make FX targets do their thing on all matching FX instances instead of only the first one.

Selector "By ID or position (legacy)"

This refers to the FX by its unique ID with its position as fallback. This was the default behavior for ReaLearn versions up to 1.11.0 and is just kept for compatibility reasons.

Deprecated: Don't use this selector anymore.

Input FX

If unchecked, the *FX* dropdown will show FX instances in the track's normal FX chain. If checked, it will show FX instances in the track's input FX chain.

Monitoring FX

This appears instead of the input FX checkbox if you select track <Master>. If you check this, you can target FX instances on REAPER's global monitoring FX chain.

Warning Because of a limitation in the REAPER API, learning and feedback for monitoring FX doesn't work at the moment!

FX must have focus

If checked, this mapping will be active only if the FX instance set in *FX* is currently focused.

If the FX instance is displayed in a floating window, *focused* means that the floating window is active. If it's displayed within the FX chain window, *focused* means that the FX chain window is currently open and the FX instance is the currently selected FX in that FX chain.

Of course, this flag doesn't have any effect if you chose <Focused> FX.

Common elements for pollable targets

The following elements are available only for the few targets that might need polling (= regular value querying) in order to support automatic feedback in all cases.

Poll for feedback

This makes ReaLearn query the current target value every few milliseconds in order to send up-to-date feedback to your controller at all times.

This is not necessary for most targets because usually ReaLearn takes advantage of REAPER's internal notification system to get notified about target value changes (which is great for performance). For the few targets for which it is, this option is enabled by default in order to give you the best feedback experience out-of-the-box.

In the probably rare case that the polling causes performance issues, you can untick this checkbox.

- For most targets, if you untick this checkbox, automatic feedback for that target will simply stop working. This means you will not receive up-to-date feedback anymore whenever you change the target value within REAPER itself (not using ReaLearn).
- The [FX parameter: Set value](#) target is an exception. Automatic feedback will still work, even without *Poll for feedback* enabled. But in the following corner cases it might not:
 - If the FX is on the monitoring FX chain.
 - If you change a preset from within the FX GUI.

Category "Real"

Global: Last touched

This will control whatever target has been last touched in REAPER. It's similar to the built-in REAPER action "Adjust last touched FX parameter" but provides the following benefits:

1. It's applicable to all ReaLearn targets that are learnable, not just FX parameters.
2. It offers feedback.
3. It can distinguish between parameter modifications caused by ReaLearn (i.e. hardware control) and those caused in other ways (e.g. via mouse).
 - **Targets → Pick!:** This opens a window that lets you pick all considered target types and types of invocations (only macOS and Windows so far). Last-touched targets not checked in this window will be ignored.

Global: Mouse

This will control the mouse.

- **Action**
 - **Move cursor to:** Moves the mouse cursor on the given axis in an absolute manner. This is a good choice for absolute mouse movement, that is, if you want to position the mouse cursor to a specific screen position. Although it's also possible to move the mouse cursor relatively with this action by controlling the target with relative messages, it's usually better to use *Move cursor by* instead.
 - **Move cursor by:** Moves the mouse cursor on the given axis in a relative manner. This is a good choice if you want to move the cursor e.g. up a bit, starting from its current position. This only works with relative control elements such as encoders or features such as [Make relative](#).
 - **Press or release button:** Presses or releases a certain mouse button, depending on the incoming control value (0% = release, anything else = press).
 - **Turn scroll wheel:** Simulates the scroll wheel.
- **Axis:** Determines the direction of movement or scrolling.
- **Button:** Determines which mouse button to use.

Tip One popular use of this target is to adjust the FX parameter under the mouse cursor. For this, it's usually best to set *Action* to "Turn scroll wheel" and *Axis* to "Y (vertical)".

Tip You can unfold the magic of this target by combining multiple mappings. E.g. one can simulate mouse dragging by using one mapping to press/release the left button and another mapping to move the cursor. [This example project](#) contains multiple examples (one per group).

Warning Feedback for this target is not fully implemented.

Global: Set automation mode override

Sets the global automation mode override to the desired value if the incoming control value is greater than 0%, otherwise removes the override.

- **Behavior:** Lets you decide between not overriding anything, bypassing all envelopes or overriding with a specific automation mode.
- **Mode:** Here you can pick the desired automation mode if *Behavior* is *Override*.

Project: Any on (solo/mute/...)

This target is most useful in feedback direction. Map it to some LED on your controller and the LED will light up if at least one of the tracks in your project is e.g. mute (depending on the track parameter in question).

If the control element is also a button, pressing the button will e.g. unmute all tracks in your project.

- **Parameter:** The track parameter in question.

Project: Invoke REAPER action

Triggers or sets the value of a particular REAPER action in the main section.

- **Invoke:** Specifies *how* the picked action is going to be controlled.
 - **Trigger:** Invokes the action with the incoming absolute control value, but only if it's greater than 0%. Most suitable for simple trigger-like actions that neither have an on/off state nor are annotated with "(MIDI CC/OSC only)" or similar.
 - **Absolute 14-bit:** Invokes the action with the incoming absolute control value, even if it's 0%. Most suitable for actions which either have an on/off state or are annotated with "(MIDI CC/OSC only)" or similar. The resolution of the invocation is 14-bit, no matter what's the resolution of your control element).
 - **Absolute 7-bit:** Just like the previous invocation mode but uses 7-bit resolution. Might be necessary for actions provided by 3rd-party extensions which don't interpret 14-bit control values correctly. In all other circumstances, 14-bit is probably the better default choice.
 - **Relative:** Invokes the action with the incoming relative control value (absolute ones are ignored). Only works for actions that are annotated with "(MIDI CC relative only)" or similar.
- **Pick!:** Opens REAPER's action dialog so you can select the desired action.
- **With track:** Allows you to choose a track which ReaLearn will select before executing the action. This makes it possible to combine ReaLearn's flexible track selection capabilities with the plethora of REAPER actions that work on the currently selected track.

The particular action decides if toggling/feedback works completely, has limitations or is not possible at all. There are multiple types of actions so it's not possible to settle with one invocation type and be done with it. The types of actions can roughly be divided into:

1. Actions that take care of toggling themselves *and* report on/off state.
 - Example: "25. Track: Toggle record arm for track 01"
 - If you want toggle behavior, you have 2 options:
 - a) Set Invoke to "Absolute" and Mode to "Toggle button" (preferred).
 - b) Set Invoke to "Trigger" and Mode to "Normal".
 - Feedback is completely supported.
2. Actions that take care of toggling themselves but *don't* report on/off state.

- Example: "40175. Item properties: Toggle mute"
- Toggle behavior is achieved as described in (1) but support for toggling and feedback has limitations (explained in (4)).

3. Actions that don't take care of toggling themselves ("trigger only").

- Example: "1007. Transport: Play"
- There's no way to make such an action toggle because the action is not designed to do so.
- If the action reports an on/off state, feedback is completely supported though, otherwise not at all!

4. Actions that have a complete range of values as state.

- Example: "994. View: Adjust vertical zoom (MIDI CC/OSC only)"
- Since ReaLearn 2 and REAPER 6.20, there's special support for this type of actions. Starting from the first time this action is triggered, ReaLearn will track its current value.
- That's why toggling is supported. Because ReaLearn itself takes care of toggling, you need to set *Invoke* to "Absolute" and *Mode* to "Toggle button".
- Feedback is also supported.
- Toggling/feedback for this type of actions comes with some inherent limitations that are related to the fact that a) REAPER itself doesn't necessarily use actions to invoke its own functions and b) MIDI CC/OSC actions don't have the concept of a "current value" (unlike e.g. toggle actions or FX parameters).
- The bottom line of these limitations is that toggling/feedback will only work if the action itself is used to trigger the change and if the action is an absolute action (not relative).
- Limitations in detail:
 - i. In most cases, feedback will not work when changing the value in REAPER directly (e.g. when adjusting vertical zoom directly via the REAPER user interface).
 - ii. It will only work for actions that support some kind of absolute value range (usually the case for all non-relative MIDI CC/OSC actions).
 - iii. When the action is invoked via ReaLearn, the feedback will only work if "Invoke" is "Trigger" or "Absolute". It won't work with "Relative".
 - iv. When the action is invoked from ReaScript or other extensions, it will only work if the invocation was done via `KBD_OnMainActionEx()` and an absolute value change.
 - v. When the action is invoked via a native REAPER action mapping, it will only work if the invocation is done using absolute MIDI CC/OSC (not relative).

Project: Invoke transport action

Invokes a transport-related action.

- **Action:** Specifies which transport action should be invoked.
 - **Play/stop:** Starts playing the containing project if the incoming absolute control value is greater than 0%, otherwise invokes stop.
 - **Play/pause:** Starts playing the containing project if the incoming absolute control value is greater than 0%, otherwise invokes pause.
 - **Stop:** Stops the containing project if the incoming absolute control value is greater than 0%. Useful for distinguishing feedback between *paused* and *stopped* state.
 - **Pause:** Pauses the containing project if the incoming absolute control value is greater than 0%. Useful for distinguishing feedback between *paused* and *stopped* state.

- **Record:** Starts/enables recording for the current project if the incoming absolute control value is greater than 0%, otherwise disables recording.
- **Repeat:** Enables repeat for the containing project if the incoming absolute control value is greater than 0%, otherwise disables it.

Project: Browse tracks

Steps through tracks. To be used with endless rotary encoders or previous/next-style "Incremental buttons".

- **Scroll TCP** and **Scroll mixer:** See [Track: Select/unselect](#) target.
- **Scope:** Decides which tracks are considered and how.
 - **All tracks:** Considers all tracks even those which are hidden.
 - **Only tracks visible in TCP:** Considers only those tracks which are visible in the track control panel.
 - **Only tracks visible in TCP (allow 2 selections):** Like "Only tracks visible in TCP" but makes it possible to have 2 selections. One for the MCP and one for the TCP. These selections can be moved independently. This can make sense if you have a bunch of tracks that you only show in the TCP and another separate bunch of tracks that you only show in the MCP.
 - **Only tracks visible in MCP:** Considers only those tracks which are visible in the mixer control panel.
 - **Only tracks visible in MCP (allow 2 selections):** See above.

Project: Seek

Allows you to use faders, knobs, encoders or incremental buttons to seek within portions of your project ... with feedback that indicates the current position!

- **Feedback:** Determines how frequently ReaLearn captures feedback and sends it to your feedback output.
 - **Beat:** Every beat.
 - **Fast:** As fast as possible, thereby giving the satisfying feeling of continuity. This obviously uses some more resources. No idea how far you can go with that. Try yourself.
- **Behavior:** Determines whether to use immediate or smooth seeking.
- **Seek play:** Doesn't just change the edit cursor but also changes the play position when the project is currently being played.
- **Move view:** Allow to scroll / change viewport when seeking.

The following options determine which time ranges will be taken into consideration as reference for seeking (control) and feedback.

1. **Use time selection:** Can use the currently set time selection as reference.
2. **Use loop points:** Can use the currently set loop points as reference.
3. **Use regions:** Can use the current region as reference.
4. **Use project:** Can use the complete project as reference, from start to end.

If you don't tick any "Use" checkbox, ReaLearn will seek within the currently visible viewport.

If you tick multiple options, this is the order of fallbacks:

- If there's no time selection, the loop points will be used.
- If there are no loop points, the current region is used.
- If there's no current region, the project will be used.
- If the project is empty, the viewport will be used.

This target supports the following additional placeholders in textual feedback expressions:

<code>target.position.project_default</code>	Position in the current transport time unit
<code>target.position.time</code>	<i>minute:second.milli</i>
<code>target.position.measures_beats_time</code>	<i>measure.beat.milli</i>
<code>target.position.measures_beats</code>	<i>measure.beat.milli</i>
<code>target.position.seconds</code>	<i>second.milli</i>
<code>target.position.samples</code>	<i>sample</i>
<code>target.position.hmsf</code>	<i>hour:minute:second:milli</i>
<code>target.position.absolute_frames</code>	<i>frames</i>
<code>target.position.project_default.mcu</code>	Like <code>target.position.project_default</code> but tailored to Mackie Control timecode displays
<code>target.position.time.mcu</code>	Like <code>target.position.time</code> but tailored to Mackie Control timecode displays
<code>target.position.measures_beats_time.mcu</code>	Like <code>target.position.measures_beats_time</code> but tailored to Mackie Control timecode displays
<code>target.position.measures_beats.mcu</code>	Like <code>target.position.measures_beats</code> but tailored to Mackie Control timecode displays
<code>target.position.seconds.mcu</code>	Like <code>target.position.seconds</code> but tailored to Mackie Control timecode displays
<code>target.position.samples.mcu</code>	Like <code>target.position.samples</code> but tailored to Mackie Control timecode displays
<code>target.position.hmsf.mcu</code>	Like <code>target.position.hmsf</code> but tailored to Mackie Control timecode displays
<code>target.position.absolute_frames.mcu</code>	Like <code>target.position.absolute_frames</code> but tailored to Mackie Control timecode displays

Project: Set playrate

Sets REAPER's master playrate.

Attention: This target doesn't currently work if the project containing ReaLearn is not the active project tab.

Project: Set tempo

Sets REAPER's master tempo.

This target is not learnable anymore via the "Learn target" button and also not eligible for the [Last touched](#) target because it caused too many "false positives".

Marker/region: Go to

Navigates to a specific marker or region. Here's the behavior in detail:

- **Regions**
 - If the project is stopped, the editor cursor immediately jumps to the start position of the given region.
 - If the project is playing, playback will continue with the given region as soon as the currently playing region (or measure if not within a region) has finished playing. This is called "smooth seek".
 - **Attention:** This currently doesn't work if the project containing ReaLearn is not the active project tab.
- **Markers**
 - If the project is stopped, the editor cursor immediately jumps to the given marker.
 - If the project is playing, playback will immediately be continued at the given marker.

The cool thing about this target compared to REAPER's built-in actions is that it allows to target arbitrarily many markers/regions (either by position or by ID) ... and that it supports visual feedback! If you assign this target to a button which has an LED, you will see which marker/region is currently playing just by looking at your controller.

Please note that this doesn't work when recording!

User interface elements specific to this target:

- **Marker/region:**
 - **Left dropdown:** This dropdown lets you choose if you want to refer to a marker/region by its user-assigned ID or by its position on the timeline.
 - **Right dropdown:** This dropdown displays the markers or regions (depending on the *Regions* checkbox state).
- **Now!:** This sets the target to the currently playing (or currently focused, if stopped) marker/region.
- **Behavior:** Determines whether to use immediate or smooth seeking.
- **Regions:** Switches between markers and regions.
- **Set loop points:** For regions, this will additionally set the loop points to the region start and end position.
- **Set time selection:** For regions, this will additionally set the time selection to the region start and end position.

This target supports the following additional placeholders in textual feedback expressions:

<code>target.bookmark.id</code>	(Numeric) ID of the bookmark
<code>target.bookmark.index</code>	Index of the bookmark (counting both markers and regions)
<code>target.bookmark.index_within_type</code>	Index of the bookmark (counting only markers or regions, respectively)
<code>target.bookmark.name</code>	Name of the bookmark

Track

A target that allows you to define a track.

The setting **Act/Tags** stands for "Action / Instance tags" and decides what happens when a control messages arrives, e.g. a button press:

- **None (feedback only):** With this setting, nothing will happen. It's suited very well as neutral target for textual feedback with an expression that contains a track property, e.g. `{{ target.track.name }}`.

- **Set (as instance track):** The button press will set the track defined in this target as [Instance track](#) *without resolving it before*. For example, if this target defines to use the currently selected track ([Selector "<Selected>"](#)), pressing the button will make the instance track dynamically reflect whatever track is selected.
- **Pin (as instance track):** The button press will resolve the track defined in this target and set the result as [Instance track](#). For example, if this target defines to use the currently selected track, pressing the button will check which track is currently selected and set the instance track to exactly this track. It will stay that way even if the user selects another track.

The text field to the right defines contains **Instance tags** of the ReaLearn instances whose instance track should be changed. If it's empty, the current instance will be affected.

Track: Arm/disarm

Arms the track for recording if the incoming absolute control value is greater than 0%, otherwise disarms the track. This disables "Automatic record-arm when track selected". If you don't want that, use the *Track: Select/unselect* target instead.

Track: Enable/disable all FX

Enables all the track's FX instances if the incoming absolute control value is greater than 0%, otherwise disables them.

Track: Enable/disable parent send

Enables the parent send routing of the track if the incoming absolute control value is greater than 0%, otherwise disables it.

Track: Mute/unmute

Mutes the track if the incoming absolute control value is greater than 0%, otherwise unmutes the track.

Track: Peak

This is a feedback-only target! It turns your feedback-capable controller into a VU meter by constantly reporting the current volume of the configured track to it.

In addition to connecting it with a LED ring or motor fader source (which should be obvious), it can also be used with a single LED to build a clipping indicator:

1. Set *Target Min* to the minimum dB value that should make your clipping LED turn on. Leave *Target Max* at 12.00 dB.
2. Make sure the *Out-of-range* behavior is set to "Min or max".
3. If you have an LED that supports multiple colors, you will probably see a rainbow of colors flashing up which can be quite confusing. Use the feedback transformation formula $x = \text{ceil}(y)$ to restrict the feedback to just two values: Min (0%) or Max (100%). You can then use *Source Min* and *Max* to adjust the off/on LED colors.

At the moment this target only reports peak volume, not RMS.

Track: Phase invert/normal

Inverts the track phase if the incoming absolute control value is greater than 0%, otherwise switches the track phase back to normal.

Track: Select/unselect

Selects the track if the incoming absolute control value is greater than 0%, otherwise unselects the track.

This target stops being learnable if you activate the REAPER preference "Mouse click on volume/pan faders and track buttons changes track selection" (because this preference would generate too many false positives). If you change the preference, ReaLearn will take it into consideration the next time you restart REAPER.

- **Scroll TCP:** Also scrolls the track control panel to the desired track.

- **Scroll mixer:** Also scrolls the mixer control panel to the desired track.

Track: Set automation mode

Sets the track to a specific automation mode if the incoming control value is greater than 0%, otherwise sets it back to REAPER's default track automation mode "Trim/Read".

- **Mode:** Here you can pick the desired automation mode.

Track: Set monitoring mode

Sets the track to a specific input monitoring mode if the incoming control value is greater than 0%, otherwise sets it back to "Off".

- **Mode:** Here you can pick the desired monitoring mode.

Track: Set automation touch state

When you use REAPER's "Touch" automation mode, REAPER needs a way to know if you are currently touching the control element which is bound to the automation envelope or not. As long as you keep touching it, it will overwrite existing automation. As soon as you release it, REAPER will leave the envelope untouched.

Classical control surfaces implement this very intuitively by providing touch-sensitive faders. With this target, you can easily reproduce exactly this behavior via ReaLearn. You do this by mapping the touch event (which is usually nothing else than a MIDI note on/off message) to this target. The touch state is scoped to a particular track and parameter type which you can choose in the ***Type** dropdown.

However, ReaLearn wouldn't be ReaLearn if it wouldn't allow you to let totally different sources take control of the touch state. For example, if you have a push encoder, you could map the "push" event to the touch state, allowing you to write automation only while you are touching the encoder. Or if you don't have a push encoder, you could just use some spare button.

Track: Set pan

Sets the track's pan value.

This target supports the following additional placeholders in textual feedback expressions:

<code>target.pan.mcu</code>	Pan value tailored to one line on a Mackie Control LCD
-----------------------------	--

Track: Set stereo pan width

Sets the track's width value (applicable if the track is in stereo pan mode).

This target supports the following additional placeholders in textual feedback expressions:

<code>target.width.mcu</code>	Width value tailored to one line on a Mackie Control LCD
-------------------------------	--

Track: Set volume

Sets the track's volume.

Track: Show/hide

Shows the track if the incoming absolute control value is greater than 0%, otherwise hides it.

- **Area:** Lets you decide if you want it to show/hide in the track control panel or the mixer.

Track: Solo/unsolo

Soloes the track if the incoming absolute control value is greater than 0%, otherwise unsoloes the track.

Provides the following additional settings:

- **Behavior:** See the REAPER user guide for details.
 - **Solo in place:** Soloes the track while respecting REAPER's routing. This is REAPER's default and since ReaLearn v2.4.0 also ReaLearn's default.
 - **Solo (ignore routing):** Soloes the track muting everything else, no matter the routing.
 - **Use REAPER preference:** Follows whatever is set in the REAPER preferences.

Learning this target by pressing the "Solo" button of the *master* track is currently not possible but of course you can just select it manually in the dropdown menu.

FX chain: Browse FXs

Steps through the FX instances in the FX chain by always having exactly one FX instance visible. To be used with endless rotary encoders or previous/next-style "Incremental buttons".

- **Display:** Here you can decide if you want to display the FX as part of the FX chain or in a dedicated floating window.

FX

A target that allows you to define an FX, in its basic variant perfect for acquiring feedback for a specific FX.

The setting **Act/Tags** allows you to optionally set/pin the declared FX as [Instance FX](#). This works pretty much the same as described in target [Track](#).

FX: Enable/disable

Enables the FX instance if the incoming absolute control value is greater than 0%, otherwise disables it.

FX: Set online/offline

Sets the FX instance online if the incoming absolute control value is greater than 0%, otherwise sets it offline.

FX: Load snapshot

Restores a certain state of a particular FX. Before using this target, you need to take a snapshot of the desired FX state using the *Take!* button. This snapshot will be saved as part of ReaLearn's state itself and as a direct consequence as a part of your project. This makes your project nicely self-contained. It's perfect for activating particular FX presets because it will always restore the desired state, even if the preset list has changed.

This target supports feedback, but only if the snapshot is loaded via ReaLearn itself.

Please note that some plug-ins have *very large* states. Therefore you should keep an eye on the snapshot size, which will be displayed once you take the snapshot. ReaLearn's own state will grow with every new snapshot mapping, so this can quickly add up and make REAPER/ReaLearn slow!

FX: Browse presets

Steps through FX presets.

This target is suited for use with knobs, encoders and incremental buttons (previous/next) because it allows you to step through the complete preset list. The minimum value always represents *No preset* whereas the maximum value always represents the last available preset.

It's *not* suited for activating a particular preset (e.g. by setting *Target Min* and *Max* to the same value), because the preset list of an FX is usually not constant. As soon as you modify the preset list, this value will might suddenly point to a completely different preset. Even worse, the actual preset might have been deleted.

If you want to activate a particular preset, please use the [FX: Load snapshot](#) target instead.

FX: Open/close

Makes the FX instance visible if the incoming control value is greater than 0%, otherwise hides it.

- **Display:** Here you can decide if you want to display the FX as part of the FX chain or in a dedicated floating window.

FX parameter: Set automation touch state

This does the same as [Track: Set automation touch state](#) but for FX parameter value changes.

FX parameter: Set value

Sets the value of a particular track FX parameter.

- **Parameter:** The parameter to be controlled. Please note that both [Selector "Particular"](#) and [Selector "At position"](#) address the FX by its position in the FX chain. The difference between the two is that [Selector "Particular"](#) shows a dropdown containing the available parameters and [Selector "At position"](#) lets you enter the position as a number in a text field. Latter is useful if at the time of choosing the position, the FX is not available.

This target supports the following additional placeholders in textual feedback expressions:

<code>target.fx_parameter.index</code>	Zero-based index of the resolved FX parameter.
<code>target.fx_parameter.name</code>	Name of the resolved FX parameter.
<code>target.fx_parameter.macro.name</code>	Name of the corresponding Pot macro parameter. Only works if this parameter is part of a preset loaded via Pot.
<code>target.fx_parameter.macro.section.name</code>	Name of the corresponding Pot macro parameter section. Only works if this parameter is part of a preset loaded via Pot.
<code>target.fx_parameter.macro.section.index</code>	Zero-based index of the corresponding Pot macro parameter section (within the current bank). Only works if this parameter is part of a preset loaded via Pot.
<code>target.fx_parameter.macro.new_section.name</code>	Name of the corresponding Pot macro parameter section, but only if this parameter marks the start of a new section. Only works if this parameter is part of a preset loaded via Pot.
<code>target.fx_parameter.macro.bank.name</code>	Name of the corresponding Pot macro parameter bank. Only works if this parameter is part of a preset loaded via Pot.

Pot: Browse filter items

This target can be used to filter the potentially very large collection of presets in [Pot: Browse presets](#). The idea is to map this target to an endless rotary encoder or previous/next buttons (using [Incremental button](#) mode) and then navigate within the available filter items, e.g. instruments or banks.

- **Kind:** Choose the kind of filter items that you want to browse. They correspond to the filters available in [Open Pot Browser](#).

This target supports the following additional placeholders in textual feedback expressions:

<code>target.item.name</code>	Name of the filter item.
<code>target.item.parent.name</code>	Name of the parent filter item if there's any. E.g. the instrument to which a bank belongs or the type to which a sub type belongs.

Pot: Browse presets

Use this target to browse a collection of presets. By default, this is the complete collection of presets available in all supported databases, so potentially thousands of presets. If you want to browse just a subset, see [Pot: Browse filter items](#).

The idea is to map this target to an endless rotary encoder or previous/next buttons (using [Incremental button mode](#)) and then navigate within the available presets. Once you have selected a preset, you can audition it via [Pot: Preview preset](#) (if it's a sound preset) and load it via [Pot: Load preset](#).

This target supports the following additional placeholders in textual feedback expressions:

<code>target.preset.name</code>	Name of the preset.
<code>target.preset.product.name</code>	Name of the product to which this preset belongs, if available.
<code>target.preset.file_ext</code>	File extension of the preset, in case it's a file-based preset.
<code>target.preset.author</code>	Name of the preset author, if available.
<code>target.preset.vendor</code>	Name of the preset vendor, if available.
<code>target.preset.comment</code>	Preset comment, if available.

Pot: Preview preset

Auditions a preset selected via [Pot: Browse presets](#). Only works if it's a sound preset and a sound preview file is available.

Pot: Load preset

Loads a preset selected via [Pot: Browse presets](#).

This needs at least REAPER version 6.69+dev1030! Also, it only works if you have the VST2/VST2i version Note of the corresponding plug-in installed. - **NKS audio file presets:** Loading supported via ReaSamplematic5000

Settings:

- **Track/FX:** You must tell the target at which FX slot to load the corresponding plug-in. The best idea is to use FX selector [Selector "At position"](#). Selectors such as [Selector "Particular"](#) or [Selector "Named"](#) are not suited because the target might replace the plug-in with another one, in which the unique FX ID and the FX name can change. Then the target would turn inactive and stop working.

This target supports the same additional placeholders for textual feedback expressions as [Pot: Browse presets](#). The only difference is that the ones in [Pot: Load preset](#) relate to the currently loaded preset, not the one that's selected in the preset browser.

Send: Automation mode

Sets the track send to a specific automation mode if the incoming control value is greater than 0%, otherwise sets it back to REAPER's default automation mode "Trim/Read".

Send: Mono/stereo

Sets the track send to mono or back to stereo.

Send: Mute/unmute

Mutes/unmutes the track send.

Send: Phase invert/normal

Inverts the track send phase or switches it back to normal.

Send: Set automation touch state

This does the same as [Track: Set automation touch state](#) but for send volume or pan adjustments.

Send: Set pan

Sets the track send's pan value.

Send: Set volume

Sets the track send's volume.

Clip: Invoke transport action

Caution Clips are a highly experimental feature of ReaLearn and still subject to many changes! Better don't rely on it at the moment!

Under construction

Clip: Seek

Under construction

Clip: Volume

Under construction

MIDI: Send message

Sends arbitrary MIDI messages (also sys-ex!) in response to incoming messages. This target turns ReaLearn into a capable and convenient MIDI → MIDI and OSC → MIDI converter.

- **Output:** Where to send the MIDI message.
 - **FX output:** Sends the MIDI message to the output of this ReaLearn instance - which usually means it flows into the FX below ReaLearn, e.g. a VST instrument.
 - **Feedback output:** Sends the MIDI message to the device which is set as *output*.
- **Pattern:** Defines the MIDI message to be sent as a sequence of bytes in hexadecimal notation. It also allows you to encode the incoming *absolute* control value as part of the message (after it has been processed by the glue section). The syntax for doing this takes some getting used to but it's very flexible. It's exactly the same syntax as used in the [Raw MIDI source](#). Please read about it there!
- ... Primarily provides predefined patterns. Just pick one here, set the destination to "Feedback output" and add a "ReaControlMIDI" FX below to see which messages ReaLearn sends.

This target is a bit special in that it carries out its processing logic exclusively in the audio thread if it's controlled by a MIDI source. This has the big advantage that receiving and producing MIDI messages happens in one go (without inter-thread-communication latency), which is often important when using MIDI message conversion.

However, this also means that the following things won't work when controlling this target using MIDI:

- Note
- It can't take the lead in [Group interaction](#).
 - If *output* is set to <FX output>, additional limitations apply:
 - It can't act as a follower in [Group interaction](#), either.
 - It can't participate in [ReaLearn: Load mapping snapshot](#).

OSC: Send message

Sends OSC messages with up to one argument in response to incoming messages. This target turns ReaLearn into a capable and convenient MIDI → OSC and OSC → OSC converter. If an argument number is entered (e.g. 1), it will encode the incoming absolute control value as that argument (after it has been processed by the glue section).

- **Output:** Where to send the OSC message.
 - **<Feedback output>:** Sends the OSC message to the device which is set as *Output*. Of course this only works if it's an OSC device.
 - **Specific device:** Sends the OSC message to a specific device.
- **Address, Argument and Range:** These correspond to the identically named settings of [OSC sources](#). Check that section for details.

ReaLearn: Enable/disable instances

This target allows you to flexibly enable or disable other ReaLearn instances based on their tags:

- **Exclusivity**
 - **Non-exclusive:** If the incoming control value is greater than 0%, all matching ReaLearn instances will be enabled (on top of the already enabled instances). If the value is 0%, all matching ReaLearn instances will be disabled.
 - **Exclusive:** If the incoming control value is greater than 0%, all matching ReaLearn instances will be enabled and all non-matching ones will be disabled. If the value is 0%, it's exactly the opposite (react to button [press only](#) if you don't want this to happen).
 - **Exclusive (on only):** Variation of *Exclusive* that applies exclusivity only if the incoming control value is greater than 0%.
- **Tags:** A ReaLearn instance matches when it is tagged with any of the tags entered in this field (comma-separated).

Tip Use the main panel to assign tags to a ReaLearn instance.

Please note:

- This really affects other ReaLearn instances only, not *this* instance.
- ReaLearn instances without tags won't be affected at all.
- Only affects instances in the same project. If *this* ReaLearn instance is on the monitoring FX chain, it only affects other instances in the monitoring FX chain.

Tip This target is great for switching between completely different controller setups!

ReaLearn: Dummy target

This target simply does nothing when invoked and also doesn't provide any meaningful feedback on its own.

It's sometimes useful to have such a dummy target, e.g. combined with [Group interaction](#). Or if you want to use ReaLearn as a MIDI filter which just "eats" an incoming MIDI message. Or if you want to send some text feedback to a hardware display, if the text is just a constant string or uses a placeholder that doesn't need target context.

ReaLearn: Enable/disable mappings

This target allows you to flexibly enable or disable other mappings in this instance based on their tags:

- **Exclusivity**
 - **Non-exclusive:** If the incoming control value is greater than 0%, all matching mappings will be enabled (on top of the already enabled mappings). If the value is 0%, all matching mappings will be disabled.

- **Exclusive:** If the incoming control value is greater than 0%, all matching mappings will be enabled and all non-matching ones will be disabled. If the value is 0%, it's exactly the opposite (react to button [press only](#) if you don't want this to happen).
- **Exclusive (on only):** Variation of *Exclusive* that applies exclusivity only if the incoming control value is greater than 0%.

- **Tags:** A mapping matches when it is tagged with any of the tags entered in this field (comma-separated).

Tip Use the text field on the top of the mapping panel to assign tags to a mapping.

Please note:

- This really affects other mappings only, not *this* mapping.
- Mappings without tags won't be affected at all.

Tip This target is a straightforward alternative to [Conditional activation](#), especially when it comes to bank switching!

ReaLearn: Load mapping snapshot

Restores target values for all or certain mappings in this ReaLearn instances.

- **Snapshot:** Choose the snapshot that you want to load.
 - **<Initial>:** Restores the initial target values for the mappings.
 - **By ID:** Restores target values contained in a snapshot that was taken via [ReaLearn: Take mapping snapshot](#). Simply enter the corresponding ID here.
- **Default:** Allows you to define a default target value to restore for each participating mapping whenever the snapshot either doesn't exist or doesn't contain a value for that mapping. If that participating mapping has [Reverse](#) checked, the inverse of the default value will be loaded.
- **Tags:** Allows you to restrict the set of mappings whose target values will be restored.
 - If this field is empty, target values of all mappings will be restored.
 - If this field contains tags (comma-separated), target values will be restored only for mappings that are tagged with any of these.
- **Active mappings only:** By default, even target values for inactive (but control-enabled) mappings are restored! If you don't like that, tick this checkbox.

Please note:

- Mappings for which control is not enabled, never participate in snapshotting.
- Some targets don't report values and therefore don't participate in snapshotting.
- Feedback of this target indicates whether the desired snapshot is the one which has last been loaded (for the given tags).

ReaLearn: Modify mapping

Triggers a modification of another ReaLearn mapping.

- **Kind:** The kind of modification.
 - **Learn target:** Switches "Learn target" on or off for the destination mapping. Use button ... to pick the considered target types and invocations to be included in the learning process.
 - **Set target to last touched:** Sets the target of the destination mapping to the last-touched target. Use button ... to pick the considered target types and invocations.
- **Instance:** Allows you to pick another ReaLearn instance.

- **Mapping:** Allows you to pick the destination mapping.

Tip This target is great to "pin" targets to certain control elements on demand.

ReaLearn: Take mapping snapshot

Memorizes target values for all or certain mappings in this ReaLearn instances and saves them in a snapshot of your choice.

- **Snapshot:** Choose the snapshot to which you want to save the mapping values.
 - **<Last loaded>:** Always chooses the snapshot which is currently active (was last loaded) for the given tags.

Tip Only works if tags are not empty and if all tags have the same last-loaded snapshot. So the best is if you always enter exactly one tag.
 - **By ID:** Enter the unique ID of the snapshot, e.g. `scene_1`.
- **Tags:** Allows you to restrict the set of mappings whose target values will be memorized.
 - If this field is empty, target values of all mappings will be memorized.
 - If this field contains tags (comma-separated), target values will be memorized only for mappings that are tagged with any of these.
- **Active mappings only:** By default, even target values of inactive (but control-enabled) mappings end up in the snapshot! If you don't like that, tick this checkbox.

ReaLearn: Browse group mappings

This target lets you choose an arbitrary mapping group in this compartment and cycle through it with an encoder/fader/knob or incremental (previous/next) buttons.

"Cycling through" means that you move from one mapping in the group to the next one by hitting the next mapping's target with the *Target Max* value in its glue section (by default 100%).

- **Group:** The group that you want to browse.
- **Exclusivity**
 - **Non-exclusive:** Really just hits the target of the mapping which is next in the line and doesn't do anything with the other mappings. In many cases this is enough, e.g. if the targets of the mappings in the cycled group are the same and just "Target Max" is different. Or if the target itself already takes care of exclusivity.
 - **Exclusive:** Doesn't just hit the target of the mapping which is next in the line but also hits the targets of all other mappings in the cycled group with their respective *Target Min* value (by default 0%). Be careful with this, you often won't need it.

Please note:

- Inactive mappings are skipped.

Mapping lend themselves perfectly for defining things that should happen *in sequence*. This target allows you to take advantage of that!

- Combine it with [ReaLearn: Enable/disable mappings](#) to browse different banks.
- Combine it with [ReaLearn: Enable/disable instances](#) to browse completely different controller setups (or banks).
- Combine it with targets that don't provide a "Browse ..." variant themselves.
- Use it as an alternative to [target value sequences](#) that allows you to have completely different targets within one sequence.

Tip

Category "Virtual"

This is exactly the counterpart of the possible [virtual sources](#). Choosing a virtual target here is like placing cables between a control element and all corresponding main mappings that use this virtual control element as source.

Glue

As mentioned before, the glue section defines the glue between a source and a target. It's divided into several sub sections some of which make sense for all kinds of sources and others only for some.

At first something important to understand: Since ReaLearn 2, a mapping can deal with both *absolute* and *relative* values, no matter what's set as *Mode*! ReaLearn checks the type of each emitted source value and interprets it correctly. The *Mode* dropdown has been sort of "degraded" because now it only applies to incoming *absolute* values and determines how to handle them (see further below). This change has been made to support virtual sources - because virtual sources can be either absolute or relative depending on the current controller mappings. ReaLearn allows you to prepare your mapping for both cases by showing all possible settings.

Relative means that the current target value is relevant and the change of the target value is calculated in terms of increments or decrements. Control elements that can emit relative values are rotary encoders and virtual multis.

Having so many settings available at the same time can be a bit daunting. ReaLearn helps you by hiding settings that don't make sense in the current context. It shows or hides them based on the following criteria:

- Is control and/or feedback enabled for the mapping?
- What are the characteristics of the source and target?
- What's the current setting of *Absolute mode* and *Make absolute*?

Reset to defaults

Resets the settings to some sensible defaults.

Reverse

If checked, this inverses the direction of the change. E.g. the target value will decrease when moving the fader upward and increase when moving it downward.

Target Min/Max

The controlled range of absolute target values. This enables you to "squeeze" target values into a specific value range.

Example: If you set this to "-6 dB to 0 dB" for a *Track volume* target, the volume will always stay within that dB range if controlled via this mapping. It wouldn't prevent the volume from exceeding that range if changed e.g. in REAPER itself.

This setting applies to targets which are controlled via absolute control values (= all targets with the exception of the [Project: Invoke REAPER action](#) if invocation type is *Relative*).

These are relevant for the control direction only:

Value sequence

Allows you to define a specific sequence of target values. This is a very powerful feature because you not only can enter single values but also ranges with customizable step sizes! Plus, it has support for true relative control. All values are entered comma-separated and using the unit chosen in the [target section](#). Just click the unit button to switch between the native target unit and percentages.

Examples:

- -20, -14, -12, -6, -3.5, 0: Enter this sequence for a volume target with target unit switched to dB. When you move your knob or rotary encoder or press a button using *Incremental button* mode, ReaLearn will step through the entered dB values for you.
- 10 - 30, 50 - 70 (5), 80 - 90 (2): Enter this sequence for a target with a continuous value range and target unit switched to %. It will first step in 1% steps from 10% to 30%, then in 2% steps from 50% to 70% and finally from 80% to 90% in 2% steps. At the moment it's important that the numbers and the range dash are separated by spaces!

- 20, 10, 10, -5, 8: When using absolute control, even duplicate values and direction changes are respected, as seen in this value sequence. However, true relative control naturally supports continuous sequences only. So if you have a rotary encoder that sends relative messages (hopefully!) or use incremental buttons, the sequence will be stepped through in a continuous manner (-5, 8, 10, 20). The benefit as always: No parameter jumps! If you want to use non-continuous sequences with encoders or incremental buttons, you can always use *Make absolute*!

Group interaction

Lets you control not just *this* mapping but also *all other mappings in the same mapping group*. Very powerful feature!

If you want to control *other* mappings only and not *this* mapping, just pick a target that doesn't have any Tip effect, for example the [ReaLearn: Dummy target](#) or an unused internal ReaLearn compartment parameter (target [FX parameter: Set value](#) with FX set to <This>).

- **None:** Switches group interaction off. This is the default. Incoming control events will just affect *this* mapping, not others.
- **Same control:** This will broadcast any incoming control value to all other mappings in the same group. The glue section of this mapping will be ignored when controlling the other mappings. The glue sections of the other mappings will be respected, including the source min/max setting.
- **Same target value:** This will set the target value of each other mapping in the same group to the target value of this mapping. Nice: It will respect the target min/max setting of both this mapping and the other mappings. All other settings of the glue section will not be processed. Needless to say, this kind of control is always absolute, which means it can lead to parameter jumps. Therefore, it's most suited for on/off targets. If you don't like this, choose *Same control* instead.
- **Inverse control:** This is like *Same control* but broadcasts the *inverse* of the incoming control value.
- **Inverse target value:** This is like *Same target value* but sets the target values of the other mappings to the *inverse* value. This is very useful in practice with buttons because it essentially gives you exclusivity within one group. It's a great alternative to the *Exclusive* setting which is available for some targets. Unlike the latter, *Inverse target value* allows for exclusivity between completely different target types and completely custom groupings - independent of e.g. organization of tracks into folders.
- **Inverse target value (on only):** Variation of *Inverse target value* that applies the inverse only when the target value is > 0%.
- **Inverse target value (off only):** Variation of *Inverse target value* that applies the inverse only when the target value is 0%.

Feedback type

Determines whether to send numeric, textual or dynamic feedback to the source.

Numeric feedback: EEL transformation

Sends numeric feedback to the source. This is the default.

The text field below allows you to specify an optional feedback transformation formula, which is much like [Control transformation \(EEL\)](#) but used for translating a target value back to a source value for feedback purposes. Be aware: Here x is the desired source value (= output value) and y is the current target value (= input value), so you must assign the desired source value to x .

Example: $x = y * 2$.

ReaLearn's feedback processing order is like this since version 2: .. Apply target interval. .. Apply reverse. .. Apply transformation. .. Apply source interval.

Textual feedback: Text expression

With this option, ReaLearn will send textual feedback values to the source. This only works with sources that are capable of displaying text: That is any [OSC source](#) with argument type *String*, [MIDI Display](#) and [MIDI Script](#).

The field below contains the *textual feedback expression*. Here you define which text is going to be sent to the source *whenever the target value changes* and also - for your convenience - immediately at the moment of entering the text. Whatever text you enter here, will be sent verbatim to the source.

Of course, entering a fixed text here is not very exciting. Most likely you want to display dynamic text such as the name of the currently selected track or the current target value, nicely formatted! You can do that by using placeholders, delimited by double braces. Example: `{{target.text_value}}`.

Which placeholders are available, depends very much on the type of the mapping target. However, there are some which are available for (almost) any target:

<code>global.realearn.time</code>	Time in milliseconds since ReaLearn has been loaded (the first instance).
<code>mapping.name</code>	Name of the mapping. Contains the explicitly assigned mapping name, never an automatically generated one.
<code>target.text_value</code>	<p>Short text representing the current target value, including a possible unit.</p> <p>If the target value can be represented using some kind of name, this name is preferred over a possibly alternative numeric representation. Example: Let's assume the 4th track in our project is called "Guitar" and the mapping target is Project: Browse tracks. Then <code>target.text_value</code> contains the text <i>Guitar</i>, not the text <i>4</i>.</p> <p>This is the default value shown if textual feedback is enabled and the textual feedback expression is empty.</p>
<code>target.available</code>	<p>A boolean value indicating whether the target is currently available or not.</p> <p>Most targets that are <i>active</i> are also <i>available</i>. But some targets can be <i>active</i> and <i>unavailable</i>. Example: Pot: Preview preset is not <i>available</i> if no preview is available for the preset currently selected in Pot browser. But the target is still considered <i>active</i> in this case!</p> <p>Usually used together with Dynamic feedback: Lua script, for example in order to display different things on displays depending on the target's availability.</p>
<code>target.discrete_value</code>	The current target value as zero-based integer. This only works for discrete targets.
<code>target.discrete_value_count</code>	The number of possible values in the current target. This only works for discrete targets.
<code>target.numeric_value</code>	<p>The current target value as a "human-friendly" number without its unit.</p> <p>The purpose of this placeholder is to allow for more freedom in formatting numerical target values than when using <code>target.text_value</code>. Future versions of ReaLearn might extend textual feedback expressions in a way so the user can define how exactly the numerical value is formatted (e.g. the number of digits after the decimal point).</p>
<code>target.numeric_value.unit</code>	Contains the unit of <code>target.numeric_value</code> (e.g. <i>dB</i>).

<code>target.normalized_value</code>	The current target value as normalized number, that is, a value between 0.0 and 1.0 (the so-called unit interval). You can think of this number as a percentage, and indeed, it's currently always formatted as percentage. This value is available for most targets and good if you need a totally uniform and predictable representation of the target value that doesn't differ between target types. By default, this number is formatted as percentage <i>without</i> the percent sign. Future versions of ReaLearn might offer user-defined formatting. This will also be the preferred form to format on/off states in a custom way (where 0% represents <i>off</i>).
<code>target.type.name</code>	Short name representing the type of the mapping target.
<code>target.type.long_name</code>	Long name representing the type of the mapping target.
<code>target.track.index</code>	Zero-based index of the first resolved target track (if supported).
<code>target.track.name</code>	Name of the first resolved target track (if supported).
<code>target.fx.index</code>	Zero-based index of the first resolved target FX (if supported).
<code>target.fx.name</code>	Name of the first resolved target FX (if supported).
<code>target.route.index</code>	Zero-based index of the first resolved target send/receive/output (if supported).
<code>target.route.name</code>	Name of the first resolved target send/receive/output (if supported).

For target-specific placeholders, please look up the corresponding [Target](#) section.

Dynamic feedback: Lua script

This feedback type puts you fully in charge about which feedback to send to the source. It does so by letting you define a Lua script that builds numeric, textual or even arbitrarily structured feedback.

General mechanics

- ReaLearn executes your script whenever one of the ReaLearn-provided properties used in your script might have changed its value.
- The script receives an input and must produce an output.
- **Script input**
 - The input is a function `context.prop` which you can use to query arbitrary properties, e.g. target or mapping properties. Those properties are the very same properties that you can use in [textual feedback](#).
 - Here's an example how to use this function:

```
local preset_name = context.prop("target.preset.name")
local param_name = context.prop("target.fx_parameter.name")
```

- Values returned by this function can be `nil`! E.g. target-related properties return a `nil` value whenever the mapping or target turns inactive, which is a very common situation. So it's important to prepare your Lua code for that, otherwise script execution fails and no feedback will be sent. One way to deal with a `nil` value returned by `context.prop` is to also return `nil` as value (see below).

• Script output

- The output that the script is supposed to return is a table which looks as in the following example:

```
return {
  feedback_event = {
    -- The feedback value (1)
    value = "Arbitrary text",
    -- An optional color (2)
    color = { r = 0, g = 255, b = 0 },
    -- An optional background color (3)
    background_color = nil,
  }
}
```

1. In this example it's a text value, but it can be anything!

2. Has the same effect as color in [Feedback style](#)

3. Has the same effect as background color in [Feedback style](#)

- The most important thing here is `value`. It can either be ...
 - ... a string (ideal for display sources)
 - ... a number (ideal for LEDs and motor faders)
 - ... `nil` (which means "turn the source off", e.g. turn off the LED, turn down the motorfader, clear the display text)
 - ... or anything else (`true`, `false` or an arbitrary table ... at the moment, this is only useful for the [MIDI Script](#) source because other sources don't know how to deal with it)

• Examples

- Displays the number of milliseconds passed since ReaLearn was loaded:

```
local millis = context.prop("global.realearn.time")
return {
  feedback_event = {
    value = "" .. millis .. "ms"
  },
}
```

- Creates an animation to make a long FX name visible on a tiny screen.

```
function create_left_right_animation(global_millis, max_char_count, frame_length, text)
  if text == nil then
    return nil
  end
  if #text > max_char_count then
    local frame_count = #text - max_char_count
    local frame_index = math.floor(global_millis / frame_length) % (frame_count * 2)
    local text_offset
    if frame_index < frame_count then
      text_offset = frame_index
    else

```

```

        local distance = frame_index - frame_count
        text_offset = frame_count - distance
    end
    return text:sub(text_offset + 1, text_offset + max_char_count)
else
    return text
end
end
end

-- The maximum number of characters we want to display at once
local max_char_count = 10
-- How many milliseconds to remain in one position
local frame_length = 150
local millis = context.prop("global.realearn.time")
local fx_name = context.prop("target.fx.name")
local animation = create_left_right_animation(millis, 10, frame_length, fx_name)
return {
    feedback_event = {
        value = animation
    },
}

```

- Returns a structured feedback value ...

```

return {
    feedback_event = {
        value = {
            available = context.prop("target.available"),
            more_info = {
                index = context.prop("target.discrete_value"),
                count = context.prop("target.discrete_value_count"),
            },
        },
    },
}

```

- ... which can then be processed by a [MIDI Script](#) source (this example is not realistic, it just shows how you can access the value table returned by the glue section feedback script):

```

return {
    address = 0x4bb0,
    messages = {
        { 0xb0, 0x4b, y.more_info.index, y.more_info.count }
    }
}

```

Feedback style

The ... button provides options to change the *feedback style*. At the moment, it's all about setting colors.

Tip If you use [Dynamic feedback: Lua script](#), changes made here don't have any effect because you are supposed to provide style properties as part of the Lua script result (which is much more flexible).

- **Color / Background color:** With this you can define the color and background color of the displayed text. Of course this will only work if the display source supports it!
 - **<Default color>:** Chooses the default color, that is the one which is preferred for the corresponding controller and display type.
 - **<Pick color...>:** Opens a color picker so you can choose the color of your choice.
 - **Property name:** Maybe you don't want a fixed color but a dynamic one that changes whenever your target changes. Choose one of the following properties to make that happen:

`target.track.color`

Custom color of the first resolved target track (if supported).

`target.bookmark.color`

Custom color of the resolved marker or region.

Only works with the [Marker/region: Go to](#) target.

Source Min/Max

The observed range of absolute source control values. By restricting that range, you basically tell ReaLearn to react only to a sub range of a control element, e.g. only the upper half of a fader or only the lower velocity layer of a key press. In relative mode, this only has an effect on absolute source control values, not on relative ones. This range also determines the minimum and maximum feedback value.

Out-of-range behavior

This determines ReaLearn's behavior if the source value is not within "Source Min/Max" or the target value not within "Target Min/Max". There are these variants:

	Control direction (absolute mode only)	Feedback direction
Min or max	<p>If the source value is $< \text{Source Min}$, ReaLearn will behave as if <i>Source Min</i> was received (or 0% if <i>Source Min = Source Max</i>).</p> <p>If the source value is $> \text{Source Max}$, ReaLearn will behave as if <i>Source Max</i> was received (or 100% if <i>Source Min = Source Max</i>).</p>	<p>If the target value is $< \text{Target Min}$, ReaLearn will behave as if <i>Target Min</i> was detected (or 0% if <i>Target Min = Target Max</i>).</p> <p>If the target value is $> \text{Target Max}$, ReaLearn will behave as if <i>Target Max</i> was detected (or 100% if <i>Target Min = Target Max</i>).</p>
Min	ReaLearn will behave as if <i>Source Min</i> was received (or 0% if <i>Source Min = Source Max</i>).	ReaLearn will behave as if <i>Target Min</i> was detected (or 0% if <i>Target Min = Target Max</i>). Useful for getting radio-button-like feedback.
Ignore	Target value won't be touched.	No feedback will be sent.

Mode ("Absolute mode")

Let's you choose an *absolute mode*, that is, the way incoming absolute source values are handled.

Not all modes make sense at all times! It mostly depends on the character of the source. If a mode doesn't tip make sense given the current source, it will be marked as NOT APPLICABLE. In this case, you should choose another mode or change the source.

Normal

Takes and optionally transforms absolute source control values *the normal way*. *Normal* means that the current target value is irrelevant and the target will just be set to whatever absolute control value is coming in (potentially transformed).

Incremental button

With this you can "go relative" without having encoders, provided your control elements are buttons. Let's assume you use the *MIDI Note velocity* and select *Incremental button* mode. Then it works like this: Each time you press the key, the target value will increase, according to the mode's settings. You can even make the amount of change velocity-sensitive! If you want the target value to decrease, just check the *Reverse* checkbox.

Toggle button

Toggle button mode is used to toggle a target between on and off states. It only makes sense for momentary buttons (which fire a value > 0 on each press).

Here's how it works in detail:

- If the current target value is within the first half of the target min/max range, it's considered as *off* and will therefore be switched *on* (set to *target max*). If it's within the second half, it's considered as *on* and will therefore be switched *off* (set to *target min*).
- It works a bit differently if *target min* and *target max* have the same value (which is a common technique to set the target to a specific value on the press of a button). Instead of toggling between *target min* and *target max*, this mode now toggles between this specific value (= *target min* = *target max*) and 0%. This is useful whenever you have a set of buttons each of which sets the same target to a different value, and you want them to toggle between the specified value and an initial value (0%).

This mode is not supported for controller mappings that have a virtual target.

Sometimes the controller itself provides a toggle mode for buttons. **Don't use it!**

Always set up your controller buttons to work in momentary mode! It's impossible for the controller to know Tip which state (on/off) a target currently has. Therefore, if you use the controller's built-in toggle function, it's quite likely that it gets out of sync with the actual target state at some point.

ReaLearn's own toggle mode has a clear advantage here.

Make relative

This converts incoming absolute fader/knob movements into relative adjustments of the target value. It somewhat resembles takeover mode [Parallel](#) but has important differences:

- It's guaranteed that a full fader/knob swipe from 0% to 100% always results in a swipe over the full target range (assuming the target was at 0% initially).
- It doesn't need to know the current target value. Which means it also works for mappings with [virtual targets](#).

Performance control

This mode emulates the behavior of a typical soft synth modulation matrix mapping: It uses the target value that has been set in REAPER (not via this ReaLearn mapping) as an offset and starts changing it from there.

Takeover mode

If you are not using motorized faders, absolute mode is inherently prone to parameter jumps. A parameter jump occurs if you touch a control element (e.g. fader) whose position in no way reflects the current target value. This can result in audible jumps because the value is changed abruptly instead of continuously. You can deal with this by setting the right takeover mode.

ReaLearn provides multiple takeover modes that decide how to deal with situations when a target parameter jump would occur.

Off

The default settings: Jumps allowed.

Pick up

This is the same as "Soft takeover" in REAPER's built-in MIDI learn. It prevents jumps by not changing the target value until your control element reaches it.

In certain cases, this mode can cause the target value to get stuck. This happens with faders/knobs that cause jumps themselves when moved very rapidly. If you don't like that, you might want to try [Pick up \(tolerant\)](#).

Pick up (tolerant)

This is like [Pick up](#) but makes extra sure that the target value doesn't get stuck.

However, unlike [Pick up](#), this mode will jump if you cause a jump on your controller! Imagine using a touch strip. This kind of control element allows you to jump to arbitrary values at any time. Tolerant mode will not prevent this kind of jumps!

Long time no see

This is similar to [Pick up](#) with the difference that the current target value will gradually "come your way". This results in seamless and fast reunification of control and target value but it can feel weird because the target value can temporarily move in the opposite direction of the fader movement. In older ReaLearn versions this was called "Slowly approach if jump too big".

Parallel

With this mode, the target will simply follow your fader moves, in exactly the same tempo - without any scaling. Reunification only happens when both control and target value meet at the "borders".

Catch up

This mode is sometimes called "Proportional" or "Value scaling" mode. It's like "Parallel" mode but the target value is allowed to move slower than the control value - hence the control can catch up (converge) faster.

Control transformation (EEL)

This feature allows you to write a formula that transforms incoming control values.

While very powerful because it allows for arbitrary transformations (velocity curves, random values - you name it), it's not everybody's cup of tea to write something like that. The formula must be written in the language [EEL2](#). Some REAPER power users might be familiar with it because REAPER's JSFX uses the same language.

Luckily, ReaLearn has a fancy editor which visualizes the formula and has some predefined templates built-in (available on Windows and macOS only at the moment). Press the "... " button to open the editor. Code changes are applied immediately.

The most simple formula is $y = x$, which means there will be no transformation at all. $y = x / 2$ means that incoming control values will be halved. You get the idea: y represents the desired target control value (= output value) and x the incoming source control value (= input value). Both are 64-bit floating point numbers between 0.0 (0%) and 1.0 (100%).

The script can be much more complicated than the mentioned examples and make use of all built-in EEL2 language features. The important thing is to assign the desired value to y at some point.

The following variables/functions are available in the formula:

Variable	Description
y	y initially contains the <i>current</i> target value. You can use that value in order to calculate the new value. With this, you can essentially craft your own relative mode!
	This contains the last value of the target before it was affected by this particular mapping.
y_last	Allows you to come up with a performance control mode typical for synth parameter mappings, just like the built-in Performance control mode but more customizable. Try this for example: $y = y_last + x * (1 - y_last)$

Variable

Description

This contains the number of milliseconds since this mapping has last been triggered with a control message coming from the source.

As soon as you use this and a control message comes in, ReaLearn will start invoking your formula *repeatedly*! That means, this variable is your entrance ticket to smooth transitions and continuous parameter modulation.

A few examples:

- Smooth transition from current value to control value: $\text{rel_time}; y = \text{abs}(x - y) < 0.05 ? \text{stop} : y + 0.1 * (x - y)$
- Sinus LFO: $y = (\sin(\text{rel_time} / 500) + 1) / 2$
- Linear transition to control value (1 second): $y = \text{abs}(x - y) < 0.05 ? \text{stop} : x * \min(\text{rel_time} / 500, 1)$
- 2 seconds chaos: $y = \text{rel_time} < 2000 ? \text{rand}(1) : \text{stop}$
- Setting a value with delay: $y = \text{rel_time} < 2000 ? \text{none} : \text{stop}(0.5)$

rel_time

In combination with `rel_time`, this stops repeated invocation of the formula until the mapping is triggered again.

Good for building transitions with a defined end.

Stopping the invocation at some point is also important if the same parameter should be controlled by other mappings as well. Otherwise, if multiple mappings continuously change the target parameter, only the last one wins.

This also exists as a function, which lets you do both, returning a target value **and** stopping the transition. Pass the desired value in the parentheses, e.g. `stop(0.5)`.

Usually, each repeated (see `rel_time`) invocation always results in a target invocation (unless the target is not retriggerable and already has the desired value). Sometimes this is not desired. In this case, one can return `none`, in which case the target will not be touched.

Good for transitions that are not continuous, especially if other mappings want to control the parameter as well from time to time.

stop and stop(...)

none

ReaLearn's control processing order is like this:

1. Apply source interval
2. Apply transformation
3. Apply reverse

4. Apply target interval
5. Apply rounding

Step size Min/Max

When you deal with relative adjustments of target values in terms of increments/decrements, then you have great flexibility because you can influence the *amount* of those increments/decrements. This is done via the *Step size* setting, which is available for all *continuous* targets.

- *Step size Min* specifies how much to increase/decrease the target value when an increment/decrement is received.
- *Step size Max* is used to limit the effect of acceleration (for rotary encoders which support acceleration and virtual control elements that are mapped as "Incremental button" and have a "Speed" > 1x) and changes in velocity (for velocity-sensitive buttons/keys that are used as "Incremental button"). If you set this to the same value as *Step size Min*, encoder acceleration or changes in velocity will have absolutely no effect on the incrementation/decrementation amount. If you set it to 100%, the effect is maximized.

Speed Min/Max

When you choose a discrete target, the *Step size* label will change into *Speed*. *Discrete* means there's a concrete number of possible values - it's the opposite of *continuous*. If a target is discrete, it cannot have arbitrarily small step sizes. It rather has one predefined atomic step size which never should be exceeded. Allowing arbitrary step size adjustment wouldn't make sense. That's why *Speed* allows you to *multiply* (positive numbers) or *"divide"* (negative numbers) value increments with a factor instead. Negative numbers are most useful for rotary encoders because they will essentially lower their sensitivity. Virtual targets are always discrete.

Example:

- Let's assume you selected the discrete target [FX: Browse presets](#), which is considered discrete because an FX with for example 5 presets has 6 well-defined possible values (including the <no preset> option), there's nothing inbetween. And let's also assume that you have a controller like Midi Fighter Twister whose rotary encoders don't support built-in acceleration. Now you slightly move an encoder clock-wise and your controller sends an increment +1. If the *Speed Min* slider was at 1 (default), this will just navigate to the next preset (+1). If the *Speed Min* slider was at 2, this will jump to the 2nd-next preset (+2). And so on.
- There are FX plug-ins out there which report their parameter as discrete with an insanely small step size (e.g. some Native Instrument plug-ins). This kind of defeats the purpose of discrete parameters and one can argue that those parameters should actually be continuous. In such a case, moving your rotary encoder might need a *lot* of turning even if you set *Speed* to the apparent maximum of 100! In this case you will be happy to know that the text field next to the slider allows you to enter values higher than 100.
- You can set the "Speed" slider to a negative value, e.g. -2. This is the opposite. It means you need to make your encoder send 2 increments in order to move to the next preset. Or -5: You need to make your encoder send 5 increments to move to the next preset. This is like slowing down the encoder movement.

Encoder filter (dropdown)

Allows you to react to clockwise or counter-clockwise encoder movements only, e.g. if you want to invoke one action on clockwise movement and another one on counter-clockwise movement. Or if you want to use different step sizes for different movements.

- **Increment & decrement:** ReaLearn will process both increments and decrements.
- **Increment only:** ReaLearn will ignore decrements.
- **Decrement only:** ReaLearn will ignore increments.

Wrap

If unchecked, the target value will not change anymore if there's an incoming decrement but the target already reached its minimum value. If checked, the target value will jump to its maximum value instead. It works analogously if there's an incoming increment and the target already reached its maximum value.

If this flag is enabled for controller mappings which have a virtual target, every main mapping controlled by that virtual control element will *rotate* - even if the main mapping itself doesn't have *rotate* enabled.

Make absolute

Check this box if you want to emulate an absolute control element with a relative encoder or with -/+ (incremental) buttons.

This is useful if you have configured your controller to be relative all the way (which is good!) but you want to use a control transformation EEL formula - which is not possible if you change the target with relative increments. It works by keeping an internal absolute value, incrementing or decrementing it accordingly and then processing it just like normal absolute control values.

By checking this box:

- You lose the possibility to be perfectly free of parameter jumps (but you can try to mitigate that loss by using the jump settings).
- You gain support for control-direction EEL transformation, non-continuous target value sequences and source range.
- You can still use some of the relative-only features: Step size and rotate!

Fire mode

Normally, when a button gets pressed, it controls the target immediately. However, by using this dropdown and by changing the values below it, you can change this behavior. This dropdown provides different fire modes that decide how exactly ReaLearn should cope with button presses.

Fire on press (or release if > 0 ms)

This mode is essential in order to be able to distinguish between different press durations.

- **Min** and **Max** decide how long a button needs to be pressed to have an effect.
- By default, both min and max will be at 0 ms, which means that the duration doesn't matter and both press (> 0%) and release (0%) will be instantly forwarded. If you change *Min* to e.g. 1000 ms and *Max* to 5000 ms, it will behave as follows:
- If you press the control element and instantly release it, nothing will happen.
- If you press the control element, wait for a maximum of 5 seconds and then release it, the control value of the press (> 0%) will be forwarded.
- It will never forward the control value of a release (0%), so this is probably only useful for targets with trigger character.
- The main use case of this setting is to assign multiple functions to one control element, depending on how long it has been pressed. For this, use settings like the following:
- Short press: 0 ms - 250 ms
- Long press: 250 ms - 5000 ms

Fire after timeout

This mode is more "satisfying" because it will let ReaLearn "fire" immediately once a certain time has passed since the press of the button. However, obviously it doesn't have the concept of a "Maximum" press duration, so it can't be used to execute different things depending on different press durations (or only as the last part in the press duration chain, so to say).

- **Timeout:** Sets the timeout in milliseconds. If this is zero, everything will behave as usual.

Fire after timeout, keep firing (turbo)

Welcome to turbo mode. It will keep hitting your target (always with the initial button press velocity) at a specific rate. Optionally with an initial delay. Epic!

- **Timeout:** This is the initial delay before anything happens. Can be zero, then turbo stage is entered instantly on press.
- **Rate:** This is how frequently the target will be hit once the timeout has passed. In practice it won't happen more frequently than about 30 ms (REAPER's main thread loop frequency).

Fire on double press

This reacts to double presses of a button (analog to double clicks with the mouse).

Fire after single press (if hold < Max ms)

If you want to do something in response to a double press, chances are that you want to do something *else* in response to just a single press. The *Normal* fire mode will fire no matter what! That's why there's an additional *Single press* mode that will not respond to double clicks. Needless to say, the response happens *slightly* delayed - because ReaLearn needs to wait a bit to see if it's going to be a double press or not.

- **Max:** It's even possible to distinguish between single, double *and* long press. In order to do that, you must set the *Max* value of the *Single press* mapping to a value that is lower than the *Timeout* value of your *After timeout* mapping. That way you can use one button for 3 different actions! Example:
 - Mapping 1 "Single press" with Max = 499ms
 - Mapping 2 "Double press"
 - Mapping 3 "After timeout" with Timeout = 500ms

Button filter (right dropdown)

This allows you to easily ignore button presses or releases.

- **Press & release:** ReaLearn will process both button presses (control value = 0%) and button releases (control value > 0%). This is the default.
- **Press only:** Makes ReaLearn ignore the release of the button. The same thing can be achieved by setting *Source Min* to 1. However, doing so would also affect the feedback direction, which is often undesirable because it will mess with the button LED color or on/off state.
- **Release only:** Makes ReaLearn ignore the press of the button (just processing its release). Rare, but possible.

Bottom section

This section has two functions:

- Providing context-sensitive help for the glue section
- Providing target control information and error reporting

Target control information

This shows information about how an incoming control value was handled and possible target control errors.

Note If the target supports MIDI real-time control and the source is a MIDI source, this currently only works if "Log target control" is enabled (see [Logging](#)).

Help

Context-sensitive help for the glue section. Whenever you touch a setting in the glue section, some text will appear which explains what this element does, both for the *control* and for the *feedback* direction (if applicable).

- **If source is a:** It often depends on the kind of source what effect a setting has. Therefore, this dropdown always contains a list of sources. It only displays relevant kinds of sources. If a source kind is impossible according to the current source settings or if it's not supported by the setting, it won't appear in the list.

9.3. Provided REAPER actions

ReaLearn provides some REAPER actions which become available as soon as at least one instance of ReaLearn is loaded. It can be useful to put a ReaLearn instance on REAPER's monitoring FX chain in order to have access to those actions at all times.

In order to find these actions, open REAPER's *Actions* menu, choose *Show action list...* and simply search for *realearn*. The most important actions:

- **ReaLearn: Find first mapping by source:** This action will ask you to touch some control element. As soon as you touch a control element which is mapped, it will open the mapping panel for the corresponding mapping. It will search within all ReaLearn instances loaded in your current project as well as the ones on the monitoring FX chain.
- **ReaLearn: Find first mapping by target:** This action is similar to *Find first mapping by source*. It asks you to touch some (learnable) REAPER parameter. As soon as you touch one that is mapped, it will open its mapping panel.
- **ReaLearn: Learn single mapping (reassigning source):** Asks you to touch a control element and target and adds a new mapping in the first ReaLearn instance that it encounters. It prefers instances in the current project over monitoring FX. It automatically chooses the instance with the correct MIDI/OSC input. If there's an instance which already has that source assigned, it will be reassigned to the new target that you touched.
- **ReaLearn: Learn single mapping (reassigning source) and open it:** Like *Learn single mapping* but additionally opens the mapping panel after having learned the mapping. This is great for subsequent fine tuning.
- **ReaLearn: Learn source for last touched target (reassigning target):** This behaves similar to REAPER's built-in MIDI learn in that it always relates to the target that has been touched last.
- **ReaLearn: Send feedback for all instances:** Makes each ReaLearn instance in all project tabs send feedback for all mappings. That shouldn't be necessary most of the time because ReaLearn usually sends feedback automatically, but there are situations when it might come in handy.

9.4. Advanced settings

This section describes the *Advanced settings* feature of the mapping panel (see section [Mapping](#)) in more detail.

The YAML language

This feature allows you enter text that conforms to the so-called [YAML](#) format. This is not a programming language, so you can't write loops, conditions or anything like that. Instead, think of it as a language for writing configuration. Do you know INI files? REAPER uses INI files to save configuration. YAML is a bit like that, just much more expressive because it allows you to not only express flat key-value pairs (e.g. `edit_fontsize=29`) but also deeply nested configuration data and lists.

Important thing 1: YAML is indentation-sensitive, so indentation matters! The bright side of this is that it always looks clean and nice. The dark side is that ReaLearn will refuse to save your settings if you messed up the indentation. Therefore: Be consistent with your indentation (e.g. use always an indentation of 2 spaces for nesting) and have an utmost attention to detail when doing copy and paste from the examples in this section!

Important thing 2: When you close the text editor and ReaLearn saves your advanced settings as part of the mapping, it will not save the text that you have entered *verbatim*. It will save a structural representation of what you entered (and it will strip comments!). That means if you open the advanced settings again, your could text could look a bit different, in particular it can have a different formatting. But don't worry, it *means* exactly the same to ReaLearn.

Why the hell did you come up with something like that?

Deciding for textual configuration and YAML in particular was a conscious decision with the goal to provide a developer-friendly framework for rapidly extending ReaLearn with advanced features that don't urgently need a graphical user interface.

- **Why ask the user to enter text instead of providing a convenient graphical user interface?**
 - That's mostly a tradeoff due to the fact that my time available for developing ReaLearn is limited.
 - It's much work to develop a graphical user interface for every feature. In fact, programming the user interface often takes most of the time whereas implementing the actual logic is not that much effort.
 - It's true that some sorts of functionality really benefit from having a fancy graphical user interface. But there's also functionality for which having it is not too important, e.g. functionality that is of configurational nature and not used that often.
 - Also, one of ReaLearn's goals is to give power users and programmers extra powers. Textual configuration can be more powerful in many situations once the user knows how to go about it.

- **Why YAML?**

- YAML has the advantage of being popular among programmers, widely supported, highly structured and relatively well readable/writable by both humans and machines.
- Many text editors offer built-in support for editing YAML.
- Makes it easy to provide data for even very complex features.

- **Why not a scripting language?**

- Using a scripting language would spoil any future possibility to add a graphical user interface on top of some of the functionality.
- It wouldn't allow ReaLearn to apply future optimizations and improvements. ReaLearn is rather declarative in nature and a scripting language would destroy this quality.
- It's hard to come up with a stable and good API.
- It's harder to use than a configuration language.

- **Why don't you save the text, just the structure?**

- Mostly because saving just the structure makes the entered data become a natural part of ReaLearn's main preset format (JSON).
- However, this is something that might change in future, depending on how it proves itself in practice.
- Once we would start saving the actual text, it would be hard to go back.

Supported configuration properties

In this section you will find examples that cover all currently supported configuration properties. You can copy and paste the stuff you need to the text editor, remove the parts that you don't need and adjust the rest. Comments (lines starting with #) will be removed automatically.

Mapping lifecycle actions

ReaLearn allows you to define MIDI messages to be sent to the output whenever a mapping turns active or inactive.

Example use cases:

- Accessing very device-specific features via system-exclusive MIDI messages.
- Choosing a different LED color/style depending on the active mapping.
- Initializing a sys-ex-controllable display with some mapping-specific text (more difficult).

A mapping can change its active/inactive state based on the following factors:

- **Preset loading/unloading:** A mapping can turn active when a ReaLearn instance or preset is loaded and turn inactive when it's changed or unloaded.
- **Conditional activation:** A mapping can turn inactive when an activation condition is not satisfied anymore and can change back to active as soon as it's satisfied again.
- **Target condition:** Targets can also have conditions (e.g. "Track must be selected"). They affect activation state changes in the same way.
- **Target validity:** A mapping can turn inactive when the target is not valid anymore, e.g. when it's a target that's based on the currently selected track but no track is currently selected. Analogously, it can turn active again once a valid target can be resolved.
- **Feedback enabled checkbox (or mapping-enabled checkbox):** A mapping can turn inactive as soon as this checkbox is unticked and turn active again when ticking it. This is also the best way to test your configuration.

(Controller) mappings with virtual targets are always considered active as long as the feedback checkbox is ticked. That's why they are perfectly suited for holding a bunch of controller initialization messages! This feature is for example used in the "PreSonus FaderPort Classic" controller preset, which needs to be put in a specific mode before being usable. ReaLearn does this automatically simply by sending some mapping on-activate MIDI messages.

These are the available configuration properties:

```
# Contains stuff to be done whenever this mapping becomes active.
on_activate:
  # A list of MIDI messages to be sent to the output when this mapping becomes active.
  #
  # At the moment, only messages of type "raw" are supported. Although this covers all possible types
  # of MIDI messages, it's a bit hard to express e.g. simple NOTE ON or CC messages with this notation.
  # In particular, you would need to know how MIDI messages are presented as byte sequences. Future ReaLearn
  # versions will provide more convenient ways to describe simple MIDI messages.
  send_midi_feedback:
    # This is an example of a system-exclusive message ("SysEx"). It's usually expressed in hexadecimal string
    # notation. Make sure to include the leading F0 and trailing F7, which is the begin and end marker of all
    # system-exclusive messages!
    - raw: F0 00 20 6B 7F 42 02 00 10 77 01 F7
    # Instead of above hexadecimal string notation, you could also use an array of decimal numbers to describe a raw
    # message. The following is a NOTE ON of note 74 on channel 1 with velocity 100.
    - raw:
      # NOTE ON on channel 1
      - 144
      # Note number 74
      - 74
      # Note velocity 100
      - 100

# Contains stuff to be done whenever this mapping becomes inactive.
on_deactivate:
  # A list of MIDI messages to be sent to the output when this mapping becomes inactive.
  send_midi_feedback:
    # Supports exactly the same kinds of messages as described above in "on_activate".
    - raw: F0 00 20 6B 7F 42 02 00 10 77 14 F7
```

Please remember that YAML comments (e.g. # The following line does this and that) *will not be saved!* In case you want to explain something, you need to write it as YAML property, such as in the following example:

```
comment: "The following configuration makes the rightmost pad of the MiniLab mkII light up in red color."
on_activate:
  send_midi_feedback:
    - raw: F0 00 20 6B 7F 42 02 00 10 77 01 F7
```

ReaLearn will ignore any unknown properties.

Tip If you use input <FX input> and find that MIDI lifecycle messages aren't sent, no matter what, make sure "Send feedback only if track armed" is disabled (see [Options](#))!

Disabling the complete ReaLearn instance will cause all mappings to deactivate. However, sending MIDI messages on deactivation in this case will only work if the output is a device! If it is <FX output>, it

Warning will not send anything because REAPER will not give that ReaLearn instance any chance to output MIDI messages once it's disabled. Instead, the MIDI message will queue up and be sent once you enable that instance again ... which is probably not what you want.

9.5. Configuration files

ReaLearn creates and/or reads a few files in REAPER's resource directory.

File	Description
Data/helgoboss	Directory which contains data such as presets or resources that need to be distributed via ReaPack
Data/helgoboss/auto-load-configs/fx.json	Contains global FX-to-preset links, see Auto-load
Data/helgoboss/archives	Directory which contains archives e.g. the compressed app, distributed via ReaPack
Data/helgoboss/doc	Contains offline documentation, e.g. this guide as PDF
Data/helgoboss/presets/controller	Contains preset for the controller compartment

File	Description
Data/helgoboss/presets/main	Contains preset for the main compartment
Helgoboss	Directory which contains rather personal or device-specific data, not touched via ReaPack
licensing.json	Contains license keys
Helgoboss/App	Contains the uncompressed App, if installed
Helgoboss/Pot/previews	Directory which contains previews recorded by Pot Browser
Helgoboss/ReaLearn/osc.json	Global OSC device configurations, see OSC devices
Helgoboss/ReaLearn/realearn.ini	<p>Very basic global configuration, currently mainly regarding ReaLearn's built-in server.</p> <p>Currently supported properties (subject to change): server_enabled, server_http_port, server_https_port, server_grpc_port, companion_web_app_url</p>
Helgoboss/Server/certificates	Contains a list of certificates and corresponding private keys in order to allow encrypted communication with ReaLearn Companion and App.