



A suite of tools developed to support
the instrument creation process

Reference Manual

CREATOR TOOLS	4
KONTAKT DEBUGGER	4
KONTAKT INSTRUMENT EDITOR.....	6

Scripting Reference

INSTRUMENT STRUCTURE	8
SCRIPTING BASICS	9
SCRIPT PATH.....	9
READ PROPERTIES AND PRINT THEM	9
ITERATE OVER CONTAINERS.....	9
WORKING WITH CONTAINERS	10

Binding Reference

TYPE	11
OPERATORS	11
FUNCTIONS.....	11
PROPERTIES.....	12
SCALARS.....	12
PROPERTIES.....	12
FUNCTIONS.....	13
VECTOR	13
CONSTRUCTORS.....	13
OPERATORS	14
PROPERTIES.....	14
FUNCTIONS.....	15
STRUCT	16
CONSTRUCTORS.....	16
OPERATORS	16
PROPERTIES.....	16
FUNCTIONS.....	17

Contents

ALGORITHMS.....	18
FREE FUNCTIONS	18
FILE SYSTEM.....	19
EXAMPLES	19
FUNCTIONS.....	19
OPERATIONAL.....	20
POSIXTIME.....	22
EXAMPLES	22
FUNCTIONS.....	22
CONVERSIONS	22
MIR FUNCTIONS.....	23
PITCH DETECTION.....	23
PEAK, RMS & LOUDNESS DETECTION.....	24

Reference Manual

Creator Tools

A suite of tools developed to support the instrument creation process. It consists of the Debugger and the Instrument Editor. Switching between the two tools is possible from the top tabs, or by using the shortcut F1 for Debugger and F2 for Instrument Editor. Creator Tools actions can also be triggered via their dedicated shortcuts. Shortcuts act according to which panel the user is focused on.

Kontakt Debugger

The debugger connects to all running instances of Kontakt, both plug-in and standalone. It logs messages, warnings and errors coming for KSP, supports inspecting script variables, provides timestamps per notification and some basic filtering options.

VARIABLE WATCHING (§/CTRL-E)

Clicking on the eye icon reveals the Variable Watching area. 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 Log above, 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.


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

PAUSE (⌘/CTRL-P) ||

Suspends the debugging session. While active, the Pause button is blinking. Once the session is resumed, all messages that were received during pause will appear.

CLEAR (⌘/CTRL-BACKSPACE) 

Clears all content of the Debugger log.

SETTINGS 

Defines the behaviour of the Log.

LOG

This is where all notifications from Kontakt appear. The Log contains 7 columns:

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

Apart from Type and Message, all other columns can be hidden. Right-clicking on the column header reveals the column menu.

Kontakt Instrument Editor

The Instrument Editor connects to a running instance of Kontakt, either plug-in or standalone, and offers programmatic access to parts of a Kontakt instrument's structure through Lua-based scripting.

It 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 if needed. Ideally, the content of the scripts' folder can be copied to "`*user*/Documents/Native Instruments/Creator Tools`".

MULTI RACK MENU

Sets the focus of the tool on the multi rack of one of the connected Kontakt instances.

INSTRUMENT MENU

Sets the focus of the tool to a specific instrument that is loaded in one of the connected Kontakt instances. Note that instruments with locked edit views, cannot be selected.

PUSH (⌘/CTRL-Shift-↑) ↑

Applies to Kontakt all changes from the Tools' side. If changes are not pushed, an indication on the button appears to notify for the pending changes.

PULL (⌘/CTRL-Shift-↓) ↓

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.

CONNECTION INDICATOR

The connection indicator on the top right corner, indicates whether a successful connection between the tools and the Kontakt instances is established.

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.

SCRIPT PANEL

Changes within the Tools happen exclusively via running a Lua script. A script can see and modify the instrument copy in the Tools. Scripts can be created and modified with an external editor. The script output will appear in the console output. All console output can be copied to system clipboard via the command ⌘/Ctrl-Alt-C.

LOAD (⌘/CTRL-L)

Opens the file explorer in order to locate a .lua file in disk and load it. 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.]

OPEN IN TEXT EDITOR (⌘/CTRL-O)

Opens the loaded script file in the system's default editor.

RUN (⌘/CTRL-R)

Executes the loaded .lua script. Changes are immediately reflected in the Instrument Tree View.

STOP (⌘/CTRL-I)

Stops the execution of the running script. The Instrument Editor state is reverted, as if the script never run.

CLEAR

Clears all content of the Script Output Panel.

Scripting Reference

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 current structure looks like this:

Instrument	Struct
name	String
groups	Vector of Group
name	String
volume	Real, -inf..12
tune	Real, -36..36
zones	Vector of Zone
uniqueID	Int
file	String
volume	Real, -inf..12
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

Future updates will allow more properties to be edited.

Scripting Basics

Scripting is based on the language Lua. 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*.

SCRIPT PATH

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

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
```

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)  
prints the number of groups of the instrument
```

```
for n=0,#instrument.groups-1 do  
    print(instrument.groups[n].name)  
end  
iterates over the groups of the instrument and prints their names
```

Note that vectors are zero-indexed! There are ways to iterate containers without using indices.

WORKING WITH CONTAINERS

```
Group()
```

This 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())
```

This adds a new empty group at the end

The next example inserts a deep copy of the 4th group at index 0 i.e. at the beginning:

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

Binding Reference

Type

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

OPERATORS

`toString`
returns a string representation describing the object

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`

PROPERTIES

`object.typeinfo`
returns type information as a string in the form `Type'Tag`

`object.parent`
returns the parent object or nil

Scalars

Basic types which contain a single value:

`Bool`
boolean, true or false

`Int`
64 bit integer (signed i.e. can be negative)

`Real`
64 bit floating point number

`String`
text

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

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 initial

FUNCTIONS

`vector:set(index, object)`
set 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)`
inserts object at the end

`vector:add(object)`
inserts object at the end

`vector:add(value)`
inserts value at the end

`vector:insert(index, object)`
inserts object or value before index

`vector:insert(index, value)`
inserts value before index

`vector:remove(index)`
removes element at index

Struct

Type-safe, record with named fields.

For a struct object the following accessors are defined:

CONSTRUCTORS

```
struct()  
returns 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 initial
```


FUNCTIONS

`vector:set(index, object)`
assigns object at index

`vector:get(index)`
returns object at index

`vector:reset()`
resets the struct to initial

`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

Algorithms

FREE FUNCTIONS

`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

File system

The Lua binding is based on the C++ library [boost filesystem](#). The [reference documentation](#) describes each function in detail. 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.

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 table filesystem.

Iterators:

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

Path:

The functions return a string which contains the modified path.

```
string 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)
```

Query:

The functions query a given path.

```
bool 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)
```

OPERATIONAL

These functions allow queries on the underlying filesystem.

Path:

```
bool filesystem.exists(path)
bool filesystem.equivalent(path1, path2)
int filesystem.fileSize(path)
string filesystem.currentPath()
string filesystem.initialPath()
string filesystem.absolute(path, [base])
string filesystem.canonical(path, [base])
string filesystem.systemComplete(path)
```

Test:

```
bool filesystem.isDirectory(path)
bool filesystem.isEmpty(path)
bool filesystem.isRegularFile(path)
bool filesystem.isSymLink(path)
bool filesystem.isOther(path)
```

Last Write Time:

```
int filesystem.lastWriteTime(path)
```

Links:

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

Note:

For convenience user can declare `fs=filesystem` and call all filesystem functions using the `fs` prefix:

Example:

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

PosixTime

Date and time related utility functions.

EXAMPLES

```
print(posixTimeToString(filesystemlastWriteTime(...)))
```

Converts filesystem lastWriteTime to a string.

FUNCTIONS

Note that all functions live in the global table `posixTime`.

CONVERSION

Note that all functions live in the global table `posixTime`.

```
string posixTime.toString(int)
```

Converts the posix-time to an ISO string.

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 table `mir`.

PITCH DETECTION

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

Pitch functions return a floating point value corresponding to the MIDI scale (69 = 440Hz). In case the pitch analysis fails, it will return a value of 0.

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

Single function call

```
pitchBatch = mir.detectPitchBatch('fullPathToFolder')
```

Batch processing

Batch processing will return a Lua table with `samplePath` as the table key and pitch as the value. It can be accessed in the following way:

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

PEAK, RMS & LOUDNESS DETECTION

Peak, RMS and Loudness 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.

Single Functions

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

```
rmsVal = mir.detectRMS('fullPathToSample')  
  
rmsVal = mir.detectRMS('fullPathToSample', frameSizeInSeconds,  
hopSizeInSeconds)
```

RMS detection

```
loudnessVal = mir.detectLoudness('fullPathToSample')  
  
loudnessVal = mir.detectLoudness('fullPathToSample',  
frameSizeInSeconds, hopSizeInSeconds)
```

Loudness detection

Batch Processing

Batch processing will return a Lua table with samplePath as the table key and peak (RMS, Loudness) as the value. It can be accessed in the following way:

```
peakValue = peakBatchData['fullPathToSample']
```

```
peakBatchData = mir.detectPeakBatch('fullPathToFolder')  
Peak detection
```

```
rmsBatchData = mir.detectRMSBatch('fullPathToFolder')  
  
rmsBatchData = mir.detectRMSBatch('fullPathToFolder',  
frameSizeInSeconds, hopSizeInSeconds)
```

RMS detection

```
loudnessBatchData = mir.detectLoudnessBatch('fullPathToFolder')  
  
loudnessBatchData = mir.detectLoudnessBatch('fullPathToFolder',  
frameSizeInSeconds, hopSizeInSeconds)
```

Loudness detection