CREATOR TOOLS

A suite of tools developed to support
the instrument creation process

Table of Contents

DISCLAIMER

Document authored by: Elpiniki Pappa, Holger Zwar

Software version: 1.2.0 (11/2019)

Special thanks to the Beta Test Team, who are invaluable not just in tracking down bugs, but in making Creator Tools a better product.

DOCUMENT CONVENTIONS

This document uses particular formatting to point out special facts and to warn you of potential issues. The icons introducing the following notes let you see what kind of information can be expected:

The speech bubble icon indicates a useful tip that may help you to solve a task more efficiently.

The exclamation mark icon highlights important information that is essential for the given context.

The warning icon warns you of serious issues and potential risks that require your full attention.

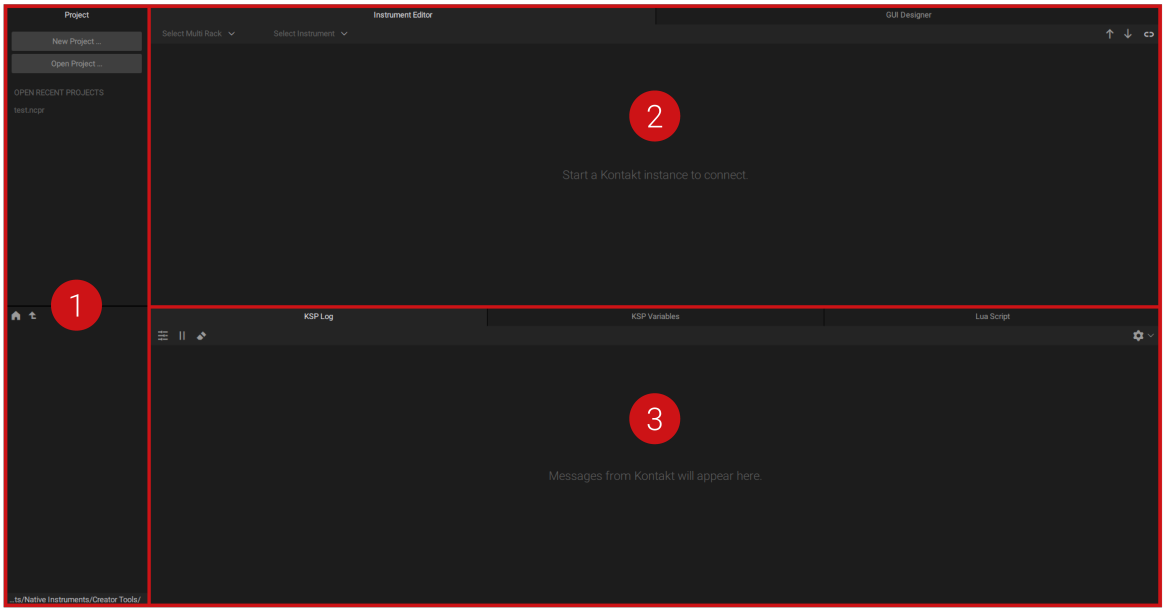
Furthermore, the following formatting is used:

- Paths to locations on your hard disk or other storage devices are printed in *italics*.
- Important names and concepts are printed in **bold**.
- Square brackets are used to reference keys on a computer's keyboard, e.g., Press [Shift] + [Enter].

1. CREATOR TOOLS

A suite of tools developed to support the creation of KONTAKT instruments.

Creator Tools consists of three panels: the **Project Panel (1)**, the **Top Panel (2)** and the **Bottom Panel (3)**. Each panel contains different tools and can be shown or hidden from the View menu of the application top menu or by using dedicated shortcuts. Switching between the tools is possible from the tabs at the top of each panel, from the Go menu of the application top menu or by using the dedicated shortcuts. These shortcuts trigger actions related to the active panel in Creator Tools.



1.1. Project Panel [⌘/Ctrl-1]

The Project Panel consists of the **Project Manager** (located at the top of the panel) and the **File Browser** (located at the bottom of the panel). It is a tool for browsing the filesystem and a project's resources and it allows files to be opened directly from Creator Tools by double-clicking on them.

1.1.1. Project Manager [F1]

A Creator Tools project file contains all data relevant for the instrument creation process. The format of a Creator Tools project file is JSON, which allows you to use source code version control systems.

The Project Manager is an editable view of a Creator Tools project file and therefore can only be used when a Creator Tools project file is open. It serves as a central access point for a project's resources and can, although not necessarily, contain all used resources.

The following items are shown in the Project Manager:

- File and Folder items in a project file reference the corresponding files and folders in the filesystem. If the referenced entity is located in the project file folder or any of its children, the ref-

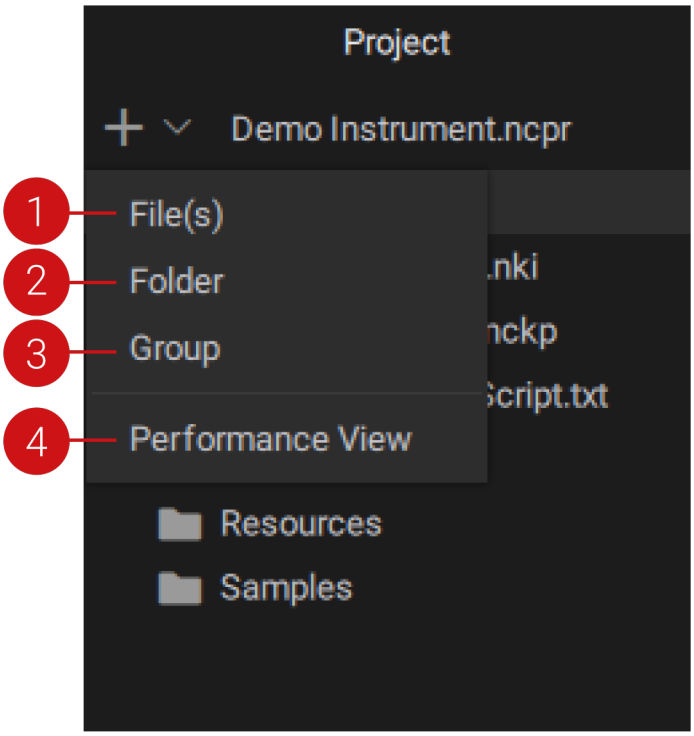
erence is encoded relative to the project file location. Otherwise, the absolute reference is used.

- Group items in a project file organize a project as a hierarchical structure. A group can contain another group, a file or a folder.

There are no rules enforced about a project file's location or structure. There's not necessarily a project folder or any form of enforced project folder structure. Multiple project files can exist side-by-side.

It is recommended to have a project file residing in a project folder together with sub-folders for common resources. When moving a project to other devices or platforms, it is important that the project is self-contained, i.e., contains only relative references.

Insert Menu [⌘/Ctrl-↵]



(**1**) **Files:** Opens the OS file selection dialog. For each selected file an entry is inserted.

(**2**) **Folder:** Opens the OS file selection dialog and inserts the selected folder. Once a folder is selected, its contents are displayed in the File Browser area below.

(**3**) **Group:** Inserts a new group. Group items can contain files, folders and groups, and can be renamed.

(**4**) **Performance View:** Opens the OS Save As dialog, which sets the saving location of the new performance view file. The new performance view item is added to the project.

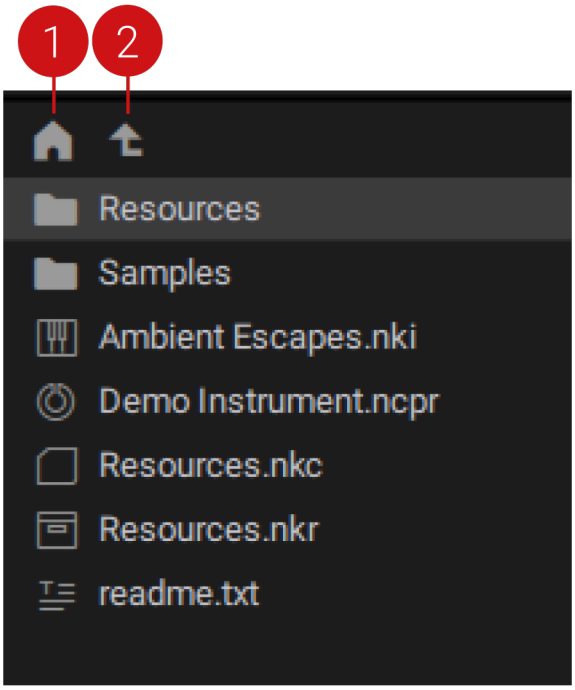
For all the items described above, if a group is selected and expanded in the Project Manager, the inserted item(s) will be placed into this group. In any other case, the inserted item(s) will be placed above the currently selected item.

Shortcuts

Up/down arrows [↑/↓]	Navigate up/down
Left/right arrows [←/→]	Expand/collapse group
Return [↵]	Rename group if the current item is a group
	Open the system file-browser if the current item is a folder
	Open the file within Creator Tools if the current item is an associated file e.g. performance view
	Open the file with the system associated app e.g. a text editor
Double-click	Rename groups if the current item is a group
	Open the file within Creator Tools if the current item is an associated file e.g. performance view
	Open the file with the system associated app e.g. a text editor

1.1.2. File Browser

The File Browser is a secondary area in the Project Panel, located below the Project Manager. It is a read-only view of the filesystem and allows navigation through files and folders. It displays a flat list of files and/or folders within the current root.



(**1**) **Home Button:** Navigates to the folder where the project file is located.

(**2**) **Parent Button:** Navigates to the parent folder of the currently displayed folder.

Shortcuts

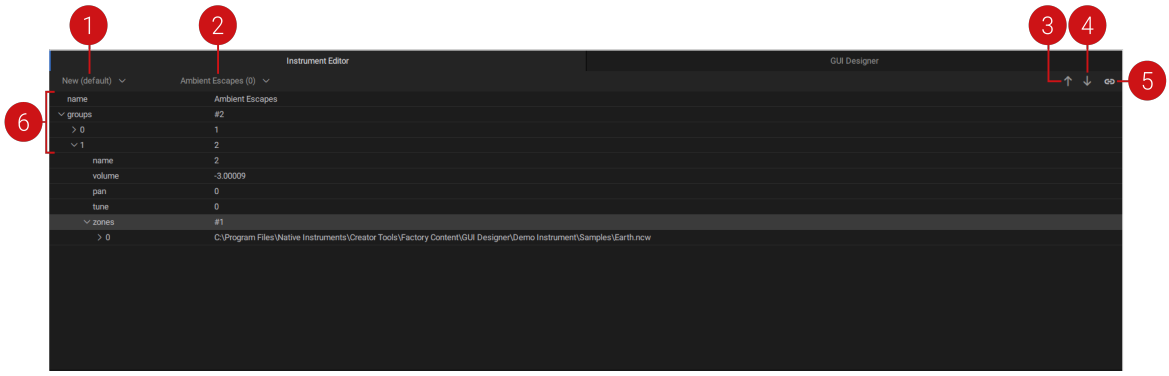
Up/down arrows [↑/↓]	Navigate up or down
Cmd/Ctrl-up [⌘↑]	Navigate to parent folder
Cmd/Ctrl-down [⌘↓]	Enter folder
Return [↵]	Open the system file-browser if the current item is a folder Open the file within Creator Tools if the current item is an associated file e.g. performance view Open the file with the system associated app e.g. a text editor
Double-click	Enter folder Open the file within Creator Tools if the current item is an associated file e.g. performance view Open the file with the system associated app e.g. a text editor

1.2. Top Panel [⌘/Ctrl-2]

The Top Panel consists of the **Instrument Editor** and the **GUI Designer**.

1.2.1. Instrument Editor [F2]

The Instrument Editor connects to a running instance of Kontakt (either plug-in or standalone) and displays the structure of an instrument in the form of a nested tree. Combined with the **Lua Script** tool in the **Bottom Panel**, it offers programmatic access to parts of a Kontakt instrument's structure through Lua-based scripting.



(**1**) **Multi Rack Menu:** Sets the focus of the tool on the Multi Rack of one of the connected Kontakt instances.

(**2**) **Instrument Menu:** Sets the focus of the tool to a specific instrument from the selected Multi Rack (**1**) that is loaded in one of the connected Kontakt instances.

Note that instruments with locked edit views cannot be selected.

(**3**) **Push:** [⌘/Ctrl-Alt-↑] Applies all changes from the Tools' side to Kontakt. If changes are not pushed, an indication on the button appears to notify for the pending changes.

(**4**) **Pull:** [⌘/Ctrl-Alt-↓] Overwrites the current Kontakt state to the tools. Whenever a change takes place on the Kontakt side, Pull needs to be manually pressed in order to apply the changes in the Tools. If changes are not pulled, an indication on the button appears to notify for the pending changes.

(**5**) **Connection Indicator:** Indicates whether a successful connection between the tools and the Kontakt instances is established.

(**6**) **Instrument Tree View:** The instrument structure is displayed in the form of a nested tree. The tree view shows the basic instrument structure and instrument properties that can be modified through Lua scripts.

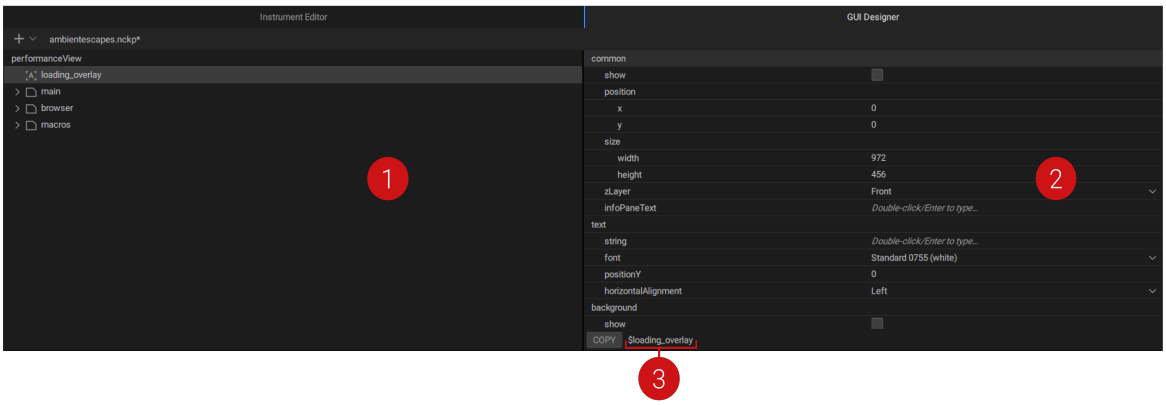
1.2.2. GUI Designer [F3]

The GUI Designer allows one to assemble, customize and reuse Kontakt performance views and controls, without the need to write code. It can generate two types of files, the performance view files (.nckp) and the control files (.nckc).

The performance view files (.nckp) contain all the information about an instrument's graphical interface. These files can then be loaded in a KSP script (see also Loading in KSP). A new performance view file can be created from the Insert menu of the Project Manager.

The control files (.nckc) are files that are created by exporting a single control or a container of controls (see also Panels). These files can then be imported in a later GUI Designer project, shared with collaborators or set the foundation for building custom UI libraries. Control files cannot be loaded in KSP.

The two main areas of the tool are the **Tree View (1)** and the **Properties (2)**.



(**1**) **Tree View:** The structure of a Kontakt performance view is displayed here in the form of a tree. A new performance view has one hierarchy level; the root level. Additional levels can be created when controls are added in Panels (see Panels).

Actions on one or more selected controls can be performed from the Tree View's context menu. The context menu actions are:

Cut [⌘/Ctrl-X]	Copies selection to the clipboard and deletes it from the tree
Copy [⌘/Ctrl-C]	Copies selection to the clipboard
Paste [⌘/Ctrl-V]	Pastes controls from the clipboard above selection
Duplicate [⌘/Ctrl-D]	Duplicates selection

Rename [↵]	Enters renaming mode for selection
Delete [⌘/Ctrl-⌫]	Deletes selection
Import [⌘/Ctrl-I]	Opens the system's file browser in order to locate and import a control file (.nckc) from the disk. The imported control will be placed above the currently selected control
Export [⌘/Ctrl-E]	Opens the system's file browser in order to save the selected control's file (.nckc) in a desired location

(**2**) **Properties:** Displays the properties of the element selected in the Tree View. Changes can be made by double clicking on a specific property. Alternatively, the arrow buttons can be used to navigate the properties and changes can be made via hitting enter. Undo and redo actions are available via the usual shortcuts: [⌘/Ctrl-Z] for Undo and [⌘-Shift-Z/Ctrl-Y] for Redo.

- **Image:** As input, image fields take the filename of a picture (.png) that is contained in the pictures subfolder of the KONTAKT Instrument' s resources folder.
- **Fonts:** From the dropdown menu, one can select to set 1 of the 25 factory fonts or a custom font. If "Custom" is selected, then the filename of the picture font is expected. Similar to the Image property, the picture font should be contained in the pictures subfolder of the KONTAKT Instrument's resources folder.

(**3**) **Variable name:** The KSP variable name of the currently selected control is displayed here. Use this name when scripting in KONTAKT. You can use the **Copy** button next to it to save the variable name to the clipboard.

Adding a Control

Insert Control Menu [⌘/Ctrl-↵]

A new control can be inserted in the performance view tree from the Insert Control menu. The menu lists all the known Kontakt UI controls, including a new control called panels (see Panels).

Import [⌘/Ctrl-I]

Previously exported controls can be added in the tree via the context menu's Import function. Select a control and right click to reveal the context menu. Click on Import and locate the control's .nckc file in the system's file browser. Select Open and the control will be added in the tree, on top of the currently selected control.

Panels

A panel is a control that can contain one or multiple controls. Unlike the other controls, panels don't have a size. They are very useful for grouping controls that are meant to be handled together. This allows one to simultaneously modify the Show, Position or zLayer property of all the controls contained in that panel. The position of a contained control is relative to the panel's position, meaning that the control's (0,0) position is the current (x,y) position of the panel.

Panels can be nested, so they can contain other panels. If panelA is contained in panelB, then panelA will appear in front of panelB. This is because children panels have a higher zLayer value than their parent panels. Use this logic to easily create hierarchies in a performance view.

Panels can also be used to keep the Tree View organized. They can be expanded or collapsed. When a panel is selected and expanded, new controls will be added on top of the panel's contained controls. When a panel is selected but collapsed, new controls will be added above it, on the same hierarchy level as the panel.

Panels in KSP

Panels, like any other control, can also be used with pure KSP outside the GUI Designer, using the following command and control parameter:

```
declare ui_panel $<my_panel_name>
```

Creates a panel

```
set_control_par(<control-to-add-ui-ID>,$CONTROL_PAR_PARENT_PANEL,<panel-ui-ID>)
```

Adds a UI control (or panel) in a panel

Example

```
declare ui_panel $mixer
declare ui_knob $volume (0, 300, 1)
set_control_par(get_ui_id($volume), $CONTROL_PAR_PARENT_PANEL,
get_ui_id($mixer))
```

Adding a volume knob in a mixer panel

1.2.3. Loading in KSP

The Resource Container is a dedicated location to store scripts, graphics, .nka files and impulse response files that can be referenced by any NKI or group of NKIs linked to the container.

When creating the Resource Container, Kontakt versions that are compatible with the GUI Designer will create a new subfolder named performance_view.

In order for a performance view to be displayed in an NKI, the performance view file (.nckp) must be stored in the *performance_view* subfolder of the NKI. It can then be loaded in the NKI via the KSP command:

```
load_performance_view("filename")
```

Where "filename" is the filename of the .nckp file without the extension. Performance view filenames can only contain letters, numbers or underscores.

When saving a .nckp file in the GUI Designer, any changes will be automatically applied to the Kontakt side. This means when editing the performance view of an NKI, if that NKI is loaded in a running Kontakt instance, all changes can be previewed in real time upon Save.

To avoid conflicts, only one performance view file can be loaded per script slot. If needed, more controls can be additionally declared in the KSP script.

Connecting UI controls to engine parameters still occurs via KSP. The variable name of a control contained in a performance view is auto generated based on its hierarchy, with underscore characters as concatenation.

Example

Control `knobVolume` is contained within panel `eqTab`, which is also contained within panel `mixerTab`. The KSP variable name of the control will be: `mixerTab_eqTab_knobVolume`.

The KSP variable name of a selected control is displayed at the bottom of the properties area. Click on the Copy button next to it (or use the shortcut [⌘/Ctrl-Alt-C]) to copy the variable name to clipboard.

1.2.4. Creating and previewing your first Performance View

To create and preview a performance view:

1. Create a new NKI in Kontakt.
2. Create the Resource Container for the NKI.
3. Create a new performance view in Project Manager and double-click it to open it in the GUI Designer. Start adding controls.
4. Make sure all the referenced data (images, picture fonts) are stored in the images subfolder of the NKI's Resource Container.
5. Save the performance view file in the performance_view subfolder of the NKI's Resource Container.
6. In the NKI script editor, write and apply the following script:

```
on init
load_performance_view ("filename")
end on
```

You can now continue to edit the performance view in the GUI Designer. Each time you want to preview any changes in the performance view, save the Creator Tools project.

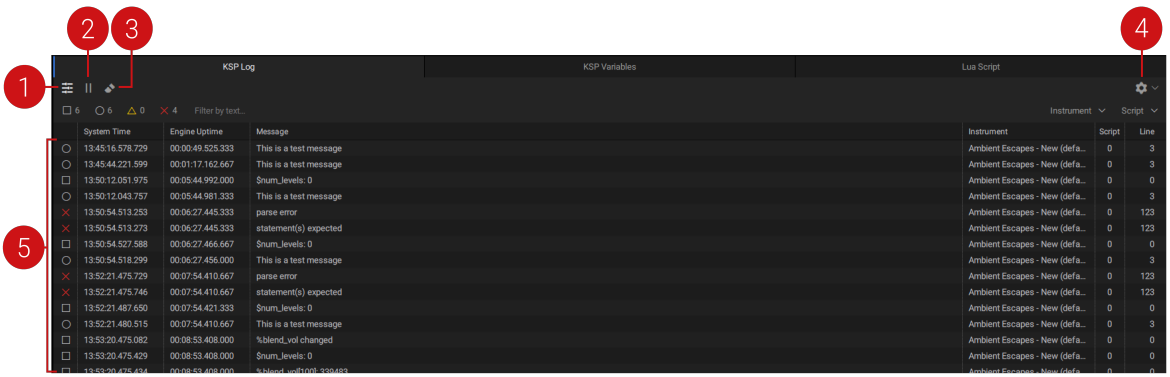
1.3. Bottom Panel [⌘/Ctrl-3]

The Bottom Panel consists of the KSP Log, KSP Variables and Lua Script.

1.3.1. KSP Log [⇧F1]

The KSP log connects to all running instances of Kontakt, both plug-in and standalone.

It logs messages, warnings and errors coming from KSP, supports inspecting script variables, provides timestamps per notification and some basic filtering options.



(**1**) **Filter:** [⌘/Ctrl-F] When active, it reveals the filtering options and applies them.

- Filter by type (Variable Watching, Message, Warning, Error)
- Filter by text (characters in the Message column)
- Filter by Instrument
- Filter by Script slot

(**2**) **Pause:** [⌘/Ctrl-P] Suspends the debugging session. When active, the Pause button blinks. Once the session is resumed, all messages that were received during pause will appear.

(**3**) **Clear:** [⌘/Ctrl-Backspace] Clears all content of the log.

(**4**) **Settings:** Defines the behavior of the log.

(**5**) **Log:** All notifications from Kontakt appear in the Log area. The Log contains seven columns:

- Type
- System Time
- Engine Time
- Message
- Instrument
- Script
- Line

Type and Message are set, but all other columns can be hidden. Right-click on the column header to reveal the column menu.

1.3.2. KSP Variables [⇧F2]

This is where the current values of all watched variables and arrays are displayed, in order of appearance. For every variable or array that is inspected, an entry is created upon initialization and updated every time a value change occurs. All value changes appear also in the KSP Log, in chronological order.

Inspecting a variable or array is possible via the dedicated KSP commands `watch_var` and `watch_array_idx`.

For example `watch_var($count)` inspects the value changes of the variable `count` and `watch_array_idx(%volume,5)` inspects the value changes of index 5 of the array `volume`.

Please also refer to the KSP Reference Manual for more details.

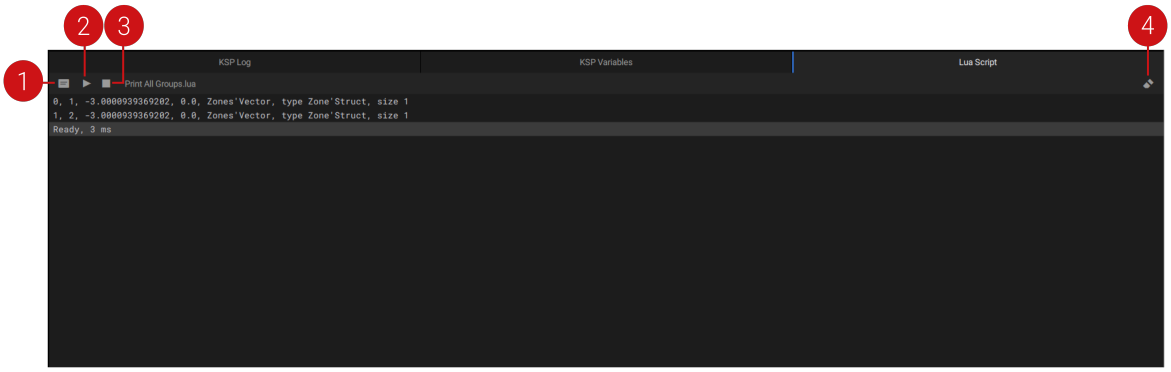
1.3.3. Lua Script [⇧F3]

Changes to an instrument's structure from within the Tools happen exclusively by running a Lua script. A script can see and modify the instrument copy in the Tools. All parameters that can be modified are displayed in the Instrument Editor tool. Scripts can be created and modified with an external editor.

The Lua Script tool loads and runs Lua scripts that have been created in a text editor and saved to disk. In this way an instrument structure can be modified. One can now easily rearrange, add or remove groups and zones, edit their names and some of their properties, like tune, volume, and mapping. Limited file system access also allows the creation of new instruments based on samples on the disk. The added MIR functions (like pitch and RMS detection) assist or automate parts of the instrument creation process.

Some Lua example and tutorial scripts are provided for the above in the application folder, to help you get started. Ideally, the content of the scripts' folder can be copied to "*user/Documents/Native Instruments/Creator Tools*".

The script output will appear in the console output. All console output can be copied to the system clipboard via the command [⌘/Ctrl-Alt-C].



(**1**) **Open in text editor:** [⌘/Ctrl-E] Opens the loaded script file in the system's default editor.

(**2**) **Run:** [⌘/Ctrl-R] Executes the loaded .lua script. Changes are immediately reflected in the Instrument Editor tool.

(**3**) **Stop:** [⌘/Ctrl-I] Stops the execution of the running script. The Instrument Editor state is reverted, as if the script never run.

(**4**) **Clear** [⌘/Ctrl-Backspace] Clears all content of the tool's output console.

Loading a script

A script can be loaded by double-clicking it in the **Project Manager** or **File Browser**, or by dragging it directly from any disk location. The filename of the loaded file will then appear in the filename area.

Currently the Creator Tools Lua runtime on Windows does not support filepaths that contain Unicode characters. Please rename the script's filepath accordingly to successfully load it.

2. SCRIPTING REFERENCE

Lua scripts can be loaded and run in the Instrument Editor tool to assist or automate tasks in the instrument creation process. This section of the documentation contains the scripting basics of the Lua language as well as extension bindings to a Kontakt instrument's structure.

2.1. Instrument Structure

An instrument is shown as a nested tree with properties and values. Containers like groups and zones are represented as vectors (lists with indices). Property values are typed and value-checked so that changes are verified and ignored if the data is invalid.

The structure with property names, types and value-ranges looks like this:

```
Instrument                                  -- Struct
    name                                    -- String
    groups                                  -- Vector of Group:
        name                                -- String
        volume                              -- Real, -inf..12
        pan                                 -- Real, -100..100
        tune                                -- Real, -36..36
        zones                               -- Vector of Zone
            file                            -- String
            volume                          -- Real, -inf..12
            pan                             -- Real, -100..100
            tune                            -- Real, -36..36
            rootKey                         -- Int, 0..127
            keyRange                        -- Struct
                low                         -- Int, 0..127
                high                        -- Int, 0..127
            velocityRange                   -- Struct
                low                         -- Int, 0..127
                high                        -- Int, 0..127
            sampleStart                     -- Int, 0..inf
            sampleStartModRange             -- Int, 0..inf
            sampleEnd                       -- Int, 4..inf
            loops                           -- Vector of Loop
                mode                        -- Int, 0..4 (see below)
                start                       -- Int, 0..inf
                length                      -- Int, 4..inf
                xfade                       -- Int, 0..1000000
                count                       -- Int, 0..1000000
                tune                        -- Real, -12..12
```

Loop modes:

- 0: Oneshot i.e. off

- 1: Until end
- 2: Until end alternating
- 3: Until release
- 4: Until release alternating

2.2. Scripting Basics

Scripting is based on the Lua language. Resources are available online e.g. www.lua.org. The core language has been extended by bindings to the instrument structure. Whenever an instrument is connected and the tree view is displayed, a script can access it via the variable instrument.

2.2.1. Script Path

The global variable `scriptPath` points to the directory of the executed script. This is useful for file I/O related workflows.

2.2.2. Read properties and print them

A script can print to the console e.g.

```
print(instrument)
```

Prints "Instrument" if an instrument is connected, otherwise "nil" i.e. nothing.

```
print(scriptPath)
```

Prints the directory of the running script.

All properties can be referenced using dots e.g.

```
print(instrument.groups[0].name)
```

Prints the name of the first group - or an error message if no instrument is connected.

```
print(instrument.groups[0].zones[0].keyRange.high)
```

Prints the highest key range value of the first zone of the first group.

2.2.3. Iterate over containers

The brackets [ ] refer to the n^{th} element of the group or zone vector. The number of elements can be read with Lua‘s length operator:

```
print(#instrument.groups)
```

This allows iterating through groups or zones:

```
for n=0,#instrument.groups-1 do
    print(instrument.groups[n].name)
end
```

Note that vectors are zero-indexed! There are other ways to iterate over containers without using indices. In the Binding Reference chapter, iteration via pairs is described for Vector and Struct and iteration via the traverse function is described under Algorithms.

2.2.4. Changing properties

Changing values works naturally:

```
instrument.groups[0].name = "Release"
```

or in a loop:

```
for n=0,#instrument.groups-1 do
    instrument.groups[n].name = 'grp_'..n
end
```

2.2.5. Working with containers

```
Group()
```

Creates a new object of type Group.

```
print(scriptPath)
```

Prints the directory of the running script.

Structural changes like add, remove, insert are possible:

```
instrument.groups:add(Group())
```

Adds a new empty group at the end.

```
instrument.groups:insert(0, instrument.groups[3])
```

Inserts a deep copy of the 4th group at index 0 i.e. at the beginning.

3. BINDING REFERENCE

3.1. Type

Base type of all object types. The following accessors are defined for objects off all derived types:

Operators	
`tostring`	Returns a string representation describing the object

Properties	
`object.typeinfo`	Returns type information as a string in the form Type'Tag
`object.parent`	Returns the parent object or nil

Functions	
`object:equals(other)`	Returns true if object value is equal to the value of other
`object:instanceOf(type)`	Returns true if object is an instance of type i.e. `object.type == type`
`object:instanceOf(name)`	Returns true if object is an instance of a type named name i.e. `object.type.name = name`
`object:childOf(other)`	Returns true if object is a direct child of other i.e. `object.parent == other`
`object:childOf(type)`	Returns true if object is a direct child of an object of type i.e. `object.parent` and `object.parent.type == type`
`object:childOf(name)`	Returns true if object is a direct child of an object of a type named name i.e. `object.parent` and `object.parent.type.name == name`

3.2. Scalars

Basic types which contain a single value:

Bool	Boolean, true or false
Int	64 bit signed integer (can be negative)
Real	64 bit floating point number
String	Text

For a scalar object of these types the following accessors are defined:

Properties	
`scalar.type`	Returns the value type
`scalar.initial`	Returns true if the value is in the initial state
`scalar.value`	Returns the value

Functions	
`scalar:reset()`	Resets the value to its initial state
`scalar:assign(other)`	Assigns a copy of the other value object

3.3. Vector

Type-safe, dynamically sized, zero-indexed, random access container.

For a vector object the following accessors are defined:

Constructors	
`vector()`	Returns new vector
`vector(other)`	Returns a copy of the other vector
`vector(size)`	Returns a new vector with size elements
`vector(args)`	Returns a new vector initialized with variadic args

Operators	
`#`	Returns the number of elements i.e. the size of the vector
`pairs`	Returns an iterator function for iterating over all elements
`value/object = vector[index]`	Returns the value if vector type is scalar, otherwise returns the object
`vector[index] = value`	Sets value at index
`vector[index] = object`	Assigns object to index

Properties	
`vector.type`	Returns the vector type
`vector.empty`	Returns true if the vector has no elements
`vector.initial`	Returns true if the vector is in its initial state

Functions	
`vector:set(index, object)`	Sets element at index to object
`vector:get(index)`	Returns object at index
`vector:reset()`	Resets the vector to initial
`vector:resize(size)`	Resizes the vector to size elements
`vector:resolve(path)`	Returns the object at path or nil
`vector:assign(other)`	Assigns a copy of the other vector
`vector:add(object)`	Inserts object at the end
`vector:add(value)`	Inserts value at the end
`vector:insert(index, object)`	Inserts object before index
`vector:insert(index, value)`	Inserts value before index
`vector:remove(index)`	Removes element at index

3.4. Struct

Type-safe record with named fields.

For a struct object the following accessors are defined:

Constructors	
`struct()`	Returns a new struct
`struct(other)`	Returns a copy of the other struct

Operators	
`#`	Returns the number of used fields
`pairs`	Returns an iterator function for iterating over used fields
`value/object = struct.field`	Returns the value if field contains a scalar, otherwise returns the object
`struct.field = value`	Sets element value of field
`struct.field = object`	Assigns object to field

Properties	
`Struct.type`	Returns the struct type
`struct.empty`	Returns true if the struct has no used fields
`struct.initial`	Returns true if the struct is in its initial state

Functions	
`struct:set(index, object)`	Assigns object at index
`struct:get(index)`	Returns object at index
`struct:reset()`	Resets the struct to its initial state
`struct:reset(index)`	Resets the field at index
`struct:reset(field)`	Resets the field
`struct:used(field)`	Returns true if the field is used
`struct:resolve(path)`	Returns the object at path
`struct:assign(other)`	Assigns a copy of the other struct

3.5. Algorithms

The following are free functions operating on any **Type**.

`path(object)`	Returns the path to object
`resolve(path)`	Returns the object at path or nil
`traverse(object, function(key, object, [level]))`	Recursively traverses object and calls function where key is the index or field name of the object in parent
`string = json(object, [indent])`	Converts objects to a json string and returns it
`object = json(type, string)`	Converts the string to an object of type and returns it

3.6. File system

The Lua binding is based on the C++ library boost filesystem. In contrary to the original C++ design, the Lua binding does not define an abstraction for path. Instead, path always refers to a Lua string.

The following chapter lists functions from the aforementioned library and the associated data type they return. For a detailed description of each function, please refer to the reference documentation.

Examples

```
for _,p in filesystem.directory(path) do
    print(p)
end
```

Lists paths in directory

```
for _,p in filesystem.directoryRecursive(path) do
    print(p)
end
```

Lists paths in directory and all sub-directories

Functions

Note that all functions live in the global filesystem.

Iterators	
Function	**Returned data type**
`filesystem.directory(path)`	`iterator`
`filesystem.directoryRecursive(path)`	`iterator`

Path	
These functions return a string which contains the modified path.	
`filesystem.native(path)`	`string`
`filesystem.rootName(path)`	`string`
`filesystem.rootDirectory(path)`	`string`
`filesystem.rootPath(path)`	`string`
`filesystem.relativePath(path)`	`string`
`filesystem.parentPath(path)`	`string`
`filesystem.filename(path)`	`string`
`filesystem.stem(path)`	`string`
`filesystem.replaceExtension(path, newExtension)`	`string`
`filesystem.extension(path)`	`string`
`filesystem.preferred(path)`	`string`

Query	
These functions query a given path.	
`filesystem.empty(path)`	`bool`
`filesystem.isDot(path)`	`bool`
`filesystem.isDotDot(path)`	`bool`
`filesystem.hasRootPath(path)`	`bool`
`filesystem.hasRootName(path)`	`bool`
`filesystem.hasRootDirectory(path)`	`bool`
`filesystem.hasRelativePath(path)`	`bool`
`filesystem.hasParentPath(path)`	`bool`
`filesystem.hasFilename(path)`	`bool`
`filesystem.hasStem(path)`	`bool`
`filesystem.hasExtension(path)`	`bool`
`filesystem.isAbsolute(path)`	`bool`
`filesystem.isRelative(path)`	`bool`

Operational	
These functions allow queries on the underlying filesystem.	
Path	
`filesystem.exists(path)`	`bool`
`filesystem.equivalent(path1, path2)`	`bool`
`filesystem.fileSize(path)`	`int`
`filesystem.currentPath()`	`string`
`filesystem.initialPath()`	`string`
`filesystem.absolute(path, [base])`	`string`
`filesystem.canonical(path, [base])`	`string`
`filesystem.systemComplete(path)`	`string`
Test	
`filesystem.isDirectory(path)`	`bool`
`filesystem.isEmpty(path)`	`bool`
`filesystem.isRegularFile(path)`	`bool`
`filesystem.isSymLink(path)`	`bool`
`filesystem.isOther(path)`	`bool`

Last Write Time	
`filesystem.lastWriteTime(path)`	`int`

Links	
`filesystem.readSymLink(path)`	`string`
`filesystem.hardLinkCount(path)`	`int`

For convenience users can declare `fs = filesystem` and call all filesystem functions using the fs prefix, demonstrated in the following example.

```
fs = filesystem
iterator fs.directory(path)
iterator fs.directoryRecursive(path)
```

3.7. PosixTime

Date and time related utility functions.

Functions	
`string posixTime.toString(int)`	Converts the posix-time to an ISO string

Note that all functions live in the global table posixTime.

Example

```
print(posixTime.toString(filesystem.lastWriteTime(...)))
```

Converts filesystem lastWriteTime to a string.

3.8. MIR functions

Music Information Retrieval (MIR) is the science of retrieving information from music. Among others, it allows the extraction of meaningful features from audio files, such as the pitch or the velocity of a sample. Creator Tools come with a collection of MIR functions, to assist or automate parts of the instrument creation process.

Single functions retrieve information from single files and take as argument an absolute filename (the full path to the sample file). Batch processing functions retrieve information from folders and take as argument an absolute folder name (the full path to the sample folder).

Note that all functions live in the global MIR table.

3.8.1. Pitch detection

The pitch detection tries to detect the fundamental frequency of a monophonic/single note sample. It corresponds to the MIDI scale (69 = 440 Hz) and ranges from semitone 15 (~20Hz) to semitone 120 (~8.4 kHz).

Functions	
`mir.detectPitch('fullPathToSample')`	Returns a floating point number corresponding to the MIDI pitch value of the audio sample specified in the 'fullPathToSample' absolute file path. If detection fails it will return `kDetectPitchInvalid`.
`mir.detectPitchBatch('fullPathToFolder')`	Returns a Lua table with samplePath as the table key and a floating point number corresponding to the detected pitch as the value. If detection fails it will return `kDetectPitchInvalid`

Examples

```
pitchVal = mir.detectPitch('fullPathToSample')
```

Sets `pitchVal` to the pitch of the sample at the 'fullPathToSample' filepath.

```
pitchBatchData = mir.detectPitchBatch('fullPathToFolder')
pitchValue = pitchBatchData['fullPathToSample']
```

Detects pitch of the samples in the 'fullPathToFolder' directory and stores them in the Lua table `pitchBatchData`. It then accesses `pitchBatchData` via the key 'fullPathToSample' and stores the resulting value in `pitchValue`

3.8.2. Peak, RMS & Loudness detection

Loudness, Peak, and RMS functions return a value in dB, with a maximum at 0dB.

The RMS and Loudness functions are calculated over small blocks of audio. The duration of those blocks is called frame size and is expressed in seconds. The process is repeated in intervals equal to the hop size (also expressed in seconds), until it reaches the end of the sample. The functions return the overall loudest/highest value of the different blocks.

If frame size and hop size are not indicated, the default values 0.4 (frame size in seconds) and 0.1 (hop size in seconds) are applied respectively.

In the case that Loudness, Peak, or RMS detection fails, they will return constants `kDetectLoudnessInvalid`, `kDetectPeakInvalid`, `kDetectRMSInvalid`, which all resolve to `1000000.0f`. When logging Peak, RMS, or Loudness values, invalid detections will be logged in numerical form.

It is advised to compare against respective `<k-type-invald>` constants in your scripts, since there is no guarantee that the numerical value will not change.

Functions	
Peak detection	
`mir.detectPeak('fullPathToSample')`	Returns a floating point number corresponding to the Peak value in dB of an audio sample at the 'fullPathToSample' absolute file path. If detection fails the constant `kDetectPeakInvalid` is returned.
`mir.detectPeakBatch('fullPathToFolder')`	Returns a Lua table with samplePath as the table key and a floating point number corresponding to the peak value in dB as the value for all samples in the 'fullPathToFolder' directory. If detection fails the constant `kDetectPeakInvalid` is returned
RMS detection	
`mir.detectRMS('fullPathToSample',frameSizeInSeconds, hopSizeInSeconds)`	Returns a floating point number corresponding to the RMS value in dB of an audio sample at the 'fullPathToSample' absolute file path. Frame size and hop size are optional arguments and default to 0.4 and 0.1 respectively if not specified. If detection fails the constant `kDetectRMSInvalid` is returned.
`mir.detectRMSBatch('fullPathToFolder', frameSizeInSeconds, hopSizeInSeconds)`	Returns a Lua table with samplePath as the table key and a floating point number corresponding to the RMS value in dB as the value for all samples in the 'fullPathToFolder' directory. Frame size and hop size are optional arguments and default to 0.4 and 0.1 respectively if not specified. If detection fails the constant `kDetectRMSInvalid` is returned.
Loudness detection	
`mir.detectLoudness('fullPathToSample', frameSizeInSeconds, hopSizeInSeconds)`	Returns a floating point number corresponding to the Loudness value in dB of an audio sample at the 'fullPathToSample' absolute file path. Frame size and hop size are optional arguments and default to 0.4 and 0.1 respectively if not specified. If detection fails the constant `kDetectLoudnessInvalid` is returned.
`mir.detectLoudnessBatch('fullPathToFolder', frameSizeInSeconds, hopSizeInSeconds)`	Returns a Lua table with samplePath as the table key and a floating point number corresponding to the Loudness value in dB as the value for all samples in the 'fullPathToFolder' directory. Frame size and hop size are optional arguments and default to 0.4 and 0.1 respectively if not specified. If detection fails the constant `kDetectLoudnessInvalid` is returned.

Examples

```
peakVal = mir.detectPeak('fullPathToSample')
```

Analyses an audio sample at the 'fullPathToSample' absolute file path and stores the resulting Peak value in `peakVal`.

```
rmsBatchData = mir.detectRMSBatch(‘fullPathToFolder', 0.02, 0.01)
```

Analyses all audio samples in the 'fullPathToFolder' directory and stores the resulting RMS values in the Lua table `rmsBatchData`. For the calculation, Frame and Hop size have been specified to 0.02 and 0.01 respectively.

3.8.3. Type detection

Type detection is a means to determine which category a given audio sample belongs to. Currently, Creator Tools supports detection of three distinct types: Sample Type, Drum Type, and Instrument Type. Sample Type is used to determine if a sample is either a drum, or an instrument. Drum Type and Instrument Type both determine which drum or instrument category a sample belongs to. For each type, an `INVALID` category is defined as helper for default initialisation.

Furthermore, in the case that Sample Type, Drum Type, or Instrument Type detection fails, they will return their respective INVALID type, which all resolve to `-1`. When logging Sample Type, Drum Type, or Instrument Type values, invalid detections will be logged in numerical form. However, it is advised to compare against respective `<type.INVALID>` constants in your scripts - since there is no guarantee that the numerical value will not change.

Note that these type detection functions are designed to process one-shot audio samples.

Functions	
Sample type detection	

`mir.detectSampleType('fullPathToSample')`	Returns the sample type of an audio sample at the 'fullPathToSample' absolute file path. Can return one of the following types:
	`DetectSampleType.INVALID`
	`DetectSampleType.INSTRUMENT`
	`DetectSampleType.DRUM`

`mir.detectSampleTypeBatch(‘fullPathToFolder’)`	Processes a batch of audio samples in the folder specified by the 'fullPathToFolder' absolute path. Returns a Lua table with samplePath as the key and the respective sample type as the value. The returned types are the same as in the single function call above.
Drum type detection	

Functions	

`mir.detectDrumType('fullPathToSample')`	Returns the drum type of an audio sample at the 'fullPathToSample' absolute file path. Can return one of the following types:
	`DetectDrumType.INVALID`
	`DetectDrumType.KICK`
	`DetectDrumType.SNARE`
	`DetectDrumType.CLOSED_HH`
	`DetectDrumType.OPEN_HH`
	`DetectDrumType.TOM`
	`DetectDrumType.CYMBAL`
	`DetectDrumType.CLAP`
	`DetectDrumType.SHAKER`
	`DetectDrumType.PERC_DRUM`
	`DetectDrumType.OTHER`

`mir.detectDrumTypeBatch('fullPathToFolder')`	Processes a batch of audio samples in the folder specified by the 'fullPathToFolder' absolute path. Returns a Lua table with samplePath as the key and the respective drum type as the value. The returned types are the same as in the single function call above.
Instrument type detection	

`mir.detectInstrumentType('fullPathToSample')`	Returns the instrument type of an audio sample at the 'fullPathToSample' absolute file path. Can return one of the following types:
	`DetectInstrumentType.INVALID`
	`DetectInstrumentType.BASS`
	`DetectInstrumentType.BOWED_STRING`
	`DetectInstrumentType.BRASS`
	`DetectInstrumentType.FLUTE`
	`DetectInstrumentType.GUITAR`
	`DetectInstrumentType.KEYBOARD`
	`DetectInstrumentType.MALLET`
	`DetectInstrumentType.ORGAN`
	`DetectInstrumentType.PLUCKED_STRING`
	`DetectInstrumentType.REED`
	`DetectInstrumentType.SYNTH`
	`DetectInstrumentType.VOCAL`

`mir.detectInstrumentTypeBatch('fullPathToFolder')`	Processes a batch of audio samples in the folder specified by the 'fullPathToFolder' absolute path. Returns a Lua table with samplePath as the key and the respective instrument type as the value. The returned types are the same as in the single function call above.

Examples

```
sampleType = mir.detectSampleType('fullPathToSample')
```

Analyses an audio sample at the 'fullPathToSample' absolute file path and stores the resulting sample type category in `sampleType`.

```
drumType = mir.detectDrumType('fullPathToSample')
```

Analyses an audio sample at the 'fullPathToSample' absolute file path and stores the resulting drum type category in `drumType`.

```
instType = mir.detectInstrumentType('fullPathToSample')
```

Analyses an audio sample at the 'fullPathToSample' absolute file path and stores the resulting instrument type category in `instType`.

```
sampleTypeBatchData = mir.detectSampleTypeBatch(‘fullPathToFolder’)
```

Analyses audio samples in the 'fullPathToFolder' directory and stores the resulting sample types in the Lua table `sampleTypeBatchData`.