Koala

June 5, 2016

An advanced scripting library for KSP developers

Concept and scripting by Davide Magni

version 1.0

Koala is a collection of functions and macros which enormously simplify the process of writing scripts.

This Library is the result of almost a year of work and it's still currently being updated and improved. The Library is open source and it is covered by MIT License.

This Manual will guide the Developer to completely understand every single feature of this extremely powerful tool.

MIT License

Contacts

info@davidemagni.com

www.davidemagni.com

Skype: magneto538

Twitter: magneto538

Contents

Part I

Introduction

This library (hereby known as "Koala") is a collection of functions and macros which enormously simplify the process of writing scripts.

The usage of this library will improve significantly the look and readability of your scripts, as well as allowing super fast access to advanced functions not available with standard KSP code.

This guide assumes you are well-aware of what Kontakt Scripting is. If you don't, please be sure to check out KSP Manual (if you own Konakt, you will find it inside Kontakt documentation files).

We are not going to explain basic Kontakt scripting features, so be sure to know them well.

1 Pre-requisites

In order to use Koala, you have to fulfill the following requirements:

1. **SublimeText 3 BETA**, which can be found at http://www.sublimetext.com/3. Download and install the latest version.
2. **Nils Liberg's SublimeKSP plugin**. This plugin is included within Koala's ZIP file: we provide you a patched version of this plugin with all our custom functions added. SublimeKSP is a plugin for SublimeText which allows you to compile your Scripts. SublimeKSP also makes the scripting process easier and faster by introducing some amazing features, as well as syntax simplifications. The complete list of available features can be found here.
3. **Any version of Kontakt FULL**. You simply cannot use Kontakt Player to execute your own Scripts. Full version is needed.

2 Configuring SublimeText and SublimeKSP Plugin

After installing SublimeText 3 BETA and Kontakt, you need to setup SublimeText in order to be able to compile your KSP scripts.

NOTE: you might already have SublimeKSP plugin installed inside your *Packages* folder. Koala comes with a patched version of SublimeKSP which includes our custom functions and constants, so make sure to replace it with our own version.

1. Open SublimeText 3 BETA. On the Menu bar, find *Preferences* and choose *Browse Packages…* . A folder called *Packages* is opening up.
2. Close SublimeText 3 BETA.
3. Go to the folder where you extracted all Koala's files and look for *SublimeKSP* folder.
4. Copy *SublimeKSP* folder inside *Packages* folder.
5. Start SublimeText 3 BETA and create a new file. Save it where you want to. Name it *<asyouwish>.ksp*. This is your KSP Script.
6. On the low-right corner, you will find a menu which displays "Plain text". Click on "Plain text" and choose "KSP" instead.
7. On the Menu bar, find *Tools*. Inside this Menu, you should see some entries with "KSP:" prefix. If you see this, you are able to use SublimeKSP to compile your own scripts. If you don't, be sure to double-check all the previous passages.

8. In Tools menu, enable the following entries:
   - "KSP: Extra syntax check"
   - "KSP: Optimize compiled code"

   This will give the compiler the instructions to include macros, functions, constants and other KSP features in the correct way.
9. Write your Script. When you are ready to move it to Kontakt, press CMD+K (Mac) or F5 (Windows) to compile the Script and copy it to clipboard. If you want to see the generated Script, simply do CTRL+v in any text editor. Paste the Script to Kontakt and press "Apply".

3 Including Koala Library into your Script

You can take a look to the example script to see how to do this.

1. Move the folder called *Koala* to the folder where you have your KSP script.
2. Open your Script and look for the very beginning of the file (line 1). Write the following instruction:

   ```
   import "Koala/Koala.ksp"
   ```

   This will import the Koala Library into your Script.
3. Inside your Script, look for the very beginning of on init callback. Complete the callback with the following instruction:

   ```
   import "Koala/Koala.ksp"
   on init
       Koala.init
       {...your script...}
   end on
   ```

   This will tell the compiler to import everything you need to run Koala Library.
4. That's it! You are ready to get into scripting!

4 Additional notes about SublimeKSP syntax

SublimeKSP introduces a simplified syntax that helps developers to save time while writing code. Therefore, writing code becomes simpler, as you can "forget" some of KSP real grammar when declaring or manipulating any part of your script.

Please refer to http://nilsliberg.se/ksp/ to check out the complete list of features that SublimeKSP introduces. We are going to summarize the most important ones here.

1. When declaring or calling a variable, constant, array etc., **you can leave out the prefixes `$` and `%`**. The compiler is going to add them for you later. As of today, though, **you still need to insert manually the prefixes for strings variables and arrays, `@` and `!`.**
2. **You can use `for` loops**, which do not exist in real KSP syntax. The compiler will translate them into `while` loops later.
3. **You can use improved constructs for `if` statements**, such as `else if`, which do not exist in real KSP syntax. The compiler will translate them into a sequence of `if` statements.
4. **`macro` calls will be unpacked** by default by the compiler: their calls will be replaced by the whole content of each `macro`.

5. **`macro` calls support the Wild Card symbol `#` when defining the macro's prototype.** `function` calls don't support this at the moment.
6. **`function` calls** are handled in two separate ways:
   (a) If a call is performed by using the syntax `call <name of function>`, **the compiler will keep the original declaration** in the code exactly as you wrote it.
   (b) If a call is performed by writing only the name of the function, **the compiler will replace entirely** that portion of code with the whole content of the declaration of the function, exactly as it happens with `macro` calls.

For any other SublimeKSP feature, please refer to Nils Liberg's website as stated before.

4.1 Notes on the syntax used in this Manual

Every now and then, this Manual will use examples to explain the concepts behind each feature of Koala Library. The examples will look like these:

1. As SublimeKSP automatically adds prefixes for **integer variables and arrays**, we are not going to specify them in the Scripts, neither in declarations nor in calls. Keep in mind that **string variables and arrays**, on the other hand, lack this feature in SublimeKSP, so we will specify their prefixes both in declarations and calls.
2. We will use, when available, SublimeKSP simplified syntax, which includes the features stated in the previous section.

Part II

Graphic User Interface features

Koala Library features a ton of GUI functions to massively reduce the time you need to spend on scripting for your GUI.

Here is the list of the features available with Koala Library.

1 Initial operations

Before anything can be done to your script, you have a set of functions that need to be declared in order to setup the GUI.

1.1 Creating your GUI

This function allows you to initialize Kontakt's GUI by setting its height and wallpaper. Be sure to declare this before anything else in your script, otherwise the GUI will not be initalized.

The lack of this declaration will cause nothing to be displayed on your GUI when you compile your Script to Kontakt.

```
create_instrument(<height>,<wallpaper>,<icon>,<title>)
```

`height`	Desired height in pixel. The maximum available height is 540 px.
`wallpaper`	Name of .png file you wish to use as wallpaper for your Instrument.
`icon`	Name of .png file you wish to use as icon for your Instrument.
`script_title`	Title of your Script.

1.2 Creating your first UI Family

Before starting to add UI Controls to your User Interface, you need to create one or more **Families**. Each Family is an ID which allows the developer to perform mass action over a selected set of UI Controls. It's extremely important to assign UI Controls to proper Families. We will explain how Families work in the next chapter.

In order to create a Family, you have to call the following function in `on init` callback:

```
create_family(<fam>,<id>)
```

Now you can assign any UI Control to this Family.

You can choose any name for the Family and any ID you like.

Keep in mind that a **Family name** works exactly as a variable, so the name **has to be unique**: neither used by other Families, nor used by constants or variables, otherwise the compiler will return an error message for redeclaration of a constant.

Family IDs have to be different too: you should be even more careful here, as the compiler is not going to return an error message if you declare two Families with the same ID (but you are going to encounter some serious troubles, as you can imagine). Please be careful to avoid this conflict.

2 Adding UI Controls to your project

One of the most annoying things in KSP is the **generation of UI Controls**.

There are lots of parameters that can be set when creating an UI Control. Koala has some functions which allow them to be generated, initialized and managed.

There are 9 different UI Controls in Kontakt. Each of them can be generated by Koala using only one line of code. Every single parameter you initialize a UI Control with - position, size, text etc. - is stored and can be retrieved in any time by proper functions (we will get into this later).

Each single UI Control type can be generated using each of the functions in this chapter.

2.1 Buttons

For info about the differences between Buttons ans Switches, please refer to the KSP Manual.

```
create_button(<name>,<fam>,<x>,<y>,<w>,<h>,<default>,<pers_status>,<text>,<picture>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`default`	Default value.
`persistence_status`	Can be set either to `PERSISTENT` or `NOT_PERSISTENT`. If set to `PERSISTENT`, the compiler will add the function `make_persistent()` to the Script; otherwise, no code will be added.
`text`	Text you want to be displayed on the UI Control.
`picture`	Name of .png file you wish to use as picture.

2.2 Switches

For info about the differences between Buttons ans Switches, please refer to the KSP Manual.

```
create_switch(<name>,<fam>,<x>,<y>,<w>,<h>,<default>,<pers_status>,<text>,<picture>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`default`	Default value.
`persistence_status`	Can be set either to `PERSISTENT` or `NOT_PERSISTENT`. If set to `PERSISTENT`, the compiler will add the function `make_persistent()` to the Script; otherwise, no code will be added.
`text`	Text you want to be displayed on the UI Control.
`picture`	Name of .png file you wish to use as picture.

2.3 Text Labels

Normally, Kontakt has only one UI Control for Labels. We decided to split it into two separate resources - Text and Picture Labels - to make the script even easier to be read.

```
create_text_label(<name>,<fam>,<x>,<y>,<w>,<h>,<text>,<font>,<alignment>,<background>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`text`	The text you want to be displayed on the Text Label.
`font`	Font type. For more info about available fonts and their names, please refer to *"UI Text"* section of this Manual.
`alignment`	Text alignment. The following constants are available: `LEFT, CENTER, RIGHT`
`background`	Name of .png file you wish to use as background picture.

2.4 Picture Labels

Normally, Kontakt has only one UI Control for Labels. We decided to split it into two separate resources - Text and Picture Labels - to make the script even easier to be read.

```
create_pic_label(<name>,<fam>,<x>,<y>,<w>,<h>,<picture>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`picture`	Name of .png file you wish to use as picture.

2.5 Sliders

```
create_slider(<name>,<fam>,<x>,<y>,<w>,<h>,<min>,<max>,<default>, ...
    <pers_status>,<mouse_behaviour>,<picture>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`min`	Lowest value the Slider can be set at.
`max`	Highest value the Slider can be set at.
`default`	Default value. You can set any control to its default value by pressing CTRL+click on it.
`persistence_status`	Can be set either to `PERSISTENT` or `NOT_PERSISTENT`. If set to `PERSISTENT`, the compiler will add the function `make_persistent()` to the Script; otherwise, no code will be added.
`mouse_behaviour`	Value from -5000 to 5000. Sets the move direction of a slider and its drag-scale.
`picture`	Name of .png file you wish to use as picture.

2.6 Knobs

Keep in mind that Knobs in Kontakt behave in a quite weird way: you cannot assign a custom picture to them and you cannot resize them, so these values are not available for this UI Control.

```
create_knob(<name>,<fam>,<x>,<y>,<min>,<max>,<default>, ...
    <disp_ratio>,<pers_status>,<unit>,<text>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`min`	Lowest value the Slider can be set at.
`max`	Highest value the Slider can be set at.
`default`	Default value. You can set any control to its default value by pressing CTRL+click on it.
`disp_ratio`	The knob value is divided by *`disp_ratio`* for display purposes.
`persistence_status`	Can be set either to `PERSISTENT` or `NOT_PERSISTENT`. If set to `PERSISTENT`, the compiler will add the function `make_persistent()` to the Script; otherwise, no code will be added.
`unit`	Assign a unit of measure to the knob. The following constants are available: `$KNOB_UNIT_NONE, $KNOB_UNIT_DB, $KNOB_UNIT_HZ, $KNOB_UNIT_PERCENT, $KNOB_UNIT_MS, $KNOB_UNIT_OCT, $KNOB_UNIT_ST`
`text`	Text you want to be displayed on the Knob's label.

2.7 Menus

```
create_menu(<name>,<fam>,<x>,<y>,<w>,<h>,<default>,<entries>, ...
    <font>,<alignment>,<pers_status>,<picture>)
```

NOTE: to add menu entries to a menu, please refer to `add_menu_entries` function.

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`default`	Default value.
`entries`	Amount of values available for this Menu. Each entry will be added with `add_menu_entry` function.
`persistence_status`	Can be set either to `PERSISTENT` or `NOT_PERSISTENT`. If set to `PERSISTENT`, the compiler will add the function `make_persistent()` to the Script; otherwise, no code will be added.
`picture`	Name of .png file you wish to use as picture.

2.8 Value Edits

```
create_value_edit(<name>,<fam>,<x>,<y>,<w>,<min>,<max>,<default>, ...
    <disp_ratio>,<pers_status>,<font>,<picture>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`min`	Lowest value the Slider can be set at.
`max`	Highest value the Slider can be set at.
`default`	Default value. You can set any control to its default value by pressing CTRL+click on it.
`disp_ratio`	The knob value is divided by *`disp_ratio`* for display purposes.
`persistence_status`	Can be set either to `PERSISTENT` or `NOT_PERSISTENT`. If set to `PERSISTENT`, the compiler will add the function `make_persistent()` to the Script; otherwise, no code will be added.
`font`	Font type. For more info about available fonts and their names, please refer to this Manual's appendix.
`picture`	Name of .png file you wish to use as picture.

2.9 File Selector

Remember: only ONE File Selector per each Script Slot is allowed. Be sure to remember this.

```
create_file_selector(<name>,<fam>,<x>,<y>,<w>,<h>,<col_width>, ...
    <file_type>,<root_folder>,<font>)
```

name	Name of the variable.
fam	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
x	Position on X axis.
y	Position on Y axis.
w	Width of UI Control.
h	Height of UI Control.
col_width	Width of each column.
file_type	The following constants are available. Please refer to KSP Reference Manual to get more info. `$NI_FILE_TYPE_MIDI, $NI_FILE_TYPE_AUDIO, $NI_FILE_TYPE_ARRAY`
root_folder	Root folder.
font	Font type. For more info about available fonts and their names, please refer to this Manual's appendix.

2.10 Tables

```
create_table(<name>,<fam>,<x>,<y>,<w>,<h>,<columns>,<range>,<pers_status>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`columns`	Number of columns of table.
`range`	Range of each Table column.
`persistence_status`	Can be set either to `PERSISTENT` or `NOT_PERSISTENT`. If set to `PERSISTENT`, the compiler will add the function `make_persistent()` to the Script; otherwise, no code will be added.

2.11 Level Meters

```
create_level_meter(<name>,<fam>,<x>,<y>,<w>,<h>,<orientation>, ...
    <group>,<slot>,<channel>,<bus>)
```

NOTE: if you want to set the Level Meter's colors, please refer to `set_level_meter_color()` function.

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`orientation`	Can be set either to `VERTICAL` or `HORIZONTAL` meter.
`group`	The index of the group you wish to access. Should be set to `-1` if not using the group level. Inside the function, this parameter is used with `attach_level_meter` KSP function.
`slot`	The index of the fx slot you wish to access. Should be set to -1 if you do not wish to access an fx slot. Inside the function, this parameter is used with `attach_level_meter` KSP function.
`channel`	Can be set either to `LEFT_CH` or `RIGHT_CH` channel. Inside the function, this parameter is used with `attach_level_meter` KSP function.
`bus`	The index of the instrument bus you wish to access. Should be set to -1 if you are not accessing the bus level. Inside the function, this parameter is used with `attach_level_meter` KSP function.

2.12 Waveforms

```
create_waveform(<name>,<fam>,<x>,<y>,<w>,<h>,<zone_name>,<zone_flags>)
```

name	Name of the variable.
fam	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
x	Position on X axis.
y	Position on Y axis.
w	Width of UI Control.
h	Height of UI Control.
zone_name	Name of the Zone that you want to attach to the waveform display. Inside the function, this parameter is used with attach_zone KSP function.
zone_flags	you can control different settings of the UI waveform via its flags. Inside the function, this parameter is used with attach_zone KSP function. The following flags are available: UI_WAVEFORM_USE_SLICES, UI_WAVEFORM_USE_TABLE, UI_WAVEFORM_TABLE_IS_BIPOLAR, UI_WAVEFORM_USE_MIDI_DRAG

2.13 Text Entries

This set of functions allows to create a custom editable text box.

```
create_text_entry(<name>,<fam>,<x>,<y>,<w>,<h>,<text>,<font>,<alignment>,<background>)
```

`name`	Name of the variable.
`fam`	Assigned UI Family. Make sure you have created a Family before (see next chapter for details).
`x`	Position on X axis.
`y`	Position on Y axis.
`w`	Width of UI Control.
`h`	Height of UI Control.
`text`	The text you want to be displayed on the Text Edit. You can edit this further.
`font`	Font type. For more info about available fonts and their names, please refer to *"UI Text"* section of this Manual.
`alignment`	Text alignment. The following constants are available: `LEFT, CENTER, RIGHT`
`background`	Name of .png file you wish to use as background picture.

3 Managing UI Controls: setting data

This Library features a wide range of functions to handle processes over UI Controls.

All the following functions have to be fed with the UI ID of the desired control. We decided to do this instead of passing the variable name in order to allow the developer to use these functions on multiple controls, without necessarily having to run through Families to set some "group" settings.

For instance, you can save all the UI IDs of a Channel of your Mixer inside an array and then you can use a loop to set all these controls' position. Here's an example:

```
on init
    {...}
    declare ui_ids[5]
    ui_ids[0] := get_ui_id(mute_btn)
    ui_ids[1] := get_ui_id(solo_btn)
    ui_ids[2] := get_ui_id(volume)
    ui_ids[3] := get_ui_id(panpot)
end on
on ui_control(button)
    for count := 0 to 3
        shift_ctrl(ui_ids, 70, 0)
    end for
end on
```

This Script features a button. Each time the button is pressed, all the UI Controls whose UI IDs were loaded inside `ui_ids[]` array will be shifted 70 px on the right from their current position. Keep in mind that the UI ID of a specific UI Control can be retrieved using the native KSP function `get_ui_id(<variable name>)`.

You can perform many actions over UI Controls such as move, show, hide and so on. You can also retrieve each UI Control's values. There are two ways to access and set data for each UI Control:

1. **By single control:** you can choose to perform an action over a specific UI Control, given its UI ID.
2. **By Family:** you can access to each declared Family in order to perform group actions over all the UI Controls of that Family.

In this section, we will focus on the first method.

Additional notes on UI Controls management

1. **Everytime any of the following functions is used in `on init` callback, the values passed to the function overrides the default values.** For example:

```
on init
    create_family(FAMILY, 1)
    create_knob(knob, FAMILY, ...)
    set_pos(get_ui_id(knob), 60, 70)
end on
```

In this case, `set_pos` overrides the default values set in create_knob (10 and 15). If you call `reset_pos`, the knob position will be set at X=60 and Y=70. If you call `set_pos` from a different callback, obviously, the default value is not overriden.

3.1 set_pos

This function allows the developer to set a specific UI Control's position given X and Y coordinates.

```
set_pos(<ui_id>,<x>,<y>)
```

ui_id	UI Control ID. Can be retrieved using get_ui_id(<variable name>).
x	Position on X axis.
y	Position on Y axis.

3.2 shift_ctrl

This function works similarly to set_pos, but instead of moving the UI Control by giving absolute coordinates, it requires to be fed with offset values. The UI Control's resulting position will be relative to its previous position, plus (or minus) an offset.

```
shift_ctrl(<ui_id>,<off_x>,<off_y>)
```

ui_id	UI Control ID. Can be retrieved using get_ui_id(<variable name>).
off_x	Shift offset on X axis.
off_y	Shift offset on Y axis.

3.3 set_ctrl_pic

This function allows the developer to set a specific UI Control's picture (if any).

```
set_ctrl_pic(<ui_id>,<picture>)
```

ui_id	UI Control ID. Can be retrieved using get_ui_id(<variable name>).
picture	Name of .png file you wish to use as picture.

3.4 set_ctrl_text

This function allows the developer to set a specific UI Control's text (if available).

```
set_ctrl_text(<ui_id>,<text>)
```

ui_id	UI Control ID. Can be retrieved using get_ui_id(<variable name>).
text	Text you want to be displayed on the UI Control.

3.5 set_font

This function allows the developer to set a specific UI Control's text font and alignment (if available).

```
set_font(<ui_id>,<font>,<alignment>)
```

ui_id	UI Control ID. Can be retrieved using get_ui_id(<variable name>).
font	Font type. For more info about available fonts and their names, please refer to this Manual's appendix.
alignment	Text alignment. The following constants are available: LEFT, CENTER, RIGHT

3.6 set_value

This function allows the developer to set a specific UI Control's value (if available).

```
set_value(<ui_id>,<value>)
```

ui_id	UI Control ID. Can be retrieved using get_ui_id(<variable name>).
value	Text you want to be displayed on the UI Control.

3.7 set_visibility

These functions allow the developer to show or hide a specific UI Control given its UI ID. <value> must be replaced either with VISIBLE or INVISIBLE.

```
set_visibility(<ui_id>,<value>)
```

4 Managing UI Controls: retrieving data

Koala Library features two types of functions to get a UI Control data:

1. The developer can retrieve the data as declared in `on init` callback.
2. The developer can retrieve the current data of the selected UI Control.

Let's get through both of these groups of functions.

NOTE: the functions to retrieve both Current and Default data ARE inline functions, therefore can be used in `if` statements, `while` loops etc.

4.1 get_def_pos_x and get_def_pos_y

These functions return the default position of the specified UI Control on each axis, as declared in `on init` callback.

These ARE inline functions, therefore they can be used directly as arguments in KSP constructs such as `if`, `while` etc.

Usage:

```
if (get_def_pos_x(<ui_id>) = ...)
    {...}
end if
```

4.2 get_def_width and get_def_height

These functions return the default size values of the specified UI Control, as declared in `on init` callback.

These ARE inline functions, therefore they can be used directly as arguments in KSP constructs such as `if`, `while` etc.

Usage:

```
if (get_def_width(<ui_id>) = ...)
    {...}
end if
```

4.3 get_def_value

This function returns the default value of the specified UI Control, as declared in `on init` callback.

This IS an inline function, therefore it can be used directly as argument in KSP constructs such as `if`, `while` etc.

Usage:

```
if (get_def_value(<ui_id>) = ...)
    {...}
end if
```

4.4 get_def_picture

This function returns the UI Control's picture name. This function returns a **string,** so be sure to load its value inside a string value or array.

This IS an inline function, therefore it can be used directly as argument in KSP constructs such as `if`, `while` etc.

Usage:

```
@picture := get_def_picture(<ui_id>)
```

4.5 get_def_visibility

This function returns the UI Control's default visibility status, as declared in `on init` callback.

This IS an inline function, therefore it can be used directly as argument in KSP constructs and functions enabled to receive a string as input.

Usage:

```
call display_picture_name(get_def_picture(<ui_id>))
```

4.6 get_def_visibility

This function returns the UI Control's default visibility status, as declared in `on init` callback.

This IS an inline function, therefore it can be used directly as argument in KSP constructs and functions enabled to receive a string as input.

Usage:

```
if (get_def_visibility(<ui_id>) = ...)
     {...}
end if
```

4.7 get_type

This function returns the UI Control's type. The Type of each UI Control is an ID set when declaring the UI Control itself according to what UI Control has been declared (Sliders, Knobs, Buttons etc.).

UI Control Type IDs are listed under *"System constants"*, *"User Interface Constants"* section of this Manual.

This IS an inline function, therefore it can be used directly as argument in KSP constructs such as `if`, `while` etc.

Usage:

```
if (get_type(<ui_id>) = ...)
     {...}
end if
```

4.8 get_pos_x and get_pos_y

These functions return the current position of the specified UI Control on each axis.

These ARE inline functions, therefore they can be used directly as arguments in KSP constructs such as `if`, `while` etc.

Usage:

```
if (get_pos_x(<ui_id>) = ...)
     {...}
end if
```

4.9 get_width and get_height

These functions return the current size values of the specified UI Control.

These ARE inline functions, therefore they can be used directly as arguments in KSP constructs such as `if`, `while` etc.

Usage:

```
if (get_width(<ui_id>) = ...)
     {...}
end if
```

4.10 get_value

This function returns the current value of the specified UI Control.

This IS an inline function, therefore it can be used directly as argument in KSP constructs such as `if`, `while` etc.

Usage:

```
if (get_value(<ui_id>) = ...)
     {...}
end if
```

4.11 get_picture

This function returns the UI Control's picture name. This function returns a **string,** so be sure to load its value inside a string value or array.

This IS an inline function, therefore it can be used directly as argument in KSP constructs and functions enabled to receive a string as input.

Usage:

```
call display_picture_name(get_picture(<ui_id>))
```

4.12 get_visibility

This function returns the UI Control's current visibility status.

This IS an inline function, therefore it can be used directly as argument in KSP constructs and functions enabled to receive a string as input.

Usage:

```
if (get_visibility(<ui_id>) = ...)
     {...}
end if
```

5 Managing UI Controls: resetting data

When declaring a UI Control, the informations about position, size, value etc. are stored inside arrays in order to be recalled later. You can reset each UI Control's data to the values declared in `on init` callback.

One more thing. When you change any of these settings in `on init` callback using the functions `set_pos`, `set_value` etc., the new value is stored as default.

The same thing occurs if you perform this action on Families: for example, if in `on init` callback you call `set_family_pos()`, the default positions of all the UI Controls belonging to that Family are updated.

The following values are stored as defaults and can be recalled by the `reset_` functions:

- Position X
- Position Y
- Picture
- Text
- Font Type
- Font Alignment
- Default Value
- Default Visibility

5.1 reset_pos

This function allows to reset the UI Control's position to its default.

```
reset_pos(<ui_id>)
```

5.2 reset_ctrl_pic

This function allows to reset the UI Control's picture to its default.

```
reset_ctrl_pic(<ui_id>)
```

5.3 reset_ctrl_text

This function allows to reset the UI Control's text to its default.

```
reset_ctrl_text(<ui_id>)
```

5.4 reset_font

This function allows to reset the UI Control's font type and alignment to their default.

```
reset_font(<ui_id>)
```

5.5 reset_value

This function allows to reset the UI Control's value to its default.

```
reset_value(<ui_id>)
```

5.6 reset_visibility

This function allows to reset the UI Control's visibility status to its default.

```
reset_visibility(<ui_id>)
```

6 Managing UI Controls: functions dedicated to specific types of UI Controls

There are a few advanced functions which allow the Developer to set various properties for specific types of UI Controls.

6.1 UI Menus: add_menu_entry

This function allows the developer to create a menu entry for a specific UI Menu. The order of the entries inside the UI Menu depends on the order of declaration.

```
add_menu_entry(<name>,<text>,<visibility>)
```

`name`	UI Menu variable name.
`text`	Text you want to be displayed on the Menu Entry.
`visibility`	Hide/show the created menu entry. Can be set either to `VISIBLE` or `INVISIBLE`.

6.2 UI Level Meters: set_level_meter_color

This function allows the developer to set the colours for UI Level Meters.

Each colour parameter must be a 6-character Hexadecimal value (e.g. `FF0000` is Red).

```
set_level_meter_color(<ui_id>,<bg_color>,<off_color>,<on_color>, ...
    <overload_color>,<peak_color>)
```

`ui_id`	UI Control ID. Can be retrieved using `get_ui_id(<variable name>)`.
`bg_color`	UI Level Meter Background color.
`off_color`	UI Level Meter Off-status color.
`on_color`	UI Level Meter On-status color.
`overload_color`	UI Level Meter Overload-status color.
`peak_color`	UI Level Meter Peak-status color.

7 UI Families: perform group actions on UI Controls

We wanted to be able to handle Scripts with a large amount of UI Controls. To do so, we introduced the concept of **Families**.

A **UI Family** is an ID assigned to each UI Control you create. This ID will be used to access to all the UI Controls assigned to a Family in order to perform mass actions on those UI Controls. For instance, you can hide and move all the UI Controls belonging to a specific Family.

It's extremely important to organise UI Controls in proper Families. This will make much easier to access each Family the way you expect to, with no "surprises".

The organization of UI Controls and Families, obviously, is up to you.

Usage of Families: an example

An example of Family can be, for instance, all the controls of a Mixer. When you define each single UI Control, you are asked to specify the Family they belong to. Then you can show, hide, move, process each single UI Control belonging to that Family in an extremely advanced manner.

```
on init
    {...}
    create_family(MIXER_FAMILY, 0)
    create_button(mute_btn, MIXER_FAMILY, ...)
    create_button(solo_btn, MIXER_FAMILY, ...)
    create_slider(volume, MIXER_FAMILY, ...)
    create_slider(panpot, MIXER_FAMILY, ...)
    {...}
end on
```

This script creates a Family called MIXER_FAMILY with ID `0`. Then, it creates four UI Controls assigned to MIXER_FAMILY.

Now you can perform various actions over this Family. We are going to check out all of them in this section.

Additional notes on UI Controls management

Everytime a function regarding the GUI is called in `on init` callback, the values passed to the function override the default values.

For example:

```
on init
    create_family(FAMILY, 1)
    create_knob(knob, FAMILY, ...)
    shift_family(FAMILY, 60, 70)
end on
```

In this case, `shift_family` overrides the default values set in `create_knob` (10 and 15). If you call `reset_pos` or `reset_family_pos`, the knob position will be set at X=60 and Y=70. If you call `set_pos` or `reset_family_pos` from a different callback, obviously, the default value is not overriden.

8 Managing UI Families: setting data

8.1 shift_family

All the UI Controls assigned to a specific Family can be translated from their actual position by a given offset.

```
shift_family(<fam>,<offset_x>,<offset_y>)
```

The translation is **offset-based:** all the UI Controls will be moved from their **actual** position to another according to `offset_x` and `offset_y`.

This means that any previous translation will be kept and summed to the values in `offset_x` and `offset_y`.

The constant `ALL_FAMILIES` can be used to perform the action over all the UI Controls.

8.2 set_family_visibility(<fam>,<value>)

All the UI Controls assigned to a specific Family can be showed or hidden. *Value* must be replaced either with `VISIBLE` or `INVISIBLE`.

```
set_family_visibility(<fam>,<value>)
```

The constant `ALL_FAMILIES` can be used to perform the action over all the UI Controls.

9 Managing UI Families: retrieving data

9.1 set_family_values and get_family_values

All the values of the UI Controls assigned to a specific Family can be stored in and retrieved from a given array. You will have to pass only the array name to the function.

Be sure to use a destination array of proper dimensions in order not to lose informations. You will get a Warning from Kontakt in this case.

```
set_family_values(<fam>,<array>)
get_family_values(<fam>,<array>)
```

These are "twins" functions. See the example below.

```
on init
    {...}
end on
on ui_control (save_preset)

    get_family_values(FAMILY, preset_array) {Save all the values of a Family inside preset_array}
end on
on ui_control (recall_preset)

    set_family_values(FAMILY, preset_array){Load all the values of a Family with the values found in pr
end on
```

In this case, the array used in both the functions is the same. This is the correct way to use these functions in order to recall the values you expect to.

The constant `ALL_FAMILIES` can be used to perform the action over all the UI Controls.

9.2 get_family_ctrl_amount

This function returns the amount of UI Controls belonging to a specific Family.

```
get_family_ctrl_amount(<fam>)
```

This function has a return value but it's not an inline function, so to use it you have to use the following syntax:

```
ui_amount := get_family_ctrl_amount(FAMILY)
```

The constant `ALL_FAMILIES` can be used to get the amount of all UI Controls in your project.

10 Managing UI Families: resetting data

As it happens for individual UI Controls, you can reset all the values of Families to their defaults.

When you change any of these settings in `on init` callback using the functions `set_pos`, `set_value` etc., the new value is stored as default; this remains valid for Families too. So if you use `shift_family()` in `on init` callback, the new values will be set as defaults.

The same thing occurs if you perform this action on Families: for example, if in on init callback you call `set_family_pos`(), the default positions of all the UI Controls belonging to that Family are updated.

The following values are stored as defaults and can be recalled by the `reset_` functions:

- Position X
- Position Y
- Picture
- Text
- Font Type
- Font Alignment
- Default Value
- Default Visibility

10.1 reset_family_pos

All the UI Controls assigned to a specific Family can be reset to their default position.

```
reset_family_pos(<fam>)
```

The constant `ALL_FAMILIES` can be used to restore the default values for all the UI Controls.

10.2 reset_family_pics

All the UI Controls assigned to a specific Family can be reset to their default picture.

```
reset_family_pics(<fam>)
```

The constant `ALL_FAMILIES` can be used to restore the default values for all the UI Controls.

10.3 reset_family_text

All the UI Controls assigned to a specific Family can be reset to their default text.

```
reset_family_text(<fam>)
```

The constant `ALL_FAMILIES` can be used to restore the default values for all the UI Controls.

10.4 reset_family_font

All the UI Controls assigned to a specific Family can be reset to their default font type and alignment.

```
reset_family_font(<fam>)
```

The constant `ALL_FAMILIES` can be used to restore the default values for all the UI Controls.

10.5 reset_family_values

All the UI Controls assigned to a specific Family can be reset to their default values (if any).

```
reset_family_values(<fam>)
```

The constant `ALL_FAMILIES` can be used to restore the default values for all the UI Controls.

10.6 reset_family_visibility

All the UI Controls assigned to a specific Family can be reset to their default visibility status.

```
reset_family_visibility(<fam>)
```

The constant `ALL_FAMILIES` can be used to restore the default values for all the UI Controls.

11 Managing UI Families: advanced actions

Sometimes, developers might need to be able to organize UI elements in a very precise way. This process usually takes lots of time, lines of code, and much effort to make the whole thing work. For these reasons, we decided to introduce some very specific functions that might help.

11.1 place_on_grid

This function allows to place all the selected UI Controls belonging to a Family on a grid. You can choose the type of controls that will be placed on the grid (e.g., all the Sliders belonging to a specific Family), as well as choose the characteristics of the grid (height and width of each cell, number of UI Controls on each row).

Let's say you have a Family called MIXER_FAMILY, which embeds Sliders for volume control and Knobs for panpot. You can place all the Sliders on a grid and leave the Knobs to their default position. The operations on each Type of UI Control are independent. You will have to use place_on_grid twice, once on Sliders and once on Knobs.

You can operate on UI Controls globally by using the constants ALL_FAMILIES and ALL_TYPES if you need to. By doing so, you could set all the UI Controls belonging to a Family on the grid, or you can set all the Sliders currently existing in the GUI on the grid:

```
on init
    create_family(MIXER_FAMILY)
    {UI Controls declarations}
    place_on_grid(MIXER_FAMILY, SLIDERS, 10, 10, 70, 0, 8)
    {...}
end on
```

Keep in mind that `place_on_grid` overrides any placement made in declaration of each UI Control you are placing on the grid. To make this even more visible, we suggest - when declaring the controls to be placed on the grid - to set to 0 both `x` and `y` positions on these controls. Nothing changes even if these two values are set differently, as their position will be ovverriden: it's just a precaution and a "reminder" inside the script.

If you use `place_on_grid()` in `on init` callback, each UI Control's default position will be overriden too. If you use the function `reset_pos`(), in this case, all the UI Controls will be restored to their original placement on grid.

```
place_on_grid(<fam>,<type>,<start_x>,<start_y>,<col_w>,<row_h>,<ctrl_per_row>)
```

`family`	The name of the Family you want to perform actions on
`type`	Text you want to be displayed on the UI Control.
`start_x`	Position on X axis of upper-left corner of the FIRST control in the grid.
`start_y`	Position on Y axis of upper-left corner of the FIRST control in the grid.
`col_w`	Width of each column.
`row_h`	Height of each row.
`ctrl_per_row`	Amount of controls per each row.

11.2 restore_visibility_settings

This function allows to refresh the current visibility settings. Useful especially for testing and debugging purposes.

```
restore_visibility_settings(<fam>)
```

Part III

System constants

Kontakt comes with a wide variety of predefined system constants. Some of them though, seem to be missing. We decided to include them in Koala library in order to help the developer retrieving some standard, widely used values.

There are a few different types of system constants. We split them into different categories in order to help retrieving them in this Manual.

1 System constants

1.1 Default values

`0_dB_12`	0 dB value for knobs with +12 dB max. Equals to 630859.
`0_dB_24`	0 dB value for knobs with +24 dB max. Equals to 396484.

1.2 Boolean constants

`ON`
`OFF`
`YES`
`NO`
`TRUE`
`FALSE`

1.3 Names

`!NOTE_NAME[]`	String array containing the name of each note. E.g. `!NOTE_NAME[60]` = "C3".
`!CC_NAME[]`	String array containing the name of each MIDI CC. E.g. `!CC_NAME[1]` = "Mod. Wheel".

2 User Interface Constants

2.1 Families and types

When declaring a UI Control with one of `create_...` functions, an ID is assigned to the control itself according to **type** of Control you declared: Sliders have their own ID, Knobs have their own ID and so on. These IDs are stored in an array: each single UI Control type can be retrieved using `get_type` function (see *Managing UI Controls: retrieving data* section for more info about this function).

Obviously, it is not possible to set a UI Control Type after its declaration (there is no point of doing that).

Here is the list of available UI Control Type IDs. They are pretty self-explanatories. As these are all Constants, can be used in `if` statements, as arguments in `while` loops etc.

ALL_FAMILIES	Used in Families functions to perform an action over all Families.
ALL_TYPES	Used in Families functions to perform an action over all UI Controls Types.
SWITCHES	Used to perform an action on all the UI Controls of a specific Type.
BUTTONS	
KNOBS	
TEXT_LABELS	
PIC_LABELS	
MENUS	
FILE_SELECTOR	
VALUE_EDITS	
SLIDERS	
TABLES	
LEVEL_METERS	
WAVEFORMS	

The following example will display a different message according to the UI Control Type. In this case, the message will be “This UI Control is a Knob.”.

```
on init
    {...}
    if (get_type(get_ui_id(knob_1)) = KNOBS)
        message(“This UI Control is a Knob.”)
    else
        message(“This UI Control is not a knob.”)
    end if
end on
```

2.2 Specific UI Controls constants

LEFT	Text alignment settings for Font-based functions.
CENTER	
RIGHT	
VISIBLE	Visibility settings.
INVISIBLE	
LEFT_CH	UI Meter's left channel ID.
RIGHT_CH	UI Meter's right channel ID.
VERTICAL	UI Meter's orientation.
HORIZONTAL	

2.3 Persistence

PERSISTENT	Set persistence settings in UI Controls declarations.
NOT_PERSISTENT	

2.4 UI Text

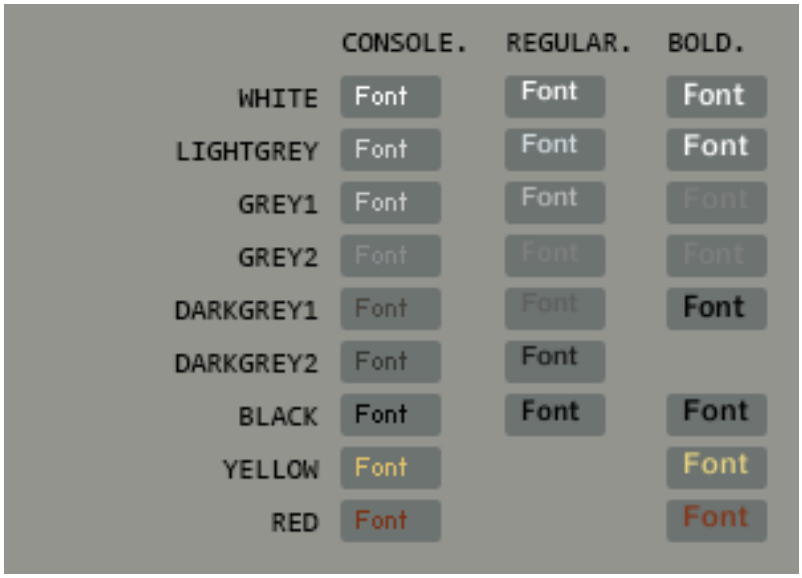
Kontakt features a default fonts set. To make things easier, we turned each font constant into an understandable and easy-to-remember constant name.

In Kontakt, there are three Fonts available: `CONSOLE`, `REGULAR` and `BOLD`. Each Font is available in a set of colours. The following table displays all the available fonts and colours.

Unfortunately, Kontakt handles fonts in a terrible way. Not all the colours are available for all the fonts, and there is no complete match between each font's available colours. The following table, though, should help the developer in choosing the desired font type and colour.

2.4.1 Usage of UI Fonts

The prototype of Font Constants is `FONT.COLOUR`. For instance, if you wish to get the Sans-serif, regular-weighted font in black, your Font constant will be `REGULAR.BLACK`.



2.4.2 UI Fonts alignments

If the desired UI Control supports this feature, its text can be aligned to the left, center or right.

```
LEFT
CENTER
RIGHT
```

The following example will display a Text Label with yellow console-like Font Style and center alignment.

```
on init
    create_text_label(text_label, FAMILY, 60, 60, 50, 18, ...
        “Text Lbl”, CONSOLE.YELLOW, CENTER, “”)
end on
```

2.5 Array-based operations

2.5.1 copy and copy_matrix modes

`NORMAL`	The position of each entry remains the same while copying.
`INVERT`	The position of each entry is inverted. The last value found in `source_arr` is the first value of `target_arr`.
`SORT_ASC`	After copying, sort `target_arr` from the smallest to the higehest value.
`SORT_DESC`	After copying, sort `target_arr` from the highest to the smallest value.

2.5.2 concat modes

`SEQUENTIAL`	Copy Array 1 first, then copy Array 2. The result would be `(ARR1[0], ARR1[1], ... ARR2[0], ARR2[1], ...)`
`MIXED`	Copy entries with the same index one after the other. The result would be `(ARR1[0], ARR2[0], ARR1[1], ARR2[1], ...)`

Part IV

Realtime Debugger

One of the most interesting features of this Library is the Realtime Debugger.

The Realtime Debugger is a set of functions which allow the developer to handle in a simple, extremely powerful way many annoying debugging processes.

Usually, when we need to display the value of an UI Control, the note played, the value of a variable, we use the KSP function `message()`. Operations like this cause the loss of a huge amount of time and weaken the script's readability, forcing the developer to distraction and causing loss of focus towards the real problems that need to be fixed.

To solve these issues, we implemented the Realtime Debugger.

Let's get a peek into this ultra-powerful tool.

1 Getting started

In order to be able to use the Realtime Debugger, the developer needs to enable the Instrument Debug Mode. To do so, the following piece of code must be inserted **at the very beginning of the Script,** before anything else, even before importing Koala Library. Therefore, the very inital part of the script will look like so:

```
SET_CONDITION(ENABLE_DEBUG)
import "Koala/Koala.ksp"
on init

    Koala.init
    create_instrument(510, "", "Script title")
    ...
```

`SET_CONDITION(ENABLE_DEBUG)` is a **preprocessor instruction** which tells the compiler to include some additional code when compiling. When this instruction is found, some parts of code are unhidden and included in the compilation process. When this instruction is not found, the Realtime Debugger's lines of code will be ignored by the compiler. You should expect a good amount of lines being added to your Script when using Realtime Debugger. Don't worry, Kontakt has no more issues when dealing with large scripts (this used to be an issue on older versions).

To disable Realtime Debugger, just comment or delete `SET_CONDITION(ENABLE_DEBUG)`.

1.1 How the GUI changes when Realtime Debugger is enabled

By default, we made some changes in how the GUI looks like when Realtime Debugger is enabled. This is necessary to give the developer more freedom in handling the debugging processes.

Here are the changes that occur when Realtime Debugger is enabled:

- **UI Height** is extended to 540 px. Any instruction given in `create_instrument()` function will be overriden, so you should expect your GUI to be taller than usual.
- **On the lower side of the Instrument** will be placed the Realtime Debugger. This tool consists in a few different windows displaying various informations in realtime regarding GUI, MIDI informations, custom variables and so on.

1.2 IMPORTANT: what to add to your Script to enable the Realtime Debugger

In order to enable the Realtime Debugger, the developer needs to add a few functions inside callbacks. These functions allow to retrieve the required data each time the callback is triggered. Be sure to add these functions exactly where they are supposed to be, otherwise you are likely to encounter issues.

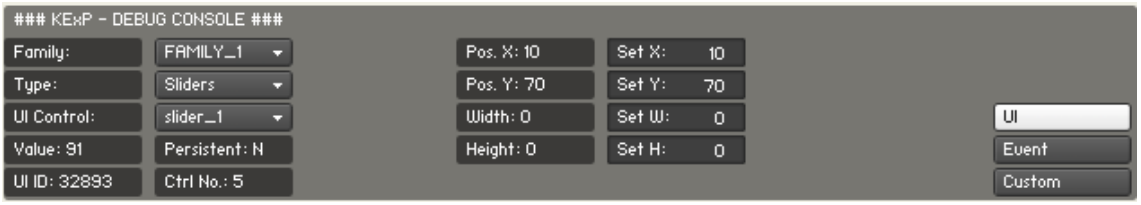
Callback	Code
`on note`	Add `DEBUG.on_note` as first line of the callback.
`on release`	Add `DEBUG.on_release` as first line of the callback.
`on controller`	Add `DEBUG.on_controller` as first line of the callback.
`on ui_control`	Add `DEBUG.on_ui_control(<control_name>)`as first line of each on `ui_control` callback. Replace `<control_name>` with the name of the UI Control.

```
on note

    DEBUG.on_note
    {...}

end on
on release

    DEBUG.on_release
    {...}

end on
on controller

    DEBUG.on_controller
    {...}

end on
on ui_control (knob_1)

    DEBUG.on_ui_control(knob_1)
    {...}

end on
```

2 UI Debugging

The first window in Realtime Debugger is called “UI”. This window displays detailed data about the selected UI Control.

Here is the list of the informations that will be displayed from the UI Window.



`Family`	Search filter. “UI Control” menu will display only the UI Controls belonging to the selected Family. This menu is updated if a UI Control is clicked (check out the next Section for more info).
`Type`	Search filter. ““UI Control” menu will display only the UI Controls belonging to the selected Type. This menu is updated if a UI Control is clicked (check out the next Section for more info).
`UI Control`	List of all the UI Controls created in GUI. Can be filtered by Family and Type in order to make searching easier.
`Value`	Current value of the selected UI Control.
`UI ID`	UI ID of the selected UI Control.
`Persistent`	“Y” means that the selected UI Control was set as `PERSISTENT` in its declaration. If it was set as `NOT_PERSISTENT` in its declaration, “N” will be displayed.
`Ctrl No.`	Number of the selected UI Control. Linked to the order you declared each UI Control in your Script.
`Pos. X`	Current position of the selected UI Control. Can be adjusted in realtime to verify the UI Control’s position. Any change made here will NOT be saved: when the desired value is set, it has to be manually copied to the Script.
`Pos. Y`	Current position of the selected UI Control. Can be adjusted in realtime to verify the UI Control’s position. Any change made here will NOT be saved: when the desired value is set, it has to be manually copied to the Script.
`Width`	Current width of the selected UI Control. Can be adjusted in realtime to verify the UI Control’s width. Any change made here will NOT be saved: when the desired value is set, it has to be manually copied to the Script.
`Height`	Current height of the selected UI Control. Can be adjusted in realtime to verify the UI Control’s height. Any change made here will NOT be saved: when the desired value is set, it has to be manually copied to the Script.

2.1 Retrieving UI data through UI Window

There are two ways to enable debugging for a certain UI Control:

- **Manual selection:** choose the desired UI Control from ““UI Control” dropdown menu. You can filter out the results by Family and Type in order to gain easier access to the desired UI Control.
- **Smart selection:** click on the desired UI Control. This is NOT applicable on non-clickable UI Controls, such as Labels, UI Meters etc. You need to use some additional code within each ui_control callback to enable this.

Let’s see in details these two solutions.

2.1.1 Manual selection

The Manual selection can be handled by using three dropdown menus, **Family, Type** and **UI Control.**

Family and **Type** are search filters: they can be used to include or exclude specific UI Controls. This should help handling very large scripts with many UI Controls. **UI Control** is populated by all the UI Controls created with Koala GUI functions (see Part II of this manual for more info). Each UI Control is displayed by its variable name. You can load any UI Control's current data by selecting it from this menu.

2.1.2 Smart selection

The Smart selection allows you to automatically select any clickable UI Control by just clicking on it.

In order to enable the Smart selection on a specific UI Control, **you need to include the following piece of code inside its `on ui_control` callback:**

```
DEBUG.on_ui_control(<name>)
```

If this line of code is found, each time the value of the UI Control is changed the Realtime Debugger will display the data of that particular UI Control.

Here is an example of this instruction used inside a macro to reduce the amount of lines of code to be written. **Pay attention to the usage of the Wild Card symbol inside the macro**. Each time a Knob is touched, the Realtime Debugger will display its data. **This is the right way to use this function.** Any other usage may return incorrect values.

```
SET_CONDITION(ENABLE_DEBUG)
import "Koala/Koala.ksp"
on init
    Koala.init
    create_family(FAMILY_1, 1)
    create_knob(knob_1, FAMILY_1, ...)
    create_knob(knob_2, FAMILY_1, ...)
    create_knob(knob_3, FAMILY_1, ...)
end on
macro knob_c(#n#)
    on ui_control (knob_#n#)
        DEBUG.on_ui_control(knob_#n#)
    end on
end macro
knob_c(1)
knob_c(2)
knob_c(3)
```

Every time you click on a UI Control, the Smart selection will update **Family** and **Type** menu to match the selected UI Control.s

2.2 Setting data through UI Window

One of the most annoying operations when dealing with KSP and UI Controls is placing them in the right position on your GUI. With custom GUI design, this can be a pain in the neck in terms of time and useless effort needed to make things work.

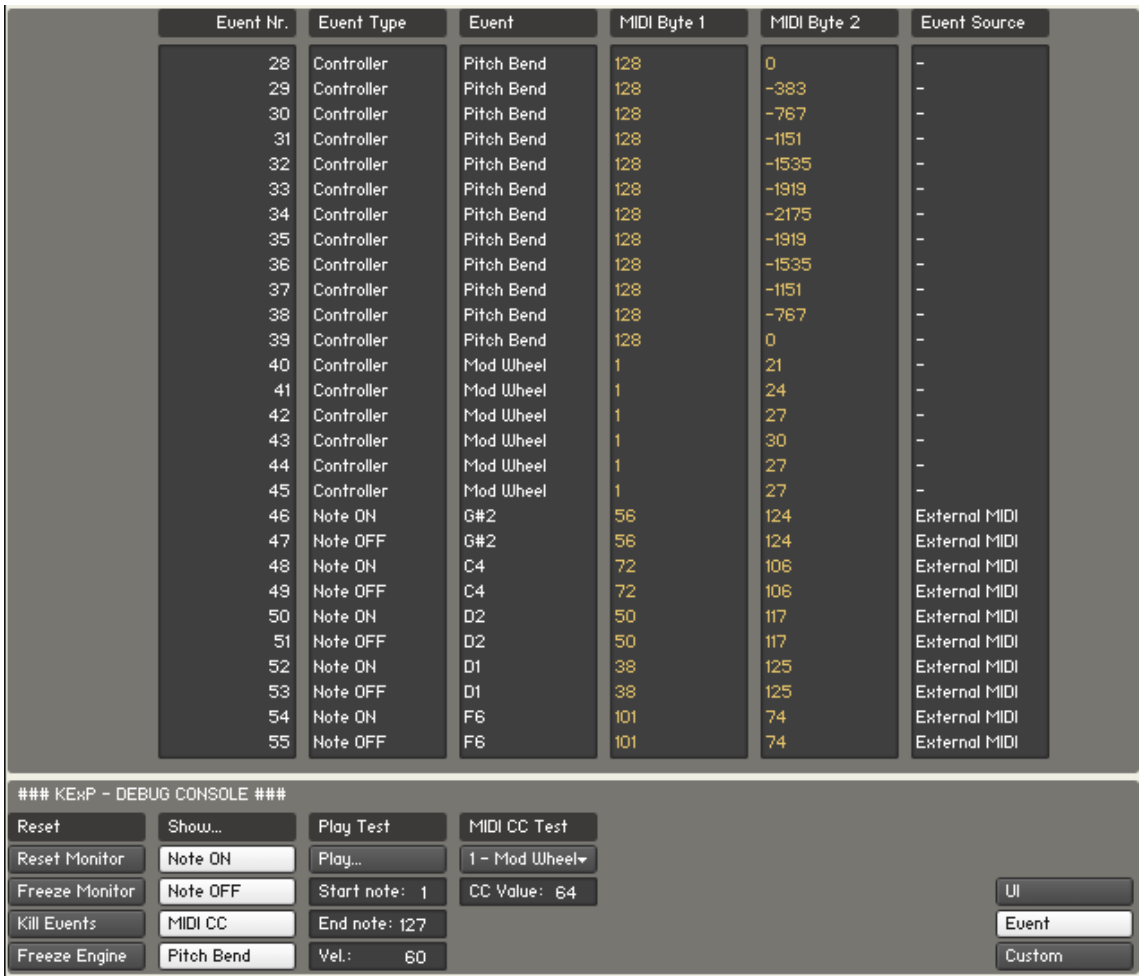
Koala Realtime Debugger allows the developer to preview any change in terms of UI Control's position and size. **Pos. X, Pos. Y, Width** and **Height** can be used for this purpose. Any of these changes will not be saved within the Script, so when you find the desired position and size it's recommended to immediately insert these data inside the Script.

3 MIDI and Event Debugging

Koala's Realtime Debugger features an extremely powerful MIDI and Event monitoring engine. The "Event" page features a detailed overview of every incoming MIDI message, along with the type of event, the value of MIDI bytes and, if available, the source of the event.

To enable MIDI and Event Debugging, you need to add the following lines at the very beginning of `on note`, `on release` and `on controller` callbacks:

```
on note

    DEBUG.on_note
    {...}

end on
on release

    DEBUG.on_release
    {...}

end on
on controller

    DEBUG.on_controller
    {...}

end on
```



3.1 MIDI Monitor: main view

The MIDI Monitor features six columns which display the following parameters:

`Event Nr.`	Displays MIDI events numbers. The MIDI Debugger is able to display the last 27 events before being reset.
`Event Type`	Can be “Note ON”, “Note OFF” or “Controller”.
`Event`	Details of the displayed Event. Controller Events show the name of the incoming MIDI CC; Note Events display the note name and octave.
`MIDI Byte 1`	First byte of incoming MIDI message. For Controller events, MIDI byte 1 is the MIDI CC number; for Note Events, MIDI byte 1 is the note number.
`MIDI Byte 2`	Second byte of incoming MIDI message. For Controller events, MIDI byte 2 is the value of MIDI CC; for Note Events, MIDI byte 2 is the velocity of the played note.
`Event Source`	Note events show here the source of the incoming MIDI event.

3.2 MIDI Monitor: Reset actions

The Reset section of MIDI and Events Debugger features four useful actions which can help in case of troubles.

`Reset Monitor`	Wipes the MIDI Monitor view.
`Freeze Monitor`	As long as this is active, the MIDI Monitor will not display any incoming MIDI event.
`Kill Events`	When pressed, three KSP actions will be executed: note_off() to all notes, MIDI CC 123 is called (“All Notes Off” command) and fade_out is performed on all the active Events. Any playing voice is stopped.
`Freeze Engine`	Any Note or Controller event is ignored. An `exit` command is executed at the beginning of `on note`, `on release` and `on controller` callback.

3.3 MIDI Monitor: Events filter

You can choose which Events are shown in the MIDI Monitor. This can be useful to keep the MIDI Monitor view clean and readable.

`Note ON`	If enabled, any Note ON Event is displayed.
`Note OFF`	If enabled, any Note OFF Event is displayed.
`MIDI CC`	If enabled, any MIDI CC Event is displayed.
`Pitch Bend`	If enabled, any Pitch Bend Event is displayed.

3.4 Play Test

The Play Test section allows the developer to play a sequence of notes to test if the Script is working properly.

`Play...`	Play the note sequence. When the note sequence is playing, this button will show the played note number and velocity.
`Start Note`	First note of the sequence.
`End Note`	Last note of the sequence.
`Vel.`	Velocity of the played note. This value can be set in realtime, even when the sequence is playing.

3.5 MIDI CC Test

The MIDI CC Test allows to set the value of any MIDI CC and check how the selected MIDI CC is changing in real time.

`MIDI CC Test Menu`	Select the desired MIDI CC. The menu displays MIDI CC numbers and their function according to MIDI standard.
`CC Value`	Current value of the selected MIDI CC. This value is updated every time you select a MIDI CC and every time the selected MIDI CC is changed.

4 Assignable UI Controls

Koala's Realtime Debugger's third page is called "Custom". This page features a good amount of assignable UI Controls which can help the developer to trigger actions, get and set values and perform basic debugging operations.

This page is useful for a few different purposes:

- Avoid using `message()` when you need to retrieve the value of a variable/function. By doing so, you can display more variables at a time and you leave the "message zone" free. This helps to find out if there are Script Warnings or such, which would be overlayed if a `message()` is used.
- Avoid declaring temporary buttons or value edits to trigger simple actions (saving arrays, displaying something etc.)

Assignable Labels	The left-hand side features 10 assignable Text Labels. These can be used to display any value, be it the value of a variable or the result of a function. To use an Assignable Label, you need to add the following piece of code in the desired place in your script (it must be inside a callback, obviously!):
	`DEBUG.show_value(<id>,<variable>)`
	`<id>` is the number of the Label. When the label is not assigned, its ID is displayed. When a label is assigned, the ID will be replaced by the name of the variable or function displayed.
	`<variable>` can be a variable, a position inside an array, a function, a constant, anything.
Assignable Buttons	You can assing up to 5 custom Buttons to perform up to 3 simple tasks, such as saving arrays, displaying values or triggering actions. To use an Assignable Button, you need to add the following piece of code at the very end of your script, outside any callback:
	`DEBUG.custom_button(<id>,<text>,<task_1>,<task_2>,<task_3>)`
	`<id>` is the number of the Button as displayed on the Button itself. When clicked, the ID will be replaced by the text inserted in `<text>`.
	`<task_1>`, `<task_2>` and `<task_3>` have to be replaced with a single line of code each. If you need to perform less than 3 tasks, just use the instruction `empty`.
Assignable Value Edits	You can assign up to 5 Value Edits to perform up to 3 simple tasks, such as setting a Parameter, triggering actions or such. To use an Assignable Value Edit, you need to add the following piece of code at the very end of your script, outside any callback:
	`DEBUG.custom_value_edit(<id>,<min>,<max>,<task_1>,<task_2>,<task_3>)`
	`<id>` is the number of the Value Edit as displayed on the Value Edit itself.
	`<min>` and `<max>` are the edge values of the Value Edit. The value of the Value Edit will be clipped to this couple of values.
	`<task_1>`, `<task_2>` and `<task_3>` have to be replaced with a single line of code each. If you need to perform less than 3 tasks, just use the instruction `empty`.

4.1 Details about Assignable UI Controls and examples

4.1.1 Assignable Labels

The following script allows you to display these values:

- on Assignable Label 1, display the played note;
- on Assignable Label 2, display the `ENGINE_UPTIME` of the played note;
- on Assignable Label 3, display the triggered MIDI CC;
- on Assignable Label 4, display the value of the MIDI CC.

```
on note
    DEBUG.show_value(1, EVENT_NOTE)
    DEBUG.show_value(2, ENGINE_UPTIME)
end on
on controller
    DEBUG.show_value(3, CC_NUM)
    DEBUG.show_value(4, CC[CC_NUM])
end on
```

4.1.2 Assignable Buttons

The following script allows you to save an array containing all the names of the MIDI notes and display a 2-slots message once this is done. Note that the task 3 is not used, so the instruction `empty` is used instead.

Each Assignable Button is declared, therefore it has a variable name. Each name follows this scheme:

```
DEBUG.custom_button_<id>
```

See the example below:

```
DEBUG.custom_button(1, "Save array", save_array(!NOTE_NAME, 1), message_2("Saved", ENGINE_UPTIME), empty
```

4.1.3 Assignable Value Edits

The following script allows you to set a Value Edit and display its value on an Assignable Label. Note that the tasks 2 and 3 are not used, so the instruction `empty` is used instead.

Each Assignable Value Edit is declared, therefore it has a variable name. Each name follows this scheme:

```
DEBUG.custom_ve_<id>
```

See the example below:

```
DEBUG.custom_value_edit(1, 0, 100, DEBUG.show_value(6, DEBUG.custom_ve_1), empty, empty)
```

Part V

Coding utilities and array operations

KSP lacks of some really important coding features which can make the developer save a lot of time. When a programmer comes to KSP from a different language, he is likely to be disappointed by the need of re-think a lot of extremely simple tasks which have their dedicated functions (or methods, or how you wish to call them) in almost any other scripting language.

Unfortunately, due to some theoretical limitations of KSP scripting language - such as the lack of dynamic arrays, just to pick one - it is still not possible to perform some tasks; what could be covered here, though, was covered in a pretty simple and clean way.

Let's check out all these features.

1 Array-based operations

Koala features a few functions dedicated to arrays. These function are dedicated to numeric arrays: trying to use them on string arrays is not possible and will return compilation errors.

In this section, we will discuss about one-dimensional arrays. Two-dimensional arrays, hereby referred to as **matrixes**, have a dedicated set of functions, so are in the next chapter.

None of the functions listed here below is an inline function, therefore you need to assign their values to a variable before being able to process the output value. See the example below:

```
value := max_value(array)
index := search(array, value)
```

1.1 max_value

This function returns the maximum value found inside an array.

```
max_value(<array>)
```

array	Name of the array to process.

If you want to retrieve the *index* of this value, you can intersect max_value with KSP native function search as in the example below. Keep in mind that two separate operations are needed to do so, as max_value is not an inline function.

```
value := max_value(array) {retrieve the maximum value and assign it to value}
index := search(array, value)
```

1.2 min_value

This function returns the maximum value found inside an array.

```
min_value(<array>)
```

array	Name of the array to process.

If you want to retrieve the *index* of this value, you can intersect min_value with KSP native function search as in the example below. Keep in mind that two separate operations are needed to do so, as min_value is not an inline function.

```
value := min_value(array) {retrieve the minimum value and assign it to value}
index := search(array, value)
```

1.3 count_entries

This function returns the amount of times a specifc value is found inside an array.

```
count_entries(<array>, <value>)
```

array	Name of the array to process.
value	Value to look for.

1.4 copy

This function allows to copy a one-dimensional array inside another one. Obviously, you have to make sure that both the source and the target array have the same size. In order to prevent errors and script warnings, though, we set up some simple safety checks which avoid wrong copying in case that the two arrays, for some reason, do have different size: the arrays will be copied only to the maximum index of the smallest array. For instance, if array_1 has size 80 and array_2 has size 155, the last index which will be copied is 79.

The copy can happen in four different modes according to how you'd like to sort the array itself. These modes are explained below.

```
copy(<source_arr>, <target_arr>, <mode>)
```

source_arr	Name of the array to copy from.
target_arr	Name of the array to copy to.
mode	Copy mode. There are four modes available.
	NORMAL: each entry has the same position on both the arrays.
	INVERT: the position of each entry is inverted. The last value found in source_arr is the first value of target_arr.
	SORT_ASC: after copying, sort target_arr from the smallest to the higehest value.
	SORT_DESC: after copying, sort target_arr from the highest to the smallest value.

1.5 reverse

This function allows to reverse the order of entries of an array, meaning that the first entry will be copied on the last position, the second entry on the second to last position and so on.

```
reverse(<array>)
```

array	Name of the array to process.

1.6 concat

This function allows to concatenate two arrays inside a third array. In order to prevent errors and script warnings, we set up some simple safety checks which allow not to overcome the edge of any of the arrays involved, preventing script warnings and compilation errors. You should make sure to have correctly sized arrays, though.

Two operating modes are available and will be explained below.

```
concat(<source_arr_1>, <source_arr_2>, <target_arr>, <mode>)
```

source_arr_1	Name of the first array to concatenate.
source_arr_2	Name of the second array to concatenate.
target_arr	Name of the array to copy to.
mode	Copy mode. There are two modes available.
	SEQUENTIAL: copy Array 1 first, then copy Array 2.
	MIXED: copy entries with the same index one after the other. The result would be (ARR1[0], ARR2[0], ARR1[1], ARR2[1], ...)

2 Matrix-based operations

Koala features a some functions dedicated to matrixes, also known as two-dimensional arrays. These function are dedicated to numeric matrixes: trying to use them on string arrays is not possible and will return compilation errors.

None of the functions listed here below is an inline function, therefore you need to assign their values to a variable before being able to process the output value. See the example below:

```
value := max_value(array)
index := search(array, value)
```

2.1 copy_matrix

This function allows to copy a two-dimensional array (a.k.a. "matrix") inside another one. Obviously, you have to make sure that both the source and the target matrix have the same size. In order to prevent errors and script warnings, though, we set up some simple safety checks which avoid wrong copying in case that the two matrixes, for some reason, do have different size: the matrixes will be copied only to the maximum index of the smallest matrix. For instance, if `matrix_1` has size `80*4` and `matrix_2` has size `155*4`, the last index which will be copied is `79*4`.

The copy can happen in four different modes according to how you'd like to sort the matrix itself. These modes are explained below.

```
copy_matrix(<source_mtx>, <target_mtx>, <source_cols>, row_to_copy, mode))
```

`source_arr`	Name of the array to copy from.
`target_arr`	Name of the array to copy to.
`mode`	Copy mode. There are four modes available.
	`NORMAL`: each entry has the same position on both the arrays.
	`INVERT`: the position of each entry is inverted. The last value found in `source_arr` is the first value of `target_arr`.
	`SORT_ASC`: after copying, sort `target_arr` from the smallest to the higehest value.
	`SORT_DESC`: after copying, sort `target_arr` from the highest to the smallest value.