KONTAKT 7

NATIVE INSTRUMENTS

THE FUTURE OF SOUND

Table of Contents

1. Disclaimer

Document authored by: Mario Krušelj, Jonas Körwer, Hannah Lockwood, Nikolas Jeroma, Yaron Eshkar, Elpiniki Pappa, Dinos Vallianatos, Adam Hanley

Software version: 7.6.0 (09/2023)

2. Welcome to KSP

Welcome to the Kontakt Script Processor (KSP) reference manual. KSP is the technology that powers thousands of instruments in our industry leading sampling platform - Kontakt. Its underlying scripting language defines how samples are being played, how MIDI information is used for interacting with the instrument, how the instrument looks and much more. From simple sequencers to large orchestral clusters with thousands of samples, KSP makes it all work.

This resource is a reference manual that covers every function, command, variable, callback and other element of the Kontakt scripting language. Where applicable, it also includes examples and short code snippets that demonstrate how a given function can be used. To learn more about Kontakt, refer to the Kontakt User Manual.

We hope you enjoy exploring KSP!

What's New in Kontakt 7.10

New Features

- Engine parameters for new Electric Grand EQ and preamp modes of EP Preamps effect.
- Engine parameters for Flex Envelope modulator (number of stages, loop start, loop end, per-stage time, level, slope).
- Built-in variables for system date and time.

Improved Features

- Waveforms in 3D mode of ui_wavetable widget now correctly display wavetable shaper adjustments.
- Effect edit panels in Instrument Edit mode now correctly update their UI upon certain engine parameter changes from KSP.

3. Callbacks

General Information

- A callback is a section within a script that is being "called" (i.e. executed) at certain times.
- All callbacks start with `on <callback-name>` and end with `end on`.
- Callbacks can be stopped by using the `exit` command.
- Each callback has a unique ID number which can be retrieved with `$NI_CALLBACK_ID`.
- You can query which callback triggered a function with `$NI_CALLBACK_TYPE` and the corresponding built-in constants.
- You can query which UI widget triggered a UI callback with `$NI_UI_ID` and the corresponding built-in constants.

Examples

```
function show_callback_type()
    if ($NI_CALLBACK_TYPE = $NI_CB_TYPE_NOTE)
        message("Function was called from note callback!")
    end if
    if ($NI_CALLBACK_TYPE = $NI_CB_TYPE_CONTROLLER)
        message("Function was called from controller callback!")
    end if
end function

on note
    call show_callback_type()
end on

on controller
    call show_callback_type()
end on
```

Query the callback type in a function.

See Also

exit

stop_wait()

on async_complete

`on async_complete`
Async complete callback, triggered after the execution of any command that is executed asynchronously, for example various loading or saving related commands.

Remarks

- To resolve synchronization issues, the commands listed in the "See Also" section below return unique IDs when being used.
- Upon completion of the command's action, the `on async_complete` callback gets triggered and the built-in variable `$NI_ASYNC_ID` is updated with the ID of the command that triggered the callback.
- If the command was completed successfully (for example if the file was found and successfully loaded), the internal value of `$NI_ASYNC_EXIT_STATUS` is set to **1**, otherwise it is **0**.

Examples

```
on init
    declare $load_midi_file_id
    declare ui_button $load_midi_file
end on

on ui_control ($load_midi_file)
    $load_midi_file_id := load_midi_file(<midi-file-path>)

    while ($load_midi_file_id # -1)
        wait(1)
    end while

    message("MIDI file loaded!")
end on

on async_complete
    if ($NI_ASYNC_ID = $load_midi_file_id)
        $load_midi_file_id := -1
    end if
end on
```

Example that pauses the ui_control callback until the file is loaded.

See Also

Load/Save Commands

set_voice_limit()

save_midi_file()

mf_insert_file()

mf_set_buffer_size()

mf_reset()

set_engine_par()

set_zone_par()

set_loop_par()

set_sample()

purge_group()
load_ir_sample()
Music Information Retrieval: MIR Commands
Built-in Variables and Constants: `$NI_ASYNC_EXIT_STATUS`, `$NI_ASYNC_ID`
Module Types and Subtypes: `$ENGINE_PAR_EFFECT_TYPE`, `$ENGINE_PAR_EFFECT_SUBTYPE`

on controller

on controller

MIDI controller callback, executed whenever a MIDI CC, Pitch Bend or Channel Pressure message is received.

Examples

```
on controller
    if (in_range($CC_NUM, 0, 127))
        message("CC Number: " & $CC_NUM & " - Value: " & %CC[$CC_NUM])
    else
        if ($CC_NUM = $VCC_PITCH_BEND)
            message("Pitch Bend - Value: " & %CC[$CC_NUM])
        end if

        if ($CC_NUM = $VCC_MONO_AT)
            message("Channel Pressure - Value: " & %CC[$CC_NUM])
        end if
    end if
end on
```

Query MIDI CC, Pitch Bend and Channel Pressure data.

See Also

set_controller()

ignore_controller

Events and MIDI: `%CC[]`, `$CC_NUM`, `$VCC_PITCH_BEND`, `$VCC_MONO_AT`

on init

`on init`

Initialization callback, executed when the script was successfully compiled without warnings or errors.

Remarks

The `on init` callback will be executed when:

- The "Apply" button is clicked in Script Editor.
- Script preset or an instrument are loaded.
- Kontakt's audio engine is restarted by clicking the "Restart Engine" button in the Monitor > Engine tab, or "!" button in Kontakt's header.
- Snapshot is loaded with `set_snapshot_type()` set to **0** or **2**
- Creator Tools is connected to a Kontakt instance, and GUI Designer's performance view file is resaved. In this case, the scripts are reinitialized automatically in order to update the performance view with most recent changes.

Examples

```
on init
    declare ui_button $Sync
    declare ui_menu $Time

    add_menu_item($Time, "16th", 0)
    add_menu_item($Time, "8th", 1)

    $Sync := 0   { sync is off by default, so hide menu }

    move_control($Time, 0, 0)
    move_control($Sync, 1, 1)

    make_persistent($Sync)
    make_persistent($Time)

    read_persistent_var ($Sync)

    if ($Sync = 1)
        move_control($time, 2, 1)
    else
        move_control($Time, 0, 0)
    end if
end on

on ui_control ($Sync)
    if ($Sync = 1)
        move_control($Time, 2, 1)
    else
        move_control($Time, 0, 0)
    end if
end on
```

init callback with read_persistent_var().

```
on init
    declare ui_button $Sync
    declare ui_menu $Time
    move_control($Sync, 1, 1)

    add_menu_item($Time, "16th", 0)
    add_menu_item($Time, "8th", 1)

    make_persistent($Sync)
    make_persistent($Time)
end on

function show_menu()
    if ($Sync = 1)
        move_control($Time, 2, 1)
    else
        move_control($Time, 0, 0)
    end if
end function

on persistence_changed
    call show_menu()
end on

on ui_control ($Sync)
    call show_menu()
end on
```

The same script functionality, now with persistence_changed callback. This is the preferred method to use, especially when utilizing snapshots!

See Also

make_persistent()

read_persistent_var()

on persistence_changed

on listener

`on listener`

Listener callback, executed at definable time intervals or whenever a transport command is received.

Remarks

- The listener callback is executed at time intervals defined with the `set_listener()` command. It can also react to the host's transport start and stop commands. This makes it the ideal callback for anything tempo-synced, like sequencers, arpeggiators, MIDI file players etc.
- In some situations (like tempo changes within the host), ticks can be occasionally left out.

Examples

```
on init
    declare ui_knob $Test (0, 99, 1)
    declare $direction
    declare $tick_counter

    set_listener($NI_SIGNAL_TIMER_MS, 10000)
end on

on listener
    if ($NI_SIGNAL_TYPE = $NI_SIGNAL_TIMER_MS)
        if ($direction = 0)
            inc($tick_counter)
        else
            dec($tick_counter)
        end if

        $Test := $tick_counter

        if ($tick_counter = 99)
            $direction := 1
        end if

        if ($tick_counter = 0)
            $direction := 0
        end if
    end if
end on
```

Not useful as such, but nice to look at.

See Also

set_listener()

change_listener_par()

Callbacks and UI: `$NI_SIGNAL_TYPE`, `$NI_SONG_POSITION`

on note

`on note`
Note callback, executed whenever a MIDI Note On message is received.

Examples

```
on note
    message("Note Number: " & $EVENT_NOTE & " - Velocity: " & $EVENT_VELOCITY)
end on
```

Query note properties.

See Also

on release

ignore_event()

set_event_par()

get_event_par()

Events and MIDI: `$EVENT_NOTE`, `$EVENT_VELOCITY`, `$EVENT_ID`

on persistence_changed

on persistence_changed
Executed after the `on init` callback, or whenever a snapshot has been loaded.

Remarks

- This callback is called whenever the persistent variables change in an instrument, i.e. it is always executed after the `on init` callback, and/or upon loading a snapshot.

Examples

```
on init
    set_snapshot_type(1)    { init callback not executed upon snapshot loading }
    reset_ksp_timer

    declare $init_flag      { 1 if init callback has been executed, 0 otherwise }
    $init_flag := 1

    declare ui_label $label (2, 2)
    set_text($label, "Init callback " & $KSP_TIMER)
end on

function add_text()
    add_text_line($label, "Persistence changed callback " & $KSP_TIMER)
end function

on persistence_changed
    if ($init_flag = 1)    { instrument has been loaded }
        call add_text()
    else    { snapshot has been loaded }
        set_text($label, "Snapshot loaded!")
    end if

    $init_flag := 0
end on
```
Query if a snapshot or instrument has been loaded. This also demonstrates the ability to call functions upon initialization, i.e. the persistence callback acts as an extension to the init callback.

See Also

on init

read_persistent_var()

set_snapshot_type()

on pgs_changed

on pgs_changed
Executed whenever any `pgs_set_key_val()` command is executed in any script slot.

Remarks

- PGS stands for Program Global Storage and is a means of communication between script slots. See the chapter on PGS for more details.

Examples

```
on init
    pgs_create_key(FIRST_KEY, 1)    { defines a key with 1 element }
    pgs_create_key(NEXT_KEY, 128)   { defines a key with 128 elements }

    declare ui_button $Push
end on

on ui_control($Push)
    pgs_set_key_val(FIRST_KEY, 0, 70 * $Push)
    pgs_set_key_val(NEXT_KEY, 0, 50 * $Push)
    pgs_set_key_val(NEXT_KEY, 127, 60 * $Push)
end on
```

Pressing the button...

```
on init
    declare ui_knob $First (0, 100, 1)
    declare ui_table %Next[128] (5, 2, 100)
end on

on pgs_changed
    { checks if FIRST_KEY and NEXT_KEY have been declared }
    if (pgs_key_exists(FIRST_KEY) and pgs_key_exists(NEXT_KEY))
        $First := pgs_get_key_val(FIRST_KEY, 0)
        %Next[0] := pgs_get_key_val(NEXT_KEY, 0)
        %Next[127] := pgs_get_key_val(NEXT_KEY, 127)
    end if
end on
```

...will change the controls in this example, regardless of the script slot order.

See Also

PGS: `pgs_create_key()`, `pgs_set_key_val()`, `pgs_get_key_val()`

on poly_at

on poly_at
Polyphonic aftertouch callback, executed whenever a MIDI Polyphonic Aftertouch message is received.

Examples

```
on init
    declare %note_id[128]
end on

on note
    %note_id[$EVENT_NOTE] := $EVENT_ID
end on

on poly_at
    change_tune(%note_id[$POLY_AT_NUM], %POLY_AT[$POLY_AT_NUM] * 1000, 0)
end on
```

A simple poly aftertouch to pitch implementation.

See Also

Events and MIDI: `%POLY_AT[]`, `$POLY_AT_NUM`, `$VCC_MONO_AT`

on release

```
on release
```

Release callback, executed whenever a MIDI Note Off message is received.

Examples

```
on init
    declare polyphonic $new_id
end on

on release
    wait(1000)
    $new_id := play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, 100000)
    change_vol($new_id, -24000, 1)
end on
```

Creating an artificial release triggered noise.

See Also

on note

ignore_event()

get_event_par(): `$EVENT_PAR_REL_VELOCITY`

on rpn/nrpn

`on rpn/nrpn`

RPN and NRPN callbacks, executed whenever a MIDI RPN or NRPN (registered/non-registered parameter number) message is received.

Examples

```
on rpn
    select ($RPN_ADDRESS)
        case 0
            message("Pitch Bend Sensitivity" & " - Value: " & $RPN_VALUE)
        case 1
            message("Fine Tuning" & " - Value: " & $RPN_VALUE)
        case 2
            message("Coarse Tuning" & " - Value: " & $RPN_VALUE)
    end select
end on
```

Query standard RPN messages.

See Also

on controller

set_rpn()/set_nrpn()

msb()

lsb()

Events and MIDI: $RPN_ADDRESS, $RPN_VALUE

on ui_control

```
on ui_control (<ui-widget-name>)
```

UI callback, executed whenever the user interacts with a particular UI widget.

Examples

```
on init
    declare ui_knob $Knob (0, 100, 1)
    declare ui_button $Button
    declare ui_switch $Switch
    declare ui_table %Table[10] (2, 2, 100)
    declare ui_menu $Menu
    declare ui_value_edit $VEdit (0, 127, 1)
    declare ui_slider $Slider (0, 100)

    add_menu_item($Menu, "Entry 1", 0)
    add_menu_item($Menu, "Entry 2", 1)
end on

on ui_control ($Knob)
    message("Knob" & " (" & $ENGINE_UPTIME & ")")
end on

on ui_control ($Button)
    message("Button" & " (" & $ENGINE_UPTIME & ")")
end on

on ui_control ($Switch)
    message("Switch" & " (" & $ENGINE_UPTIME & ")")
end on

on ui_control (%Table)
    message("Table" & " (" & $ENGINE_UPTIME & ")")
end on

on ui_control ($Menu)
    message("Menu" & " (" & $ENGINE_UPTIME & ")")
end on

on ui_control ($VEdit)
    message("Value Edit" & " (" & $ENGINE_UPTIME & ")")
end on

on ui_control ($Slider)
    message("Slider" & " (" & $ENGINE_UPTIME & ")")
end on
```

Various UI controls and their corresponding UI callbacks.

See Also

on ui_controls

on ui_update

Callbacks And UI: `$NI_UI_ID`

Control Parameters: `$CONTROL_PAR_CUSTOM_ID`, `$CONTROL_PAR_TYPE`

on ui_controls

`on ui_controls`

Global UI callback, executed whenever the user interacts with *any* particular UI widget.

Remarks

- When interacting with a particular UI widget, this callback will always be executed first, then the individually declared `on ui_control` callback (if it exists).
- Intended use of this callback type is to provide a more general approach of dealing with UI widgets, and also allowing for a sort of separation between the UI widgets (the Controller) and actual parameter data (the Model), according to MVC methodology (where UI widget covers the Controller and View parts). This results in UI widgets generally not needing to be made persistent, instead all parameter data could be held in a single persistent array (or two, in case of mixing integer and real values in the Model). As a consequence of not requiring persistence, variable names of UI widgets - and even widget types themselves - can change without affecting the state of the data in the script.

Examples

```
on init
    message("")
    set_snapshot_type(3)

    declare const $NUM_PARAMS := 4
    declare const $FILT_SLOT  := 0

    declare $i

    declare $uiid
    declare $param_id

    declare @str

    { this array actually holds our parameter values, not the widgets,
      and it's the only persistent variable! }
    declare %params[$NUM_PARAMS] := (1000000, 0, 630000, 500000)

    make_persistent(%params)

    declare const $FILT_CUT := 0
    declare const $FILT_RES := 1

    declare const $OUT_VOL  := 2
    declare const $OUT_PAN  := 3

    declare ui_slider $Cut (0, 1000000)
    declare ui_slider $Res (0, 1000000)

    declare ui_slider $Vol (0, 1000000)
    declare ui_slider $Pan (0, 1000000)

    { link between parameter IDs and UI IDs }
    declare %pid_to_uid[$NUM_PARAMS]
    %pid_to_uid[$FILT_CUT] := get_ui_id($Cut)
    %pid_to_uid[$FILT_RES] := get_ui_id($Res)
    %pid_to_uid[$OUT_VOL]  := get_ui_id($Vol)
    %pid_to_uid[$OUT_PAN]  := get_ui_id($Pan)
    { this is a quick example of course, but you can easily see how this can be used
      to scale up to larger instruments, since UI IDs can get out of order, but this
      doesn't matter when $CONTROL_PAR_CUSTOM_ID holds the "pointer" to the
parameter }
    $i := 0
    while ($i < $NUM_PARAMS)
        set_control_par(%pid_to_uid[$i], $CONTROL_PAR_CUSTOM_ID, $i)
        { set the default values for first time script apply case, optionally }
        set_control_par(%pid_to_uid[$i], $CONTROL_PAR_VALUE, %params[$i])

        inc($i)
    end while

    { set up a filter in group 1 so that we have something to work with }
    set_engine_par($ENGINE_PAR_EFFECT_TYPE, $EFFECT_TYPE_FILTER, 0, $FILT_SLOT, -1)
end on

function SetParam()
    select ($param_id)
        case $FILT_CUT
            set_engine_par($ENGINE_PAR_CUTOFF, %params[$param_id], 0, $FILT_SLOT,
-1)
        case $FILT_RES
            set_engine_par($ENGINE_PAR_RESONANCE, %params[$param_id], 0,
$FILT_SLOT, -1)
        case $OUT_VOL
            set_engine_par($ENGINE_PAR_VOLUME, %params[$param_id], 0, -1, -1)
        case $OUT_PAN
            set_engine_par($ENGINE_PAR_PAN, %params[$param_id], 0, -1, -1)
    end select
end function

function SetLabel()
    select ($param_id)
        case $FILT_CUT
            @str := get_engine_par_disp($ENGINE_PAR_CUTOFF, 0, $FILT_SLOT, -1) & "
Hz"
        case $FILT_RES
            @str := get_engine_par_disp($ENGINE_PAR_RESONANCE, 0, $FILT_SLOT, -1) &
" %"
        case $OUT_VOL
            @str := get_engine_par_disp($ENGINE_PAR_VOLUME, 0, -1, -1) & " dB"
        case $OUT_PAN
            @str := get_engine_par_disp($ENGINE_PAR_PAN, 0, -1, -1)
    end select

    set_control_par_str(%pid_to_uid[$param_id], $CONTROL_PAR_LABEL, @str)
end function

on persistence_changed
    { easy refreshing of parameter values and automation labels! }
    $i := 0
    while ($i < $NUM_PARAMS)
        $param_id := $i

        call SetParam()
        call SetLabel()

        set_control_par(%pid_to_uid[$i], $CONTROL_PAR_VALUE, %params[$i])

        inc($i)
    end while
end on

{ that's all you need, no need to write individual callbacks anymore! }
on ui_controls
    $param_id := get_control_par($NI_UI_ID, $CONTROL_PAR_CUSTOM_ID)
    %params[$param_id] := get_control_par($NI_UI_ID, $CONTROL_PAR_VALUE)

    call SetParam()
    call SetLabel()
end on
```
A small showcase of benefits provided by the `on ui_controls`callback.

See Also

on ui_control

User-defined Functions

Callbacks And UI: `$NI_UI_ID`

Control Parameters: `$CONTROL_PAR_CUSTOM_ID`, `$CONTROL_PAR_TYPE`

on ui_update

`on ui_update`

UI update callback, executed with every GUI change in Kontakt.

Remarks

- This callback can be executed very often in Kontakt, so use it with caution!

Examples

```
on init
    declare ui_knob $Volume (0, 1000000, 1)

    set_knob_unit($Volume, $KNOB_UNIT_DB)
    set_knob_defval($Volume, 630000)

    $Volume := get_engine_par($ENGINE_PAR_VOLUME, -1, -1, -1)
    set_knob_label($Volume, get_engine_par_disp($ENGINE_PAR_VOLUME, -1, -1, -1))
end on

on ui_update
    $Volume := get_engine_par($ENGINE_PAR_VOLUME, -1, -1, -1)
    set_knob_label($Volume, get_engine_par_disp($ENGINE_PAR_VOLUME, -1, -1, -1))
end on

on ui_control ($Volume)
    set_engine_par($ENGINE_PAR_VOLUME, $Volume, -1, -1, -1)
    set_knob_label($Volume, get_engine_par_disp($ENGINE_PAR_VOLUME, -1, -1, -1))
end on
```

Mirroring instrument volume with a KSP control.

See Also

on ui_control

on ui_controls

4. Variables

General Information

- All user-defined variables must be declared in the `on init` callback.
- Variable names may contain only alphanumerical characters (`0-9`, `a-z`, `A-Z`) and underscore (_).
- Please do **NOT** create variables with the following prefixes, as these are used for internal variables and constants:
    - `$NI_`
    - `$CONTROL_PAR_`
    - `$EVENT_PAR_`
    - `$ENGINE_PAR_`
    - `$ZONE_PAR_`
    - `$LOOP_PAR_`

$ (integer variable)

```
declare $<variable-name>
```

Declares a user-defined variable to store a single, 32-bit signed integer value.

Remarks

- Valid numbers that can be stored in an integer variable are in range **-2147483648 ... 2147483647**.

Examples

```
on init
    declare $test
    $test := -1
end on
```

Creating a variable.

```
on init
    declare $test := -1
end on
```

Creating a variable similar to above, but with in-line value initialization.

See Also

on init

make_persistent()

read_persistent_var()

real()

int()

% (integer array)

```
declare %<variable-name>[<num-of-elements>]
```

Declares a user-defined array to store multiple 32-bit signed integer values at specific indices.

Remarks

- The maximum size of arrays is **1000000** indices.
- The number of elements must be defined with a constant value, variables cannot be used for this purpose.
- It is possible to initialize an array with one value - refer to the second example below.
- If initializing an array with several values, but number of initializers is less than array size, the last initializer will be used to initialize the rest of the array.
- Valid numbers that can be stored in an integer array are in range **-2147483648 ... 2147483647**.

Examples

```
on init
    declare %presets[10 * 8] := ( ...
    {  1 }  8, 8, 8, 0,   0, 0, 0, 0, ...
    {  2 }  8, 8, 8, 8,   0, 0, 0, 0, ...
    {  3 }  8, 8, 8, 8,   8, 8, 8, 8, ...
    {  4 }  0, 0, 5, 3,   2, 0, 0, 0, ...
    {  5 }  0, 0, 4, 4,   3, 2, 0, 0, ...
    {  6 }  0, 0, 8, 7,   4, 0, 0, 0, ...
    {  7 }  0, 0, 4, 5,   4, 4, 2, 2, ...
    {  8 }  0, 0, 5, 4,   0, 3, 0, 0, ...
    {  9 }  0, 0, 4, 6,   7, 5, 3, 0, ...
    { 10 }  0, 0, 5, 6,   4, 4, 3, 2  )
end on
```

Creating an array for storing preset data.

```
on init
    declare %presets[10 * 8] := (4)
end on
```

Quick way of initializing the same array with a specific value.

See Also

Array Commands

Group Commands

make_persistent()

~ (real variable)

```
declare ~<variable-name>
```

Declares a user-defined variable to store a single, 64-bit (double precision) real value.

Remarks

- Real numbers must always be defined with a decimal point, even if the number is a whole number. For example **2.0** should be used instead of only **2**.

Examples

```
on init
    declare ~test
    ~test := 0.5
end on
```

Creating a variable.

```
on init
    declare ~test := 0.5
end on
```

Creating a variable similar to the above, but with in-line value initialization.

See Also

on init

make_persistent()

read_persistent_var()

real()

int()

? (real array)

`declare ?<variable-name>[<num-of-elements>]`

Declares a user-defined array to store multiple 64-bit (double precision) real values at specific indices

Remarks

- The maximum size of arrays is **1000000** indices.
- The number of elements must be defined with a constant integer value, regular variables cannot be used for this purpose.
- It is possible to initialize an array with one value - refer to the second example below.
- If initializing an array with several values, but number of initializers is less than array size, the last initializer will be used to initialize the rest of the array.
- `array_equal()` and `search()` commands do not work with real arrays.

Examples

```
on init
    declare ?presets[5 * 4] := ( ...
    { 1 }   1.0, 1.0, 1.0, 1.0, ...
    { 2 }   0.5, 0.7, 0.1, 0.5, ...
    { 3 }   1.0, 0.6, 0.6, 0.2, ...
    { 4 }   0.0, 0.0, 0.5, 0.3, ...
    { 5 }   0.0, 1.0, 0.4, 0.1  )
end on
```

Creating an array for storing preset data.

```
on init
    declare ?presets[10 * 8] := (1.0)
end on
```

Quick way of initializing the same array with a specific value.

See Also

Array Commands

Group Commands

make_persistent()

@ (string variable)

`declare @<variable-name>`
Declares a user-defined string variable to store text.

Remarks

- You cannot declare and in-line initialize a string variable as you can with an integer variable.
- The maximum length of text that can be stored in a string variable is **320** characters.

Examples

```
on init
    declare @text
    @text := "Last received note number played or released: "
end on

on note
    message(@text & $EVENT_NOTE)
end on

on release
    message(@text & $EVENT_NOTE)
end on
```

Use string variables to display long text.

See Also

! (string array)

ui_text_edit

make_persistent()

! (string array)

```
declare !<array-name>[<num-of-elements>]
```

Declares a user-defined string array to store text strings at specified indices.

Remarks

- The maximum size of arrays is **1000000** indices.
- Just like with string variables, the contents of a string array cannot be in-line initialized on the same line as the declaration.
- The maximum length of a string at any given indice is **320** characters.
- Please be aware that large string arrays with long strings can require a lot of memory if they are made persistent (roughly, one character = 1 byte, 320 characters = 320 bytes, 1000000 strings with 320 characters = **305 megabytes**)

Examples

```
on init
    declare $count

    declare !note[12]
    !note[0] := "C"
    !note[1] := "C#"
    !note[2] := "D"
    !note[3] := "D#"
    !note[4] := "E"
    !note[5] := "F"
    !note[6] := "F#"
    !note[7] := "G"
    !note[8] := "G#"
    !note[9] := "A"
    !note[10] := "Bb"
    !note[11] := "B"

    declare !name [128]

    while ($count < 128)
        !name[$count] := !note[$count mod 12] & (($count / 12) - 2)

        inc($count)
    end while
end on

on note
    message("Note played: " & !name[$EVENT_NOTE])
end on
```

Creating a string array with all MIDI note names.

See Also

@ (string variable)

const $ (constant integer)

```
declare const $<name>
```

Declares a user-defined constant to store a single integer value.

Remarks

- As the name implies, the value of constant variables can only be read, not changed.
- It is quite common (and recommended) to write the names of constants in uppercase.
- Valid numbers that can be stored in an integer constant are in range **-2147483648 ... 2147483647**.

Examples

```
on init
    declare const $NUM_PRESETS := 10
    declare const $NUM_PARAMETERS := 5

    declare %preset_data[$NUM_PRESETS * $NUM_PARAMETERS]
end on
```

Creating constants – useful when creating preset arrays.

See Also

on init

const ~ (real constant)

```
declare const ~<name>
```

Declares a user-defined constant to store a single, 64-bit (double precision) real value.

Remarks

- As the name implies, the value of constant variables can only be read, not changed.
- It is quite common (and recommended) to write the names of constants in uppercase.

Examples

```
on init
    declare const ~BIG_NUMBER := 100000.0
    declare const ~SMALL_NUMBER := 0.00001
end on
```

Just a couple of real constants, man.

See Also

on init

polyphonic $ (polyphonic integer)

```
declare polyphonic $<variable-name>
```

Declares a user-defined polyphonic variable to store a single integer value per note event.

Remarks

- A polyphonic variable acts as a unique variable for each executed note event, avoiding conflicts in callbacks that are executed in parallel, for example when using `wait()`.
- A polyphonic variable retains its value in the `on release` callback of the corresponding note.
- Polyphonic variables need much more memory than regular variables - since the note event queue in Kontakt supports a maximum of **8192** events, 32 bits * 8192 = **32 kilobytes**.
- Polyphonic variables can only be used within note and release callbacks.

Examples

```
on init
    declare polyphonic $a
    { declare $a }
end on

on note
    ignore_event($EVENT_ID)

    $a := 0
    while ($a < 13 and $NOTE_HELD = 1)
        play_note($EVENT_NOTE + $a, $EVENT_VELOCITY, 0, $DURATION_QUARTER / 2)

        inc($a)

        wait($DURATION_QUARTER)
    end while
end on
```

To hear the effect of the polyphonic variable, play and hold an octave: both notes will ascend chromatically. Then make ***$a*** *a normal variable and play the octave again:* ***$a*** *will be shared by both executed callbacks, thus both notes will ascend in larger intervals.*

```
on init
    declare $counter
    declare polyphonic $polyphonic_counter
end on

on note
    message($polyphonic_counter & "  " & $counter)
    inc($counter)
    inc($polyphonic_counter)
end on
```

Since a polyphonic variable is always unique per callback, ***`$polyphonic_counter`*** *will always be 0 in the displayed message.*

make_instr_persistent()

`make_instr_persistent(<variable>)`
Retains the value of a variable within the instrument only.

Remarks

- `make_instr_persistent()` is similar to `make_persistent()`, however the value of a variable is only saved with the instrument, not with snapshots. It can be used to prevent UI elements from being changed when loading snapshots.

Examples

```
on init
    set_snapshot_type(1)    { init callback not executed upon snapshot loading }

    declare ui_knob $knob_1 (0, 2, 1)
    set_text($knob_1, "Pers")
    make_persistent($knob_1)

    declare ui_knob $knob_2 (0, 2, 1)
    set_text($knob_2, "Inst Pers")
    make_instr_persistent($knob_2)

    declare ui_knob $knob_3 (0, 2, 1)
    set_text($knob_3, "Not Pers")
end on
```

The second knob will not be changed when loading snapshots.

See Also

read_persistent_var()

make_persistent()

set_snapshot_type()

make_persistent()

```
make_persistent(<variable>)
```

Retainsy the value of a variable with the instrument and snapshot.

Remarks

- The state of the variable is saved not only with the instrument,multi or host chunk, but also when a script is saved as a Kontakt preset (.nkp file).
- The state of the variable is read at the end of the init callback. To load a stored value manually within the init callback, insert `read_persistent_var()` before using the stored value.
- You can also use the `on persistence_changed` callback for retrieving the values of persistent variables.
- When updating script code by replacing old code with new one, the values of persistent variables that have identical names will be retained.
- Sometimes, when working on more complex scripts, you might want to flush the values of persistent variables by resetting the script. You can do this by loading the **- INIT Script -** preset from the Script Editor's Preset menu, then applying your code again.

Examples

```
on init
    declare ui_knob $Preset (1, 10, 1)
    make_persistent($Preset)
end on
```

User interface elements, such as knobs, should usually retain their value when reloading the instrument.

See Also

read_persistent_var()

on persistence_changed

make_instr_persistent()

read_persistent_var()

`read_persistent_var(<variable>)`

Instantly reloads the value of a variable that was saved via the `make_persistent()` command.

Remarks

- This command can only be used within the `on init` callback.
- You can also use the `on persistence_changed` callback for retrieving the values of persistent variables.

Examples

```
on init
    declare ui_label $label (1, 1)
    declare ui_button $button
    set_text($button, "$a := 10000")

    declare $a
    make_persistent($a)
    { read_persistent_var($a) }
    set_text($label, $a)
end on

on ui_control ($button)
    $a := 10000
    set_text($label, $a)
end on
```

After applying this script, click on the button and then save and close the NKI. After reloading it, the label will display 0 because the value of ***`$a`*** *is initialized at the very end of the init callback. Now remove the comment around* ***`read_persistent_var`*** *and apply the script again to see the difference.*

See Also

make_persistent()

on persistence_changed

watch_var()

`watch_var(<variable>)`
Sends an event to the Creator Tools KSP Log for every change of the watched variable's value.

Remarks

- This command can only be used within the `on init` callback.
- This command has no effect on Kontakt's status bar – the events only appear in Creator Tools.
- This command does not work with built-in variables (`$ENGINE_UPTIME`, `$NOTE_HELD`, `$KSP_TIMER`, etc.)

Examples

```
on init
    declare $intVar

    watch_var($intVar)
    make_persistent($intVar)
end on

on note
    $intVar := $EVENT_VELOCITY
end on
```

Try playing some notes while having Creator Tools running. Make sure you have the KSP Variables or KSP Log panel focused.

watch_array_idx()

`watch_array_idx(<array>, <array_idx>)`

Sends an event to the Creator Tools KSP Log for every change of the watched array index's value.

Remarks

- This command can only be used within the `on init` callback.
- This command has no effect on Kontakt's status bar – the events only appear in Creator Tools.
- This command does not work with built-in array variables (`%KEY_DOWN`, `%CC`, `%EVENT_PAR`, etc.).

Examples

```
on init
    declare %mykeys[128]

    watch_array_idx(%mykeys, 60)
    watch_array_idx(%mykeys, 61)
    watch_array_idx(%mykeys, 62)
    watch_array_idx(%mykeys, 63)
    watch_array_idx(%mykeys, 64)

    declare ui_button $Save
    declare ui_button $Load
end on

on note
    %mykeys[$EVENT_NOTE] := $EVENT_VELOCITY
end on

on ui_control($Save)
    save_array(%mykeys, 0)
end on

on ui_control($Load)
    load_array(%mykeys, 0)
end on
```

Try playing some notes or clicking on the save and load buttons while having Creator Tools running. Make sure you have the KSP Variables or KSP Log panel focused.

5. Arithmetic Commands & Operators

Basic Operators

The following operators work on both integers and real numbers.	
`x := y`	Assignment (the value of `y` is assigned to `x`).
`x + y`	Addition.
`x - y`	Subtraction.
`x * y`	Multiplication.
`x / y`	Division.
`-x`	Negative value.
`abs(x)`	Absolute value.
`signbit(x)`	Sign bit (returns **1** if the number is negative, **0** otherwise).
`sgn(x)`	Signum function (returns **-1** if the number is negative, **0** if it's zero, **1** if it's positive).

Integer Number Commands

The following commands and operators can only be performed on integer variables and values.

`inc(x)`

Increments an expression by 1 (`x := x + 1`).

`dec(x)`

Decrements an expression by 1 (`x := x - 1`).

`x mod y`

Modulo operator. Returns the remainder after integer division.
e.g. `13 mod 8` will return **5**.

Real Number Commands

The following commands can only be performed on real numbers.

`x mod y`

Modulo operator. Returns the remainder after division.
e.g. `4.5 mod 2.0` returns the value **0.5**.

`exp(x)`

Exponential function (returns the value of e^x).

`exp2(x)`

Binary exponential function (returns the value of 2^x).

`log(x)`

Natural logarithmic function (base *e*).

`log2(x)`

Binary logarithmic function (base 2).

`log10(x)`

Common logarithmic function (base 10).

`pow(x, y)`

Power function (returns the value of x^y).

`sqrt(x)`

Square root function.

`cbrt(x)`

Cube root function.

Rounding Commands

Rounding commands can only be performed on real numbers.

`ceil(x)`

Ceiling (round up).

```
ceil(2.3) = 3.0
```

`floor(x)`

Floor (round down).

```
floor(2.8) = 2.0
```

`round(x)`

Round (round to nearest).

```
round(2.3) = 2.0
round(2.8) = 3.0
```

Trigonometric Commands

Trigonometric commands can only be performed on real numbers.

`cos(x)`

Cosine function.

`sin(x)`

Sine function.

`tan(x)`

Tangent function.

`acos(x)`

Arccosine (inverse cosine) function.

`asin(x)`

Arcsine (inverse sine) function.

`atan(x)`

Arctangent (inverse tangent) function.

Bitwise Operators

The following bitwise operators can be used:

`x .and. y`	Bitwise `AND`.
`x .or. y`	Bitwise `OR`.
`x .xor. y`	Bitwise `XOR`.
`.not. x`	Bitwise `NOT` (negation).
`sh_left(<expression>, <shift-bits>)`	Shifts the bits in `<expression>` by the amount of `<shift-bits>` to the left. Effectively multiplies the expression by 2.
`sh_right(<expression>, <shift-bits>)`	Shifts the bits in `<expression>` by the amount of `<shift-bits>` to the right. Effectively divides the expression by 2.

random()

random(<min>, <max>)
Generates a random integer between (and including) `<min>` and `<max>`.

Examples

```
on init
    declare $rnd_amt
    declare $new_vel
end on

on note
    $rnd_amt := $EVENT_VELOCITY * 10 / 100
    $new_vel := random($EVENT_VELOCITY - $rnd_amt, $EVENT_VELOCITY + $rnd_amt)

    { mirror invalid velocity values into the allowed velocity range }
    if ($new_vel > 127)
        $new_vel := 127 - ($new_vel mod 127)
    end if

    if ($new_vel < 1)
        $new_vel := 1 + abs($new_vel)
    end if

    change_velo($EVENT_ID, $new_vel)
end on
```

Randomly changing velocities by ±10 percent.

real()

```
real(<integer-value>)
```

Converts an integer value into a real number.

Examples

```
on init
    declare ~velocity_disp
end on

on note
    ~velocity_disp := real($EVENT_VELOCITY) / 127.0
    message(~velocity_disp)
end on
```

Displays the event velocity in the range from 0.0 to 1.0.

See Also

int()

int()

`int(<real-value>)`

Converts a real number into an integer.

Remarks

- Using this command without any rounding function will cause the real value to be truncated, so performing this function with real values **2.2** and **2.8** will both return an integer value of **2**
- Be aware that this command reduces precision from 64-bit to 32-bit, which means that valid real numbers outside of 32-bit signed integer range (**-2147483648 ... 2147483647**) will not be properly converted, since they end up in overflow.

Examples

```
on init
    declare $test_int
    declare ~test_real := 2.8

    $test_int := int(~test_real)
    message($test_int)
end on
```

Converting a variable from real to integer and then displaying it.

See Also

real()

Rounding Commands: `ceil()`, `floor()`, `round()`

msb()

`msb(<value>)`

Returns the most significant byte portion of a 14-bit value.

Examples

```
on rpn
    message(msb($RPN_VALUE))
end on
```
Commonly used when working with RPN and NRPN messages.

```
on init
    declare ui_value_edit $Value (0, 16383, 1)
end on

on ui_control ($Value)
    message("MSB: " & msb($Value) & " - LSB: " & lsb($Value))
end on
```
Understanding MSB and LSB.

See Also

lsb()

Events and MIDI: $RPN_ADDRESS, $RPN_VALUE

lsb()

`lsb(<value>)`

Returns the least significant byte portion of a 14-bit value.

Examples

```
on rpn
    message(lsb($RPN_VALUE))
end on
```

Commonly used when working with RPN and NRPN messages.

```
on init
    declare ui_value_edit $Value (0, 16383, 1)
end on

on ui_control ($Value)
    message("MSB: " & msb($Value) & " - LSB: " & lsb($Value))
end on
```

Understanding MSB and LSB.

See Also

msb()

Events and MIDI: $RPN_ADDRESS, $RPN_VALUE

6. Control Statements

Boolean Operators

`x > y`	Greater than.
`x < y`	Less than.
`x >= y`	Greater than or equal.
`x <= y`	Less than or equal.
`x = y`	Equal.
`x # y`	Not equal.
`in_range(x, y, z)`	True if x is between (and including) y and z.
`not a`	True if **a** is false and vice versa.
`a and b`	True if **a** is true and **b** is true.
`a or b`	True if **a** is true or **b** is true.
`a xor b`	True **only** if either **a** or **b** is true, but not both.

Remarks

- Boolean operators are used in `if` and `while` statements, since they return if the condition is either true or false. In the list above. x, y and z denote numerals, **a** and **b** denote Boolean values.

continue

```
while (<condition>)
  if (<condition>)
    continue
  end if
  ...
end while
```

Continue statement. Skips the rest of the current iteration if the condition is met.

Remarks

- `continue` statement can only be used inside a `while` loop.

Examples

```
on init
    declare $i
    declare $skipped

    $i := 0
    while ($i < 5)
      inc($i)

      if ($i <= 3)
        continue
      end if

      inc($skipped)
    end while

    message($i & ", " & $skipped)
end on
```

Skipping first three iterations of the while loop. The message will print 5, 2.

See Also

while ()

if ... else ... end if

```
if
  <statements>
else
  <statements>
end if
```

Conditional if statement. `else` branch is not required.

Examples

```
on controller
    if (in_range($CC_NUM, 0, 127))
        message("CC Number: " & $CC_NUM & " - Value: " & %CC[$CC_NUM])
    else
        if ($CC_NUM = $VCC_PITCH_BEND)
            message("Pitch Bend - Value: " & %CC[$CC_NUM])
        end if

        if ($CC_NUM = $VCC_MONO_AT)
            message("Channel Pressure - Value: " & %CC[$CC_NUM])
        end if
    end if
end on
```

Display different text depending on various MIDI controller messages.

See Also

select ()

select ()

```
select (<variable>)
  case <constant>
    <statements>
  case <constant> to <constant>
    <statements>
end select
```

Select-case statement.

Remarks

- This statement is similar to the `if` statement, except that it has an arbitrary number of branches. The expression after the `select` keyword is evaluated and matched against individual `case` branches. The first `case` branch that matches is executed.
- The `case` branches may consist of either a single constant number, or a number range expressed by the term "`x to y`").
- While there is no `else` or `default` case branch in KSP, one can be achieved by using `case 08000000H to 07FFFFFFFH`, which covers the whole range of 32-bit signed integer values, effectively catching all cases that don't have literally specified branches.

Examples

```
on controller
    if ($CC_NUM = $VCC_PITCH_BEND)
        select (%CC[$VCC_PITCH_BEND])
            case -8191 to -1
                message("Pitch Bend down")
            case 0
                message("Pitch Bend center")
            case 1 to 8191
                message("Pitch Bend up")
            case 080000000H to 07FFFFFFH
                message("We're not sure how you got this Pitch Bend value!")
        end select
    end if
end on
```

Query the state of the pitch bend wheel.

See Also

if ... else ... end if

while ()

```
while (<condition>)
  <statements>
  <increment, decrement or wait>
end while
```

While loop.

Remarks

- If a `while` loop does not contain any counter or a `wait()`/`wait_ticks()` command, it will be stopped and exited after 10 million iterations.

Examples

```
on note
    ignore_event($EVENT_ID)

    while ($NOTE_HELD = 1)
        play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, $DURATION_QUARTER / 2)
        wait($DURATION_QUARTER)
    end while
end on
```

Repeating held notes at the rate of one quarter note.

See Also

wait()

wait_ticks()

Events and MIDI: `$NOTE_HELD`

7. User Interface Widgets

ui_button

```
declare ui_button $<variable-name>
```

Creates a button in the performance view.

Remarks

- UI callback for button is triggered when releasing the mouse (on mouse up).
- A button cannot be MIDI learned or host automated.

Examples

```
on init
    declare ui_button $free_sync_button
    $free_sync_button := 1

    set_text($free_sync_button, "Sync")

    make_persistent($free_sync_button)
    read_persistent_var($free_sync_button)

    if ($free_sync_button = 0)
        set_text($free_sync_button, "Free")
    else
        set_text($free_sync_button, "Sync")
    end if
end on

on ui_control ($free_sync_button)
    if ($free_sync_button = 0)
        set_text($free_sync_button, "Free")
    else
        set_text($free_sync_button, "Sync")
    end if
end on
```

A simple freerun/tempo sync button implementation.

See Also

ui_switch

ui_file_selector

`declare ui_file_selector $<variable-name>`

Creates a file selector in the performance view.

Examples

```
on init
    message("")
    set_ui_height(5)

    declare $load_mf_id := -1
    declare @file_name
    declare @file_path
    declare @basepath
    { set browser path here, for example:
    @basepath := "/Users/username/Desktop/MIDI Files/" }

    declare ui_file_selector $file_browser

    declare $browser_id
    $browser_id := get_ui_id($file_browser)

    set_control_par_str($browser_id, $CONTROL_PAR_BASEPATH, @basepath)
    set_control_par($browser_id, $CONTROL_PAR_FILE_TYPE, $NI_FILE_TYPE_MIDI)
    set_control_par($browser_id, $CONTROL_PAR_COLUMN_WIDTH, 180)
    set_control_par($browser_id, $CONTROL_PAR_HEIGHT, 170)
    set_control_par($browser_id, $CONTROL_PAR_WIDTH, 550)

    move_control_px($file_browser, 66, 2)

    declare ui_button $prev
    declare ui_button $next

    move_control($prev, 5, 1)
    move_control($next, 6, 1)
end on

on async_complete
    if ($NI_ASYNC_ID = $load_mf_id)
        $load_mf_id := -1

        if ($NI_ASYNC_EXIT_STATUS = 0)
            message("MIDI file not found!")
        else
            message("Loaded MIDI File: " & @file_name)
        end if
    end if
end on

on ui_control ($file_browser)
    @file_name := fs_get_filename($browser_id, 0)
    @file_path := fs_get_filename($browser_id, 2)
    $load_mf_id := load_midi_file(@file_path)
end on

on ui_control ($prev)
    { calls 'on ui_control ($file_browser)' }
    fs_navigate($browser_id, 0)
    $prev := 0
```

```
end on

on ui_control ($next)
    { calls 'on ui_control ($file_browser)' }
    fs_navigate($browser_id, 1)
    $next := 0
end on
```

Loading MIDI files via the file selector.

See Also

fs_navigate()

ui_label

`declare ui_label $<variable-name> (<grid-width>, <grid-height>)`	
Creates a text or image label in the performance view	
`<grid-width>`	The width of the label in grid units (**1** to **6**)
`<grid-height>`	The height of the label in grid units (**1** to **16**)

Examples

```
on init
    declare ui_label $label_1 (1, 1)
    set_text($label_1, "Small Label")

    declare ui_label $label_2 (3, 6)
    set_text($label_2, "Big Label")
    add_text_line($label_2, "…with a second text line")
end on
```

Two labels with different sizes

```
on init
    declare ui_label $label_1 (1, 1)
    set_text($label_1, "Small Label")
    hide_part($label_1, $HIDE_PART_BG)
end on
```

Hide the background of a label (also possible with other UI widgets)

See Also

set_text()

add_text_line()

hide_part()

ui_level_meter

```
declare ui_level_meter $<variable-name>
```

Creates a level meter in the performance view.

Remarks

- The level meter can display the output levels of instrument buses, main instrument output, and gain reduction from compressor and limiter effects.

Examples

```
on init
    declare ui_level_meter $Level1
    declare ui_level_meter $Level2

    attach_level_meter(get_ui_id($Level1), -1, -1, 0, -1)
    attach_level_meter(get_ui_id($Level2), -1, -1, 1, -1)
end on
```

Creating two volume meters, each displaying one channel of Kontakt’s instrument output.

See Also

attach_level_meter()

Specific: `$CONTROL_PAR_BG_COLOR`, `$CONTROL_PAR_OFF_COLOR`, `$CONTROL_PAR_ON_COLOR`, `$CONTROL_PAR_OVERLOAD_COLOR`, `$CONTROL_PAR_PEAK_COLOR`, `$CONTROL_PAR_VERTICAL`, `$CONTROL_PAR_RANGE_MIN`, `$CONTROL_PAR_RANGE_MAX`

ui_knob

`declare ui_knob $<variable-name> (<min>, <max>, <display-ratio>)`	
Creates a knob in the performance view.	
`<min>`	The minimum value of the knob.
`<max>`	The maximum value of the knob.
`<display-ratio>`	The knob value is divided by `<display-ratio>` for display purposes.

Examples

```
on init
    declare ui_knob $Knob_1 (0, 1000, 1)
    declare ui_knob $Knob_2 (0, 1000, 10)
    declare ui_knob $Knob_3 (0, 1000, 100)
    declare ui_knob $Knob_4 (0, 1000, 20)
    declare ui_knob $Knob_5 (0, 1000, -10)
end on
```
Various display ratios.

```
on init
    declare $count
    declare !note_class[12]
    !note_class[0] := "C"
    !note_class[1] := "C#"
    !note_class[2] := "D"
    !note_class[3] := "D#"
    !note_class[4] := "E"
    !note_class[5] := "F"
    !note_class[6] := "F#"
    !note_class[7] := "G"
    !note_class[8] := "G#"
    !note_class[9] := "A"
    !note_class[10] := "Bb"
    !note_class[11] := "B"
    declare !note_names [128]

    while ($count < 128)
        !note_names[$count] := !note_class[$count mod 12] & (($count / 12) - 2)

        inc($count)
    end while

    declare ui_knob $Note (0, 127, 1)

    make_persistent($Note)
    read_persistent_var($Note)

    set_knob_label($Note, !note_names[$Note])
end on

on ui_control ($Note)
    set_knob_label($Note, !note_names[$Note])
end on
```
Knob displaying MIDI note names.

ui_menu

```
declare ui_menu $<variable-name>
```

Creates a drop-down menu in the performance view.

Examples

```
on init
    declare ui_menu $menu

    add_menu_item($menu, "First Entry", 0)
    add_menu_item($menu, "Second Entry", 1)
    add_menu_item($menu, "Third Entry", 2)
end on
```

A simple menu.

```
on init
    declare $count
    declare ui_menu $menu

    $count := 0
    while ($count < 20)
        add_menu_item($menu, "Entry #" & $count + 1, $count)

        inc($count)
    end while
end on
```

Quickly create a menu with many entries.

See Also

add_menu_item()

get_menu_item_str()

get_menu_item_value()

get_menu_item_visibility()

set_menu_item_str()

set_menu_item_value()

set_menu_item_visibility()

ui_mouse_area

`declare ui_mouse_area $<variable-name>`

Creates a mouse area in the performance view.

Remarks

- A mouse area supports drag and drop of the following file types: audio (WAV, AIF, AIFF, NCW), MIDI and KSP array (NKA).
- It is possible to define which types of files are accepted as drop targets and whether to accept just one or multiple files.
- The mouse area widget is invisible, but the drop target can be shown or hidden, like any other UI widget.

Examples

```
on init
    declare ui_mouse_area $waveDnD

    set_control_par(get_ui_id($waveDnD), $CONTROL_PAR_DND_ACCEPT_AUDIO,
$NI_DND_ACCEPT_ONE)
    set_control_par(get_ui_id($waveDnD), $CONTROL_PAR_DND_ACCEPT_ARRAY,
$NI_DND_ACCEPT_ONE)
    set_control_par(get_ui_id($waveDnD), $CONTROL_PAR_WIDTH, 90)
    set_control_par(get_ui_id($waveDnD), $CONTROL_PAR_HEIGHT, 32)
    set_control_par(get_ui_id($waveDnD), $CONTROL_PAR_RECEIVE_DRAG_EVENTS, 1)

    move_control_px($waveDnD, 66, 2)
end on
```

A mouse area widget which can accept a single audio or NKA file.

The on ui_control callback is triggered by a drop action. It has 3 built-in arrays:

```
!NI_DND_ITEMS_AUDIO
!NI_DND_ITEMS_MIDI
!NI_DND_ITEMS_ARRAY
```

Example UI callback

```
on ui_control ($waveDnD)
    if ($NI_MOUSE_EVENT_TYPE = $NI_MOUSE_EVENT_TYPE_DRAG)
        message("DRAG")
        message("MOUSE OVER CONTROL: " & $NI_MOUSE_OVER_CONTROL)
    end if

    if ($NI_MOUSE_EVENT_TYPE = $NI_MOUSE_EVENT_TYPE_DROP)
        if (num_elements(!NI_DND_ITEMS_AUDIO) = 1)
            wait_async(set_sample(%NI_USER_ZONE_IDS[0], !NI_DND_ITEMS_AUDIO[0]))
        end if
    end if
end on
```

See Also

Specific: `$NI_MOUSE_EVENT_TYPE`, `$NI_MOUSE_EVENT_TYPE_DND_DROP`, `$NI_MOUSE_EVENT_TYPE_DND_DRAG`, `$NI_MOUSE_OVER_CONTROL`

ui_panel

`declare ui_panel $<variable-name>`

Creates a panel for grouping widgets in the performance view.

Remarks

- A panel is a control which can contain one or multiple widgets. Unlike the rest of the UI widgets, panels don't have size. They are very useful for grouping controls that are meant to be handled together, then one can simultaneously modify the `$CONTROL_PAR_HIDE`, `$CONTROL_PAR_POS_X`, `$CONTROL_PAR_POS_Y` or `$CONTROL_PAR_Z_LAYER` properties of all the controls contained in that panel. The position of a contained control is relative to the panel's position. This means 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 Z-layer value than their parent panels. One could use this logic to easily create hierarchies in a performance view.

Examples

```
on init
    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))
end on
```

Adds the volume knob in the mixer panel.

See Also

General: `$CONTROL_PAR_PARENT_PANEL`

ui_slider

`declare ui_slider $<variable-name> (<min>, <max>)`	
Creates a slider in the performance view.	
`<min>`	The minimum value of the slider.
`<max>`	The maximum value of the slider.

Examples

```
on init
    declare ui_slider $test (0, 100)
    set_control_par(get_ui_id($test), $CONTROL_PAR_DEFAULT_VALUE, 50)
end on
```

Slider with default value.

```
on init
    declare ui_slider $test (-100, 100)
    declare $id

    $id := get_ui_id($test)

    $test := 0

    set_control_par($id, $CONTROL_PAR_MOUSE_BEHAVIOUR, 2000)
    set_control_par($id, $CONTROL_PAR_DEFAULT_VALUE, 0)
    set_control_par_str($id, $CONTROL_PAR_PICTURE, "slider")
end on
```

Creating a bipolar slider by loading a different picture background. See the chapter on the Resource Container *in order to learn more about how to use graphical assets with KSP.*

See Also

ui_knob

set_control_par_arr()

Specific: `$CONTROL_PAR_MOUSE_BEHAVIOUR`

ui_switch

```
declare ui_switch $<variable-name>
```

Creates a switch in the performance view.

Remarks

- UI callback for switch is triggered when pressing the mouse (on mouse down).
- A switch can be MIDI learned and host automated, as opposed to a button.

Examples

```
on init
    declare ui_switch $rec_button

    declare $rec_button_id
    $rec_button_id := get_ui_id($rec_button)

    set_control_par($rec_button_id, $CONTROL_PAR_POS_X, 250)
    set_control_par($rec_button_id, $CONTROL_PAR_POS_Y, 5)
    set_control_par($rec_button_id, $CONTROL_PAR_WIDTH, 60)
    set_control_par($rec_button_id, $CONTROL_PAR_HEIGHT, 20)
    set_control_par($rec_button_id, $CONTROL_PAR_TEXT_ALIGNMENT, 1)
    set_control_par_str($rec_button, $CONTROL_PAR_TEXT, "Record")
end on
```

Switch with various settings utilizing set_control_par() and set_control_par_str().

See Also

ui_button

ui_table

`declare ui_table %<array-name>[num-elements] (<grid-width>, <grid-height>, <range>)`	
Creates a table in the performance view.	
`<width>`	The width of the table in grid units (**1 ... 6**).
`<height>`	The height of the table in grid units (**1 ... 16**).
`<range>`	The range of the table. If negative values are used, a bipolar table is created.

Remarks

- The maximum number of columns (elements) in the table is **128**.

Examples

```
on init
    declare ui_table %table_uni[10] (2, 2, 100)
    declare ui_table %table_bi[10] (2, 2, -100)
end on
```
Unipolar and bipolar tables.

```
on init
    declare ui_table %table[128] (5, 2, 100)
    declare ui_value_edit $Steps (1, 127, 1)

    $Steps := 16
    set_table_steps_shown(%table, $Steps)
end on

on ui_control ($Steps)
    set_table_steps_shown(%table, $Steps)
end on
```
Changes the amount of shown steps (columns) in a table.

```
on init
    declare ui_table %table[20] (4, 4, 100)
    declare ui_button $button
    declare ui_label $value (1, 1)
end on

on ui_control($button)
    if ($button = 1)
        hide_part(%table, $HIDE_PART_VALUE)
    else
        hide_part(%table, $HIDE_PART_NOTHING)
    end if

    set_text($value, "Step " & $NI_CONTROL_PAR_IDX + 1 & ": " &
%table[$NI_CONTROL_PAR_IDX])
end on
```
Hiding the value indicator in top left corner of the table and replacing it with a value readout in a label.

See Also

set_table_steps_shown()

hide_part()

Specific: `$NI_CONTROL_PAR_IDX`

ui_text_edit

```
declare ui_text_edit @<variable-name>
```

Creates a text edit field in the performance view.

Examples

```
on init
    declare ui_text_edit @label_name
    declare ui_label $pattern_lbl (1, 1)

    set_control_par_str(get_ui_id(@label_name), $CONTROL_PAR_TEXT, "empty")
    set_control_par(get_ui_id(@label_name), $CONTROL_PAR_FONT_TYPE, 25)
    set_control_par(get_ui_id(@label_name), $CONTROL_PAR_POS_X, 73)
    set_control_par(get_ui_id(@label_name), $CONTROL_PAR_POS_Y, 2)

    set_text($pattern_lbl, "")

    move_control_px($pattern_lbl, 66, 2)

    make_persistent(@label_name)
end on

on ui_control (@label_name)
    message(@label_name & " it is!")
end on
```

A text edit field on top of a label.

See Also

@ (string variable)

ui_value_edit

`declare ui_value_edit $<variable-name> (<min>, <max>, <$display-ratio>)`	
Creates a value edit field (number box) in the performance view.	
`<min>`	The minimum value of the value edit.
`<max>`	The maximum value of the value edit.
`<display-ratio>`	The value is divided by `<display-ratio>` for display purposes. You can also use `$VALUE_EDIT_MODE_NOTE_NAMES` here to display note names instead of numbers.

Examples

```
on init
    declare ui_value_edit $test (0, 127, $VALUE_EDIT_MODE_NOTE_NAMES)
    set_text($test, "")
    set_control_par(get_ui_id($test), $CONTROL_PAR_WIDTH, 45)
    move_control_px($test, 66, 2)
end on

on note
    $test := $EVENT_NOTE
end on
```

Value edit displaying note names.

```
on init
    declare ui_value_edit $test (0, 10000, 1000)
    set_text($test, "Value")
end on
```

Value edit with three decimal spaces.

See Also

Specific: `$CONTROL_PAR_SHOW_ARROWS`, `$VALUE_EDIT_MODE_NOTE_NAMES`

ui_waveform

`declare ui_waveform $<variable-name> (<grid-width>, <grid-height>)`	
Creates a waveform display for displaying samples and optionally their slices. This widget can also be used to control specific parameters per slice and for MIDI drag and drop functionality.	
`<width>`	The width of the waveform in grid units (**1 ... 6**).
`<height>`	The height of the waveform in grid units (**1 ... 16**).

Examples

```
on init
    declare ui_waveform $Waveform (6, 6)
    attach_zone($Waveform, find_zone(”Test”), 0)
end on
```

Displays the zone which has the name “Test” in the waveform widget. Use a sample named Test.wav (or .aiff, etc.) to test the above code.

See Also

set_ui_wf_property()

get_ui_wf_property()

attach_zone()

Zone and Slice Functions: `find_zone()`

Specific: Waveform Flag Constants, Waveform Property Constants, `$CONTROL_PAR_WAVE_COLOR`, `$CONTROL_PAR_BG_COLOR`, `$CONTROL_PAR_WAVE_CURSOR_COLOR`, `$CONTROL_PAR_SLICEMARKERS_COLOR`, `$CONTROL_PAR_BG_ALPHA`

ui_wavetable

`declare ui_wavetable $<variable-name>`

Creates a wavetable display in the performance view, visualizing the state of a zone which is used as a wavetable.

Examples

```
on init
    declare ui_wavetable $wavetable
    set_control_par(get_ui_id($wavetable), $CONTROL_PAR_WT_ZONE,
find_zone("Wavetable01"))
end on
```

Displays the zone "Wavetable01" in the wavetable widget. Use a wavetable named Wavetable01.wav (or .aiff, etc.) to test the above code.

See Also

set_control_par()

Zone and Slice Functions: `find_zone()`

Specific: `$CONTROL_PAR_WT_VIS_MODE`, `$NI_WT_VIS_2D`, `$NI_WT_VIS_3D`, `$CONTROL_PAR_WAVE_COLOR`, `$CONTROL_PAR_BG_COLOR`, `$CONTROL_PAR_BG_ALPHA`, `$CONTROL_PAR_WAVE_COLOR`, `$CONTROL_PAR_WAVE_ALPHA`, `$CONTROL_PAR_WAVE_END_COLOR`, `$CONTROL_PAR_WAVE_END_ALPHA`, `$CONTROL_PAR_WAVETABLE_END_COLOR`, `$CONTROL_PAR_WAVETABLE_END_ALPHA`, `$CONTROL_PAR_PARALLAX_X`, `$CONTROL_PAR_PARALLAX_Y`, `$CONTROL_PAR_WT_ZONE`

ui_xy

```
declare ui_xy ?<array-name>[num-elements]
```

Creates an XY pad in the performance view.

Remarks

- The range of each axis on the XY pad is always between **0.0** and **1.0**.
- The number of cursors in the XY pad, i.e. the interactive elements, is defined by the size of the array. Each index in the array represents one axis of one cursor, so two indices are needed for each cursor. Applying this, if you wanted to create an XY pad with **3** cursors, the size of the XY array would have to be **6**.
- The maximum size of the XY array is **32** elements, so the maximum number of cursors in a single XY pad is **16**.
- The even indices of the array hold the X axis value of the cursors, and the odd indices hold the Y axis values. So index **0** is the X value of the first cursor, and index **1** is the Y value of the first cursor.
- It is possible to define how the XY pad reacts to mouse interaction using the `$CONTROL_PAR_MOUSE_MODE` control parameter.
- Querying `$NI_MOUSE_EVENT_TYPE` within the `on ui_control` callback allows identification of the mouse event type that triggered it.

Examples

```
on init
    message("")

    make_perfview
    set_ui_height(7)

    declare ui_xy ?myXY[4]

    declare $xyID
    $xyID := get_ui_id(?myXY)

    { define the mouse behaviour }
    set_control_par($xyID, $CONTROL_PAR_MOUSE_MODE, 0)
    set_control_par($xyID, $CONTROL_PAR_MOUSE_BEHAVIOUR_X, 1000)
    set_control_par($xyID, $CONTROL_PAR_MOUSE_BEHAVIOUR_Y, 1000)

    { set automation IDs and parameter names }
    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 0, 0)
    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 1, 1)
    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 2, 2)
    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 3, 3)

    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Cutoff", 0)
    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Resonance", 1)
    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Delay Pan", 2)
    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Delay Feedback",
3)

    { position and size }
    move_control_px(?myXY, 216, 50)
    set_control_par($xyID, $CONTROL_PAR_WIDTH, 200)
    set_control_par($xyID, $CONTROL_PAR_HEIGHT, 200)
    { move the cursors around the XY pad }
    ?myXY[0] := 0.25 { cursor 1, X axis }
    ?myXY[1] := 0.75 { cursor 1, Y axis }
    ?myXY[2] := 0.75 { cursor 2, X axis }
    ?myXY[3] := 0.25 { cursor 2, Y axis }
end on
```

Creating an XY pad control with two cursors and automation information.

See Also

set_control_par_arr()

General: `set_control_par_str_arr()`

Specific: `$HIDE_PART_CURSOR`, `$NI_CONTROL_PAR_IDX`, `$CONTROL_PAR_MOUSE_MODE`, `$CONTROL_PAR_ACTIVE_INDEX`, `$CONTROL_PAR_CURSOR_PICTURE`, `$CONTROL_PAR_MOUSE_BEHAVIOUR_X`, `$CONTROL_PAR_MOUSE_BEHAVIOUR_Y`

8. User-defined Functions

```
function <function-name>()
  <statements>
end function
```

Declares a function.

```
call <function-name>
```

Calls a previously declared function

Remarks

- The function has to be declared before it is called.
- Empty parentheses can be optionally used when declaring and calling the function.

Examples

```
on init
    declare $root_note := 60

    declare ui_button $button_1
    declare ui_button $button_2
    declare ui_button $button_3

    set_text($button_1, "Play C Major")
    set_text($button_2, "Play F# Major")
    set_text($button_3, "Play C7 (b9, #11)")
end on

function func_play_triad()
    play_note($root_note, 100, 0, 300000)
    play_note($root_note + 4, 100, 0, 300000)
    play_note($root_note + 7, 100, 0, 300000)
end function

on ui_control ($button_1)
    $root_note := 60
    call func_play_triad()

    $button_1 := 0
end on

on ui_control ($button_2)
    $root_note := 66
    call func_play_triad()

    $button_2 := 0
end on

on ui_control ($button_3)
    $root_note := 60
    call func_play_triad()

    $root_note := 66
    call func_play_triad()
    $button_3 := 0
end on
```

Jazz Harmony 101.

9. General Commands

disable_logging()

`disable_logging(<event-type>)`	
Disables emission of messages, warnings or watched variable events to both the Kontakt status bar and Creator Tools Debugger.	
`<event-type>`	Which event type emission to deactivate. Available options: • `$NI_LOG_MESSAGE` • `$NI_LOG_WARNING` • `$NI_LOG_WATCHING`

Remarks

- This command is only available in the `on init` callback.

Examples

```
on init
    disable_logging($NI_LOG_MESSAGE)
    disable_logging($NI_LOG_WARNING)
    disable_logging($NI_LOG_WATCHING)
end on
```

Keep the lines above commented out while development and bring them back in right before shipping your product to disable any debugging-related content.

See Also

watch_var()

watch_array_idx()

exit

`exit`

Immediately stops a callback or exits a function.

Remarks

- `exit` is a very strong command. Be careful when using it, especially when dealing with larger scripts.
- If used within a function, `exit` only exits the function, but not the entire callback.

Examples

```
on note
    if (not in_range($EVENT_NOTE, 60, 71))
        exit
    end if

    { from here on, only notes between C3 to B3 will be processed }
end on
```

Useful for quickly setting up key ranges to be affected by the script.

See Also

wait()

stop_wait()

ignore_controller

`ignore_controller`
Ignores a MIDI Continuous Controller, Pitch Bend or Channel Aftertouch event in the `on controller` callback, or Polyphonic Aftertouch event in the on poly_at callback.

Examples

```
on controller
    if ($CC_NUM = 1)
        ignore_controller
        set_controller($VCC_MONO_AT, %CC[1])
    end if
end on
```

Transform the modwheel into aftertouch.

See Also

ignore_event()

set_controller()

on controller

message()

```
message(<variable-or-string>)
```

Displays text in the status line of Kontakt.

Remarks

- This command is intended to be used for debugging and testing while programming a script. Since there is only one status line in Kontakt, it should not be used as a generic means of communication with the user. Use a label widget instead.
- Make it a habit to write `message("")` at the start of the `on init` callback. You can then be sure that all previous messages (by the script or by the system) are deleted and you see only new messages.

Examples

```
on init
    message("Hello, world!")
end on
```

The inevitable implementation of "Hello, world!" in KSP.

```
on note
    message("Note " & $EVENT_NOTE & " received at " &  $ENGINE_UPTIME & "
milliseconds")
end on
```

Concatenating strings and expressions in a message() command.

See Also

reset_ksp_timer

ui_label

set_text()

Time and Transport: `$ENGINE_UPTIME`, `$KSP_TIMER`

note_off()

`note_off(<event-id>)`	
Sends a MIDI Note Off message for a specific note event ID.	
`<event-id>`	Unique identification number of the note event to be changed.

Remarks

- `note_off()` is equivalent to releasing a key, thus it will always trigger an `on release` callback, as well as jump to the release portion of a volume envelope. Notice the difference between `note_off()` and `fade_out()`, since `fade_out()` works on voice level.

Examples

```
on controller
    if ($CC_NUM = 1)
        note_off($ALL_EVENTS)
    end if
end on
```

A custom "All Notes Off" implementation triggered by the modwheel.

```
on init
    declare polyphonic $new_id
end on

on note
    ignore_event($EVENT_ID)
    $new_id := play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, 0)
end on

on release
    ignore_event($EVENT_ID)
    wait(200000)
    note_off($new_id)
end on
```

Delaying the release of each note by 200 milliseconds.

See Also

fade_out()

play_note()

play_note()

`play_note(<note-number>, <velocity>, <sample-offset>, <duration>)`	
Generates a note event, i.e. a Note On message followed by a Note Off message.	
`<note-number>`	The MIDI note number to be generated (**0 ... 127**).
`<velocity>`	Velocity of the generated note (**1 ... 127**).
`<sample-offset>`	Sample offset in microseconds.
`<duration>`	Length of the generated note in microseconds. This parameter also accepts two special values: **-1**: releasing the note which started the callback stops the sample. **0**: the entire sample is played (be careful with looped samples, as they would be played indefinitely in this case!).

Remarks

- In DFD mode, the sample offset is dependent on the **Sample Mod (S. Mod)** value of the respective zones (found in Kontakt's Wave Editor). Sample offset value greater than the zone's S. Mod setting will be ignored and no sample offset will be applied.
- You can retrieve the event ID of the played note event in a variable by writing:
  `<variable> := play_note(<note>, <velocity>, <sample-offset>, <duration>)`

Examples

```
on note
    play_note($EVENT_NOTE + 12, $EVENT_VELOCITY, 0, -1)
end on
```

Harmonizes the played note with the upper octave.

```
on init
    declare $new_id
end on

on controller
    if ($CC_NUM = 64)
        if (%CC[64] = 127)
            $new_id := play_note(60, 100, 0, 0)
        else
            note_off($new_id)
        end if
    end if
end on
```

Trigger a MIDI note by pressing the sustain pedal.

See Also

note_off()

set_controller()

`set_controller(<controller>, <value>)`	
Sends a MIDI Continuous Controller, Pitch Bend or Channel Pressure message	
`<controller>`	This parameter sets the type, and in the case of MIDI CCs, sets the CC number: • A number (**0 ... 127**) designates a MIDI CC number • `$VCC_PITCH_BEND` indicates MIDI Pitch Bend • `$VCC_MONO_AT` indicates MIDI Channel Pressure (monophonic aftertouch)
`<value>`	The value of the specified controller. • MIDI CC and Channel Pressure value range: **0 ... 127** • MIDI Pitch Bend value range: **-8192 ... 8191**

Remarks

- `set_controller()` cannot be used in the `on init` callback. If for some reason you wat to send a controller value upon instrument load, use `on persistence_changed` callback.

```
on note
    if ($EVENT_NOTE = 36)
        ignore_event($EVENT_ID)
        set_controller($VCC_MONO_AT, $EVENT_VELOCITY)
    end if
end on

on release
    if ($EVENT_NOTE = 36)
        ignore_event($EVENT_ID)
        set_controller($VCC_MONO_AT, 0)
    end if
end on
```

If you have a keyboard with no aftertouch, press C1 instead.

See Also

ignore_controller

Events and MIDI: `$VCC_PITCH_BEND`, `$VCC_MONO_AT`

set_rpn()/set_nrpn()

`set_rpn(<address>, <value>)`
`set_nrpn(<address>, <value>)`	

Sends a MIDI RPN or NRPN message.	
`<address>`	The RPN or NRPN address (**0 ... 16383**).
`<value>`	The value of the RPN or NRPN message (**0 ... 16383**).

Remarks

- Kontakt cannot handle RPN or NRPN messages as external modulation sources. You can however use these messages for simple inter-script communication.

See Also

on rpn/nrpn

set_controller()

msb()

lsb()

Events and MIDI: $RPN_ADDRESS, $RPN_VALUE

set_snapshot_type()

`set_snapshot_type(<type>)`	
Configures the behavior of all five slots when a snapshot is saved or recalled.	
`<type>`	The available types are: **0**: The init callback will always be executed upon snapshot change, then the on persistence_changed callback will be executed (default behavior). **1**: the init callback will not be executed upon loading a snapshot, only the on persistence_callback will be executed. **2**: same as type **0**, but only KSP variables are saved with the snapshot. **3**: same as type **1**, but only KSP variables are saved with the snapshot.

Remarks

- This command acts globally, i.e. it can applied in any script slot.
- In snapshot types **1** and **3**, values of persistent and instrument persistent variables are preserved.
- Loading a snapshot always resets Kontakt's audio engine, i.e. audio is stopped and all active events are deleted.

Examples

```
on init
    set_snapshot_type(1)

    declare ui_knob $knob_1 (0, 127, 1)
    set_text($knob_1, "Knob")
    make_persistent($knob_1)

    declare ui_button $gui_btn
    set_text($gui_btn, "Page 1")
end on

function show_gui()
    if ($gui_btn = 1)
        set_control_par(get_ui_id($knob_1), $CONTROL_PAR_HIDE, $HIDE_PART_NOTHING)
    else
        set_control_par(get_ui_id($knob_1), $CONTROL_PAR_HIDE, $HIDE_WHOLE_CONTROL)
    end if
end function

on persistence_changed
    call show_gui()
end on

on ui_control ($gui_btn)
    call show_gui()
end on
```

Retaining the GUI upon loading snapshots.

See Also

on init

on persistence_changed

10. Event Commands

by_marks()

`by_marks(<mark>)`	
A user-defined group of events.	
`<mark>`	One of 28 marks, `$MARK_1 ... $MARK_28`, which was assigned to the event. You can also select more than one event group by using the bitwise `.or.` operator, or by simply summing the event marks.

Remarks

- `by_marks()` is a user-defined group of events which can be set with `set_event_mark()`. It can be used with all commands which take event ID as an argument, like `note_off()`, `change_tune()` etc.

Examples

```
on note
    if ($EVENT_NOTE mod 12 = 0) { if played note is a C }
        set_event_mark($EVENT_ID, $MARK_1)
        change_tune(by_marks($MARK_1), %CC[1] * 1000, 0)
    end if
end on

on controller
    if ($CC_NUM = 1)
        change_tune(by_marks($MARK_1), %CC[1] * 1000, 0)
    end if
end on
```

Moving the modwheel changes the tuning of all C notes (C-2, C-1…C8).

See Also

set_event_mark()

Events and MIDI: `$EVENT_ID`, `$ALL_EVENTS`, `$MARK_1 … $MARK_28`

change_note()

`change_note(<event-id>, <note-number>)`
Changes the note number of a specific event ID.

Remarks

- `change_note()` is only allowed in the `on note` callback and only works before the first `wait()` statement. If the voice is already running, only the value of the `$EVENT_NOTE` variable changes.
- Once the note number of a particular note event is changed, it becomes the new `$EVENT_NOTE`.
- It is not possible to address events via event groups like `$ALL_EVENTS`.

Examples

```
on init
    declare %black_keys[5] := (1, 3, 6, 8, 10)
end on

on note
    if (search(%black_keys, $EVENT_NOTE mod 12) # -1)
        change_note($EVENT_ID, $EVENT_NOTE - 1)
    end if
end on
```

Constrain all notes to white keys, i.e. C major.

See Also

change_velo()

Events and MIDI: `$EVENT_NOTE`

change_pan()

`change_pan(<event-id>, <panorama>, <relative-bit>)`	
Changes the pan position of a specific note event.	
`<event-id>`	Unique identification number of the note event to be changed.
`<panorama>`	The pan position of the note event, from **-1000** (left) to **1000** (right).
`<relative-bit>`	If set to **0**, the amount is **absolute**, i.e. the amount overwrites any previous set values of that event. Note that this mode *also* overwrites any zone volume adjustments! If set to **1**, the amount is **relative** to the actual value of the event. If set to **2**, it behaves like mode **0** (**absolute** adjustment), except any zone volume adjustments are preserved. The different implications between absolute and relative adjustments are more apparent with more than one `change_pan()` statement applied to the same event.

Remarks

- `change_pan()` works on the note event level and does not change any panorama settings in the instrument itself. It is also not related to any modulations regarding panorama.

Examples

```
on init
    declare $pan_position
end on

on note
    $pan_position := ($EVENT_NOTE * 2000 / 127) - 1000
    change_pan($EVENT_ID, $pan_position, 0)
end on
```
Panning the entire key range from left to right, i.e. C-2 all the way to the left, G8 all the way to the right.

```
on note
    if ($EVENT_NOTE < 60)
        change_pan($EVENT_ID, 1000, 0)
        wait(500000)
        change_pan($EVENT_ID, -1000, 0) { absolute, pan is at -1000 }
    else
        change_pan($EVENT_ID, 1000, 1)
        wait(500000)
        change_pan($EVENT_ID, -1000, 1) { relative, pan is at 0 }
    end if
end on
```
Notes below C3 utilize a relative bit of 0. C3 and above utilize a relative bit of 1.

See Also

change_vol()

change_tune()

change_tune()

change_tune(<event-id>, <tune-amount>, <relative-bit>)	
Changes the tuning of a specific note event in millicents.	
<event-id>	Unique identification number of the note event to be changed.
<tune-amount>	The tune amount in millicents. **100000** equals **100** cents (one semitone).
<relative-bit>	If set to **0**, the amount is **absolute**, i.e. the amount overwrites any previous set values of that event. If set to **1**, the amount is **relative** to the actual value of the event. The different implications between absolute and relative adjustments are more apparent with more than one change_tune() statement applied to the same event.

Remarks

- change_tune() works on the note event level and does not change any tune settings in the instrument itself. It is also not related to any modulations regarding tuning.

Examples

```
on init
    declare $tune_amount
end on

on note
    $tune_amount := random(-50000, 50000)
    change_tune($EVENT_ID, $tune_amount, 1)
end on
```

Randomly detune every played note by ±50 cents

See Also

change_vol()

change_pan()

change_velo()

`change_velo(<event-id>, <velocity>)`

Changes the velocity of a specific note event ID.

Remarks

- `change_velo()` is only allowed in the on note callback and only works before the first `wait()` statement. If the voice is already running, only the value of the variable changes.
- Once the velocity of a particular note event is changed, it becomes the new `$EVENT_VELOCITY`.
- It is not possible to address events via event groups like `$ALL_EVENTS`.

Examples

```
on note
    change_velo($EVENT_ID, 100)
    message($EVENT_VELOCITY)
end on
```

All velocities are set to 100. Note that `$EVENT_VELOCITY` will also change to 100.

See Also

change_note()

Events and MIDI: `$EVENT_VELOCITY`

change_vol()

`change_vol(<event-id>, <volume>, <relative-bit>)`	
Changes the volume of a specific note event in millidecibels.	
`<event-id>`	Unique identification number of the note event to be changed.
`<volume>`	The volume change in millidecibels (**1000** millidecibels = **1** decibel).
`<relative-bit>`	If set to **0**, the amount is **absolute**, i.e. the amount overwrites any previous set values of that event. Note that this mode *also* overwrites any zone volume adjustments! If set to **1**, the amount is **relative** to the actual value of the event. If set to **2**, it behaves like mode **0** (**absolute** adjustment), except any zone volume adjustments are preserved. The different implications between absolute and relative adjustments are more apparent with more than one `change_vol()` statement applied to the same event.

Remarks

- `change_vol()` works on the note event level and does not change any tune settings in the instrument itself. It is also not related to any MIDI modulations regarding volume (e.g. MIDI CC #7).

Example

```
on init
    declare $vol_amount
end on

on note
    $vol_amount := (($EVENT_VELOCITY - 1) * 12000 / 126) - 6000
    change_vol ($EVENT_ID, $vol_amount, 1)
end on
```

A simple dynamic expander: lightly played notes will be softer, harder played notes will be louder.

See Also

change_tune()

change_pan()

fade_in()

fade_out()

delete_event_mark()

`delete_event_mark(<event-id>, <mark>)`	
Delete an event mark, i.e. ungroup the specified event from an event group.	
`<event-id>`	Unique identification number of the note event to be ungrouped.
`<mark>`	One of 28 marks, `$MARK_1 ... $MARK_28`, which was assigned to the event.

See Also

set_event_mark()

by_marks()

Events and MIDI: `$EVENT_ID`, `$ALL_EVENTS`, `$MARK_1` … `$MARK_28`

event_status()

`event_status(<event-id>)`

Retrieves the status of a particular note event (or MIDI event in the multi script). These are the possible states:

- `$EVENT_STATUS_NOTE_QUEUE` (note is active - instrument script only)
- `$EVENT_STATUS_MIDI_QUEUE` (MIDI event is active - multi script only)
- `$EVENT_STATUS_INACTIVE` (note or MIDI event is inactive)

Remarks

- `event_status()` can be used to find out if a note event is still "alive" or not.

Examples

```
on init
    declare %key_id[128]
end on

on note
    if (event_status(%key_id[$EVENT_NOTE]) = $EVENT_STATUS_NOTE_QUEUE)
        fade_out(%key_id[$EVENT_NOTE], 10000, 1)
    end if

    %key_id[$EVENT_NOTE] := $EVENT_ID
end on
```

Limit the number of active note events to one per MIDI key.

See Also

get_event_ids()

Events and MIDI: `$EVENT_STATUS_INACTIVE`, `$EVENT_STATUS_NOTE_QUEUE`, `$EVENT_STATUS_MIDI_QUEUE`

fade_in()

`fade_in(<event-id>, <fade-time>)`	
Performs a fade in for a specific note event.	
`<event-id>`	Unique identification number of the note event to be faded in.
`<fade-time>`	The fade in time in microseconds.

Examples

```
on init
    declare $note_1_id
    declare $note_2_id
end on

on note
    $note_1_id := play_note($EVENT_NOTE + 12, $EVENT_VELOCITY, 0, -1)
    $note_2_id := play_note($EVENT_NOTE + 19, $EVENT_VELOCITY, 0, -1)

    fade_in ($note_1_id, 1000000)
    fade_in ($note_2_id, 5000000)
end on
```

Fading in the first two harmonics.

See Also

change_vol()

fade_out()

fade_out()

`fade_out(<event-id>, <fade-time>, <stop-voice>)`	
Performs a fade-out for a specific note event.	
`<event-id>`	Unique identification number of the note event to be faded out.
`<fade-time>`	The fade out time in microseconds.
`<stop-voice>`	If set to **1**, the voice is stopped after the fade out. If set to **0**, the voice will still be running after the fade out.

Examples

```
on controller
    if ($CC_NUM = 1)
        if (%CC[1] mod 2 # 0)
            fade_out($ALL_EVENTS, 5000, 0)
        else
            fade_in($ALL_EVENTS, 5000)
        end if
    end if
end on
```

Use the modwheel on held notes to create a stutter effect.

```
on controller
    if ($CC_NUM = 1)
        fade_out($ALL_EVENTS, 5000, 1)
    end if
end on
```

A custom All Sound Off implementation triggered by the modwheel.

See Also

change_vol()

fade_out()

get_event_ids()

`get_event_ids(<array-name>)`	
Fills the specified array with all active event IDs.	
`<array-name>`	Array to be filled with active event IDs.

Remarks

- The command overwrites all existing values as long as there are active events, and writes **0** if no events are active anymore. If there are more active events than array indices, the array will be filled until it is full, ignoring the remaining event IDs.
- If there are less active events than array indices, the array will be filled from the beginning with all event IDs, followed by one array index with its value set to **0**.

Examples

```
on init
    declare const $ARRAY_SIZE := 500

    declare $a
    declare $note_count
    declare %test_array[$ARRAY_SIZE]
end on

on note
    get_event_ids(%test_array)

    $note_count := 0
    $a := 0
    while($a < $ARRAY_SIZE and %test_array[$a] # 0)
        inc($note_count)
        inc($a)
    end while

    message("Active Events: " & $note_count)
end on
```

Monitoring the number of active events.

See Also

event_status()

get_event_mark()

`get_event_mark(<event-id>, <mark>)`	
Checks if the specified event ID belongs to a specific event group (returns **1** if the bit mark is set, **0** otherwise).	
`<event-id>`	Unique identification number of the note event to be checked.
`<mark>`	The bit mark, `$MARK_1 ... $MARK_28`. You can also select more than one event group by using the bitwise `.or.` operator, or by simply summing the event marks.

Examples

```
on note
    if ($EVENT_NOTE mod 12 = 0)
        set_event_mark($EVENT_ID, $MARK_1)
    end if
end on

on release
    if (get_event_mark($EVENT_ID, $MARK_1) = 1)
        message("You've played a C!")
    else
        message("")
    end if
end on
```

A rather long-winded way to check if you've released a C key.

See Also

set_event_mark()

get_event_par()

`get_event_par(<event-id>, <parameter>)`	
Returns the value of a specific event parameter of the specified event.	
`<event-id`	Unique identification number of the note event to be changed.
`<parameter>`	The event parameter, either one of four freely assignable event parameters: • `$EVENT_PAR_0` • `$EVENT_PAR_1` • `$EVENT_PAR_2` • `$EVENT_PAR_3` or the "built-in" parameters of a note event: • `$EVENT_PAR_VOLUME` • `$EVENT_PAR_PAN` • `$EVENT_PAR_TUNE` • `$EVENT_PAR_NOTE` • `$EVENT_PAR_VELOCITY` • `$EVENT_PAR_REL_VELOCITY` • `$EVENT_PAR_MIDI_CHANNEL` • `$EVENT_PAR_SOURCE` • `$EVENT_PAR_PLAY_POS` • `$EVENT_PAR_ZONE_ID` (use with care, see below)

Remarks

- A note event always carries certain information like the note number, the played velocity, but also volume, pan and tune. With `get_event_par()`, you can get either these parameters or use the freely assignable parameters like `$EVENT_PAR_0`. This is especially useful when chaining scripts, i.e. set an event parameter for an event in script slot 1, then retrieve this information in script slot 2 by using `get_event_par()`.

Examples

```
on note
    message(get_event_par($EVENT_ID, $EVENT_PAR_NOTE))
end on
```

The same functionality as ***`message($EVENT_NOTE)`***.

```
on note
    message(get_event_par($EVENT_ID, $EVENT_PAR_SOURCE))
end on
```

Check if the event comes from outside (returns ***-1*** *in this case) or if it was created in one of the five script slots (returns* ***0-4****).*

```
on note
    wait(1)
    message(get_event_par($EVENT_ID, $EVENT_PAR_ZONE_ID))
end on
```

An event itself does not carry a zone ID (only a voice can carry zone IDs), therefore you need to insert ***`wait(1)`*** *in order to properly retrieve the zone ID.*

See Also

set_event_par()
ignore_event()
set_event_par_arr()
get_event_par_arr()

get_event_par_arr()

`get_event_par_arr(<event-id>, <parameter>, <index>)`	
Retrieves the value of a specified event parameter of a specific event.	
`<event-id>`	Unique identification number of the note event.
`<parameter>`	Can be one of the following: • `$EVENT_PAR_ALLOW_GROUP` • `$EVENT_PAR_CUSTOM` • `$EVENT_PAR_MOD_VALUE_ID` • `$EVENT_PAR_MOD_VALUE_EX_ID`
`<index>`	When used with: • `$EVENT_PAR_ALLOW_GROUP`: the group index (**0 ... 4095**, however this depends on the amount of groups present in a particular Kontakt instrument) • `$EVENT_PAR_CUSTOM`: the event parameter index (**0 ... 15**) • `$EVENT_PAR_MOD_VALUE_ID/$EVENT_PAR_MOD_VALUE_EX_ID`: the "from script" modulator index, as set in the modulation assignment in Kontakt (**0 ... 1000**)

Remarks

- `get_event_par_arr()` is the array variant of `get_event_par()`. You can use it to retrieve the group allow state of a specific event, or if you need to access more than the four standard event parameters. You can also use it to retrieve the value of event-specific modulations, facilitated by "from script" modulators in Kontakt.

Examples

```
on init
    declare $count
    declare ui_label $label (2, 4)
    set_text($label, "")
end on

on note
    set_text($label, "")

    $count := 0
    while ($count < $NUM_GROUPS)
        if (get_event_par_arr($EVENT_ID, $EVENT_PAR_ALLOW_GROUP, $count) = 1)
            add_text_line($label,"Group ID " & $count & " allowed")
        else
            add_text_line($label,"Group ID " & $count & " disallowed")
        end if

        inc($count)
    end while
end on
```

A simple group monitor.

See Also

set_event_par_arr()

get_event_par()

Events and MIDI: `$EVENT_PAR_ALLOW_GROUP`, `$EVENT_PAR_MOD_VALUE_ID`, `$EVENT_PAR_CUSTOM` , `%GROUPS_AFFECTED`

ignore_event()

`ignore_event(<event-id>)`

Ignores a note event in `on note` or `on release` callbacks.

Remarks

- If you ignore an event, any volume, tune or pan information is lost. You can however retrieve this infomation with get_event_par(), refer to the two examples below.
- `ignore_event()` is a very "strong" command. Always check if you can get the same results with the various `change_...()` commands without having to ignore the event.

Examples

```
on note
    ignore_event($EVENT_ID)
    wait(500000)
    play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, -1)
end on
```

Delaying all notes by half a second. Not bad, but if you, for example insert a microtuner before this script, the tuning information will be lost.

```
on init
    declare $new_id
end on

on note
    ignore_event($EVENT_ID)
    wait(500000)
    $new_id := play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, -1)

    change_vol($new_id, get_event_par($EVENT_ID, $EVENT_PAR_VOLUME), 1)
    change_tune($new_id, get_event_par($EVENT_ID, $EVENT_PAR_TUNE), 1)
    change_pan($new_id, get_event_par($EVENT_ID, $EVENT_PAR_PAN), 1)
end on
```

Better: the tuning (plus volume and pan, to be precise) information is retrieved and applied to the played note.

See Also

ignore_controller

get_event_par()

redirect_output()

`redirect_output(<event-id>, <output-type>, <index>)`	
Routes the audio signal of the specified event to a specific output or bus.	
`<event-id>`	Unique identification number of the note event to be routed.
`<output-type>`	Can be one of the following: • `$OUTPUT_TYPE_DEFAULT`: The audio signal of the event is routed to the default instrument output. • `$OUTPUT_TYPE_MASTER_OUT`: The audio signal of the event is routed directly to the output channel specified with `<index>` (**0 ... 63**, depending on number of output channels defined in Kontakt). The audio signal will not be affected by any instrument effect. • `$OUTPUT_TYPE_AUX_OUT`: The audio signal of the event is routed directly to the Aux channel specified with `<index>`. (**0 ... 3**) The audio signal will not be affected by any instrument effect. • `$OUTPUT_TYPE_BUS_OUT`: The audio signal of the event is routed to the instrument bus specified with `<index>` (**0 ... 15**).
`<index>`	Specifies the output channel, aux channel or instrument bus, depending on `<output-type>`. Has no effect when `<output-type>` is set to `$OUTPUT_TYPE_DEFAULT`.

Remarks

- When using `redirect_output()`, the output selection of a group is completely ignored.

Examples

```
on init
    declare $new_id_0
    declare $new_id_1
    decalre $new_id_2
end on

on note
    ignore_event($EVENT_ID)

    $new_id_0 := play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, -1)
    $new_id_1 := play_note($EVENT_NOTE + 4, $EVENT_VELOCITY, 0, -1)
    $new_id_2 := play_note($EVENT_NOTE + 7, $EVENT_VELOCITY, 0, -1)

    redirect_output($new_id_0, $OUTPUT_TYPE_BUS_OUT, 0)
    redirect_output($new_id_1, $OUTPUT_TYPE_BUS_OUT, 1)
    redirect_output($new_id_2, $OUTPUT_TYPE_BUS_OUT, 2)
end on
```

Creating a major triad and routing each note to a separate instrument bus.

set_event_mark()

`set_event_mark(<event-id>, <mark>)`	
Assigns the specified event ID to a specific event group.	
`<event-id>`	Unique identification number of the note event.
`<mark>`	One of 28 marks, from `$MARK_1` to `$MARK_28` which will be assigned to the event. You can also assign more than one mark to a single event, either by typing the command again, or by using the bitwise `.or.` operator, or by simply summing the event marks.

Remarks

- When working with commands that deal with event IDs, you can group events by using `by_marks()` instead of using individual IDs, as Kontakt needs to know that you want to address marks, and not IDs.

Examples

```
on init
    declare $new_id
end on

on note
    set_event_mark($EVENT_ID, $MARK_1)

    $new_id := play_note($EVENT_NOTE + 12, 120, 0, -1)
    set_event_mark($new_id, $MARK_1 + $MARK_2)

    change_pan(by_marks($MARK_1), -1000, 1) { both notes panned to left }
    change_pan(by_marks($MARK_2), 2000, 1) { new note panned to right }
end on
```

The played note belongs to event mark 1, the harmonized note belongs to both event marks 1 and 2.

See Also

by_marks()

delete_event_mark()

Events and MIDI: `$EVENT_ID`, `$ALL_EVENTS`, `$MARK_1` … `$MARK_28`

set_event_par()

`set_event_par(<event-id>, <parameter>, <value>)`	
Assigns a specific event parameter value to a specific event.	
`<event-id>`	Unique identification number of the note event.
`<parameter>`	The event parameter, either one of four freely assignable event parameters: • `$EVENT_PAR_0` • `$EVENT_PAR_1` • `$EVENT_PAR_2` • `$EVENT_PAR_3` or the "built-in" parameters of a note event: • `$EVENT_PAR_VOLUME` • `$EVENT_PAR_PAN` • `$EVENT_PAR_TUNE` • `$EVENT_PAR_NOTE` • `$EVENT_PAR_VELOCITY` • `$EVENT_PAR_REL_VELOCITY` • `$EVENT_PAR_MIDI_CHANNNEL`
`<value>`	The value of the event parameter.

Remarks

- A note event always carries certain information like the note number, the played velocity, but also volume, pan and tune. With `set_event_par()`, you can set either these parameters or use the freely assignable parameters like `$EVENT_PAR_0`. This is especially useful when chaining scripts, i.e. set an event parameter for an event in script slot 1, then retrieve this information in script slot 2 by using `get_event_par()`.
- If you need access to more than four custom parameters, please use `set_event_par_arr()` with `$EVENT_PAR_CUSTOM`.

Examples

```
on note
    set_event_par($EVENT_ID, $EVENT_PAR_NOTE, 60)
end on
```

Setting all notes to middle C3, same as `change_note($EVENT_ID, 60)`.

```
on init
    message("")
    declare ui_switch $switch

    declare ui_label $midiChan1 (1, 1)
    declare ui_label $midiChan2 (1, 1)
    declare ui_label $midiChan3 (1, 1)
    declare ui_label $midiChan4 (1, 1)
    declare ui_label $midiChan5 (1, 1)
    declare ui_label $midiChan6 (1, 1)
    declare ui_label $midiChan7 (1, 1)
    declare ui_label $midiChan8 (1, 1)
    declare ui_label $midiChan9 (1, 1)
    declare ui_label $midiChan10 (1, 1)
    declare ui_label $midiChan11 (1, 1)
    declare ui_label $midiChan12 (1, 1)
    declare ui_label $midiChan13 (1, 1)
    declare ui_label $midiChan14 (1, 1)
    declare ui_label $midiChan15 (1, 1)
    declare ui_label $midiChan16 (1, 1)

    declare %midiChans[16]
    %midiChans[0] := get_ui_id($midiChan1)
    %midiChans[1] := get_ui_id($midiChan2)
    %midiChans[2] := get_ui_id($midiChan3)
    %midiChans[3] := get_ui_id($midiChan4)
    %midiChans[4] := get_ui_id($midiChan5)
    %midiChans[5] := get_ui_id($midiChan6)
    %midiChans[6] := get_ui_id($midiChan7)
    %midiChans[7] := get_ui_id($midiChan8)
    %midiChans[8] := get_ui_id($midiChan9)
    %midiChans[9] := get_ui_id($midiChan10)
    %midiChans[10] := get_ui_id($midiChan11)
    %midiChans[11] := get_ui_id($midiChan12)
    %midiChans[12] := get_ui_id($midiChan13)
    %midiChans[13] := get_ui_id($midiChan14)
    %midiChans[14] := get_ui_id($midiChan15)
    %midiChans[15] := get_ui_id($midiChan16)
end on

on release
    if ($switch = 1)
        set_event_par($EVENT_ID, $EVENT_PAR_REL_VELOCITY, 127)
    end if

    set_control_par_str(%midiChans[$MIDI_CHANNEL], $CONTROL_PAR_TEXT,
get_event_par($EVENT_ID, $EVENT_PAR_REL_VELOCITY))
end on
```

Release velocity in MPE (MIDI Polyphonic Expression) context.

See Also

get_event_par()

ignore_event()

set_event_par_arr()

get_event_par_arr()

set_event_par_arr()

`set_event_par_arr(<event-id>, <parameter>, <value>, <index>)`	
Assigns an event parameter array to a specific event.	
`<event-id>`	Unique identification number of the note event.
`<parameter>`	Can be one of the following: • `$EVENT_PAR_ALLOW_GROUP` • `$EVENT_PAR_CUSTOM` • `$EVENT_PAR_MOD_VALUE_ID` • `$EVENT_PAR_MOD_VALUE_EX_ID`
`<value>`	When used with: • `$EVENT_PAR_ALLOW_GROUP`: the allow state for the group (**1** for allowed, **0** for disallowed) • `$EVENT_PAR_CUSTOM`: the value of the specified event parameter • `$EVENT_PAR_MOD_VALUE_ID`: the modulation value to be sent to "from script" modulator (clamped internally between **-1000000** and **1000000**) • `$EVENT_PAR_MOD_VALUE_EX_ID`: the modulation value to be sent to "from script" modulator (unbounded value range)
`<index>`	When used with: • `$EVENT_PAR_ALLOW_GROUP`: the group index (**0 ... 4095**, however this depends on the amount of groups present in a particular Kontakt instrument) • `$EVENT_PAR_CUSTOM`: the event parameter index (**0 ... 15**) • `$EVENT_PAR_MOD_VALUE_ID/$EVENT_PAR_MOD_VALUE_EX_ID`: the "from script" modulator index, as set in the modulation assignment in Kontakt (**0 ... 1000**)

Remarks

- `set_event_par_arr()` is the array variant of `set_event_par()`. You can use it to set the group allow state of a specific event, or if you need to access more than the four standard event parameters. You can also use it to set up event-specific modulations, facilitated by "from script" modulators in Kontakt.

Examples

```
on note
    if (get_event_par_arr($EVENT_ID, $EVENT_PAR_ALLOW_GROUP, 0) = 0)
        set_event_par_arr($EVENT_ID, $EVENT_PAR_ALLOW_GROUP, 1, 0)
    end if
end on
```

Making sure the first group is always played.

```
on init
    declare const $CUSTOM_EVENT_PAR_4 := 4
end on

on note
    set_event_par_arr($EVENT_ID, $EVENT_PAR_CUSTOM, $ENGINE_UPTIME,
$CUSTOM_EVENT_PAR_4)
end on
```

```
on release
    message(get_event_par_arr($EVENT_ID, $EVENT_PAR_CUSTOM, $CUSTOM_EVENT_PAR_4))
end on
```

Simple implementation of `$EVENT_PAR_CUSTOM`.

```
on note
    if ($EVENT_NOTE = 60)
        set_event_par_arr($EVENT_ID, $EVENT_PAR_MOD_VALUE_ID, 500000, 1)
    end if
end on
```

Only middle C (MIDI note 60) will have any modulation applied, facilitated by "from script" modulator that has its ID set to "1".

See Also

allow_group()

disallow_group()

get_event_par_arr()

set_event_par()

Events and MIDI: `$EVENT_PAR_ALLOW_GROUP`

set_map_editor_event_color()

`set_map_editor_event_color(<hex-value>)`	
Assigns the specified color to events generated in the current script slot, visible in Kontakt's Mapping Editor.	
`<hex value>`	The hexadecimal color value in the following format: `0ff0000h {red}` The **0** at the start lets Kontakt know the value is a number. The **h** at the end indicates that it is a hexadecimal value. You can also use uppercase **H**.

Remarks

- This command is only available in `on init` callback.
- The specified color will always be drawn 50% opaque, so that cases of overlapping events from multiple script slots could be discerned.
- This command will only work if script generated events are allowed to be displayed in Kontakt's Mapping Editor (option available in Mapping Editor's Edit menu).

Examples

```
on init
    set_map_editor_event_color(000FF00h)
end on

on note
    play_note(($EVENT_NOTE + 12) mod 127, $EVENT_VELOCITY, 0, -1)
end on
```

Add a note played an octave up. This event will be shown as a green blip in Kontakt's Mapping Editor.

11. Array Commands

array_equal()

`array_equal(<array-variable>, <array-variable>)`

Checks the values of two arrays. Returns **1** if all values are equal, **0** if not.

Remarks

- This command does not work with arrays of real numbers.

Examples

```
on init
    declare %array_1[10]
    declare %array_2[11]

    if (array_equal(%array_1, %array_2))
        message("Arrays are not equal!")
    else
        message("Arrays are equal!")
    end if
end on
```

This script will produce an error message as the two arrays don't have the same size.

See Also

sort()

num_elements()

search()

num_elements()

`num_elements(<array-variable>)`

Returns the number of elements in an array.

Remarks

- With this function you can, e.g., check how many groups are affected by the current event, using `num_elements(%GROUPS_AFFECTED)`.

Examples

```
on note
    message(num_elements(%GROUPS_AFFECTED))
end on
```

Outputs the number of groups that are playing when you press a key.

See Also

array_equal()

sort()

search()

Events and MIDI: `%GROUPS_AFFECTED`

search()

```
search(<array-variable>, <value>)
search(<array-variable>, <value>, <from>, <to>)
```

Searches the specified array for the specified value (optionally within the range specified by `<from>` and `<to>`) and returns the index of its first position. If the value is not found, the function returns **-1**.

`<array-variable>`	Array to be searched through
`<value>`	Value to be found in the specified array.
`<from>`	Optional argument which specifies the array index from which to start the searching operation.
`<to>`	Optional argument which specifies the array index at which searching operation will end.

Remarks

- This command does not work with arrays of real numbers.

Examples

```
on init
    declare ui_table %array[10] (2, 2, 5)
    declare ui_button $check

    set_text($check, "Zero present?")
end on

on ui_control ($check)
    if (search(%array, 0) = -1)
        message("No")
    else
        message("Yes")
    end if

    $check := 0
end on
```

Checking if a specific value is present in an array.

```
on init
    declare const $SEARCH_FOR  := 54321
    declare const $SEARCH_FROM := 54000
    declare const $SEARCH_TO   := 55000

    declare $i
    declare %array[100000]

    declare ui_button $Check

    set_text($Check, $SEARCH_FOR & " present?")

    { fill the array with sequential numbers, just to have something to search
through }
    while ($i < num_elements(%array))
        %array[$i] := $i

        inc($i)
    end while
end on

on ui_control ($Check)
    if (search(%array, $SEARCH_FOR, $SEARCH_FROM, $SEARCH_TO) = -1)
        message("No")
    else
        message("Yes")
    end if

    $Check := 0
end on
```

Searching for a specific value in a smaller part of a large array - much more performant than doing the same thing with a while loop directly in KSP!

See Also

array_equal()

num_elements()

sort()

sort()

`sort(<array-variable>, <direction>)`
`sort(<array-variable>, <direction>, <from>, <to>)`	

Sorts an array in ascending or descending order (optionally within the range specified by `<from>` and `<to>`).	
`<array-variable>`	The array to be sorted.
`<direction>`	When equal to **0**, the array is sorted in ascending order. When *not* equal to **0**, the array is sorted in descending order.
`<from>`	Optional argument which specifies the array index from which to start the sorting operation.
`<to>`	Optional argument which specifies the array index at which sorting operation will end.

Examples

```
on init
    declare $count

    declare ui_button $Invert
    declare ui_table %array[128] (3, 3, 127)

    while ($count < 128)
        %array[$count] := $count

        inc($count)
    end while
end on

on ui_control ($Invert)
    sort(%array, $Invert)
end on
```

Quickly inverting a linear curve display.

```
on init
    declare const $ARRAY_SIZE := 32

    declare ui_table %T[$ARRAY_SIZE](6, 4, -100)
    declare ui_value_edit $From (1, $ARRAY_SIZE, 1)
    declare ui_value_edit $To (1, $ARRAY_SIZE, 1)
    declare ui_button $Randomize
    declare ui_button $Direction
    declare ui_button $SortAll
    declare ui_button $SortRange

    make_persistent(%T)
    make_persistent($From)
    make_persistent($To)

    $From := 4
    $To := 8
end on

on ui_control ($Randomize)
    $i := 0
    while ($i < num_elements(%T))
        %T[$i] := random(-100, 100)

        inc($i)
    end while

    $Randomize := 0
end on

on ui_control ($SortAll)
    sort(%T, $Direction)

    $SortAll := 0
end on

on ui_control ($SortRange)
    sort(%T, $Direction, $From, $To)

    $SortRange := 0
end on
```

Comparing sorting the whole array versus sorting a range within the array.

See Also

array_equal()

num_elements()

sort()

12. Group Commands

allow_group()

`allow_group(<group-index>)`

Allows the specified group, i.e. makes it available for playback.

Remarks

- This commmand is only available in `on note` and `on release` callbacks.
- The numbering of the group index is zero-based, i.e. index of the first instrument group is **0**.
- The group allow states can only be changed if the voice is not running.

Examples

```
on note
    disallow_group($ALL_GROUPS)
    allow_group(0)
end on
```

Only the first group will play back.

See Also

disallow_group()

set_event_par_arr()

Events and MIDI: `$ALL_GROUPS`, `$EVENT_PAR_ALLOW_GROUP`

disallow_group()

`disallow_group(<group-index>)`

Disallows the specified group, i.e. makes it unavailable for playback.

Remarks

- This commmand is only available in `on note` and `on release` callbacks.
- The numbering of the group index is zero-based, i.e. index of the first instrument group is **0**.
- The group disallow states can only be changed if the voice is not running.

Examples

```
on init
    declare $count
    declare ui_menu $groups_menu

    add_menu_item($groups_menu, "Play All", -1)

    while ($count < $NUM_GROUPS)
        add_menu_item($groups_menu, "Mute: " & group_name($count), $count)

        inc($count)
    end while
end on

on note
    if ($groups_menu # -1)
        disallow_group($groups_menu)
    end if
end on
```

Muting one specific group of an instrument.

See Also

allow_group()

set_event_par_arr()

Events and MIDI: `$ALL_GROUPS`, `$EVENT_PAR_ALLOW_GROUP`

get_group_idx()

`get_group_idx(<group-name>)`

Returns the group index for the specified group name.

Remarks

- If no group with the specified name is found, this command will return `$NI_NOT_FOUND`.

Examples

```
on init
    declare $group_idx
end on

on note
    $group_idx := get_group_idx("Accordion")

    if ($group_idx # $NI_NOT_FOUND)
        disallow_group($group_idx)
    end if
end on
```

A simple, yet useful script.

See Also

allow_group()

disallow_group()

group_name()

get_purge_state()

`get_purge_state(<group-index>)`	
Returns the purge state of the specified group. **0:** The group is purged. **1:** The group is not purged, i.e. the samples are loaded.	
`<group-index>`	The index number of the group that should be checked.

Examples

```
on init
    declare ui_button $purge
    declare ui_button $checkpurge

    set_text($purge, "Purge first group")
    set_text($checkpurge, "Check purge status")
end on

on ui_control ($purge)
    { 1 - $purge inverts the behaviour of the button, here }
    purge_group(0, 1 - $purge)
end on

on ui_control ($checkpurge)
    if (get_purge_state(0) = 0)
        message(“Group is purged.”)
    else
        message(“Group is not purged.”)
    end if
end on
```

A simple purge check.

See Also

purge_group()

group_name()

`group_name(<group-index>)`

Returns the group name for the specified group.

Remarks

- The numbering of the group index is zero-based, i.e. index of the first instrument group is **0**.

Examples

```
on init
    declare $count
    declare ui_menu $groups_menu

    $count := 0
    while ($count < $NUM_GROUPS)
        add_menu_item ($groups_menu, group_name($count), $count)

        inc($count)
    end while
end on
```

Quickly creating a menu with all available groups.

```
on init
    declare $count
    declare ui_label $label (2, 6)

    set_text($label, "")
end on

on note
    $count := 0
    while ($count < num_elements(%GROUPS_AFFECTED))
        add_text_line($label, group_name(%GROUPS_AFFECTED[$count]))

        inc($count)
    end while
end on

on release
    set_text($label, "")
end on
```

Query the status of the first 1001 zone IDs.

See Also

allow_group()

disallow_group()

get_group_idx()

output_channel_name()

Events and MIDI: `$ALL_GROUPS`, `$NUM_GROUPS`

purge_group()

`purge_group(<group-index>, <mode>)`	
Purges (i.e. unloads from RAM) the samples of the specified group.	
`<group-index>`	The index number of the group which contains the samples to be purged.
`<mode>`	If set to 0, the samples of the specified group are unloaded. If set to 1, the samples are reloaded.

Remarks

- When using `purge_group()` in a while loop, don't use any wait() commands within the loop.
- `purge_group()` can only be used in on ui_control, on ui_controls and on persistence_changed callbacks.
- It is recommended not to use the `purge_group()` command in UI callbacks of automatable controls.
- It is possible to supply an async ID to the `purge_group()` function and get a return in the on async_complete callback.

Examples

```
on init
    declare $async_id := -1
    declare ui_button $purge

    set_text($purge,"Purge first group")
end on

on ui_control ($purge)
    $async_id := purge_group(0, abs($purge - 1))
end on

on async_complete
    if ($NI_ASYNC_ID = $async_id)
        if (get_purge_state(0) = 0)
            message("Group is purged")
        else
            message("Group is not purged")
        end if
    end if
end on
```

Unloading all samples of the first group.

See Also

get_purge_state()

13. Time-Related Commands

change_listener_par()

`change_listener_par(<signal-type>, <parameter>)`	
Changes the parameters of the `on listener` callback. It can be used in any callback.	

`<signal-type>`	The signal to be changed, can be:
	`$NI_SIGNAL_TIMER_MS`
	`$NI_SIGNAL_TIMER_BEAT`

`<parameter>`	Dependent on the specified signal type: `$NI_SIGNAL_TIMER_MS`: Time interval in microseconds (minimum value is **1000** microseconds, which equals one millisecond) `$NI_SIGNAL_TIMER_BEAT`: Time interval in fractions of a beat/quarter note (**1 ... 24**)

Remarks

- It is also possible to completely disable a particular listener signal by setting `<parameter>` to **0**.

Examples

```
on init
    declare ui_value_edit $Tempo (20, 300, 1)
    declare ui_switch $Play

    make_persistent($Tempo)
    read_persistent_var($Tempo)

    $Tempo := 120

    set_listener($NI_SIGNAL_TIMER_MS, 60000000 / $Tempo)
end on

on listener
    if ($NI_SIGNAL_TYPE = $NI_SIGNAL_TIMER_MS and $Play = 1)
        play_note(60, 127, 0, $DURATION_EIGHTH)
    end if
end on

on ui_control($Tempo)
    change_listener_par($NI_SIGNAL_TIMER_MS, 60000000 / $Tempo)
end on
```

A very basic metronome.

See Also

on listener

set_listener()

Callbacks and UI: `$NI_SIGNAL_TYPE`

ms_to_ticks()

```
ms_to_ticks(<microseconds>)
```

Converts a microseconds value into a value in tempo-dependent MIDI ticks.

Examples

```
on init
    declare ui_label $bpm (1, 1)
    set_text($bpm, ms_to_ticks(60000000) / 960)
end on
```

Displaying the current host tempo.

See Also

ticks_to_ms()

Time and Transport: `$NI_SONG_POSITION`

set_listener()

`set_listener(<signal-type>, <parameter>)`	
Sets the signals on which the listener callback should react to. Can only be used in the `on init` callback.	

`<signal-type>`	The event on which the listener callback should react. The following types are available:
	`$NI_SIGNAL_TRANSP_STOP`
	`$NI_SIGNAL_TRANSP_START`
	`$NI_SIGNAL_TIMER_MS`
	`$NI_SIGNAL_TIMER_BEAT`

`<parameter>`	User defined parameter, dependant on the specified signal type: `$NI_SIGNAL_TIMER_MS`: Time interval in microseconds (minimum value is **1000**, which equals one millisecond) `$NI_SIGNAL_TIMER_BEAT`: Time interval in fractions of a beat/quarter note (**1 ... 24**) `$NI_SIGNAL_TRANSP_START`: Set to **1** if the listener callback should react to the host's transport start command `$NI_SIGNAL_TRANSP_STOP`: Set to **1** if the listener callback should react to the host's transport stop command

Remarks

- When using `$NI_SIGNAL_TIMER_BEAT`, the maximum resolution is **24** ticks per beat/quarter note.

Examples

```
on init
    set_listener($NI_SIGNAL_TIMER_BEAT, 1)
end on

on listener
    if ($NI_SIGNAL_TYPE = $NI_SIGNAL_TIMER_BEAT)
        message($ENGINE_UPTIME)
    end if
end on
```

Triggering the listener callback every beat. Triggering will occur even when transport is stopped.

See Also

change_listener_par()

Callbacks and UI: `$NI_SIGNAL_TYPE`

stop_wait()

`stop_wait(<callback-id>, <parameter>)`	
Stops wait commands in the specified callback.	
`<callback-id>`	The callback's ID number in which the wait commands will be stopped.
`<parameter>`	**0**: stops only the current wait. **1**: stops the current wait and ignores all following wait commands in this callback.

Remarks

- Be careful with while loops when stopping all `wait()` commands in a callback!

Examples

```
on init
    declare $id

    declare ui_button $Play
end on

on ui_control ($Play)
    if ($Play = 1)
        $id := $NI_CALLBACK_ID

        play_note(60, 127, 0, $DURATION_QUARTER)

        wait($DURATION_QUARTER)

        if ($Play = 1)
            play_note(64, 127, 0, $DURATION_QUARTER)
        end if

        wait($DURATION_QUARTER)

        if ($Play = 1)
            play_note(67, 127, 0, $DURATION_QUARTER)
        end if
    else
        stop_wait($id, 1)
        fade_out($ALL_EVENTS, 10000, 1)
    end if
end on
```

The Play button triggers a simple triad arpeggio. Without the stop_wait() command, parallel callbacks could occur when pressing the Play button quickly in succession resulting in multiple arpeggios.

See Also

wait()

wait_async()

wait_ticks()

Callbacks and UI: Callback Type Variables and Constants

reset_ksp_timer

`reset_ksp_timer`
Resets the KSP timer built-in variable (`$KSP_TIMER`) to zero.

Remarks

- Note that the `$KSP_TIMER` variable, due to its 32-bit signed nature, will reach its limit after 2147483648 microseconds, or roughly 35 minutes and 47 seconds.
- Since the KSP timer is based on the CPU clock, the main reason to use it is for debugging and optimization. It is a great tool to measure the efficiency of certain script passages. However, it should not be used for musical timing, as it remains at a real-time constant rate, even if Kontakt is being used in an offline bounce.

Examples

```
on init
    declare $a
    declare $b
    declare $c
end on

on note
    reset_ksp_timer

    $c := 0
    while ($c < 128)
        $a := 0
        while($a < 128)
            set_event_par($EVENT_ID, $EVENT_PAR_TUNE, random(-1000, 1000))

            inc($a)
        end while

        inc($c)
    end while

    message($KSP_TIMER)
end on
```
A nested while loop – takes about 5400 to 5800 microseconds.

See Also

Time and Transport: `$ENGINE_UPTIME`, `$KSP_TIMER`

ticks_to_ms()

`ticks_to_ms(<ticks>)`

Converts a tempo-dependent MIDI ticks value into a value in microseconds.

Remarks

- Since the returned value is in microseconds, note that due to its 32-bit signed nature it will not return correct values if specified number of ticks at the current tempo exceeds 2147483648 microseconds, or roughly 35 minutes and 47 seconds.

Examples

```
on init
    declare $msec
    declare $sec
    declare $min


    declare ui_label $label (2, 1)

    set_listener($NI_SIGNAL_TIMER_MS, 1000)
end on

on listener
    if ($NI_SIGNAL_TYPE = $NI_SIGNAL_TIMER_MS)
        $msec := ticks_to_ms($NI_SONG_POSITION) / 1000
        $sec := $msec / 1000
        $min := $sec / 60

        set_text($label, $min & ":" & $sec mod 60 & "." & $msec mod 1000)
    end if
end on
```

Displaying the song position in realtime.

See Also

ms_to_ticks()

Time and Transport: `$NI_SONG_POSITION`

wait()

`wait(<wait-time>)`

Pauses the callback for the specified time in microseconds.

Remarks

- `wait()` stops the callback at the position in the script for the specified time. In other words, it freezes the callback, although other callbacks can still be processed during this time. After the specified time, period the callback continues.

Examples

```
on note
    ignore_event($EVENT_ID)

    if ($DURATION_BAR = 0)
        wait(($DURATION_QUARTER * 4) - $DISTANCE_BAR_START)
    else
        wait($DURATION_BAR - $DISTANCE_BAR_START)
    end if

    play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, -1)
end on
```

Quantize all notes to the downbeat of the next measure. This script also takes care of the fact that in Kontakt standalone, $DURATION_BAR returns 0, so instead of that we use the quarter note duration to make up a 4/4 bar.

See Also

stop_wait()

wait_async()

wait_ticks()

while ()

Time and Transport: `$DURATION_QUARTER`

wait_async()

`wait_async(<async-id>)`

Waits until the async command identified by the `<async-id>` is finished.

Remarks

- When performing multiple operations it is also possible to collect them together and then calling the `wait_async()` function on the collection. When the operations are collected in this manner they will be calculated in one block, resulting in a performance gain. If the async operation is not in the pipeline anymore or is invalid, there is no waiting and the script continues.

Examples

```
wait_async(set_engine_par($ENGINE_PAR_EFFECT_TYPE, ... $EFFECT_TYPE_CHORUS, -1, 2,
1))
```

Performing a single async operation.

```
%asyncid[0] := async_operation
%asyncid[1] := another_async_operation
...
%asyncid[x] := last_async_operation

$i := 0
while($i < num_elements(%asyncid))
    wait_async(%asyncid[$i])

    inc($i)
end while
```

Performing multiple async operations.

See also

General: `$NI_ASYNC_EXIT_STATUS`, `$NI_ASYNC_ID`

wait_ticks()

```
wait_ticks(<ticks>)
```

Pauses the callback for the specified time in MIDI ticks.

Remarks

- Same as `wait()` but with MIDI ticks as the wait time parameter.
- 960 MIDI ticks equals one quarter note.

See Also

stop_wait()

wait()

14. User Interface Commands

add_menu_item()

`add_menu_item(<variable>, <text>, <value>)`	
Create an entry in a `ui_menu` widget.	
`<variable>`	The variable name of the `ui_menu` widget.
`<text>`	The text of the menu entry.
`<value>`	The value of the menu entry.

Remarks

- You can create menu entries only in the `on init` callback, but you can change their text and value afterwards by using `set_menu_item_str()` and `set_menu_item_value()`. You can add as many menu entries as you want, then show or hide them dynamically by using `set_menu_item_visibility()`.
- Using the `$CONTROL_PAR_VALUE` control parameter with the `get_control_par()` command will return the currently selected menu *index*, **not** the currently selected *value*. If you want to get the menu value instead, use the `get_menu_item_value()` command.

Examples

```
on init
    declare ui_menu $menu
    add_menu_item($menu, "First Entry", 0)
    add_menu_item($menu, "Second Entry", 1)
    add_menu_item($menu, "Third Entry", 2)
end on
```

A simple menu.

See Also

get_menu_item_str()

get_menu_item_value()

get_menu_item_visibility()

set_menu_item_str()

set_menu_item_visibility()

ui_menu

Specific: `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`

add_text_line()

`add_text_line(<variable>, <text>)`	
Add a new text line in the specified `ui_label`, without erasing existing text.	
`<variable>`	The variable name of the `ui_label` widget.
`<text>`	The text to be displayed.

Examples

```
on init
    declare $count
    declare ui_label $label (1, 4)

    set_text($label, "")
end on

on note
    inc($count)

    select ($count)
        case 1
            set_text($label, $count & ": " & $EVENT_NOTE)
        case 2 to 4
            add_text_line($label, $count & ": " & $EVENT_NOTE)
    end select

    if ($count = 4)
        $count := 0
    end if
end on
```
Monitoring the last four played notes.

See Also

set_text()

ui_label

attach_level_meter()

`attach_level_meter(<ui-id>, <group>, <slot>, <channel>, <generic>)`	
Attach a `ui_level_meter` to a certain location within the instrument to read volume data.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<group>`	The index of the group you want to access. Should be set to **-1** if not using the group level.
`<slot>`	The index of the FX slot you wish to access. Should be set to **-1** if not accessing an FX slot.
`<channel>`	Select from **0** to **15** to set the audio channel the level meter will be displaying.
`<generic>`	Can be one of the following: • `NI_LEVEL_METER_MAIN`: used to access Main FX chain • `NI_LEVEL_METER_GROUP`: used to access the Group FX chain • `NI_LEVEL_METER_INSERT`: used to access the Insert FX chain • `0 ... 15`: used to access individual instrument buses

Remarks

- Level meters can be attached to the output of an instrument bus and the instrument main output. They can also be attached to compressor and limiter effects to display gain reduction data, with the ability to set minimum and maximum display values and inverting the display by using `$CONTROL_PAR_RANGE_MIN` and `$CONTROL_PAR_RANGE_MAX` control parameters.

Examples

```
on init
    declare const $GROUP_IDX := 0
    declare const $BUS_IDX := 0
    declare const $SLOT_IDX := 0
    declare const $CHANNEL_L := 0
    declare const $CHANNEL_R := 1

    declare ui_label $InstOutputL (1, 1)
    declare ui_label $InstOutputR (1, 1)
    declare ui_label $BusOutput (1, 1)
    declare ui_label $MainFX (1, 1)
    declare ui_label $BusFX (1, 1)
    declare ui_label $GroupFX (1, 1)

    declare ui_level_meter $inst_output_l_lvl
    declare ui_level_meter $inst_output_r_lvl
    declare ui_level_meter $bus_output_lvl
    declare ui_level_meter $mainfx_output_lvl
    declare ui_level_meter $busfx_output_lvl
    declare ui_level_meter $groupfx_output_lvl

    attach_level_meter(get_ui_id($inst_output_l_lvl), -1, -1, $CHANNEL_L, -1)
    attach_level_meter(get_ui_id($inst_output_r_lvl), -1, -1, $CHANNEL_R, -1)
    attach_level_meter(get_ui_id($bus_output_lvl), -1, -1, $CHANNEL_L, $BUS_IDX)
    attach_level_meter(get_ui_id($mainfx_output_lvl), -1, $SLOT_IDX, $CHANNEL_L, -2)
    attach_level_meter(get_ui_id($busfx_output_lvl), -1, $SLOT_IDX, $CHANNEL_L,
$BUS_IDX)
    attach_level_meter(get_ui_id($groupfx_output_lvl), $GROUP_IDX, $SLOT_IDX,
$CHANNEL_L, -1)
end on
```

Various level meters.

See Also

ui_level_meter

Specific: `$CONTROL_PAR_BG_COLOR`, `$CONTROL_PAR_OFF_COLOR`, `$CONTROL_PAR_ON_COLOR`, `$CONTROL_PAR_OVERLOAD_COLOR`, `$CONTROL_PAR_PEAK_COLOR`, `$CONTROL_PAR_VERTICAL`, `$CONTROL_PAR_RANGE_MIN`, `$CONTROL_PAR_RANGE_MAX`

attach_zone()

`attach_zone(<variable>, <zone-id>, <flags>)`	
Connects the corresponding zone to the waveform so that it shows up on the `ui_waveform` widget.	
`<variable>`	The variable name of the `ui_waveform` widget.
`<zone-id>`	The ID number of the zone that you want to attach.

`<flags>`	You can control different settings of the widget via the following flags:
	`$UI_WAVEFORM_USE_SLICES`
	`$UI_WAVEFORM_USE_TABLE`
	`$UI_WAVEFORM_TABLE_IS_BIPOLAR`
	`$UI_WAVEFORM_USE_MIDI_DRAG`

Remarks

- Use the bitwise `.or.` operator to combine flags.
- The `$UI_WAVEFORM_USE_TABLE` and `$UI_WAVEFORM_USE_MIDI_DRAG` flags will only work if `$UI_WAVEFORM_USE_SLICES` is already set.

Examples

```
on init
    declare ui_waveform $Waveform (6, 6)
    attach_zone ($Waveform,find_zone(”Test”), $UI_WAVEFORM_USE_SLICES .or.
$UI_WAVEFORM_USE_TABLE)
end on
```

Attaches a zone named “Test” to the ui_waveform widget, also showing the zone’s slices and a table.

See Also

set_ui_wf_property()

get_ui_wf_property()

ui_waveform

Zone and Slice Functions: `find_zone()`

Specific: Waveform Flag Constants, Waveform Property Constants

fs_get_filename()

`fs_get_filename(<ui-id>, <return-parameter>)`	
Return the filename of the last selected file in a `ui_file_selector` widget.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<return-parameter>`	**0:** Returns the filename without extension. **1:** Returns the filename with extension. **2:** Returns the whole path.

Remarks

- This command is only available in the `on ui_control` callbacks of `ui_file_selectors`.

See Also

fs_navigate()

ui_file_selector

fs_navigate()

`fs_navigate(<ui-id>, <direction>)`	
Jump to the previous or next file in a ui_file_selector widget and trigger its callback.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<direction>`	**0:** The previous file (in relation to the currently selected one) is selected **1:** The next file (in relation to the currently selected one) is selected

Remarks

- This command is only available in the `on ui_control` callbacks of `ui_file_selectors`.
- It will always call the UI callback of the `ui_file_selector` it is pointed to.

See Also

fs_get_filename()

ui_file_selector

get_control_par()

`get_control_par(<ui-id>, <control-parameter>)`	
Retrieve various parameters of the specified UI widget.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<control-parameter>`	Parameter of the UI widget we want to retrieve, i.e. `$CONTROL_PAR_WIDTH`.

Remarks

- `get_control_par_str()` is an additional flavor of the command for use with strings (i.e. retrieving text from `ui_label` or automation name from `ui_slider`).

Examples

```
on init
    declare ui_value_edit $Test (0, 100, 1)
    message(get_control_par(get_ui_id($Test), $CONTROL_PAR_WIDTH))
end on
```

Retrieving the width of a value edit in pixels.

See Also

set_control_par()

General: `$CONTROL_PAR_KEY_SHIFT`, `$CONTROL_PAR_KEY_ALT`, `$CONTROL_PAR_KEY_CONTROL`

get_control_par_arr()

`get_control_par_arr(<ui-id>, <control-parameter>, <index>)`	
Retrieve various parameters of the specified UI widget	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<control-parameter>`	Parameter of the UI widget we want to retrieve, i.e. `$CONTROL_PAR_WIDTH`.
`<index>`	Array index of the UI widget we want to retrieve.

Remarks

- `get_control_par_arr()` comes in two additional flavors:
  - `get_control_par_str_arr()` (i.e. retrieving automation name of a particular `ui_xy` cursor)
  - `get_control_par_real_arr()` (i.e. retrieving values of `ui_xy` cursor X and Y axes)

Examples

```
on init
    declare ui_xy ?XY1[2]
    declare ui_xy ?XY2[2]
    declare ui_xy ?XY3[2]
    declare ui_xy ?XY4[2]
    declare ui_button $Random

    declare $i
    declare ~val
end on

on ui_control ($Random)
    $i := 0
    while ($i < 8)
        { randomize X axis value }
        ~val := int_to_real(random(0, 1000000)) / 1000000.0
        set_control_par_real_arr(get_ui_id(?XY1) + $i / 2, $CONTROL_PAR_VALUE,
~val, $i mod 2)

        { randomize Y axis value }
        ~val := int_to_real(random(0, 1000000)) / 1000000.0
        set_control_par_real_arr(get_ui_id(?XY1) + $i / 2, $CONTROL_PAR_VALUE,
~val, ($i mod 2) + 1)

        inc($i)
    end while

    $Random := 0
end on
```

Randomize the values of the first cursor for 4 different XY pads in one loop.

See Also

get_control_par()

set_control_par()

set_control_par_arr()

General: `$CONTROL_PAR_KEY_SHIFT`, `$CONTROL_PAR_KEY_ALT`, `$CONTROL_PAR_KEY_CONTROL`

get_font_id()

`get_font_id(<file-name>)`	
Returns a font ID generated for a custom font based on an image file. This font ID can be used on any control that has dynamic text elements.	
`<file-name>`	Name of the image, without extension. The image has to be in PNG format and reside in the "pictures" subfolder of the resource container.

Remarks

- This command is only available in the on init callback.
- Custom font images need to be formatted in a special way to be interpreted correctly as custom fonts. All characters need to be placed side-by-side, following the Windows-1252 character set, with a fully red (#FF0000) pixel at the top left of every character frame. Also, alpha layer of this image needs to be solid (contain only one color). We recommend using the SuperPNG addon for Adobe Photoshop (use the "Clean Transparent" option during export), or KSP Font Generator plugin for Figma.

Examples

```
on init
    declare ui_text_edit @textEdit
    set_control_par(get_ui_id(@textEdit), $CONTROL_PAR_FONT_TYPE,
get_font_id("Font1"))
end on
```

Using a custom font on a ui_text_edit control.

See Also

set_control_par()

General: $CONTROL_PAR_FONT_TYPE

get_menu_item_str()

`get_menu_item_str(<ui-id>, <index>)`	
Returns the string value of a particular `ui_menu` entry.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<index>`	The index (*not* the value!) of the menu item.

Remarks

- The `<index>` is defined by the order in which the menu items are added within the `on init` callback; it cannot be changed afterwards.

Examples

```
on init
    declare ui_button $button
    declare ui_menu $menu

    add_menu_item($menu, "First Entry", 0)
    add_menu_item($menu, "Second Entry", 5)
    add_menu_item($menu, "Third Entry", 10)
end on

on ui_control ($button)
    message(get_menu_item_str(get_ui_id($menu), 1))
end on
```

Displays the message "Second Entry" when clicking on the button, since we are reading the text of menu item index 1.

See Also

add_menu_item()

get_menu_item_value()

get_menu_item_visibility()

get_menu_item_str()

set_menu_item_value()

set_menu_item_visibility()

Specific: `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`

get_menu_item_value()

`get_menu_item_value(<ui-id>, <index>)`	
Returns the value of a particular ui_menu entry.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<index>`	The index (*not* the value!) of the menu item.

Remarks

- The `<index>` is defined by the order in which the menu items are added within the on init callback; it cannot be changed afterwards.

Examples

```
on init    declare ui_button $button    declare ui_menu $menu
add_menu_item($menu, "First Entry", 0)    add_menu_item($menu, "Second Entry",
5)    add_menu_item($menu, "Third Entry", 10)end onon ui_control ($button)
message(get_menu_item_value(get_ui_id($menu), 1))end on
```

Displays the number 5, since we're reading the value of menu item index 1.

See Also

add_menu_item()

get_menu_item_str()

get_menu_item_visibility()

set_menu_item_str()

set_menu_item_value()

set_menu_item_visibility()

Specific: `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`

get_menu_item_visibility()

`get_menu_item_visibility(<ui-id>, <index>)`	
Returns **1** if a particular ui_menu entry is visible, otherwise **0**.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<index>`	The index (*not* the value!) of the menu entry.

Remarks

- The `<index>` is defined by the order in which the menu items are added within the on init callback; it cannot be changed afterwards.

Examples

```
on init
    declare ui_button $visibility
    declare ui_button $value
    declare ui_menu $menu

    add_menu_item($menu, "First Entry", 0)
    add_menu_item($menu, "Second Entry", 5)
    add_menu_item($menu, "Third Entry", 10)
end on

on ui_control ($visibility)
    set_menu_item_visibility(get_ui_id($menu), $visibility))
end on

on ui_control ($value)
    message(get_menu_item_visibility(get_ui_id($menu), 1))
end on
```

Clicking on Visibility button shows or hides the second menu entry, while clicking on Value button shows the visibility state of that same menu entry.

See Also

add_menu_item()

get_menu_item_str()

get_menu_item_value()

set_menu_item_str()

set_menu_item_value()

set_menu_item_visibility()

Specific: `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`

get_ui_id()

`get_ui_id(<variable>)`

Retrieves the UI ID number of a UI widget.

Remarks

- UI IDs are assigned sequentially from the very first variable or constant declared in the script, which starts at **32768**.
- Even regular variables and constants (those that are not UI widgets) get a UI ID assigned, however this ID cannot be used with various `get_control_par()`/`set_control_par()` commands!

Examples

```
on init
    declare const $NUM_KNOBS := 4

    declare ui_knob $Knob_1 (0, 100, 1)
    declare ui_knob $Knob_2 (0, 100, 1)
    declare ui_knob $Knob_3 (0, 100, 1)
    declare ui_knob $Knob_4 (0, 100, 1)

    declare ui_value_edit $Set (0, 100, 1)

    declare $i
    declare %ID[$NUM_KNOBS]

    while ($i < $NUM_KNOBS)
        %ID[$i] := get_ui_id($Knob_1) + $i

        inc($i)
    end while
end on

on ui_control ($Set)
    $i := 0
    while ($i < $NUM_KNOBS)
        set_control_par(%ID[$i], $CONTROL_PAR_VALUE, $Set)
        inc($i)
    end while
end on
```

Store IDs in an array and use those IDs to set multiple knobs to the same value.

See Also

set_control_par()

get_control_par()

get_ui_wf_property()

get_ui_wf_property(<variable>, <property>, <index>)	
Returns the values of different properties pertaining to the `ui_waveform` widget.	
`<variable>`	Variable name of the `ui_waveform` widget.

`<property>`	The following properties are available:
	`$UI_WF_PROP_PLAY_CURSOR`
	`$UI_WF_PROP_FLAGS`
	`$UI_WF_PROP_TABLE_VAL`
	`$UI_WF_PROP_TABLE_IDX_HIGHLIGHT`
	`$UI_WF_PROP_MIDI_DRAG_START_NOTE`

`<index>`	The index of the slice.

Examples

```
on init
    declare $play_pos

    declare ui_waveform $Waveform (6, 6)

    attach_zone($Waveform, find_zone("Test"), 0)
end on

on note
    while ($NOTE_HELD = 1)
        $play_pos := get_event_par($EVENT_ID, $EVENT_PAR_PLAY_POS)

        set_ui_wf_property($Waveform, $UI_WF_PROP_PLAY_CURSOR, $play_pos)
        message(get_ui_wf_property($Waveform, $UI_WF_PROP_PLAY_CURSOR, 0))

        wait(10000)
    end while
end on
```

Displays the current play position value.

See Also

set_ui_wf_property()

ui_waveform

attach_zone()

Zone and Slice Functions: `find_zone()`

Specific: Waveform Flag Constants, Waveform Property Constants

hide_part()

`hide_part(<variable>, <hide-mask>)`	
Hide specific parts of various widgets.	
`<variable>`	The variable name of the widget.
`<hide-mask>`	Bitmask of visibility states for various parts of UI controls, consisting of the following constants: `$HIDE_PART_BG` (background of `ui_knob`, `ui_label`, `ui_value_edit` and `ui_table`) `$HIDE_PART_VALUE` (value of `ui_knob`, and `ui_table`) `$HIDE_PART_TITLE` (title of `ui_knob`,) `$HIDE_PART_MOD_LIGHT` (mod ring light of `ui_knob`,) `$HIDE_PART_CURSOR` (hide a particular `ui_xy` cursor)

Examples

```
on init
    declare ui_knob $Knob (0, 100, 1)

    hide_part($Knob, $HIDE_PART_BG .or. $HIDE_PART_MOD_LIGHT .or.
$HIDE_PART_TITLE .or. $HIDE_PART_VALUE)
end on
```
A naked knob.

```
on init
    declare ui_label $label_1 (1, 1)
    set_text($label_1, "Small Label")
    hide_part($label_1, $HIDE_PART_BG)
end on
```
Hide the background of a label. This is also possible with other UI elements.

See Also

Specific: `$CONTROL_PAR_HIDE`, `$HIDE_PART_NOTHING`, `$HIDE_WHOLE_CONTROL`

load_performance_view()

`load_performance_view(<filename>)`	
Loads a performance view file (.nckp) that was created in the Creator Tools GUI Designer.	
`<filename>`	The filename of the .nckp file without extension, entered as a string.

Remarks

- Only one performance view file can be loaded per script slot.
- This command is only available in the `on init` callback.
- This command cannot be used alongside `make_perfview`.
- The performance view file should be in the `performance_view` subfolder of the resource container.
- All contained controls are accessible as if they were declared and set up in KSP; variable names can be identified in Creator Tools.
- More information in the Creator Tools manual.

Examples

```
on init
    load_performance_view(“performanceView”)
end on

on ui_control ($testButton)
    if ($testButton = 0)
        set_control_par(get_ui_id($testSlider), $CONTROL_PAR_HIDE,
$HIDE_PART_WHOLE_CONTROL)
    else
        set_control_par(get_ui_id($testSlider), $CONTROL_PAR_HIDE,
$HIDE_PART_NOTHING)
    end if
end on
```

Loads a performance view file and then defines some basic behavior involving two of the contained controls.

make_perfview

`make_perfview`
Activates the performance view for the respective script slot.

Remarks

- This command can only be used in the `on init` callback.
- Cannot be used alongside the `load_performance_view()` command.

Examples

```
on init
    message("")

    make_perfview
    set_script_title("My Cool Instrument")
    set_ui_height(6)
end on
```

Many KSP scripts will start something like this.

See Also

set_skin_offset()

set_ui_height()

set_ui_height_px()

set_ui_width_px()

set_ui_color()

move_control()

`move_control(<variable>, <x-position>, <y-position>)`	
Position UI widgets in the standard Kontakt grid.	
`<variable>`	The variable name of the UI widget.
`<x-position>`	The horizontal position of the widget in grid units (**0 ... 6**).
`<y-position>`	The vertical position of the widget in grid units (**0 ... 16**).

Remarks

- `move_control()` can be used in all callbacks.
- Note that using `move_control()` outside of the `on init` callback is more CPU intensive, so handle with care.
- `move_control(<variable>, 0, 0)` will hide the UI widget.
- Pixel-based control parameters cannot be mixed with grid-based ones, so if you want to set $CONTROL_PAR_WIDTH for a ui_label that is positioned to grid coordinates (2, 1), this will not work - you would have to use $CONTROL_PAR_GRID_WIDTH instead.

Examples

```
on init
    set_ui_height(3)

    declare ui_label $label (1, 1)
    set_text($label, "Move the wheel!")
    move_control($label, 3, 6)
end on

on controller
    if ($CC_NUM = 1)
        move_control($label, 3,  6 - ((%CC[1] * 5) / 127))
    end if
end on
```

Move a UI element with the modwheel.

See Also

move_control_px()

General: `$CONTROL_PAR_HIDE`

move_control_px()

`move_control_px(<variable>, <x-position>, <y-position>)`	
Position UI widgets in pixels.	
`<variable>`	The variable name of the UI widget.
`<x-position>`	The horizontal position of the widget in pixels (**0 ... 1000**).
`<y-position>`	The vertical position of the widget in pixels (**0 ... 750**).

Remarks

- Pixel-based control parameters cannot be mixed with grid-based ones, so if you want to set $CONTROL_PAR_WIDTH for a ui_label that is positioned to grid coordinates (2, 1), this will not work - you would have to use $CONTROL_PAR_GRID_WIDTH instead.
- `move_control_px()` can be used in all callbacks.
- Note that using `move_control_px()` outside of the `on init` callback is more CPU intensive, so handle with care.
- In order to match Kontakt standard grid sizes to pixel position, the following formulae can be used:
    - X position: `((grid_value - 1) * 92) + 66`
    - Y position: `((grid_value - 1) * 21) + 2`
    - Width (`$CONTROL_PAR_WIDTH`): `(grid_value * 92) - 5`
    - Height (`$CONTROL_PAR_HEIGHT`): `(grid_value * 21) - 3`

Examples

```
on init
    declare ui_label $label (1, 1)
    set_text($label, "Move the wheel!")
    move_control_px($label, 66, 2)
end on

on controller
    if ($CC_NUM = 1)
        move_control_px($label, 66 + %CC[1], 2)
    end if
end on
```

Transform CC values into pixel position. This might be useful for reference.

See Also

move_control()

General: `$CONTROL_PAR_POS_X`, `$CONTROL_PAR_POS_Y`

set_control_help()

`set_control_help(<variable>, <text>)`	
Assigns a text string to be displayed when hovering over a UI widget. The text will appear in Kontakt's info pane.	
`<variable>`	The variable name of the UI widget.
`<text>`	The info text to be displayed.

Remarks

- The text string used can contain a maximum of **320** characters.

Examples

```
on init
    declare ui_knob $Knob (0, 100, 1)
    set_control_help($Knob, "I'm the only knob, folks!")
end on
```

set_control_help() in action.

See Also

set_script_title()

General:

`$CONTROL_PAR_HELP`

set_control_par()

`set_control_par(<ui-id>, <control-parameter>, <value>)`	
Change various parameters of the specified UI widget.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<control-parameter>`	Parameter of the UI control we wish to set, i.e. `$CONTROL_PAR_WIDTH`.
`<value>`	The value of the control parameter we wish to set.

Remarks

- `set_control_par_str()` is an additional flavor of the command for use with strings (i.e. setting the text of a `ui_label`, or automation name of a `ui_slider`).

Examples

```
on init
    declare ui_value_edit $test (0, 100, $VALUE_EDIT_MODE_NOTE_NAMES)
    set_text($test,"")
    set_control_par(get_ui_id($test), $CONTROL_PAR_WIDTH, 45)
    move_control_px($test, 100, 10)
end on
```

Changing the width of a value edit to 45 pixels. Note that you also have to specify its position in pixels once you use pixel-based control parameters.

```
on init
    declare ui_label $test (1, 1)
    set_control_par_str(get_ui_id($test), $CONTROL_PAR_TEXT, "This is Text")
    set_control_par(get_ui_id($test), $CONTROL_PAR_TEXT_ALIGNMENT, 1)
end on
```

Set and center text in labels.

See Also

get_control_par()

set_control_par_arr()

get_ui_id()

set_control_par_arr()

`set_control_par_arr(<ui-id>, <control-parameter>, <value>, <index>)`	
Change various parameters of an element within an array-based UI widget, e.g. `ui_xy` cursors.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<control-parameter>`	Parameter of the UI widgetwe wish to set, e.g. `$CONTROL_PAR_AUTOMATION_ID`.
`<value>`	The value of the control parameter we wish to set.
`<index>`	The array index of the UI control we wish to set.

Remarks

- `set_control_par_arr()` comes in two additional flavors:
  - `set_control_par_str_arr()` (i.e. setting automation names of individual `ui_xy` cursors)
  - `set_control_par_real_arr()` (i.e. values of individual `ui_xy` cursor X and Y axes)

Examples

```
on init
    make_perfview
    set_ui_height_px(350)

    declare ui_xy ?myXY[4]
    declare $xyID
    $xyID := get_ui_id(?myXY)

    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 0, 0)
    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 1, 1)
    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 2, 2)
    set_control_par_arr($xyID, $CONTROL_PAR_AUTOMATION_ID, 3, 3)

    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Cutoff", 0)
    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Resonance", 1)
    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Delay Pan", 2)
    set_control_par_str_arr($xyID, $CONTROL_PAR_AUTOMATION_NAME, "Delay Feedback",
3)
end on
```

Setting automation IDs and names of an XY pad with two cursors.

See Also

General: `$CONTROL_PAR_AUTOMATION_ID`, `$CONTROL_PAR_AUTOMATION_NAME`

Specific: `$CONTROL_PAR_CURSOR_PICTURE`, `$HIDE_PART_CURSOR`

set_knob_defval()

`set_knob_defval(<variable>, <value>)`

Assign a default value to a `ui_knob` to which it will be reset when pressing `[Ctrl]` (on Windows) or `[Cmd]` (on macOS) and clicking the knob.

Remarks

- In order to assign a default value to a `ui_slider`, use `set_control_par()` with `$CONTROL_PAR_DEFAULT_VALUE` control parameter.

Examples

```
on init
    declare ui_knob $Knob (-100, 100, 0)
    set_knob_defval($Knob, 0)
    $Knob := 0

    declare ui_slider $Slider (-100, 100)
    set_control_par(get_ui_id($Slider), $CONTROL_PAR_DEFAULT_VALUE, 0)
    $Slider := 0
end on
```

Assigning default values for a knob and a slider.

See Also

General: `$CONTROL_PAR_DEFAULT_VALUE`

set_knob_label()

```
set_knob_label(<variable>, <text>)
```

Assign a text string to a `ui_knob`.

Examples

```
on init
    declare !rate_names[18]
    !rate_names[ 0] := "1/128"
    !rate_names[ 1] := "1/64"
    !rate_names[ 2] := "1/32"
    !rate_names[ 3] := "1/16 T"
    !rate_names[ 4] := "1/32 D"
    !rate_names[ 5] := "1/16"
    !rate_names[ 6] := "1/8 T"
    !rate_names[ 7] := "1/16 D"
    !rate_names[ 8] := "1/8"
    !rate_names[ 9] := "1/4 T"
    !rate_names[10] := "1/8 D"
    !rate_names[11] := "1/4"
    !rate_names[12] := "1/2 T"
    !rate_names[13] := "1/4 D"
    !rate_names[14] := "1/2"
    !rate_names[15] := "1/1 T"
    !rate_names[16] := "1/2 D"
    !rate_names[17] := "1/1"

    declare ui_knob $Rate (0, 17, 1)
    set_knob_label($Rate, !rate_names[$Rate])
end on

on persistence_changed
    set_knob_label($Rate, !rate_names[$Rate])
end on

on ui_control ($Rate)
    set_knob_label($Rate, !rate_names[$Rate])
end on
```

Useful for displaying rhythmical values.

See Also

General: `$CONTROL_PAR_LABEL`

set_knob_unit()

`set_knob_unit(<variable>, <knob-unit-constant>)`

Assign a unit mark to a `ui_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`

Examples

```
on init
    declare ui_knob $Time (0, 1000, 10)
    set_knob_unit($Time, $KNOB_UNIT_MS)

    declare ui_knob $Octave (1, 6, 1)
    set_knob_unit($Octave, $KNOB_UNIT_OCT)

    declare ui_knob $Volume (-600, 600, 100)
    set_knob_unit($Volume, $KNOB_UNIT_DB)

    declare ui_knob $Scale (0, 100, 1)
    set_knob_unit($Scale, $KNOB_UNIT_PERCENT)

    declare ui_knob $Tune (4300, 4500, 10)
    set_knob_unit($Tune, $KNOB_UNIT_HZ)
end on
```

Various knob unit marks.

See Also

General: `$CONTROL_PAR_UNIT`

set_menu_item_str()

`set_menu_item_str(<ui-id>, <index>, <string>)`	
Sets the value of a `ui_menu` entry.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<index>`	The index of the menu item.
`<string>`	The text you wish to set for the selected menu item.

Remarks

- The `<index>` is defined by the order in which the menu items are added within the `on init` callback; it can't be changed afterwards.

Examples

```
on init
    declare ui_menu $menu
    declare ui_button $button
    add_menu_item ($menu, "First Entry", 0)
    add_menu_item ($menu, "Second Entry", 5)
    add_menu_item ($menu, "Third Entry", 10)
end on

on ui_control ($button)
    set_menu_item_str(get_ui_id($menu), 1, "Renamed")
end on
```

Renaming the second menu entry.

See Also

add_menu_item()

get_menu_item_str()

get_menu_item_value()

get_menu_item_visibility()

set_menu_item_value()

set_menu_item_visibility()

Specific: `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`

set_menu_item_value()

`set_menu_item_value(<ui-id>, <index>, <value>)`	
Sets the value of a `ui_menu` entry.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<index>`	The index of the menu item.
`<value>`	The value you want to give the menu item.

Remarks

- The `<index>` is defined by the order in which the menu items are added within the `on init` callback; it can't be changed afterwards.
- The `<value>` is set by the third parameter of the `add_menu_item()` command.

Examples

```
on init
    declare ui_menu $menu
    add_menu_item ($menu, "First Entry", 0)
    add_menu_item ($menu, "Second Entry", 5)
    add_menu_item ($menu, "Third Entry", 10)

    set_menu_item_value(get_ui_id($menu), 1, 20)
end on
```

Changing the value of the second menu entry to 20.

See Also

add_menu_item()

get_menu_item_str()

get_menu_item_value()

get_menu_item_visibility()

set_menu_item_str()

set_menu_item_visibility()

Specific: `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`

set_menu_item_visibility()

`set_menu_item_visibility(<ui-id>, <index>, <visibility>)`	
Sets the visibility of a `ui_menu` entry.	
`<ui-id>`	The ID number of the UI widget. You can retrieve it with get_ui_id().
`<index>`	The index of the menu item.
`<visibility>`	Set to either **0** (invisible) or **1** (visible).

Remarks

- The `<index>` is defined by the order in which the menu items are added within the `on init` callback; it can't be changed afterwards.
- Add as many menu entries as you would possibly need within the `on init` callback, then show or hide them dynamically by using `set_menu_item_visibility()`.
- If you set the currently selected menu item to invisible, the item will remain visible until it is no longer selected.

Examples

```
on init
    declare ui_menu $menu
    declare ui_button $button
    add_menu_item ($menu, "First Entry", 0)
    add_menu_item ($menu, "Second Entry", 5)
    add_menu_item ($menu, "Third Entry", 10)
end on

on ui_control ($button)
    set_menu_item_visibility(get_ui_id($menu), 1, $button)
end on
```

Hiding the second menu entry with a button.

See Also

add_menu_item()

get_menu_item_str()

get_menu_item_value()

get_menu_item_visibility()

set_menu_item_str()

set_menu_item_value()

Specific: `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`

set_table_steps_shown()

`set_table_steps_shown(<variable>, <num-of-steps>)`	
Changes the number of displayed columns in a `ui_table` widget.	
`<variable>`	The variable name of the `ui_table` widget.
`<num-of-steps>`	The number of displayed steps.

Examples

```
on init
    declare ui_table %table[32] (5, 4, 127)
    declare ui_value_edit $Steps (2, 32, 1)

    $Steps := 16
    set_table_steps_shown(%table, $Steps)
end on

on ui_control($Steps)
    set_table_steps_shown(%table, $Steps)
end on
```

Changing the number of shown steps.

See Also

ui_table

set_script_title()

`set_script_title(<text>)`

Set the script title.

Remarks

- This command overrides any manually set script titles.

Examples

```
on init
    message("")

    make_perfview
    set_script_title("My Cool Instrument")
    set_ui_height(6)
end on
```

Many performance view scripts start like this.

See Also

make_perfview

set_skin_offset()

```
set_skin_offset(<offset-in-px>)
```

Offsets the chosen background picture file by the specified number of pixels.

Remarks

- If a PNG file used for the background has been set and is larger than the maximum height of the performance view, you can use this command to offset the background graphic, thus creating separate backgrounds for each of the script slots while only using one picture file.

Examples

```
on init
    make_perfview
    set_ui_height(1)
end on

on controller
    if ($CC_NUM = 1)
        set_skin_offset(%CC[1])
    end if
end on
```

See Also

make_perfview

set_ui_height_px()

set_text()

`set_text(<variable>, <text>)`

When applied to a `ui_label`: replace the text currently visible in the specified label and add new text.

When applied to `ui_knob`, `ui_button`, `ui_switch` and `ui_value_edit`: set the display name of the widget.

Examples

```
on init
    declare ui_label $label_1 (1, 1)
    set_text($label_1, "Small Label")

    declare ui_label $label_2 (3, 6)
    set_text($label_2, "Big Label")
    add_text_line($label_2, "...with a second text line")
end on
```

Two labels with different sizes.

```
on init
    declare ui_label $label_1 (1, 1)
    set_text ($label_1, "Small Label")
    hide_part ($label_1, $HIDE_PART_BG)
end on
```

Hide the background of a label. This is also possible with other widgets.

See Also

add_text_line()

set_control_par(): `set_control_par_str()`

General: `$CONTROL_PAR_TEXT`

set_ui_color()

`set_ui_color(<hex-value>)`	
Set the main background color of the performance view.	
`<hex value>`	The hexadecimal color value in the following format: `0ff0000h {red}` The **0** at the start lets Kontakt know the value is a number. The **h** at the end indicates that it is a hexadecimal value. You can also use uppercase **H**.

Remarks

- This command can be used in all callbacks.

Examples

```
on init
    make_perfview
    set_ui_color(0000000H)
end on
```

Paint it black.

See Also

set_ui_height()

set_ui_height_px()

set_ui_height()

`set_ui_height(<height>)`	
Set the height of a script performance view in grid units.	
`<height>`	The height of script in grid units (**1 ... 8**).

Remarks

- This command can only be used in the `on init` callback.

Examples

```
on init
    message("")

    make_perfview
    set_script_title("My Cool Instrument")
    set_ui_height(6)
end on
```

Many performance view scripts start like this.

See Also

set_ui_height_px()

set_ui_height_px()

`set_ui_height_px(<height>)`	
Set the height of a script performance view in pixels.	
`<height>`	The height of script in pixels (**50 ... 750**).

Remarks

- This command can only be used in the `on init` callback.

Examples

```
on init
    make_perfview

    declare const $SIZE := 1644    { picture height }
    declare const $NUM_FRAMES := 4
    declare const $HEADER_SIZE := 68

    declare ui_value_edit $Slide (1, $NUM_SLIDES, 1)

    set_ui_height_px(($SIZE / $NUM_FRAMES) - $HEADER_SIZE)
    set_skin_offset(($Slide - 1) * ($SIZE / $NUM_FRAMES))
end on

on ui_control ($Slide)
    set_skin_offset(($Slide - 1) * ($SIZE / $NUM_FRAMES))
end on
```

See Also

set_ui_height()

set_ui_height_px()

set_ui_width_px()

`set_ui_width_px(<width>)`	
Set the width of a script performance view in pixels.	
`<width>`	The width of the script in pixels (**633 ... 1000**).

Remarks

- This command can only be used in the `on init` callback.

Examples

```
on init
    make_perfview
    set_ui_height_px(750)
    set_ui_width_px(1000)
end on
```

Making a performance view with the largest possible size.

See Also

set_ui_height_px()

set_ui_wf_property()

`set_ui_wf_property(<variable>, <property>, <index>, <value>)`	
Sets different properties for the `ui_waveform` widget.	
`<variable>`	Variable name of the `ui_waveform` widget.

`<property>`	The following properties are available:
	`$UI_WF_PROP_PLAY_CURSOR`
	`$UI_WF_PROP_FLAGS`
	`$UI_WF_PROP_TABLE_VAL`
	`$UI_WF_PROP_TABLE_IDX_HIGHLIGHT`
	`$UI_WF_PROP_MIDI_DRAG_START_NOTE`

`<index>`	The index of the slice to which the selected property applies. Only valid for `$UI_WF_PROP_TABLE_IDX_HIGHLIGHT` and `$UI_WF_PROP_TABLE_VAL`
`<value>`	The value of the selected property.

Examples

```
on init
    declare $play_pos
    declare ui_waveform $Waveform (6, 6)

    attach_zone($Waveform, find_zone("Test"), 0)
end on

on note
    while ($NOTE_HELD = 1)
        $play_pos := get_event_par($EVENT_ID, $EVENT_PAR_PLAY_POS)
        set_ui_wf_property($Waveform, $UI_WF_PROP_PLAY_CURSOR, 0, $play_pos)
        wait(10000)
    end while
end on
```

Attaches a zone named “Test” to the waveform display and shows a play cursor within the waveform as long as you play a note.

See Also

get_ui_wf_property()

ui_waveform

attach_zone()

Zone and Slice Functions: `find_zone()`

Specific: Waveform Flag Constants, Waveform Property Constants

15. Keyboard Commands

get_key_color()

`get_key_color(<note-number>)`

Returns the color constant of the specified note number.

Examples

```
on init
    message("")

    declare $count

    while ($count < 128)
        set_key_color($count, $KEY_COLOR_INACTIVE)

        inc($count)
    end while

    declare $random_key
    $random_key := random(60, 71)

    set_key_color($random_key, $KEY_COLOR_RED)
end on

on note
    if (get_key_color($EVENT_NOTE) = $KEY_COLOR_RED)
        message("Bravo!")

        set_key_color($random_key, $KEY_COLOR_INACTIVE)
        $random_key := random(60, 71)
        set_key_color($random_key, $KEY_COLOR_RED)
    else
        message("Try again!")
    end if
end on

on release
    message("")
end on
```

Catch me if you can.

See Also

set_key_color()

get_key_name()

`get_key_name(<note-number>)`
Returns the name of the specified note number.

Examples

```
on init
    declare $count

    while ($count < 128)
        set_key_name($count, "")

        inc($count)
    end while

    set_key_name(60, "Middle C")
end on

on note
    message(get_key_name($EVENT_NOTE))
end on
```

See Also

set_key_name()

get_key_triggerstate()

`get_key_triggerstate(<note-number>)`

Returns the pressed state of the specified note number, i.e. key, on the Kontakt keyboard. It can be either **1** (key pressed) or **0** (key released).

Remarks

- `get_key_triggerstate()` only works when `set_key_pressed_support()` is set to **1**.

Examples

```
on init
    set_key_pressed_support(1)
end on

on note
    set_key_pressed($EVENT_NOTE, 1)
    message(get_key_triggerstate($EVENT_NOTE))
end on

on release
    set_key_pressed($EVENT_NOTE, 0)
    message(get_key_triggerstate($EVENT_NOTE))
end on
```

See Also

set_key_pressed()

set_key_pressed_support()

get_key_type()

```
get_key_type(<note-number>)
```

Returns the key type constant of the specified note number.

See Also

set_key_type()

get_keyrange_min_note()

`get_keyrange_min_note(<note-number>)`

Returns the lowest note of the specified key range.

Remarks

- Since a key range cannot have overlapping notes, it is sufficient with all `get_keyrange_...` commands to specify the key range with one note number only.

Examples

```
on init
    declare $count
    while ($count < 128)
        remove_keyrange($count)

        inc($count)
    end while

    set_keyrange(36, 72, "Middle Range")
end on

on note
    message(get_keyrange_min_note($EVENT_NOTE))
end on
```

See Also

set_keyrange()

get_keyrange_max_note()

`get_keyrange_max_note(<note-number>)`

Returns the highest note of the specified key range.

Remarks

- Since a key range cannot have overlapping notes, it is sufficient with all `get_keyrange_...` commands to specify the key range with one note number only.

Examples

```
on init
    declare $count
    while ($count < 128)
        remove_keyrange($count)

        inc($count)
    end while

    set_keyrange(36, 72, "Middle Range")
end on

on note
    message(get_keyrange_max_note($EVENT_NOTE))
end on
```

See Also

set_keyrange()

get_keyrange_name()

`get_keyrange_name(<note-number>)`
Returns the name of the specified key range.

Remarks

- Since a key range cannot have overlapping notes, it is sufficient with all `get_keyrange_...` commands to specify the key range with one note number only.

Examples

```
on init
    declare $count
    while ($count < 128)
        remove_keyrange($count)

        inc($count)
    end while

    set_keyrange(36, 72, "Middle Range")
end on

on note
    message(get_keyrange_name($EVENT_NOTE))
end on
```

See Also

set_keyrange()

set_key_color()

`set_key_color(<note-number>, <key-color-constant>)`	
Sets the color of the specified key, i.e. MIDI note, on the Kontakt virtual keyboard.	
`<note-number>`	MIDI note number of the key (**0 ... 127**).

`<key-color-constant>`	One of available key color constant to specify the color used. The following constants are available:
	`$KEY_COLOR_RED`
	`$KEY_COLOR_ORANGE`
	`$KEY_COLOR_LIGHT_ORANGE`
	`$KEY_COLOR_WARM_YELLOW`
	`$KEY_COLOR_YELLOW`
	`$KEY_COLOR_LIME`
	`$KEY_COLOR_GREEN`
	`$KEY_COLOR_MINT`
	`$KEY_COLOR_CYAN`
	`$KEY_COLOR_TURQUOISE`
	`$KEY_COLOR_BLUE`
	`$KEY_COLOR_PLUM`
	`$KEY_COLOR_VIOLET`
	`$KEY_COLOR_PURPLE`
	`$KEY_COLOR_MAGENTA`
	`$KEY_COLOR_FUCHSIA`
	`$KEY_COLOR_DEFAULT` Sets the key to Kontakt's standard color for mapped notes
	`$KEY_COLOR_INACTIVE` Resets the key to standard black and white
	`$KEY_COLOR_NONE` Resets the key to its normal Kontakt color, e.g. red for internal keyswitches

Remarks

- The keyboard colors reside outside of KSP, i.e. changing the color of a key is similar to changing a Kontakt knob with `set_engine_par()`. It is therefore a good practice to set all keys to either `$KEY_COLOR_INACTIVE` or `$KEY_COLOR_NONE` in the `on init` callback.

Example

```
on init
    message("")

    declare ui_button $Color

    declare $count
    declare $note_count
    declare $color_count
    declare %white_keys[7] := (0, 2, 4, 5, 7, 9, 11)
    declare %colors[16] := ( ...
        $KEY_COLOR_RED, ...
        $KEY_COLOR_ORANGE, ...
        $KEY_COLOR_LIGHT_ORANGE, ...
        $KEY_COLOR_WARM_YELLOW, ...
        $KEY_COLOR_YELLOW, ...
        $KEY_COLOR_LIME, ...
        $KEY_COLOR_GREEN, ...
        $KEY_COLOR_MINT, ...
        $KEY_COLOR_CYAN, ...
        $KEY_COLOR_TURQUOISE, ...
        $KEY_COLOR_BLUE, ...
        $KEY_COLOR_PLUM, ...
        $KEY_COLOR_VIOLET, ...
        $KEY_COLOR_PURPLE, ...
        $KEY_COLOR_MAGENTA, ...
        $KEY_COLOR_FUCHSIA)

    $count := 0
    while ($count < 128)
        set_key_color($count, $KEY_COLOR_NONE)

        inc($count)
    end while
end on

on ui_control ($Color)
    if ($Color = 1)
        $count := 0
        while ($count < 128)
            set_key_color($count, $KEY_COLOR_INACTIVE)
            inc($count)
        end while

        $note_count := 0
        $color_count := 0
        while ($color_count < 16)
            if (search(%white_keys, (60 + $note_count) mod 12) # -1)
                set_key_color(60 + $note_count, %colors[$color_count])
                inc($color_count)
            end if

            inc($note_count)
        end while
    else
        $count := 0
        while ($count < 128)
            set_key_color($count, $KEY_COLOR_NONE)

            inc($count)
        end while
    end if
end on
```

Kontakt rainbow.

See Also

set_control_help()

get_key_color()

set_key_name()

set_keyrange()

set_key_name()

`set_key_name(<note-number>, <name>)`	
Assigns a text string to the specified note number.	
`<note-number>`	MIDI note number of the key (**0 ... 127**).
`<name>`	Text string to assign.

Remarks

- Key names are instrument parameters and reside outside of KSP, i.e. changing the key name is similar to changing a Kontakt knob with `set_engine_par()`. Make sure to always reset all key names in the `on init` callback.
- Key names and ranges are displayed in Kontakt's info pane when hovering the mouse over the key on the Kontakt keyboard.

Examples

```
on init
    declare $count

    while ($count < 128)
        set_key_name($count, "")

        inc($count)
    end while

    set_key_name(60, "Middle C")
end on
```

See Also

set_keyrange()

get_key_name()

set_key_pressed()

`set_key_pressed(<note-number>, <value>)`	
Sets the trigger state of the specified key on Kontakt's keyboard.	
`<note-number>`	MIDI note number of the key (**0 ... 127**).
`<value>`	**0:** Key is released **1:** Key is pressed

Remarks

- By using `set_key_pressed()` in combination with `set_key_pressed_support()` set to **1**, it is possible to show script generated notes on Kontakt's keyboard. The typical use case would be if an instrument features a built-in sequencer, arpeggiator or harmonizer, and the triggered notes should be shown on the keyboard.

Examples

```
on init
    set_key_pressed_support(1)
end on

on note
    set_key_pressed($EVENT_NOTE, 1)
end on

on release
    set_key_pressed($EVENT_NOTE, 0)
end on
```

Insert this after an arpeggiator or harmonizer script.

See Also

set_key_pressed_support()

get_key_triggerstate()

set_key_pressed_support()

`set_key_pressed_support(<mode>)`	
Sets the pressed state support mode for Kontakt's keyboard.	
`<mode>`	**0:** Kontakt handles all pressed states. `set_key_pressed()` commands are ignored (this is the default). **1:** Kontakt's keyboard is only affected by set_key_pressed() commands.

Remarks

- The pressed state mode resides outside of KSP, i.e. changing the mode is similar to changing a Kontakt knob with `set_engine_par()`. Make sure to always set the desired mode in the `on init` callback.

Examples

```
on init
    declare ui_button $Enable
    set_key_pressed_support(0)
end on

on ui_control ($Enable)
    set_key_pressed_support($Enable)
end on

on note
    play_note($EVENT_NOTE + 4, $EVENT_VELOCITY, 0, -1)
    play_note($EVENT_NOTE + 7, $EVENT_VELOCITY, 0, -1)

    set_key_pressed($EVENT_NOTE, 1)
    set_key_pressed($EVENT_NOTE + 4, 1)
    set_key_pressed($EVENT_NOTE + 7, 1)
end on

on release
    set_key_pressed($EVENT_NOTE, 0)
    set_key_pressed($EVENT_NOTE + 4, 0)
    set_key_pressed($EVENT_NOTE + 7, 0)
end on
```

Press the button and you will see what you hear.

See Also

set_key_pressed()

get_key_triggerstate()

set_key_type()

`set_key_type(<note-number>, <key-type-constant>)`	
Assigns a key type to the specified key.	
`<note-number>`	MIDI note number of the key (**0 ... 127**).
`<key-type-constant>`	The following key types are available: `$NI_KEY_TYPE_DEFAULT` Normally mapped keys that produce sound. `$NI_KEY_TYPE_CONTROL` Keyswitches or other keysthat do not produce sound. `$NI_KEY_TYPE_NONE` Resets the key to its standard Kontakt behaviour.

Remarks

- Setting the key type is useful for supported hosts like Komplete Kontrol, where keys with control functionality, e.g. keyswitches, should not be affected by any note processing.

Examples

```
on init
    declare $count

    $count := 0
    while ($count < 128)
        set_key_type($count, $NI_KEY_TYPE_NONE)

        inc($count)
    end while

    $count := 36
    while ($count <= 96)
        select ($count)
            case 36 to 47 { e.g. key switch }
                set_key_type($count, $NI_KEY_TYPE_CONTROL)
            case 48 to 96 { e.g. playable notes }
                set_key_type($count, $NI_KEY_TYPE_DEFAULT)
        end select

        inc($count)
    end while
end on
```

See Also

get_key_type()

set_keyrange()

`set_keyrange(<min-note>, <max-note>, <name>)`	
Assigns a text string to the specified range of keys.	
`<min-note>`	First key of the key range (**0 ... 127**).
`<max-note>`	Last key of the key range (**0 ... 127**).
`<name>`	Text string specifying the name of the key range.

Remarks

- Key ranges are instrument parameters and reside outside of KSP, i.e. changing the key range is similar to changing a Kontakt knob with `set_engine_par()`. Make sure to always remove all key ranges in the `on init` callback or whenever you want to change them later.
- There can be up to **16** key ranges per instrument.
- Key names and ranges are displayed in Kontakt's info pane when hovering the mouse over the key on the Kontakt keyboard. The range name is followed by the key name, separated by a dash.

Examples

```
on init
    declare $count

    while ($count < 128)
        remove_keyrange($count)

        inc($count)
    end while

    set_keyrange(36, 72, "Middle Range")
end on
```

See Also

remove_keyrange()

set_key_name()

remove_keyrange()

`remove_keyrange(<note-number>)`
Removes the range of keys to which the specified note number belongs.

Remarks

- Key ranges are instrument parameters and reside outside of KSP, i.e. changing the key range is similar to changing a Kontakt knob with`set_engine_par()`. Make sure to always remove all key ranges in the `on init` callback or whenever you want to change them later.

Examples

```
on init
    declare $count

    while ($count < 128)
        remove_keyrange($count)

        inc($count)
    end while

    set_keyrange(36, 72, "Middle Range")
end on
```

See Also

set_keyrange()

16. Engine Parameter Commands

get_mod_idx()

`get_mod_idx(<group-index>, <mod-name>)`	
Returns the slot index of an internal modulator or external modulation slot.	
`<group-index>`	The index of the group (see Index column in Monitor -> Groups pane in Kontakt).
`<mod-name>`	The name of the internal (LFO, envelope, step modulator...) or external (velocity, key position, mono aftertouch...) modulator.

Remarks

- Each modulator has a predefined name, based on its type and the parameter it targets.
- This name can be changed by enabling developer options in Kontakt's Options → Developer pane, then right-clicking on the modulator or modulator target strip.

Examples

```
on init
    declare $grp_idx := 0
    declare $env_idx

    $env_idx := get_mod_idx($grp_idx, "VOL_ENV")

    declare ui_knob $Attack (0, 1000000, 1)

    $Attack := get_engine_par($ENGINE_PAR_ATTACK, $grp_idx, $env_idx, -1)

    set_knob_unit($Attack, $KNOB_UNIT_MS)
    set_knob_label($Attack, get_engine_par_disp($ENGINE_PAR_ATTACK, $grp_idx,
$env_idx, -1))
end on

on ui_control ($Attack)
    set_engine_par($ENGINE_PAR_ATTACK, $Attack, $grp_idx, $env_idx, -1)
    set_knob_label($Attack, get_engine_par_disp($ENGINE_PAR_ATTACK, $grp_idx,
$env_idx, -1))
end on
```

Controlling the attack time of the volume envelope of the first group. Note: the envelope has been manually renamed to "VOL_ENV".

```
on init
    declare $count
    declare $mod_idx

    $mod_idx := get_mod_idx(0, "VEL_VOLUME")

    declare ui_slider $VelAmt (0, 1000000)

    if ($mod_idx # $NI_NOT_FOUND)
        $VelAmt := get_engine_par($ENGINE_PAR_MOD_TARGET_INTENSITY, 0, $mod_idx, -1)
    end if
    make_persistent($VelAmt)
end on

on ui_control ($VelAmt)
    $count := 0
    while($count < $NUM_GROUPS)
        $mod_idx := get_mod_idx($count, "VEL_VOLUME")

        if ($mod_idx # $NI_NOT_FOUND)
            set_engine_par($ENGINE_PAR_MOD_TARGET_INTENSITY, $VelAmt, $count,
$mod_idx, -1)
        end if

        inc($count)
    end while
end on
```

Creating a slider which controls the velocity to volume modulation intensity of all groups, if they exist.

See Also

get_target_idx()

set_engine_par()

get_target_idx()

`get_target_idx(<group-index>, <mod-index>, <target-name>)`	
Returns the modulation target slot index of an internal modulator	
`<group-index>`	The index of the group (see Index column in Monitor -> Groups pane in Kontakt).
`<mod-index>`	The slot index of an internal modulator (LFO, envelope, step modulator...). Can be retrieved with `get_mod_idx()`.
`<target-name>`	The name of the modulation target slot.

Remarks

- Each modulator has a predefined name, based on its type and the parameter it targets.
- This name can be changed by enabling developer options in Kontakt's Options → Developer pane, then right-clicking on the modulator or modulator target strip.

Examples

```
on init
    declare $mod_idx
    declare $target_idx

    $mod_idx := get_mod_idx(0, "FILTER_ENV")
    $target_idx := get_target_idx(0, $mod_idx, "FILTER_ENV > CUTOFF")

    declare ui_knob $FilterEnv (-1000, 1000, 10)

    set_knob_unit($FilterEnv, $KNOB_UNIT_PERCENT)

    make_persistent($FilterEnv)
end on

on ui_control ($FilterEnv)
    if ($mod_idx # $NI_NOT_FOUND and $target_idx # $NI_NOT_FOUND)
        set_engine_par($ENGINE_PAR_MOD_TARGET_MP_INTENSITY, 500000 + ($FilterEnv *
500), 0, $mod_idx, $target_idx)
    end if
end on
```

Controlling the envelope to filter cutoff modulation amount in the first group. Note: the filter envelope has been manually renamed to "FILTER_ENV", and the target to "FILTER_ENV > CUTOFF".

See Also

get_mod_idx()

set_engine_par()

get_engine_par()

`get_engine_par(<parameter>, <group>, <slot>, <generic>)`	
Returns the value of a specific engine parameter.	
`<parameter>`	Specifies the parameter by using one of the built-in engine parameter constants.
`<group>`	The index (zero-based) of the group in which the specified parameter resides. If the specified parameter resides on an Instrument level, enter **-1**. Buses and Main FX also reside on Instrument level, so you must set <group> to **-1** if you want to address a bus.
`<slot>`	The slot index (zero-based) of the specified parameter. It applies only to group/instrument effects, modulators and modulation intensities. For group/instrument effects, this parameter specifies the slot in which the effect resides (zero-based). For modulators and modulation intensities, this parameters specifies the index which you can retrieve by using `get_mod_idx()`. For all other applications, set this parameter to **-1**.
`<generic>`	This parameter applies to instrument effects and to internal modulators. For instrument effects, this parameter distinguishes between: `$NI_SEND_BUS`: Send Effect `$NI_INSERT_BUS`: Insert Effect `$NI_MAIN_BUS`: Main Effect For buses, this parameter specifies the actual bus: `$NI_BUS_OFFSET + [0 ... 15]` One of the 16 buses For internal modulators, this parameter specifies the modulation slider which you can retrieve by using `get_target_idx()`. For Flex Envelope, this parameter specifies which envelope stage to target when using `$ENGINE_PAR_FLEXENV_STAGE_TIME`, `$ENGINE_PAR_FLEXENV_STAGE_LEVEL` and `$ENGINE_PAR_FLEXENV_STAGE_SLOPE`. For Step Modulator, this parameter specifies which step value to target when using `$ENGINE_PAR_STEPSEQ_STEP_VALUE`. For all other applications, set this parameter to **-1**.

Examples

```
on init
    declare $i

    declare ui_label $label (2, 6)
    set_text($label,"Release Trigger Groups:")

    while ($i < $NUM_GROUPS)
        if (get_engine_par($ENGINE_PAR_RELEASE_TRIGGER, $i, -1, -1) = 1)
            add_text_line($label, group_name($i) & " (Index: " & $i & ")")
        end if

        inc($i)
    end while
end on
```

Output the name and index of release trigger group

```
on init
    declare ui_label $label (2, 6)
    declare ui_button $Refresh

    declare $i
    declare !effect_name[128]
    !effect_name[$EFFECT_TYPE_NONE] := "None"
    !effect_name[$EFFECT_TYPE_PHASER] := "Phaser"
    !effect_name[$EFFECT_TYPE_CHORUS] := "Chorus"
    !effect_name[$EFFECT_TYPE_FLANGER] := "Flanger"
    !effect_name[$EFFECT_TYPE_REVERB] := "Reverb"
    !effect_name[$EFFECT_TYPE_DELAY] := "Delay"
    !effect_name[$EFFECT_TYPE_IRC] := "Convolution"
    !effect_name[$EFFECT_TYPE_GAINER] := "Gainer"

    while ($i < 8)
        add_text_line($label, "Slot: " & $i + 1 & ": " & ...
                      !effect_name[get_engine_par($ENGINE_PAR_SEND_EFFECT_TYPE, -1,
$i, -1)])

        inc($i)
    end while
end on

on ui_control ($Refresh)
    set_text($label, "")

    while ($i < 8)
        add_text_line($label, "Slot: " & $i + 1 & ": " & ...
                      !effect_name[get_engine_par($ENGINE_PAR_SEND_EFFECT_TYPE, -1,
$i, -1)])

        inc($i)
    end while

    $Refresh := 0
end on
```

Output the effect types of all eight send effect slots.

See Also

`Module Types and Subtypes`

get_engine_par_disp()

`get_engine_par_disp(<parameter>, <group>, <slot>, <generic>)`	
Returns the displayed value of a specific engine parameter, as a string.	
`<parameter>`	Specifies the parameter by using one of the built-in engine parameter constants.
`<group>`	The index (zero-based) of the group in which the specified parameter resides. If the specified parameter resides on an Instrument level, enter **-1**. Buses and Main FX also reside on Instrument level, so you must set <group> to **-1** if you want to address a bus.
`<slot>`	The slot index (zero-based) of the specified parameter. It applies only to group/ instrument effects, modulators and modulation intensities. For group/instrument effects, this parameter specifies the slot in which the effect resides (zero-based). For modulators and modulation intensities, this parameters specifies the index which you can retrieve by using `get_mod_idx()`. For all other applications, set this parameter to **-1**.
`<generic>`	This parameter applies to instrument effects and to internal modulators. For instrument effects, this parameter distinguishes between: `$NI_SEND_BUS`: Send Effect `$NI_INSERT_BUS`: Insert Effect `$NI_MAIN_BUS`: Main Effect For buses, this parameter specifies the actual bus: `$NI_BUS_OFFSET + [0 ... 15]` One of the 16 buses For internal modulators, this parameter specifies the modulation slider which you can retrieve by using `get_target_idx()`. For Flex Envelope, this parameter specifies which envelope stage to target when using `$ENGINE_PAR_FLEXENV_STAGE_TIME`, `$ENGINE_PAR_FLEXENV_STAGE_LEVEL` and `$ENGINE_PAR_FLEXENV_STAGE_SLOPE`. For Step Modulator, this parameter specifies which step value to target when using `$ENGINE_PAR_STEPSEQ_STEP_VALUE`. For all other applications, set this parameter to **-1**.

Examples

```
on init
    declare $i

    declare ui_label $label (2, 6)
    set_text($label, "Group Volume Settings:")

    while ($i < $NUM_GROUPS)
        add_text_line($label, group_name($i) & ": " &
get_engine_par_disp($ENGINE_PAR_VOLUME, $i, -1, -1) & " dB")

        inc($i)
    end while
end on
```

Query the group volume settings in an instrument.

get_voice_limit()

`get_voice_limit(<voice-type>)`	
Returns the voice limit for the Time Machine Pro sampler mode of the Source module.	
`<voice-type>`	The voice type, can be one of the following: • `$NI_VL_TMPRO_STANDARD`: Standard quality mode • `$NI_VL_TMPRO_HQ`: High quality mode

Examples

```
on init
    declare ui_label $label (3, 2)

    add_text_line($label, "Standard Voice Limit: " &
get_voice_limit($NI_VL_TMPRO_STANDARD))
    add_text_line($label, "HQ Voice Limit: " & get_voice_limit($NI_VL_TMPRO_HQ))
end on
```

Displaying TM Pro voice limits.

See Also

set_voice_limit()

output_channel_name()

`output_channel_name(<output-number>)`	
Returns the channel name for the specified output.	
`<output-number>`	The number of the output channel (zero-based, i.e. the first output is **0**).

Examples

```
on init
    declare $i

    declare ui_menu $menu
    add_menu_item($menu, "Default", -1)

    $i := 0
    while ($i < $NUM_OUTPUT_CHANNELS)
        add_menu_item($menu, output_channel_name($i), $i)

        inc($i)
    end while

    $menu := get_engine_par($ENGINE_PAR_OUTPUT_CHANNEL, 0, -1, -1)
end on

on ui_control ($menu)
    set_engine_par($ENGINE_PAR_OUTPUT_CHANNEL,$menu, 0, -1, -1)
end on
```

Mirroring the output channel assignment menu of the first group.

See Also

General: `$NUM_OUTPUT_CHANNELS`, `$ENGINE_PAR_OUTPUT_CHANNEL`

set_engine_par()

`set_engine_par(<parameter>, <value>, <group>, <slot>, <generic>)`	
Control various Kontakt parameters and buttons.	
`<parameter>`	Specifies the parameter by using one of the built-in engine parameter constants.
`<value>`	The value to which the specified parameter is set. The range of values is always **0** to **1000000**, except for switches in which case it is **0** or **1**.
`<group>`	The index (zero-based) of the group in which the specified parameter resides. If the specified parameter resides on an Instrument level, enter **-1**. Buses and Main FX also reside on Instrument level, so you must set <group> to **-1** if you want to address a bus.
`<slot>`	The slot index (zero-based) of the specified parameter. It applies only to group/instrument effects, modulators and modulation intensities. For group/instrument effects, this parameter specifies the slot in which the effect resides (zero-based). For modulators and modulation intensities, this parameters specifies the index which you can retrieve by using `get_mod_idx()`. For all other applications, set this parameter to **-1**.
`<generic>`	This parameter applies to instrument effects and to internal modulators. For instrument effects, this parameter distinguishes between: `$NI_SEND_BUS`: Send Effect `$NI_INSERT_BUS`: Insert Effect `$NI_MAIN_BUS`: Main Effect For buses, this parameter specifies the actual bus: `$NI_BUS_OFFSET + [0 ... 15]` One of the 16 buses For internal modulators, this parameter specifies the modulation slider which you can retrieve by using `get_target_idx()`. For Flex Envelope, this parameter specifies which envelope stage to target when using `$ENGINE_PAR_FLEXENV_STAGE_TIME`, `$ENGINE_PAR_FLEXENV_STAGE_LEVEL` and `$ENGINE_PAR_FLEXENV_STAGE_SLOPE`. For Step Modulator, this parameter specifies which step value to target when using `$ENGINE_PAR_STEPSEQ_STEP_VALUE`. For all other applications, set this parameter to **-1**.

Examples

```
on init
    declare ui_knob $Volume (0, 1000000, 1000000)
end on

on ui_control ($Volume)
    set_engine_par($ENGINE_PAR_VOLUME, $Volume, -1, -1, -1)
end on
```

A knob controls the instrument volume.

```
on init
    declare ui_knob $Freq (0, 1000000, 1000000)
    declare ui_button $Bypass
end on

on ui_control ($Freq)
    set_engine_par($ENGINE_PAR_CUTOFF, $Freq, 0, 0, -1)
end on

on ui_control ($Bypass)
    set_engine_par($ENGINE_PAR_EFFECT_BYPASS, $Bypass, 0, 0, -1)
end on
```

Controlling the cutoff and bypass button of any filter module in the first slot of the first group.

```
on init
    declare ui_knob $Knob (-1000, 1000, 10)

    declare $mod_idx
    $mod_idx := get_mod_idx(0, "FILTER_ENV")

    declare $target_idx
    $target_idx := get_target_idx(0, $mod_idx, "ENV_AHDSR_CUTOFF")
end on

on ui_control ($Knob)
    set_engine_par($ENGINE_PAR_MOD_TARGET_MP_INTENSITY, $Knob * 1000, 0, $mod_idx,
$target_idx)
end on
```

Controlling the filter envelope amount of an envelope to filter cutoff modulation in the first group.
Note: the filter envelope has been manually renamed to "FILTER_ENV".

```
on init
    declare ui_knob $Vol (0, 1000000, 1000000)
end on

on ui_control ($Vol)
    set_engine_par($ENGINE_PAR_VOLUME, $Vol, -1, -1, $NI_BUS_OFFSET + 15)
end on
```

Controlling the amplifier volume of 16th bus.

set_voice_limit()

`set_voice_limit(<voice-type>, <value>)`	
Sets the voice limit for the Time Machine Pro mode of the Source module.	
`<voice-type>`	The voice type, can be one of the following: • `$NI_VL_TMPRO_STANDARD`: Standard quality mode • `$NI_VL_TMPRO_HQ`: High quality mode
`<value>`	The voice limit of the Time Machine Pro mode.

Remarks

- Changing voice limits is an asynchronous operation. This means that one cannot reliably access the newly allocated voices immediately after instantiation. To resolve this, the `set_voice_limit()` command returns an `$NI_ASYNC_ID` and triggers the `on async_complete` callback.
- Use this command to adjust the memory requirement of your instrument. Time Machine Pro uses its own memory allocation that is separate from memory used by loaded samples. This can be monitored in Kontakt's side pane, Monitor → Engine tab.

Examples

```
on init
    declare $change_voices_id

    declare ui_value_edit $Voices (1, 8, 1)

    make_persistent($Voices)
end on

on ui_control ($Voices)
    $change_voices_id := set_voice_limit($NI_VL_TMPRO_STANDARD, $Voices)
end on

on async_complete
    if ($NI_ASYNC_ID = $change_voices_id)
        message("New TMPro Std Voice Limit: " &
get_voice_limit($NI_VL_TMPRO_STANDARD))
    end if
end on
```

Changing TM Pro voice limits.

See Also

get_voice_limit()

17. Zone Commands

User Zone Information

User zones are a special kind of a zone that allow for zone creation and manipulation “on the fly”, and can be used to allow user interaction with the sampled content within an instrument (for example, in conjunction with sample drag-and-drop). These zones must be declared in the `on init` callback.

When a user zone is created, all zone parameters will be set to **0** by default (root key, high velocity, high note, low note etc…). Therefore, the zone will not show in the Mapping Editor’s normal view, however it will be listed and present in the Mapping Editor's List View and in Kontakt's Monitor → Zones pane.

Zone and Slice Functions

`find_zone(<zone-name>)`

Returns the zone ID for the specified zone name. Only available in the `on init` callback.

`get_sample_length(<zone-id>)`

Returns the length of the specified zone's sample in microseconds.

`num_slices_zone(<zone-id>)`

Returns the number of slices in the specified zone.

`zone_slice_length(<zone-id>, <slice-index>)`

Returns the length of the specified slice in microseconds, with respect to the current tempo.

`zone_slice_start(<zone-id>,<slice-index>)`

Returns the absolute start point of the specified slice in microseconds, independent of the current tempo.

`zone_slice_idx_loop_start(<zone-id>, <loop-index>)`

Returns the index number of the slice at the loop start.

`zone_slice_idx_loop_end(<zone-id>, <loop-index>)`

Returns the index number of the slice at the loop end.

`zone_slice_loop_count(<zone-id>, <loop-index>)`

Returns the loop count of the specified loop.

`dont_use_machine_mode(<ID-number>)`

Play the specified event in Sampler mode (only makes sense when the groups allowed for playback are in one of Machine modes).

get_loop_par()

`get_loop_par(<zone-id>, <loop-index>, <parameter>)`	
Returns the value of a particular loop parameter of a zone.	
`<zone-id>`	The ID of the zone.
`<loop-index>`	The index of the loop (**0 ... 7**).

`<parameter>`	The following parameters are available:
	`$LOOP_PAR_MODE`
	`$LOOP_PAR_START`
	`$LOOP_PAR_LENGTH`
	`$LOOP_PAR_XFADE`
	`$LOOP_PAR_COUNT`
	`$LOOP_PAR_TUNING`

Remarks

- `get_loop_par()` works on both normal and user zones.

Examples

```
message(get_loop_par($myZoneId, 0, $LOOP_PAR_MODE))
```

See Also

set_loop_par()

get_num_zones()

`get_num_zones()`
Returns the total number of all zones that are present in the instrument (normal and user zones).

Examples

```
on init
    message(get_num_zones())
end on
```

Quite self-explanatory.

get_sample()

get_sample(<zone-id>, <return-parameter>)	
Returns paths, file names and extensions of samples.	
<zone-id>	The ID of the zone.
<return-parameter>	The following parameters are available: $NI_FILE_NAME $NI_FILE_FULL_PATH $NI_FILE_FULL_PATH_OS $NI_FILE_EXTENSION

Remarks

- get_sample() works on both normal and user zones.

Examples

```
message(get_sample(%NI_USER_ZONE_IDS[0], $NI_FILE_NAME))
```

See Also

set_sample()

Sample Parameters

get_sel_zones_idx()

`get_sel_zones_idx(<array-name>)`	
Fills the specified array with indices of all selected zones in Kontakt's Mapping Editor.	
`<array-name>`	Array to be filled with indices of selected zones.

Remarks

- The command overwrites all existing values as long as there are selected zones. If there are more selected zones than array indices, the array will be filled until it is full, ignoring the remaining selected zones.
- If there are less selected zones than array indices, the array will be filled from the beginning with all selected zone indices, followed by one array index with its value set to **-1**.
- This command allows relative adjustment of various zone parameters like zone volume, pan, tune, etc.

Examples

```
on init
    message("")

    set_snapshot_type(3) { Must be 2 or 3 or else set_zone_par works only on user
zones }

    declare $a
    declare $i
    declare %sel_zones[1000]
    declare %zone_par[4] := ($ZONE_PAR_VOLUME, $ZONE_PAR_VOLUME, $ZONE_PAR_TUNE,
$ZONE_PAR_TUNE)
    declare %offset[4] := (-1, 1, -1, 1)
end on

on note
    { pressing the lowest A on an 88-key MIDI controler will gather all selected
zones }
    if ($EVENT_NOTE = 21)
        get_sel_zones_idx(%sel_zones)
    end if

    { pressing the following 4 keys will adjust volume down and up 0.01 dB, then
tuning down and up 1 cent }
    if (in_range($EVENT_NOTE, 22, 25))
        ignore_event($EVENT_ID)

        $a := $EVENT_NOTE - 22
        $i := 0
        while ($i < num_elements(%sel_zones))
            if (%sel_zones[$i] > -1)
                set_zone_par(%sel_zones[$i], ...
                             %zone_par[$a], ...
                             get_zone_par(%sel_zones[$i], %zone_par[$a]) +
%offset[$a])
            end if

            inc($i)
        end while

        exit
    end if
end on
```

Using the lowest keys of an 88 key MIDI controller to finely adjust zone volume and tuning in 0.01 steps

get_zone_id()

`get_zone_id(<zone-index>)`	
Returns the ID of the zone with the specified zone index.	
`<zone-index>`	Index of the zone (see Index column in Monitor → Zones pane in Kontakt) .

Examples

```
on init
    set_snapshot_type(3) { Must be 2 or 3 or else set_zone_par works only on user
zones }

    declare ui_slider $slider (0, 127)
    declare ui_label $label (1, 1)

    $slider := get_zone_par(get_zone_id(0), $ZONE_PAR_ROOT_KEY)

    message("This instrument contains " & get_num_zones() & " zones.")
end on

on ui_control ($slider)
    set_zone_par(get_zone_id(0), $ZONE_PAR_ROOT_KEY, $slider)
    set_text($label, get_zone_par(0, $ZONE_PAR_ROOT_KEY))
end on
```

Adjusting the root key of the first zone in the instrument.

get_zone_par()

`get_zone_par(<zone-id>, <parameter>)`	
Returns the value of a particular zone parameter.	
`<zone-id>`	The ID of the zone.

`<parameter>`	The following parameters are available:
	`$ZONE_PAR_HIGH_KEY`
	`$ZONE_PAR_LOW_KEY`
	`$ZONE_PAR_HIGH_VELO`
	`$ZONE_PAR_LOW_VELO`
	`$ZONE_PAR_ROOT_KEY`
	`$ZONE_PAR_FADE_LOW_KEY`
	`$ZONE_PAR_FADE_HIGH_KEY`
	`$ZONE_PAR_FADE_LOW_VELO`
	`$ZONE_PAR_FADE_HIGH_VELO`
	`$ZONE_PAR_VOLUME`
	`$ZONE_PAR_PAN`
	`$ZONE_PAR_TUNE`
	`$ZONE_PAR_GROUP`
	`$ZONE_PAR_SAMPLE_START`
	`$ZONE_PAR_SAMPLE_END`
	`$ZONE_PAR_SAMPLE_MOD_RANGE`
	`$ZONE_PAR_SAMPLE_RATE`
	`$ZONE_PAR_SELECTED`
	`$ZONE_PAR_BPM`

Remarks

- `get_zone_par()` works on both normal and user zones.
- `$ZONE_PAR_BPM` returns the BPM value multiplied by **1000**, so 120 BPM would be **120000**.

Examples

```
get_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_PAN)
```

See Also

set_zone_par()

get_zone_status()

`get_zone_status(<zone-ID>)`	
Queries the status of the zone ID in question. Zone status has four possible states: • `$NI_ZONE_STATUS_EMPTY` Zone is a user zone and has no sample loaded • `$NI_ZONE_STATUS_LOADED` Zone is a user zone and has a sample loaded • `$NI_ZONE_STATUS_PURGED` Zone is purged from memory (valid for both regular and user zones) • `$NI_ZONE_STATUS_IGNORED` Zone is ignored by the user response in the Content Missing dialog (valid for both regular and user zones)	
`<zone-ID>`	The ID of the zone.

Remarks

- This command requires a valid zone ID that exists in the instrument. For example, if when using `get_event_par()` a zone ID is not found (which would happen when attempting playback of an empty user zone or a purged zone), `get_zone_status()` will throw a script warning.
- `get_zone_status()` works on both normal and user zones.

Examples

```
on init
    declare ui_value_edit $ZoneID (0, 1000, 1)
end on

on ui_control ($ZoneID)
    select (get_zone_status($ZoneID))
        case $NI_ZONE_STATUS_EMPTY
            message("Zone ID " & $zoneID & " is empty!")
        case $NI_ZONE_STATUS_LOADED
            message("Zone ID " & $zoneID & " is loaded!")
        case $NI_ZONE_STATUS_PURGED
            message("Zone ID " & $zoneID & " is purged!")
        case $NI_ZONE_STATUS_IGNORED
            message("Zone ID " & $zoneID & " is ignored!")
    end select
end on
```

Query the status of the first 1001 zone IDs.

set_loop_par()

`set_loop_par(<zone-id>, <loop-index>, <parameter>, <value>)`	
Sets the loop parameters of a user zone.	
`<zone-id>`	The ID of the zone.
`<loop-index>`	The index of the loop (**0 ... 7**).

`<parameter>`	The following parameters are available:
	`$LOOP_PAR_MODE`
	`$LOOP_PAR_START`
	`$LOOP_PAR_LENGTH`
	`$LOOP_PAR_XFADE`
	`$LOOP_PAR_COUNT`
	`$LOOP_PAR_TUNING`

`<value>`	The value of the loop parameter.

Remarks

- `set_loop_par()` only works on user zones.
- When executed in the `on init` callback, this function runs synchronously and returns **-1**.
- When executed outside of the `on init` callback, this function returns an async ID and triggers the `on async_complete` callback.
- Since Kontakt 7.2, the performance of this command has been improved. Previously, every execution of this command required suspending then resuming the audio engine. Moving forward, execution of this command has been delayed until the first `wait()` command, or until the end of the callback, whichever comes first. So, if it is required to set a lot of zone parameters in batch, it is recommended to first collect all the async IDs into an array, and only then run any `wait()` commands, as shown in the description of `wait_async()` command.

Examples

```
wait_async(set_loop_par(%NI_USER_ZONE_IDS[0], 0, $LOOP_PAR_MODE, $SampleLoopOnA))
```

See Also

get_loop_par()

wait_async()

set_num_user_zones()

`set_num_user_zones(<value>)`	
Creates empty user zones.	
`<value>`	Defines the number of user zones to be created. `%NI_USER_ZONE_IDS` is the array of size `<value>` with all the user zone IDs.

Remarks

- A maximum of **1024** user zones per instrument can be created.
- User zones are shown with a different color in Kontakt's Mapping Editor.
- User zones cannot be modified from Mapping Editor or Wave Editor.
- In order to manipulate the user zones, the IDs stored in the `%NI_USER_ZONE_IDS` array should be used instead of the hard-coded zone IDs.

Examples

```
on init
    set_num_user_zones(2)

    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_GROUP, 0)
    set_zone_par(%NI_USER_ZONE_IDS[1], $ZONE_PAR_GROUP, 1)
end on
```

Create two empty zones and place each to its own group.

set_sample()

`set_sample(<zone-id>, <path>)`	
Sets the user sample in a zone.	
`<zone-id>`	The ID of the zone.
`<path>`	The file path to the sample to be loaded.

Remarks

- `set_sample()` only works on user zones.
- When executed in the `on init` callback, this function runs synchronously and returns **-1**.
- When executed outside of the `on init` callback, this function returns an async ID and triggers the `on async_complete` callback.
- Since Kontakt 7.2, the performance of this command has been improved. Previously, every execution of this command required suspending then resuming the audio engine. Moving forward, execution of this command has been delayed until the first `wait()` command, or until the end of the callback, whichever comes first. So, if it is required to set a lot of zone parameters in batch, it is recommended to first collect all the async IDs into an array, and only then run any `wait()` commands, as shown in the description of `wait_async()` command.

Examples

```
on ui_control ($myMouseArea)
    if ($NI_MOUSE_EVENT_TYPE = $NI_MOUSE_EVENT_TYPE_DROP)
        if (num_elements(!NI_DND_ITEMS_AUDIO) = 1)
            wait_async(set_sample(%NI_USER_ZONE_IDS[0], !NI_DND_ITEMS_AUDIO[0]))
        end if
    end if
end on
```

See Also

get_sample()

wait_async()

set_zone_par()

`set_zone_par(<zone-id>, <parameter>, <value>)`	
Sets the user zone parameters.	
`<zone-id>`	The ID of the zone.

`<parameter>`	The following flags are available:
	`$ZONE_PAR_HIGH_KEY`
	`$ZONE_PAR_LOW_KEY`
	`$ZONE_PAR_HIGH_VELO`
	`$ZONE_PAR_LOW_VELO`
	`$ZONE_PAR_ROOT_KEY`
	`$ZONE_PAR_FADE_LOW_KEY`
	`$ZONE_PAR_FADE_HIGH_KEY`
	`$ZONE_PAR_FADE_LOW_VELO`
	`$ZONE_PAR_FADE_HIGH_VELO`
	`$ZONE_PAR_VOLUME`
	`$ZONE_PAR_PAN`
	`$ZONE_PAR_TUNE`
	`$ZONE_PAR_GROUP`
	`$ZONE_PAR_SAMPLE_START`
	`$ZONE_PAR_SAMPLE_END`
	`$ZONE_PAR_SAMPLE_MOD_RANGE`
	`$ZONE_PAR_BPM`

`<value>`	The value of the zone parameter

Remarks

- `set_zone_par()` only works on user zones when using snapshot modes **0** and **1**. In case of using snapshot modes **2** and **3**, `set_zone_par()` will work on both normal and user zones, from any callback (please excercise caution with "fast" callback types, like `on controller` or `on listener`)!
- When executed in the `on init` callback, this function runs synchronously and returns **-1**.
- When executed outside of the `on init` callback, this function returns an async ID and triggers the `on async_complete` callback.
- `$ZONE_PAR_BPM` requires an input value multiplied by **1000**, so 120 BPM would be **120000**. Valid BPM input range is **0.1** to **400** BPM. Setting the BPM is not supported for REX files!
- Since Kontakt 7.2, the performance of this command has been improved. Previously, every execution of this command required suspending then resuming the audio engine. Moving forward, execution of this command has been delayed until the first `wait()` command, or until the end of the callback, whichever comes first. So, if it is required to set a lot of zone parameters in batch, it is recommended to first collect all the async IDs into an array, and only then run any `wait()` commands, as shown in the description of `wait_async()` command.

Examples

```
set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_GROUP, 0)
```

See Also

get_zone_par()

wait_async()

18. Load/Save Commands

General Information

File Formats

It is possible to load and save the following file formats:

- Kontakt arrays (.nka files)
- MIDI files (.mid) to be used with the file commands in KSP
- Samples (.wav, .aif, .aiff, .ncw) to be used with Kontakt's convolution effect or user zones (loading only)

Asynchronous Handling

Loading and saving files cannot be executed in real time. This is why all load/save commands return a unique value upon completion of their action - the async ID. You can use this value in combination with `$NI_ASYNC_ID` and `$NI_ASYNC_EXIT_STATUS` within the `on async_complete` callback to check whether the the command has completed its action, and whether or not the loading or saving was successful.

Path Handling

All file paths in KSP use a slash character (/) as a folder separator. Backslash characters are not supported. The full path also has to start with a slash character **/**.

Examples

Factory folder on macOS:

```
/Library/Application Support/Native Instruments/Kontakt 7/
```

Factory folder on Windows:

```
C:\Program Files\Common Files\Native Instruments\Kontakt 7\
```

When loading or saving files with an absolute path, as opposed to loading from the resource container, always use path variables in combination with `get_folder()`.

See Also

on async_complete

General: `$NI_ASYNC_ID`, `$NI_ASYNC_EXIT_STATUS`

get_folder()

`get_folder(<path-variable>)`	
Returns the path specified with the built-in path variable.	

`<path-variable>`	The following path variables are available:
	`$GET_FOLDER_LIBRARY_DIR`
	If used with an NKI which belongs to a Kontakt Player encoded library: library folder.
	If used with an unencoded NKI: the User Content folder, which is found as follows:
	macOS: `/Users/<UserName>/Documents/Native Instruments/User Content/`
	Windows: `C:\Users\<UserName>\Documents\Native Instruments\User Content\`
	`$GET_FOLDER_FACTORY_DIR`
	The factory data folder of Kontakt, mainly used for loading factory IR samples.
	Note: this is not the Kontakt Factory Library folder!
	`$GET_FOLDER_PATCH_DIR`
	The folder in which the patch was saved.
	If the patch was not saved before, an empty string is returned.

Example

```
on init
    message(get_folder($GET_FOLDER_FACTORY_DIR))
end on
```

Displaying the path of Kontakt's factory data folder.

See Also

load_ir_sample()

General: `$GET_FOLDER_LIBRARY_DIR`, `$GET_FOLDER_FACTORY_DIR`, `$GET_FOLDER_PATCH_DIR`

load_array()

`load_array(<array-variable>, <mode>)`	
Loads an array from an external .nka file.	
`<array-variable>`	The name of the array variable. This name must be present as the first line of the .nka file.
`<mode>`	0: A dialog window pops up, allowing you to select an .nka file to load. This mode can only be used in on persistence_changed, on ui_control and on pgs_changed callbacks (asynchronously). 1: The array is directly loaded from the `Data` folder. For user instruments, the `Data` folder is located beside the resource container. For Kontakt Player encoded library instruments, the `Data` folder is located here: macOS: `/Users/<UserName>/Library/Application Support/<LibraryName>/` Windows: `C:\Users\<UserName>\AppData\Local\<LibraryName>\` This mode can be used synchronously in on init, and asynchronously in on persistence_changed, on ui_control and on pgs_changed callbacks. 2: The array is directly loaded from the `data` folder inside the resource container. This mode can be used synchronously in on init, and asynchronously in on persistence_changed, on ui_control and on pgs_changed callbacks.

Remarks

- It is also possible to load real and string arrays from .nka files.
- It is not possible to load an array named `%xyz` from an .nka file into array `%abc`. The variable names have to match precisely.
- The array data is not directly available after the `load_array()` command has been executed, since the command works asynchronously. The only situation in which the values are instantly available is when using mode **1** or mode **2** inside on init callback.
- When using mode **0**, the callback continues even if the loading dialog is still open.
- When loading an array within the on init callback, please note that the array will implicitly be made persistent (as if make_persistent() command was used) , which results in loaded data being overwritten at the end of the callback. Use read_persistent_var() before loading the array to avoid this problem.
- .nka files loaded from the resource container should always have a newline character at the end of the file. If this last newline is missing, Kontakt cannot know the file has ended and will continue to try and load other data from the resource container. Files generated by the save_array() command have this automatically, but if you are creating NKA files via different means, this is something to be aware of.

Example

```
on init
    declare $count
    declare $load_arr_id := -1
    declare $save_arr_id := -1
    declare %preset[8]

    declare ui_button $Load
    declare ui_button $Save
    declare ui_table %table[8] (2,2,100)

    make_persistent(%table)
end on

on ui_control (%table)
    $count := 0
    while ($count < 8)
        %preset[$count] := %table[$count]

        inc($count)
    end while
end on

on ui_control ($Load)
    $load_arr_id := load_array(%preset, 0)
end on

on ui_control ($Save)
    $save_arr_id := save_array(%preset, 0)
end on

on async_complete
    if ($NI_ASYNC_ID = $load_arr_id)
        $load_arr_id := -1
        $Load := 0

        if ($NI_ASYNC_EXIT_STATUS = 1)
            $count := 0
            while($count < 8)
                %table[$count] := %preset[$count]

                inc($count)
            end while
        end if
    end if

    if ($NI_ASYNC_ID = $save_arr_id)
        $save_arr_id := -1
        $Save := 0
    end if
end on
```

Exporting and loading the contents of a ui_table widget.

See Also

on async_complete

save_array()

General: `$NI_ASYNC_ID`, `$NI_ASYNC_EXIT_STATUS`

load_array_str()

`load_array_str(<array-variable>, <path>)`	
Loads an array from an external .nka file, using an absolute path to the file.	
`<array-variable>`	The name of the array variable, this must be present as the first line of the .nka file.
`<path>`	The absolute path of the .nka file.

Remarks

- The behaviour is similar to load_array() with mode set to **0**, but instead of manually choosing an .nka file, you can specify it with an absolute path.
- This command can be used synchronously in on init, and asynchronously in on persistence_changed, on ui_control and on pgs_changed callbacks.
- When loading an array within the on init callback, please note that the array will implicitly be made persistent (as if make_persistent() command was used) , which results in loaded data being overwritten at the end of the callback. Use read_persistent_var() before loading the array to avoid this problem.

Example

```
on init
    message("")
    set_ui_height(2)

    declare $count
    declare $load_arr_id := -1
    declare %preset[8]
    declare @file_path
    declare @basepath_browser
    { set browser path here, for example:
    @basepath_browser := "/Users/<username>/Desktop/Arrays" }

    declare ui_file_selector $file_browser
    declare ui_table %table[8] (2, 2, 100)

    declare $browser_id
    $browser_id := get_ui_id($file_browser)

    set_control_par_str($browser_id, $CONTROL_PAR_BASEPATH, @basepath_browser)
    set_control_par($browser_id, $CONTROL_PAR_WIDTH, 112)
    set_control_par($browser_id, $CONTROL_PAR_HEIGHT, 68)
    set_control_par($browser_id, $CONTROL_PAR_COLUMN_WIDTH, 110)
    set_control_par($browser_id, $CONTROL_PAR_FILE_TYPE, $NI_FILE_TYPE_ARRAY)

    make_persistent(@file_path)
    make_persistent(%table)

    move_control_px($file_browser, 66, 2)
    move_control(%table, 3, 1)
end on

on async_complete
    if ($NI_ASYNC_ID = $load_arr_id)
        $load_arr_id := -1

        if ($NI_ASYNC_EXIT_STATUS = 0)
```

```
            message("Array not found!")
        else
            message("")

            $count := 0
            while ($count < num_elements(%preset))
                %table[$count] := %preset[$count]

                inc($count)
            end while
        end if
    end if
end on

on ui_control ($file_browser)
    @file_path := fs_get_filename($browser_id, 2)
    $load_arr_id := load_array_str(%preset, @file_path)
end on
```

Loading different table presets with a browser. Make sure to first set the browser path of the file selector to point to a folder with compatible .nka files.

load_ir_sample()

`load_ir_sample(<path-or-filename>, <slot>, <generic>)`	
Loads an impulse response sample into Kontakt's convolution effect.	
`<path-or-filename>`	The absolute file path of the IR sample. If no path is specified, the command will look for the specified sample within the `ir_samples` folder of the resource container. If no resource container is available, the folder `ir_samples` within the Kontakt user folder will be checked. The Kontakt user folder is located here: macOS: `/Users/<username>/Documents/Native Instruments/Kontakt 7/` Windows: `C:\Users\<username>\Documents\Native Instruments\Kontakt 7\`
`<slot>`	The slot index of the convolution effect (zero-based).
`<generic>`	Specifies whether the convolution effect is used as an: `$NI_SEND_BUS`: Send effect `$NI_INSERT_BUS`: Insert effect `$NI_MAIN_BUS`: Main effect For buses, this parameter specifies the actual bus: `$NI_BUS_OFFSET + [0 ... 15]`: One of the 16 buses

Remarks

- Please note that any subfolders inside the `ir_samples` folder of the resource container will not be scanned, and it is not recommended to add them manually via text strings. Doing so could lead to problems, because subfolders will be ignored during the creation of a resource container monolith.

Example

```
on init
    declare $load_ir_id := -1
    declare ui_button $Load
end on

on ui_control ($Load)
    $load_ir_id := load_ir_sample("Small Ambience.wav", 0, $NI_SEND_BUS)
    $Load := 0
end on

on async_complete
    if ($NI_ASYNC_ID = $load_ir_id)
        $load_ir_id := -1

        if ($NI_ASYNC_EXIT_STATUS = 0)
            message("IR sample not found!")
        else
            message("IR sample loaded!")
        end if
    end if
end on
```

Load an IR sample into a convolution reverb, placed in the first slot of send effect chain.

See Also

get_folder()

on async_complete

General: `$NI_ASYNC_ID`

save_array()

`save_array(<array-variable>, <mode>)`	
Saves an array to an external .nka file	
`<array-variable>`	The name of the array variable to be saved.
`<mode>`	**0:** A dialog window pops up, allowing you to save the .nka file. This mode can only be used in `on persistence_changed`, `on ui_control` and `on pgs_changed` callbacks. **1:** The array is directly saved in the `Data` folder. For user instruments, the `Data` folder is located beside the resource container. For Kontakt Player encoded library instruments, the `Data` folder is located here: OS X: `/Users/<UserName>/Library/Application Support/<LibraryName>/` Win: `C:\Users\<UserName>\AppData\Local\<LibraryName>\` This mode can be used synchronously in `on init`, and asynchronously in `on persistence_changed`, `on ui_control` and `on pgs_changed` callbacks.

Remarks

- It is also possible to save real and string arrays into .nka files.
- The exported .nka file consists of the name of the array followed by all its values, one value per line.
- When using mode **0**, the callback continues even if the loading dialog is still open.

See Also

on async_complete

load_array()

General: `$NI_ASYNC_ID`, `$NI_ASYNC_EXIT_STATUS`

save_array_str()

`save_array_str(<array-variable>, <path>)`	
Saves an array to an external .nka file with the specified absolute path.	
`<array-variable>`	The name of the array variable to be saved.
`<path>`	The absolute path of the .nka file to be saved.

Remarks

- The behaviour is similar to `save_array()`, but instead of manually choosing a save location, you can directly save the file to the specified location.
- If the file does not exist, but the folder does, a new .nka file will be created.
- This command can be used synchronously in `on init`, and asynchronously in `on persistence_changed`, `on ui_control` and `on pgs_changed` callbacks.

Example

```
on init
    message("")
    set_ui_height(2)

    declare $count
    declare $save_arr_id := -1
    declare %preset[8]
    declare @path
    { set save path here, for example:
    @path := "/Users/<username>/Desktop/Arrays/" }

    declare ui_button $Save
    declare ui_label $pattern_lbl (1, 1)
    declare ui_text_edit @preset_name
    declare ui_table %table[8] (2, 2, 100)

    make_persistent(%table)
    make_persistent(@preset_name)

    set_control_par(get_ui_id(@preset_name), $CONTROL_PAR_FONT_TYPE, 10)

    move_control_px(@preset_name, 73 + (3 * 92), 2)
    move_control_px($pattern_lbl, 66 + (3 * 92), 2)

    set_control_par_str(get_ui_id(@preset_name), $CONTROL_PAR_TEXT, "<empty>")
    set_text($pattern_lbl, "")
end on

on ui_control (%table)
    $count := 0
    while ($count < num_elements(%preset))
        %preset[$count] := %table[$count]

        inc($count)
    end while
end on

on ui_control ($Save)
    $save_arr_id := save_array_str(%preset, @path & @preset_name & ".nka")
```

```
end on

on async_complete
    if ($NI_ASYNC_ID = $save_arr_id)
        $save_arr_id := -1
        $Save := 0
    end if
end on
```

Save table presets with custom names. Make sure to set the path where the .nka files will be saved.

See Also

save_array()

load_array_str()

save_midi_file()

`save_midi_file(<path>)`	
Saves a MIDI file with the range specified by the `mf_set_export_area()` command.	
`<path>`	The absolute path of the MIDI file to be saved.

Example

```
on init
    message("")

    declare $save_mf_id := -1
    declare @path
    { set save path here, for example
    @path := "/Users/<username>/Desktop/MIDI Files/" }

    declare ui_text_edit @file_name
    declare ui_label $file_name_lbl (1, 1)
    declare ui_button $Save

    make_persistent(@file_name)

    set_control_par(get_ui_id(@file_name), $CONTROL_PAR_FONT_TYPE, 10)
    set_control_par_str(get_ui_id(@file_name), $CONTROL_PAR_TEXT, "<empty>")

    set_text($file_name_lbl, "")

    move_control($Save, 2, 1)
    move_control_px(@file_name, 73, 2)
    move_control_px($file_name_lbl, 66, 2)
end on

on ui_control ($Save)
    $save_mf_id := save_midi_file(@path & @file_name & ".mid")
end on

on async_complete
    if ($NI_ASYNC_ID = $save_mf_id)
        $save_mf_id := -1
        $Save := 0
    end if
end on
```
Saving a MIDI file.

See Also

mf_insert_file()

mf_set_export_area()

19. Music Information Retrieval

General Information

Music Information Retrieval (MIR) allows the extraction of meaningful features from audio files, such as pitch or the volume level of a sample. New KSP commands allow extraction of such parameters from samples via script. MIR functions are not asyncronous in the `on init` callback, but in all other callbacks they will run asynchronous.

Note: the type detection functions listed below (Sample Type, Drum Type, and Instrument Type) are designed to process one-shot audio samples.

detect_pitch()

`detect_pitch(<zone-id>, <pitch-result>)`	
Returns a real value representing the fundamental frequency of an audio sample, in semitones and cents. If detection fails, the function will return `~NI_DETECT_PITCH_INVALID`.	
`<zone-id>`	The ID of the zone.
`<pitch-result>`	The MIDI note value of the detected pitch.

detect_loudness()

`detect_loudness(<zone-id>, <loudness-result>)`	
Returns a real value representing the loudness of an audio sample in decibels. Loudness is measured according to the standard established by the International Telecommunication Union: *Algorithms to measure audio program loudness and true-peak audio level* - ITU-R BS.1770-4 (2015). If detection fails, the function will return `~NI_DETECT_LOUDNESS_INVALID`.	
`<zone-id>`	The ID of the zone.
`<loudness-result>`	The real value of the detected loudness in decibels.

detect_peak()

`detect_peak(<zone-id>, <peak-result>)`	
Returns a real value representing peak level of an audio sample in decibels. Peak is measured according to the standard established by the International Telecommunication Union: *Algorithms to measure audio program loudness and true-peak audio level - ITU-R BS.1770-4 (2015)*. If detection fails, the function will set `<peak-result>` to `~NI_DETECT_PEAK_INVALID`.	
`<zone-id>`	The ID of the zone.
`<peak-result>`	The real value of the detected peak level in decibels.

detect_rms()

`detect_rms(<zone-id>, <rms-result>)`	
Returns a real value representing the RMS level of an audio sample in decibels. If detection fails, the function will return ~`NI_DETECT_RMS_INVALID`.	
`<zone-id>`	The ID of the zone.
`<rms-result>`	The real value of the RMS level of the audio sample in decibels.

detect_sample_type()

`detect_sample_type(<zone-id>, <sample-type-result>)`	
Assigns `<sample-type-result>` a `$NI_DETECT_SAMPLE_TYPE` tag describing the whether an audio sample is a drum or an instrument. If detection fails, the function will return `$NI_DETECT_SAMPLE_TYPE_INVALID`.	
`<zone-id>`	The ID of the zone.

`<sample-type-result>`	The detected sample type, can be one of the following:
	`$NI_DETECT_SAMPLE_TYPE_INVALID`
	`$NI_DETECT_SAMPLE_TYPE_INSTRUMENT`
	`$NI_DETECT_SAMPLE_TYPE_DRUM`

detect_drum_type()

detect_drum_type(<zone-id>, <drum-type-result>)	
Assigns <drum-type-result> a $NI_DETECT_DRUM_TYPE tag describing the drum type of an audio sample. You can use this function if detect_sample_type() determines that a given audio sample is of type $NI_DETECT_SAMPLE_TYPE_DRUM. If detection fails, the function will return ~NI_DETECT_DRUM_TYPE_INVALID.	
<zone-id>	The ID of the zone.
<drum-type-result>	The detected drum type, can be one of the following: $NI_DETECT_DRUM_TYPE_INVALID $NI_DETECT_DRUM_TYPE_KICK $NI_DETECT_DRUM_TYPE_SNARE $NI_DETECT_DRUM_TYPE_CLOSED_HH $NI_DETECT_DRUM_TYPE_OPEN_HH $NI_DETECT_DRUM_TYPE_TOM $NI_DETECT_DRUM_TYPE_CYMBAL $NI_DETECT_DRUM_TYPE_CLAP $NI_DETECT_DRUM_TYPE_SHAKER $NI_DETECT_DRUM_TYPE_PERC_DRUM $NI_DETECT_DRUM_TYPE_PERC_OTHER

detect_instrument_type()

detect_instrument_type(<zone-id>, <instr-type-result>)	
Assigns `<instr-type-result>` a `$NI_DETECT_INSTRUMENT_TYPE` tag describing the instrument type of an audio sample. Hint: use this function if `detect_sample_type()` determines that a given audio sample is of type `$NI_DETECT_SAMPLE_TYPE_INSTRUMENT`. If detection fails, the function will return `$NI_DETECT_INSTRUMENT_TYPE_INVALID`.	
`<zone-id>`	The ID of the zone

`<instr-type-result>`	The detected instrument type, can be one of the following:
	`$NI_DETECT_INSTRUMENT_TYPE_INVALID`
	`$NI_DETECT_INSTRUMENT_TYPE_BASS`
	`$NI_DETECT_INSTRUMENT_TYPE_BOWED_STRING`
	`$NI_DETECT_INSTRUMENT_TYPE_BRASS`
	`$NI_DETECT_INSTRUMENT_TYPE_FLUTE`
	`$NI_DETECT_INSTRUMENT_TYPE_GUITAR`
	`$NI_DETECT_INSTRUMENT_TYPE_KEYBOARD`
	`$NI_DETECT_INSTRUMENT_TYPE_MALLET`
	`$NI_DETECT_INSTRUMENT_TYPE_ORGAN`
	`$NI_DETECT_INSTRUMENT_TYPE_PLUCKED_STRING`
	`$NI_DETECT_INSTRUMENT_TYPE_REED`
	`$NI_DETECT_INSTRUMENT_TYPE_SYNTH`
	`$NI_DETECT_INSTRUMENT_TYPE_VOCAL`

Examples

```
on init
    message("")

    declare ~pitch_result

    set_num_user_zones(1)

    declare ui_mouse_area $Drop
    declare ui_label $Info (1, 1)

    set_text($Info, "Drop sample here!")

    move_control_px($Drop, 66, 2)
    move_control_px($Info, 66, 2)

    set_control_par(get_ui_id($Drop), $CONTROL_PAR_DND_ACCEPT_AUDIO,
$NI_DND_ACCEPT_ONE)
    set_control_par(get_ui_id($Drop), $CONTROL_PAR_WIDTH, 150)
    set_control_par(get_ui_id($Info), $CONTROL_PAR_WIDTH, 150)

    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_HIGH_KEY, 127)
    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_LOW_VELO, 1)
    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_HIGH_VELO, 127)
    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_GROUP, 0)
end on

on ui_control ($Drop)
    if ($NI_MOUSE_EVENT_TYPE = $NI_MOUSE_EVENT_TYPE_DROP)
        wait_async(set_sample(%NI_USER_ZONE_IDS[0], !NI_DND_ITEMS_AUDIO[0]))

        wait_async(detect_pitch(%NI_USER_ZONE_IDS[0], ~pitch_result))
        wait_async(set_zone_par(%NI_USER_ZONE_IDS[0], ...
                                $ZONE_PAR_ROOT_KEY, ...
                                int(round(~pitch_result))))
        wait_async(set_zone_par(%NI_USER_ZONE_IDS[0], ...
                                $ZONE_PAR_TUNE, ...
                                int(100.0 * (round(~pitch_result) - ~pitch_result))
    end if
end on
```

Set the zone root key by rounding the pitch result to an integer value, then set the zone tune to correct for the pitch offset.

```
on init
    message("")

    set_num_user_zones(1)

    declare ui_mouse_area $Drop
    declare ui_label $Info (1, 1)

    declare $sample_type
    declare $instrument_type
    declare $drum_type
    declare !drum_types[10]
    !drum_types[$NI_DETECT_DRUM_TYPE_KICK]       := "Kick"
    !drum_types[$NI_DETECT_DRUM_TYPE_SNARE]      := "Snare"
    !drum_types[$NI_DETECT_DRUM_TYPE_CLOSED_HH]  := "Closed Hi-Hat"
    !drum_types[$NI_DETECT_DRUM_TYPE_OPEN_HH]    := "Open Hi-Hat"
    !drum_types[$NI_DETECT_DRUM_TYPE_TOM]        := "Tomr"
    !drum_types[$NI_DETECT_DRUM_TYPE_CYMBAL]     := "Cymbal"
    !drum_types[$NI_DETECT_DRUM_TYPE_CLAP]       := "Clap"
    !drum_types[$NI_DETECT_DRUM_TYPE_SHAKER]     := "Shaker"
    !drum_types[$NI_DETECT_DRUM_TYPE_PERC_DRUM]  := "Drum Percussion"
    !drum_types[$NI_DETECT_DRUM_TYPE_PERC_OTHER] := "Misc Percussion"
    declare !instrument_types[12]
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_BASS]           := "Bass"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_BOWED_STRING]   := "Bowed String"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_BRASS]          := "Brass"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_FLUTE]          := "Flute"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_GUITAR]         := "Guitar"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_KEYBOARD]       := "Keyboard"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_MALLET]         := "Mallet"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_ORGAN]          := "Organ"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_PLUCKED_STRING] := "Plucked String"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_REED]           := "Reed"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_SYNTH]          := "Synth"
    !instrument_types[$NI_DETECT_INSTRUMENT_TYPE_VOCAL]          := "Vocal"

    set_text($Info, "Drop sample here!")

    move_control_px($Drop, 66, 2)
    move_control_px($Info, 66, 2)

    set_control_par(get_ui_id($Drop), $CONTROL_PAR_DND_ACCEPT_AUDIO,
$NI_DND_ACCEPT_ONE)
    set_control_par(get_ui_id($Drop), $CONTROL_PAR_WIDTH, 150)
    set_control_par(get_ui_id($Info), $CONTROL_PAR_WIDTH, 150)

    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_HIGH_KEY, 127)
    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_LOW_VELO, 1)
    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_HIGH_VELO, 127)
    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_GROUP, 0)
end on

on ui_control ($Drop)
    if ($NI_MOUSE_EVENT_TYPE = $NI_MOUSE_EVENT_TYPE_DROP)
        wait_async(set_sample(%NI_USER_ZONE_IDS[0], !NI_DND_ITEMS_AUDIO[0]))
        wait_async(detect_sample_type(%NI_USER_ZONE_IDS[0], $sample_type))

        select ($sample_type)
            case $NI_DETECT_SAMPLE_TYPE_INSTRUMENT
                wait_async(detect_instrument_type(%NI_USER_ZONE_IDS[0],
$instrument_type))
            case $NI_DETECT_SAMPLE_TYPE_DRUM
                wait_async(detect_drum_type(%NI_USER_ZONE_IDS[0], $drum_type))
            case $NI_DETECT_SAMPLE_TYPE_INVALID
                set_text($Info, "Cannot recognize sample type!")
        end select

        if ($sample_type = $NI_DETECT_SAMPLE_TYPE_INSTRUMENT)
            if ($instrument_type = $NI_DETECT_INSTRUMENT_TYPE_INVALID)
                set_text($Info, "Cannot recognize instrument type!")
            else
                set_text($Info, "Instrument: " & !
instrument_types[$instrument_type])
            end if
        else
            if ($drum_type = $NI_DETECT_DRUM_TYPE_INVALID)
                set_text($Info, "Cannot recognize drum type!")
            else
                set_text($Info, "Instrument: " & !drum_types[$drum_type])
            end if
        end if
    end if
end on
```

Detect whether a sample is of type instrument or drum, and detect the corresponding drum or instrument type.

20. MIDI Object Commands

General Information

You can use only one MIDI object at a time within an NKI. The MIDI object is held in memory and can be accessed by any of the script slots. It is possible to add, remove and edit MIDI events within the object, as well as import and export MIDI files.

The Multi Script can also hold one MIDI object, and handles it in the same way as an NKI.

Please note that in Kontakt version 5.2, the MIDI file handling has been significantly updated. Commands and working methods from before the 5.2 release will remain in order to keep backwards compatibility; however this reference will document the post-5.2 working method.

Creating, Importing and Exporting MIDI files

When you initialize an instrument, an empty MIDI object is initialized with it. You can either start editing the object by defining a buffer size and inserting events, or by inserting a whole MIDI file.

If you want to create a MIDI sequence from scratch, you first need to assign a buffer size, which effectively creates a number of inactive MIDI events. From this point you can activate, i.e. insert, and edit MIDI events using the MIDI event commands.

You can also load a MIDI file to use or edit the data in a script. Depending on the command and variables you use, this will either be combined with any existing MIDI data, or will replace the existing data. It should be noted that loading a MIDI file is an asynchronous command, and thus the common asynchronous loading commands and working methods apply.

MIDI objects can be exported from Kontakt either by using the `save_midi_file()` command or via a drag and drop activated label widget. In either case, it is possible to define the export area, both in terms of start and end times, as well as the start and end tracks, by using the `mf_set_export_area()` command.

It is also possible to define up to 512 different export areas, each capable of pointing at different time and track range. The workflow here is to first set the amount of export areas with `mf_set_num_export_areas()`, then creating a matching number of ui_label widgets and assigning them to individual export area indices using `$CONTROL_PAR_MIDI_EXPORT_AREA_IDX` control parameter. From there, consider export area **0** as the "edit buffer", from which you then copy various time/track selections of interest into other export areas by using `mf_copy_export_area()` command.

Navigating and Editing

MIDI events in Kontakt's MIDI object are given event parameters, which are accessed using either the `mf_get_event_par()` or `mf_set_event_par()` commands. A unique event ID can be used to access a specific event, or you can navigate through events by position. The event ID is assigned whenever a MIDI event is created or loaded.

In order to access the event data of a loaded MIDI file, you can navigate around the MIDI events with a position marker, something analogous to a play-head. The position marker will focus on one single event at a time, allowing you to use a variety of commands to access or edit the event's parameters. You have the option to either navigate from one event to the next, or to specify exact positions in MIDI ticks.

It should be noted that MIDI Note Off messages are not used. When you load a MIDI file using the `mf_insert_file()` command, the Note Off events are used to give a length parameter to the respective Note On event, and then discarded.

by_marks()

`by_marks(<mark>)`	
Used to access a user-defined group of MIDI object events.	
`<mark>`	The event mark number, `$MARK_1 ... $MARK_10` .

See Also

mf_insert_file()

mf_insert_event()

mf_remove_event()

Events and MIDI: `$ALL_EVENTS`, `$CURRENT_EVENT`

by_track()

mf_set_mark()

mf_get_mark()

mf_get_id()

save_midi_file()

by_track()

`by_track(<track>)`	
Used to access events grouped by their track number.	
`<track>`	The track number of the events you wish to access.

Remarks

- Similar in functionality to the `by_marks()` command.

See Also

mf_insert_file()

mf_insert_event()

mf_remove_event()

Events and MIDI: `$ALL_EVENTS`, `$CURRENT_EVENT`

by_marks()

mf_set_mark()

mf_get_mark()

mf_get_id()

save_midi_file()

mf_copy_export_area()

`mf_copy_export_area(<index>)`
Copies the contents of MIDI export area **0** to the export area with the specified `<index>`.

Example

```
on init
    message("")
    make_perfview

    declare const $DND_AREAS := 4

    declare ui_label $label1 (1,1)
    declare ui_label $label2 (1,1)
    declare ui_label $label3 (1,1)
    declare ui_label $label4 (1,1)

    declare $i
    declare %ID[$DND_AREAS]
    %ID[0] := get_ui_id($label1)
    %ID[1] := get_ui_id($label2)
    %ID[2] := get_ui_id($label3)
    %ID[3] := get_ui_id($label4)
    declare !track_names[$DND_AREAS]
    !track_names[0] := "Synth 1"
    !track_names[1] := "Synth 2"
    !track_names[2] := "Bass"
    !track_names[3] := "Melody"

    mf_insert_file(get_folder($GET_FOLDER_PATCH_DIR) & "my_midi.mid", 0, 0, 0)
    mf_set_num_export_areas($DND_AREAS + 1)

    $i := 0
    while ($i < $DND_AREAS)
        set_control_par(%ID[$i], $CONTROL_PAR_DND_BEHAVIOUR, 1)
        set_control_par(%ID[$i], $CONTROL_PAR_MIDI_EXPORT_AREA_IDX, $i + 1)
        set_control_par_str(%ID[$i], $CONTROL_PAR_TEXT, !track_names[$i])

        mf_set_export_area(!track_names[$i], -1, -1, $i, $i)
        mf_copy_export_area($i + 1)

        inc($i)
    end while
end on
```

Loads a MIDI file and distributes the content found in the first four MIDI channels to four separate MIDI areas.

See Also

mf_set_export_area()

mf_set_num_export_areas()

mf_get_buffer_size()

```
mf_get_buffer_size()
```

Returns the size of the MIDI object's event buffer.

Remarks

- The maximum buffer size is **1000000** events, including both active and inactive events.
- Inserting a MIDI event will decrease the buffer size by one. Removing an event will increase it by one.

See Also

mf_insert_file()

mf_set_buffer_size()

mf_reset()

mf_insert_event()

mf_remove_event()

save_midi_file()

mf_get_event_par()

`mf_get_event_par(<event-id>, <parameter>)`	
Returns the value of an event parameter.	
`<event-id>`	The ID of the event to be edited.

`<parameter>`	The event parameter, either one of four freely assignable event parameters:
	`$EVENT_PAR_0`
	`$EVENT_PAR_1`
	`$EVENT_PAR_2`
	`$EVENT_PAR_3`
	or the built-in parameters of a event:
	`$EVENT_PAR_MIDI_CHANNEL`
	`$EVENT_PAR_MIDI_COMMAND`
	`$EVENT_PAR_MIDI_BYTE_1`
	`$EVENT_PAR_MIDI_BYTE_2`
	`$EVENT_PAR_POS`
	`$EVENT_PAR_NOTE_LENGTH`
	`$EVENT_PAR_ID`
	`$EVENT_PAR_TRACK_NR`

Remarks

- You can access all events in the MIDI object by using the `$ALL_EVENTS` constant as the event ID.
- You can access the currently selected event by using the `$CURRENT_EVENT` constant.
- You can also access events by track, or group them with event marks by using the `by_track()` and `by_marks()` commands.

See Also

mf_insert_file()

mf_insert_event()

mf_remove_event()

mf_get_id()

save_midi_file()

Events and MIDI: `$CURRENT_EVENT`

mf_get_first()

`mf_get_first(<track-index>)`	
Moves the position marker to the first event in the MIDI track.	
`<track-index>`	The number of the track you want to edit. **-1** refers to the whole file.

Remarks

- Using this command will also select the event at the position marker for editing.

See Also

mf_insert_file()

mf_get_next()

mf_get_next_at()

mf_get_num_tracks()

mf_get_prev()

mf_get_prev_at()

mf_get_last()

save_midi_file()

mf_get_id()

`mf_get_id()`
Returns the ID of the currently selected event, when using the navigation commands like `mf_get_first()`, `mf_get_next()`, etc.

See Also

mf_get_first()

mf_get_next()

mf_get_next_at()

mf_get_prev()

mf_get_prev_at()

mf_get_last()

mf_get_last()

`mf_get_last(<track-index>)`	
Moves the position marker to the last event in the MIDI track.	
`<track-index>`	The number of the track you want to edit. **-1** refers to the whole file.

Remarks

- Using this command will also select the event at the position marker for editing.

See Also

mf_insert_file()

mf_get_first()

mf_get_next()

mf_get_next_at()

mf_get_num_tracks()

mf_get_prev()

mf_get_prev_at()

save_midi_file()

mf_get_last_filename()

`mf_get_last_filename()`
Returns the filename (not the full path!) of the last MIDI file that was inserted into Kontakt, either via `mf_insert_file()`, or via drag and drop operation on `ui_mouse_area`.

Remarks

- This command will pair Note On and Note Off events to a single Note On with a Note Length parameter. The Note Off events will be discarded.

Example

```
on init
    message("")
    make_perfview

    declare const $DND_AREAS := 4

    declare ui_label $label1 (1,1)
    declare ui_label $label2 (1,1)
    declare ui_label $label3 (1,1)
    declare ui_label $label4 (1,1)

    declare $i
    declare %ID[$DND_AREAS]
    %ID[0] := get_ui_id($label1)
    %ID[1] := get_ui_id($label2)
    %ID[2] := get_ui_id($label3)
    %ID[3] := get_ui_id($label4)
    declare !track_names[$DND_AREAS]
    !track_names[0] := "Synth 1"
    !track_names[1] := "Synth 2"
    !track_names[2] := "Bass"
    !track_names[3] := "Melody"

    mf_insert_file(get_folder($GET_FOLDER_PATCH_DIR) & "my_midi.mid", 0, 0, 0)
    { declare export areas, area 0 serves as an edit buffer, so add one extra }
    mf_set_num_export_areas($DND_AREAS + 1)

    $i := 0
    while ($i < $DND_AREAS)
        set_control_par(%ID[$i], $CONTROL_PAR_DND_BEHAVIOUR, 1)
        set_control_par(%ID[$i], $CONTROL_PAR_MIDI_EXPORT_AREA_IDX, $i + 1)
        set_control_par_str(%ID[$i], $CONTROL_PAR_TEXT, mf_get_last_filename & " "
& !track_names[$i])

        mf_set_export_area(!track_names[$i], -1, -1, $i, $i)
        mf_copy_export_area($i + 1)

        inc($i)
    end while
end on
```

MIDI file loader which allows exporting first four tracks as individual MIDI files. Utilizes `mf_get_last_filename()` to show the exact name of the MIDI file that was loaded.

See Also

mf_insert_file()

mf_get_mark()

`mf_get_mark(<event-id>, <mark>)`	
Checks if an event is marked or not. Returns **1** if it is marked or **0** if it is not.	
`<event-id>`	The ID of the event to be edited
`<mark>`	The event mark number, `$MARK_1 ... $MARK_10`. You can also assign more than one mark to a single event, either by typing the command again, or by using the bitwise `.or.` operator, or by simply summing the event marks.

See Also

mf_insert_file()

mf_insert_event()

mf_remove_event()

Events and MIDI: `$ALL_EVENTS`, `$CURRENT_EVENT`

by_marks()

by_track()

mf_set_mark()

mf_get_mark()

mf_get_id()

save_midi_file()

mf_get_next()

`mf_get_next(<track-index>)`	
Moves the position marker to the next event in the MIDI track.	
`<track-index>`	The number of the track you want to edit. **-1** refers to the whole file.

Remarks

- Using this command will also select the event at the position marker for editing.

See Also

`load_midi_file()`

mf_get_first()

mf_get_next_at()

mf_get_num_tracks()

mf_get_prev()

mf_get_prev_at()

mf_get_last()

save_midi_file()

mf_get_next_at()

`mf_get_next_at(<track-index>, <pos>)`	
Moves the position marker to the next event in the MIDI track right after the defined position.	
`<track-index>`	The number of the track you want to edit. **-1** refers to the whole file.
`<pos>`	Position in MIDI ticks.

Remarks

- Using this command will also select the event at the position marker for editing.

See Also

`load_midi_file()`

mf_get_first()

mf_get_next()

mf_get_num_tracks()

mf_get_prev()

mf_get_prev_at()

mf_get_last()

save_midi_file()

mf_get_num_tracks()

```
mf_get_num_tracks()
```

Returns the number of tracks in the MIDI object.

See Also

mf_insert_file()

mf_get_first()

mf_get_next()

mf_get_next_at()

mf_get_prev()

mf_get_prev_at()

mf_get_last()

save_midi_file()

mf_get_prev()

`mf_get_prev(<track-index>)`	
Moves the position marker to the previous event in the MIDI track.	
`<track-index>`	The number of the track you want to edit. **-1** refers to the whole file.

Remarks

- Using this command will also select the event at the position marker for editing.

See Also

`load_midi_file()`

mf_get_first()

mf_get_next()

mf_get_next_at()

mf_get_num_tracks()

mf_get_prev_at()

mf_get_last()

save_midi_file()

mf_get_prev_at()

`mf_get_prev_at(<track-index>, <pos>)`	
Moves the position marker to the first event before the defined position.	
`<track-index>`	The number of the track you want to edit. **-1** refers to the whole file.
`<pos>`	Position in MIDI ticks.

Remarks

- Using this command will also select the event at the position marker for editing.

See Also

`load_midi_file()`

mf_get_first()

mf_get_next()

mf_get_next_at()

mf_get_num_tracks()

mf_get_prev()

mf_get_last()

save_midi_file()

mf_insert_event()

`mf_insert_event(<track>, <pos>, <command>, <byte1>, <byte2>)`	
Activates an inactive MIDI event in the MIDI object. However, because the command and position are defined in this command, it can be considered as an insertion.	
`<track>`	The track into which the event will be inserted.
`<pos>`	The position at which the event will be inserted, in ticks.

`<command>`	Defines the command type of the event, can be one of the following:
	`$MIDI_COMMAND_NOTE_ON`
	`$MIDI_COMMAND_POLY_AT`
	`$MIDI_COMMAND_CC`
	`$MIDI_COMMAND_PROGRAM_CHANGE`
	`$MIDI_COMMAND_MONO_AT`
	`$MIDI_COMMAND_PITCH_BEND`

`<byte1>`	The first byte of the command.
`<byte2>`	The second byte of the command.

Remarks

- It is not possible to insert MIDI events without first setting an event buffer size with the `mf_set_buffer_size()` command.
- Using this command when the buffer is full, i.e. has a size of zero, will do nothing.
- You can retrieve the event ID of the inserted event into a variable by writing:
  ```
  <variable> := mf_insert_event(<track>, <pos>, <command>, <byte1>,
  <byte2>)
  ```

See Also

mf_insert_file()

mf_set_buffer_size()

mf_get_buffer_size()

mf_reset()

mf_remove_event()

save_midi_file()

mf_insert_file()

`mf_insert_file(<path>, <track-offset>, <position-offset>, <mode>)`	
Inserts a MIDI file into the object.	
`<path>`	The absolute path of the MIDI file, including the file name.
`<track-offset>`	Applies a track offset to the MIDI data.
`<position-offset>`	Applies a position offset, in ticks, to the MIDI data.
`<mode>`	Defines the mode of insertion: **0:** Replaces all existing events **1:** Replaces only overlapping events **2:** Merges all events

Remarks

- This command runs synchronously in `on init` callback and asynchronously in all other callbacks, so it is advised to use the `on async_complete` callback to verify the loading status.
- This command will pair Note On and Note Off events to a single Note On with a length parameter. The Note Off events will be discarded.

Example

```
on init
    declare $load_mf_id := -1
    declare @file_name
    declare @filepath

    @file_name := "test.mid"
    @filepath := get_folder($GET_FOLDER_FACTORY_DIR) & @file_name

    declare ui_button $load_file
end on

on ui_control($load_file)
    $load_mf_id := mf_insert_file(@filepath, 0, 0, 0)
end on

on async_complete
    if ($NI_ASYNC_ID = $load_mf_id)
        $load_mf_id := -1

        if ($NI_ASYNC_EXIT_STATUS = 0)
            message("ERROR: MIDI file not found!")
        else
            message("Loaded MIDI file: " & @file_name & "!")
        end if
    end if
end on
```

Loading a MIDI file with a button. In order for this to work, you will need to put a MIDI file called "test.mid" into your Kontakt factory data folder. Otherwise, the defined error message will be displayed.

See Also

on async_complete

save_midi_file()

mf_set_event_par()

mf_get_event_par()

General: `$NI_ASYNC_ID`, `$NI_ASYNC_EXIT_STATUS`

mf_remove_event()

`mf_remove_event(<event-id>)`	
Deactivates an event in the MIDI object, effectively removing it.	
`<event-id>`	The ID of the event to be deactivated.

Remarks

- Using this command will decrease the MIDI event buffer size by one.

See Also

mf_insert_file()

mf_set_buffer_size()

mf_get_buffer_size()

mf_reset()

mf_insert_event()

save_midi_file()

mf_reset()

```
mf_reset()
```

Resets the MIDI object, sets the event buffer to zero, and removes all events.

Remarks

- This command purges all data in the MIDI object. Use with caution!
- This command is asynchronous, thus it returns an async ID and calls the `on async_complete` callback.

See Also

mf_insert_file()

mf_set_buffer_size()

mf_reset()

mf_insert_event()

mf_remove_event()

save_midi_file()

mf_set_buffer_size()

`mf_set_buffer_size(<num-events>)`	
Defines a number of inactive MIDI events, that can be activated and edited.	
`<num-events>`	The size of the MIDI object edit buffer.

Remarks

- Using the `mf_insert_event()` and `mf_remove_event()` technically activates or deactivates events in the buffer.
- It is not possible to insert MIDI events without setting a buffer size first.
- The maximum buffer size is **1000000** events, including both active and inactive events.
- This command runs synchronously in `on init` callback and asynchronously in all other callbacks, so it is advised to use the `on async_complete` callback to verify the loading status.
- Inserting a MIDI event will decrease the buffer size by one. Removing an event will increase it by one.
- Inserting a MIDI file will not affect the buffer.

See Also

mf_insert_file()

mf_get_buffer_size()

mf_reset()

mf_insert_event()

mf_remove_event()

save_midi_file()

mf_set_event_par()

`mf_set_event_par(<event-id>, <parameter>, <value>)`	
Sets an event parameter.	
`<event-id>`	The ID of the event to be edited.

`<parameter>`	The event parameter, either one of four freely assignable event parameters:
	`$EVENT_PAR_0`
	`$EVENT_PAR_1`
	`$EVENT_PAR_2`
	`$EVENT_PAR_3`
	Or the built-in parameters of a event:
	`$EVENT_PAR_MIDI_CHANNEL`
	`$EVENT_PAR_MIDI_COMMAND`
	`$EVENT_PAR_MIDI_BYTE_1`
	`$EVENT_PAR_MIDI_BYTE_2`
	`$EVENT_PAR_POS`
	`$EVENT_PAR_NOTE_LENGTH`
	`$EVENT_PAR_ID`
	`$EVENT_PAR_TRACK_NR`

`<value>`	The value of the event parameter.

Remarks

- You can control all events in the MIDI object by using the `$ALL_EVENTS` constant as the event ID.
- You can access the currently selected event by using the `$CURRENT_EVENT` constant.
- You can also control events by track, or group them with event marks by using the `by_track()` and `by_marks()` commands.

See Also

mf_insert_file()

mf_insert_event()

mf_remove_event()

Events and MIDI: `$ALL_EVENTS`, `$CURRENT_EVENT`

by_marks()

by_track()

mf_set_mark()

mf_get_id()

save_midi_file()

mf_set_export_area()

`mf_set_export_area(<name>, <start-pos>, <end-pos>, <start-track>, <end-track>)`	
Defines the part of the object that will be exported when using a drag and drop area, or when using `save_midi_file()` command.	
`<name>`	Sets the name of the exported file.
`<start-pos>`	Defines the start position (in ticks) of the export area. Use **-1** to set this to the start of the object.
`<end-pos>`	Defines the end position (in ticks) of the export area. Use **-1** to set this to the end of the object.
`<start-track>`	Defines the first track to be included in the export area. Use **-1** to set this to the first track of the object.
`<end-track>`	Defines the last track to be included in the export area. Use **-1** to set this to the last track of the object.

Remarks

- If a start point is given a value greater than the end point, the values will be swapped.
- When this command is executed, the events in the range are checked if they are valid MIDI commands. The command will return a value of **0** if all events are valid, otherwise it will return the event ID of the first invalid event.

Example

```
on init
    declare $area_status
    declare @filepath
    @filepath := get_folder($GET_FOLDER_FACTORY_DIR) & "test.mid"

    declare ui_button $CheckMIDI

    mf_insert_file(@filepath, 0, 0, 0)
end on

on ui_control($CheckMIDI)
    $area_status := mf_set_export_area(“name”, -1, -1,-1,-1)

    if ($area_status = 0)
        message(“All Good”)
    else
        message(“Error: Check event with ID ” & $area_status & "!")
    end if

    $CheckMIDI := 0
end on
```

A simple script, using this command to check if all events in a MIDI file are valid. If there is an error it will display the event ID of the first invalid event. In order for this to work you will have to put a MIDI file called "test.mid" into your Kontakt factory data folder.

See Also

mf_insert_file()

save_midi_file()

Specific: `$CONTROL_PAR_DND_BEHAVIOUR`

mf_set_mark()

`mf_set_mark(<event-id>, <mark>, <status>)`	
Marks an event, so that you may group events together and process that group quickly.	
`<event-id>`	The ID of the event to be marked.
`<mark>`	The event mark number, `$MARK_1 ... $MARK_10`. You can also assign more than one mark to a single event, either by typing the command again, or by using the bitwise `.or.` operator, or by simply summing the event marks.
`<status>`	Set this to **1** to mark an event or to **0** to unmark an event.

See Also

mf_insert_file()

mf_insert_event()

mf_remove_event()

Events and MIDI: `$ALL_EVENTS`, `$CURRENT_EVENT`

by_marks()

by_track()

mf_get_mark()

mf_get_id()

save_midi_file()

mf_set_num_export_areas()

```
mf_set_num_export_areas(<num-areas>)
```

Sets the number of export areas, with a maximum of **512**.

Remarks

- Area index **0** is always set with `mf_set_export_area()`.
- The contents of area index **0** can be copied to other areas by calling `mf_copy_export_area()`.

See Also

mf_set_export_area()

mf_copy_export_area()

21. Built-in Variables and Constants

General

`$CURRENT_SCRIPT_SLOT`

This variable returns the script slot of the current script (zero-based, i.e. the first script slot is **0**).

`%GROUPS_SELECTED[<group-idx>]`

Each index of this variable array points to the group with the same index.
If a group is selected for editing, the corresponding array cell contains a **1**, otherwise **0**.

`$NI_ASYNC_EXIT_STATUS`

This variable returns a value of **1** if the command that triggered the `on async_complete` callback has successfully completed its action. **0** if the command could not complete its action, e.g. file not found.

`$NI_ASYNC_ID`

This variable returns the ID of the command that triggered the `on async_complete` callback.

`$NI_BUS_OFFSET`

This constant is to be used in the `<generic>` part of the engine parameter commands to point to the instrument bus level. Add the index of the bus you wish to address, e.g. `$NI_BUS_OFFSET + 2` will point to instrument bus **3**.

`$NUM_GROUPS`

This variable returns the total amount of groups in an instrument.

`$NUM_OUTPUT_CHANNELS`

This variable returns the total amount of output channels of the respective Kontakt multi, not counting Aux channels.

`$NUM_ZONES`

This variable returns the total amount of zones in an instrument.

`$PLAYED_VOICES_INST`

This variable returns the amount of played voices for the current instrument.

`$PLAYED_VOICES_TOTAL`

This variable returns the amount of played voices for all instruments.

`$REF_GROUP_IDX`
This variable returns the group index of the currently viewed group in Kontakt's instrument edit view.

Path Variables

`$GET_FOLDER_LIBRARY_DIR`
If used with a Kontakt Player encoded NKI: library folder.
If used with an unencoded NKI: the user content directory.
`$GET_FOLDER_FACTORY_DIR`
The factory folder of Kontakt, mainly used for loading factory IR samples.
Note: this is not the Kontakt Factory Library folder!
`$GET_FOLDER_PATCH_DIR`
The directory in which the patch was saved.
If the patch was not saved before, an empty string is returned.

Time Machine Pro Variables

User access for the two voice limits (Standard and High Quality) of the Time Machine Pro, to be used with `set_voice_limit()` and `get_voice_limit()`.
`$NI_VL_TMPRO_STANDARD`
`$NI_VL_TMRPO_HQ`

Events and MIDI

`$ALL_GROUPS`

This constant addresses all groups in the instrument when used with `disallow_group()`, `allow_group()` or set_event_par_arr() functions.

`$ALL_EVENTS`

This constant addresses all events in functions which deal with an event ID number.

It also works with MIDI object commands that require a MIDI event ID.

`$MARK_1 ... $MARK_28`

These constants can be used to create a bitmask that is used to group events at will.
It is intended to be used with `by_marks()`, `set_event_mark()`, `get_event_mark()`, `delete_event_mark()`, `mf_set_mark()` and `mf_get_mark()` commands.

`%CC[<controller-number>]`

This variable array contains the current value for the specified MIDI CC (continuous controller) message.

`$CC_NUM`

This variable contains the MIDI CC (continuous controller) number of the controller which triggered the `on controller` callback.

`%CC_TOUCHED[<controller-number>]`

This variable array updates itself whenever a MIDI CC value is changed - **1** if the specified MIDI CC value has changed, **0** otherwise.

`$EVENT_ID`

This variable contains the unique ID number of the event which triggered the `on note` or `on release` callback.

`$CURRENT_EVENT`

This variable contains the unique ID number of the currently selected MIDI event, i.e. the MIDI event at the position marker.

`$EVENT_NOTE`

This variable contains the note number of the event which triggered the `on note` or `on release` callback.

`$EVENT_VELOCITY`

This variable contains the velocity of the note which triggered the `on note` callback.

Event Parameter Constants

Event parameters to be used with `set_event_par()` and `get_event_par()`:

- `$EVENT_PAR_0 ... $EVENT_PAR_3`
- `$EVENT_PAR_VOLUME`
- `$EVENT_PAR_PAN`
- `$EVENT_PAR_TUNE`
- `$EVENT_PAR_NOTE`
- `$EVENT_PAR_VELOCITY`
- `$EVENT_PAR_MIDI_CHANNEL`

Event parameters to be used with `set_event_par_arr()` and `get_event_par_arr()`:

- `$EVENT_PAR_ALLOW_GROUP`
- `$EVENT_PAR_CUSTOM`
- `$EVENT_PAR_MOD_VALUE_ID`
- `$EVENT_PAR_MOD_VALUE_EX_ID`

Event parameters to be used with `get_event_par()` **only**:

- `$EVENT_PAR_SOURCE` (**-1** if event originates from outside, otherwise slot number **0 ... 4**)
- `$EVENT_PAR_PLAY_POS` (Returns the absolute position of the play cursor within a zone in microseconds)
- `$EVENT_PAR_ZONE_ID` (Returns the zone ID of the event- Can only be used with active events. Returns **-1** if no zone is triggered. Returns the highest zone ID if more than one zone is triggered by the event. Make sure the voice is running by writing e.g. `wait (1)` before retrieving the zone ID!)

Event parameters to be used with `set_event_par()` and `get_event_par()` in multi scripts **only**:

- `$EVENT_PAR_MIDI_COMMAND`
- `$EVENT_PAR_MIDI_BYTE_1`
- `$EVENT_PAR_MIDI_BYTE_2`

Event parameters to be used with `mf_set_event_par()` and `mf_get_event_par()`:

- `$EVENT_PAR_POS`
- `$EVENT_PAR_NOTE_LENGTH`
- `$EVENT_PAR_ID`
- `$EVENT_PAR_TRACK_NR`

`%EVENT_PAR[<event-par>]`

This variable array contains values of `$EVENT_PAR_0 ... $EVENT_PAR_3` along with `$EVENT_PAR_CUSTOM` indices 4-15, valid for `$EVENT_ID`.

Event Status Constants

These are values returned by `event_status()` command:

`$EVENT_STATUS_INACTIVE`

`$EVENT_STATUS_NOTE_QUEUE`

Multi script only:

`$EVENT_STATUS_MIDI_QUEUE`

`%GROUPS_AFFECTED`

This variable array contains indices of those groups that are affected by the current MIDI Note On or Note Off events.

The size of the array changes depending on the number of groups the event affects, so use the `num_elements()` command to get the correct array size.

The returned indices come before any `allow_group()` or `disallow_group()` commands, so it can be used to analyze the mapping of the instrument.

`$NOTE_HELD`

This variable contains **1** if the key which triggered the callback is still held, **0** otherwise.

`%POLY_AT[<note-number>]`

This variable array contains current values of MIDI Polyphonic Aftertouch for all MIDI note numbers.

`$POLY_AT_NUM`

This variable contains the note number of the MIDI Polyphonic Aftertouch event which triggered the `on poly_at` callback.

`$RPN_ADDRESS`

This variable contains the address of a received RPN or NRPN message (**0 ... 16383**).

`$RPN_VALUE`

This variable contains the value of a received RPN or NRPN message (**0 ... 16383**).

`$VCC_MONO_AT`

This constant specifies Virtual Continuous Controller for mono aftertouch (MIDI Channel Pressure message), mainly for use in `on controller`.

`$VCC_PITCH_BEND`

This constant specifies Virtual Continuous Controller for MIDI Pitch Bend messages, mainly for use in `on controller`.

`%KEY_DOWN[<note-number>]`

This variable array contains the current state of all keys. **1** if the key is held, **0** otherwise.

`%KEY_DOWN_OCT[<note-number>]`

This variable array contains **1** if a note, independent of the octave, is held. **0** otherwise. Due to this, the note number should be a value between **0** (C) and **11** (B).

Time and Transport

Date And Time Variables

These variables return the current date and time as set by the operating system, in form of integer values.

`$NI_DATE_YEAR`

`$NI_DATE_MONTH` (**1 ... 12**)

`$NI_DATE_DAY` (**1 ... 31**)

`$NI_TIME_HOUR` (**0 ... 23**)

`$NI_TIME_MINUTE` (**0 ... 59**)

`$NI_TIME_SECOND` (**0 ... 59**)

`$DISTANCE_BAR_START`

This variable returns the time of a MIDI Note On message in microseconds from the beginning of the current bar, with respect to the current tempo.

`$DURATION_BAR`

This variable returns the duration in microseconds of one bar with respect to the current tempo.

It only works if the clock is running, otherwise it will return **0**.

You can also retrieve the duration of one bar by using `$SIGNATURE_NUM` and `$SIGNATURE_DENOM` in combination with various `$DURATION_` constants.

`$DURATION_QUARTER`

This variable returns the duration of a quarter note in microseconds, with respect to the current tempo. Also available:

`$DURATION_EIGHTH`

`$DURATION_SIXTEENTH`

`$DURATION_QUARTER_TRIPLET`

`$DURATION_EIGHTH_TRIPLET`

`$DURATION_SIXTEENTH_TRIPLET`

`$ENGINE_UPTIME`

This variable returns the time period in milliseconds (not microseconds!) that has passed since the instantiation of Kontakt. The engine uptime is calculated from the sample rate and can thus be used in musical contexts, (eg. building arpeggiators or sequencers) as it remains in sync, even in an offline bounce.

`$KSP_TIMER`

This variable returns the time period in microseconds that has passed since Kontakt was instantiated.

It can be reset with `reset_ksp_timer`.

The KSP timer is based on the CPU clock and thus runs at a constant rate, regardless of whether or not Kontakt is being used in real time. As such, it should be used to test the efficiency of scripts and not to make musical calculations.

`$NI_SONG_POSITION`

This variable returns the host’s current song position at 960 PPQ (pulses per quarter note).

`$NI_TRANSPORT_RUNNING`

This variable contains **1** if the host's transport is running, **0** otherwise.

`$SIGNATURE_NUM`

This variable contains the numerator of the current time signature, i.e. **4**/4.

`$SIGNATURE_DENOM`

This variable contains the denominator of the current time signature, i.e. 4/**4**.

Tempo Unit Constants

These constants are used to control the unit parameter of time-related controls (e.g. Delay Time, Envelope Attack etc.) with engine parameter variables like `$ENGINE_PAR_DL_TIME_UNIT`.

`$NI_SYNC_UNIT_ABS` (not synchronized to tempo)

`$NI_SYNC_UNIT_WHOLE`

`$NI_SYNC_UNIT_WHOLE_TRIPLET`

`$NI_SYNC_UNIT_HALF`

`$NI_SYNC_UNIT_HALF_TRIPLET`

`$NI_SYNC_UNIT_QUARTER`

`$NI_SYNC_UNIT_QUARTER_TRIPLET`

`$NI_SYNC_UNIT_8TH`

`$NI_SYNC_UNIT_8TH_TRIPLET`

`$NI_SYNC_UNIT_16TH`

`$NI_SYNC_UNIT_16TH_TRIPLET`

`$NI_SYNC_UNIT_32ND`

`$NI_SYNC_UNIT_32ND_TRIPLET`

`$NI_SYNC_UNIT_64TH`

`$NI_SYNC_UNIT_64TH_TRIPLET`

`$NI_SYNC_UNIT_256TH`

`$NI_SYNC_UNIT_ZONE` (only applies to the Speed parameter in certain Source module sampler modes)

`%NOTE_DURATION[<note-number>]`

This variable array contains the duration since note start in microseconds for each key. Mostly only makes sense in `on release` callback.

`$NI_BAR_START_POSITION`

This variable returns the start of current bar in ticks (at 960 PPQ) from the start of the host project.

Note: Since the AAX plugin format doesn't provide this information, this variable will return **-1** in ProTools!

Callbacks and UI

`$NI_CALLBACK_ID`

This variable returns the ID number of the callback. Every callback has a unique ID number which remains the same within a user function.

`$NI_CALLBACK_TYPE`

These constants return the callback type of a specific callback ID. Useful for retrieving the callback type that triggered a specific user function. The following constants are available:

`$NI_CB_TYPE_ASYNC_OUT`

`$NI_CB_TYPE_CONTROLLER`

`$NI_CB_TYPE_INIT`

`$NI_CB_TYPE_LISTENER`

`$NI_CB_TYPE_NOTE`

`$NI_CB_TYPE_NRPN`

`$NI_CB_TYPE_PERSISTENCE_CHANGED`

`$NI_CB_TYPE_PGS`

`$NI_CB_TYPE_POLY_AT`

`$NI_CB_TYPE_RELEASE`

`$NI_CB_TYPE_RPN`

`$NI_CB_TYPE_UI_CONTROL`

`$NI_CB_TYPE_UI_CONTROLS`

`$NI_CB_TYPE_UI_UPDATE`

Multi script only:

`$NI_CB_TYPE_MIDI_IN`

Knob Unit Mark Constants

These constants are to be used with `set_knob_unit()` or `$CONTROL_PAR_UNIT`.

`$KNOB_UNIT_NONE`

`$KNOB_UNIT_DB`

`$KNOB_UNIT_HZ`

`$KNOB_UNIT_PERCENT`

`$KNOB_UNIT_MS`

`$KNOB_UNIT_ST`

`$KNOB_UNIT_OCT`

`$NI_UI_ID`

This variable returns the ID of the UI widget which triggered the on ui_control or on ui_controls callbacks.

`$NI_SIGNAL_TIMER_BEAT`
`$NI_SIGNAL_TIMER_MS`
`$NI_SIGNAL_TRANSP_START`
`$NI_SIGNAL_TRANSP_STOP`

These constants can be used with `set_listener()` or `change_listener_par()` to set which signals will trigger the on listener callback.

`$NI_SIGNAL_TYPE`

This variable can be used in the `on listener` callback to determine which signal type triggered it.

`$NI_KONTAKT_IS_HEADLESS`

This variable returns **1** if the GUI of Kontakt is not available or loadable. Currently this is only possible when Kontakt is used in Maschine+. When Kontakt is used in a regular host, it will still return **0** even if the GUI is not visible.

`$NI_KONTAKT_IS_STANDALONE`

This variable returns **1** if Kontakt is running standalone, **0** if it's loaded as a plugin in a host.

Mathematical Constants

`~NI_MATH_PI`
This constant returns the value of π (approximately **3.14159…**).

`~NI_MATH_E`
This constant returns the value of *e* (approximately **2.71828…**).

22. Control Parameters

General

`$CONTROL_PAR_NONE`

Nothing will be applied to the widget.

`$CONTROL_PAR_CUSTOM_ID`

Sets or returns a custom value bound to a specific UI widget. This allows custom tagging of UI widgets, for instance to bind them to a specific parameter in the script.

`$CONTROL_PAR_TYPE`

Returns the type of the UI widget.

Only works with `get_control_par()`.

Possible return values are:

`$NI_CONTROL_TYPE_NONE` (UI ID belongs to a normal variable, not a UI widget)

`$NI_CONTROL_TYPE_BUTTON`

`$NI_CONTROL_TYPE_KNOB`

`$NI_CONTROL_TYPE_MENU`

`$NI_CONTROL_TYPE_VALUE_EDIT`

`$NI_CONTROL_TYPE_LABEL`

`$NI_CONTROL_TYPE_TABLE`

`$NI_CONTROL_TYPE_WAVEFORM`

`$NI_CONTROL_TYPE_WAVETABLE`

`$NI_CONTROL_TYPE_SLIDER`

`$NI_CONTROL_TYPE_TEXT_EDIT`

`$NI_CONTROL_TYPE_FILE_SELECTOR`

`$NI_CONTROL_TYPE_SWITCH`

`$NI_CONTROL_TYPE_XY`

`$NI_CONTROL_TYPE_LEVEL_METER`

`$NI_CONTROL_TYPE_MOUSE_AREA`

`$NI_CONTROL_TYPE_PANEL`

Size, Position and Look

$CONTROL_PAR_POS_X

Sets or returns the horizontal position in pixels.

$CONTROL_PAR_POS_Y

Sets or returns the vertical position in pixels.

$CONTROL_PAR_GRID_X

Sets or returns the horizontal position in grid units.

$CONTROL_PAR_GRID_Y

Sets or returns the vertical position in grid units.

$CONTROL_PAR_WIDTH

Sets or returns the width of the control in pixels.

$CONTROL_PAR_HEIGHT

Sets or returns the height of the control in pixels.

$CONTROL_PAR_GRID_WIDTH

Sets or returns the width of the control in grid units.

$CONTROL_PAR_GRID_HEIGHT

Sets or returns the height of the control in grid units.

$CONTROL_PAR_HIDE

Sets or returns the hide status. Can be used with the following built-in constants:

`$HIDE_PART_BG` (background of `ui_knob`, `ui_label`, `ui_value_edit` and `ui_table`)

`$HIDE_PART_VALUE` (value of `ui_knob` and `ui_table`)

`$HIDE_PART_TITLE` (title of `ui_knob`)

`$HIDE_PART_MOD_LIGHT` (mod ring light of `ui_knob`)

`$HIDE_PART_NOTHING` (show all elements of the widget)

`$HIDE_PART_CURSOR` (cursor of `ui_xy`)

`$HIDE_WHOLE_CONTROL`

$CONTROL_PAR_PICTURE

Sets or returns the picture name. Full path and extension are not required. If the instrument references a resource container (in this case, Kontakt will look for the specified filename in the `pictures` subfolder). If the NKI does not reference a resource container, it will first look in the user `pictures` folder (`Documents/Native Instruments/Kontakt 7/pictures`), then in the Kontakt factory `pictures` folder.

$CONTROL_PAR_PICTURE_STATE

Sets or returns the picture state of the control for `ui_label`, `ui_table`, `ui_value_edit` and `ui_xy`.

$CONTROL_PAR_PARENT_PANEL

Assigns a widget to a particular `ui_panel`. The value should be the UI ID of the panel. A given widget can only belong to a single parent panel.

$CONTROL_PAR_Z_LAYER

Sets or returns the Z layer position of the widget. Widgets can be placed in one of three layers. Within these layers they are then positioned by widget type, and then by declaration order.

0: Default layer. All widgets are assigned to this layer by default.

-1: Background layer. Widgets in this layer are placed below the default layer.

1: Foreground layer. Widgets in this layer are placed on top of the default and background layers.

Z layer order by widget type (from top to bottom):

1. Mouse Area
2. Text Edit
3. XY Pad
4. Table
5. Menu
6. Value Edit
7. Button
8. Switch
9. Slider
10. Knob
11. Label
12. Level Meter
13. Wavetable
14. Waveform
15. File Selector

Values

`$CONTROL_PAR_MIN_VALUE`

Returns the minimum declared value of the widget. This control parameter only makes sense for `ui_knob`, `ui_slider` and `ui_value_edit`.

Only works with `get_control_par()`.

`$CONTROL_PAR_MAX_VALUE`

Returns the minimum declared value of the widget. This control parameter only makes sense for `ui_knob`, `ui_slider` and `ui_value_edit`.

Only works with `get_control_par()`.

`$CONTROL_PAR_VALUE`

Sets or returns the value of the widget. Note that doing this does **NOT** execute the UI callback of the widget!

`$CONTROL_PAR_DEFAULT_VALUE`

Sets or returns the default value of the widget. A widget is set to the default value when clicking it with [Ctrl] (Windows) or [Cmd] (macOS) key held. This is only applicable to `ui_knob` and `ui_slider`.

Text

`$CONTROL_PAR_TEXT`

Sets or returns the widget text, similar to `set_text()`.

`$CONTROL_PAR_TEXTLINE`

Adds a text line to multiline labels, similar to `add_text_line()`.

`$CONTROL_PAR_IDENTIFIER`

Returns the name of the UI widget as declared in the script, without the type specifier ($, %, @, ?).

Only works with `get_control_par()`.

`$CONTROL_PAR_HELP`

Sets or returns the help text, which is displayed in Kontakt's Info pane when hovering above the widget.

`$CONTROL_PAR_LABEL`

Sets or returns the widget label, similar to `set_knob_label()`, except this control parameter is not limited to knobs only - it also works with `ui_slider`, `ui_switch` and `ui_xy`.

This is also the string published to the host when using host automation.

`$CONTROL_PAR_SHORT_NAME`

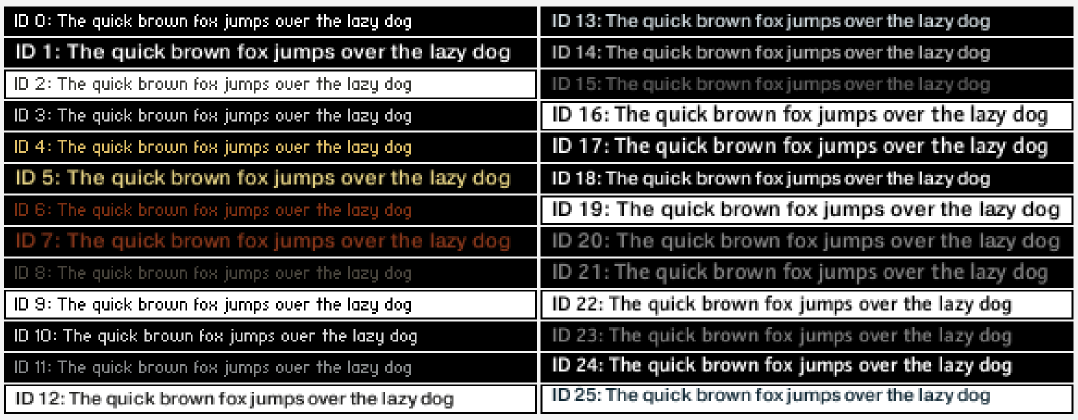
Sets or returns the short name of the widget.

`$CONTROL_PAR_UNIT`

Sets or returns the knob unit, similar to `set_knob_unit()`.

$CONTROL_PAR_FONT_TYPE

Sets or returns the font type. Numbers **0 ... 25** are used to select any of the 26 factory fonts, as shown below. Combine with `get_font_id()` to use custom fonts.



ID 0: The quick brown fox jumps over the lazy dog
ID 1: The quick brown fox jumps over the lazy dog
ID 2: The quick brown fox jumps over the lazy dog
ID 3: The quick brown fox jumps over the lazy dog
ID 4: The quick brown fox jumps over the lazy dog
ID 5: The quick brown fox jumps over the lazy dog
ID 6: The quick brown fox jumps over the lazy dog
ID 7: The quick brown fox jumps over the lazy dog
ID 8: The quick brown fox jumps over the lazy dog
ID 9: The quick brown fox jumps over the lazy dog
ID 10: The quick brown fox jumps over the lazy dog
ID 11: The quick brown fox jumps over the lazy dog
ID 12: The quick brown fox jumps over the lazy dog
ID 13: The quick brown fox jumps over the lazy dog
ID 14: The quick brown fox jumps over the lazy dog
ID 15: The quick brown fox jumps over the lazy dog
ID 16: The quick brown fox jumps over the lazy dog
ID 17: The quick brown fox jumps over the lazy dog
ID 18: The quick brown fox jumps over the lazy dog
ID 19: The quick brown fox jumps over the lazy dog
ID 20: The quick brown fox jumps over the lazy dog
ID 21: The quick brown fox jumps over the lazy dog
ID 22: The quick brown fox jumps over the lazy dog
ID 23: The quick brown fox jumps over the lazy dog
ID 24: The quick brown fox jumps over the lazy dog
ID 25: The quick brown fox jumps over the lazy dog

For responsive widgets (`ui_button`, `ui_switch` and `ui_menu`), the font can also be set separately for each of the states via the following control parameters:

`$CONTROL_PAR_FONT_TYPE_ON`

`$CONTROL_PAR_FONT_TYPE_OFF_PRESSED`

`$CONTROL_PAR_FONT_TYPE_ON_PRESSED`

`$CONTROL_PAR_FONT_TYPE_OFF_HOVER`

`$CONTROL_PAR_FONT_TYPE_ON_HOVER`

Not using any of the five additional state fonts will result in the default (`$CONTROL_PAR_FONT_TYPE`) being used instead.

$CONTROL_PAR_DISABLE_TEXT_SHIFTING

Deactivates text position shifting when clicking on `ui_button` or `ui_switch`.

$CONTROL_PAR_TEXTPOS_Y

Shifts the vertical position in pixels of text in `ui_button`, `ui_switch`, `ui_label`. `ui_menu`, and parameter *name* in `ui_value_edit`.

$CONTROL_PAR_TEXT_ALIGNMENT

The text alignment in `ui_button`, `ui_switch`, `ui_label`, `ui_menu` and `ui_text_edit`:

0: Left

1: Centered

2: Right

$CONTROL_PAR_VALUEPOS_Y

Shifts the vertical position in pixels of the parameter *value* in `ui_value_edit`.

Automation

$CONTROL_PAR_ALLOW_AUTOMATION

Sets or returns if a UI widget can be automated (**1**) or not (**0**). By default, automation is allowed for all automatable widgets (`ui_knob`, `ui_slider`, `ui_switch`, `ui_xy` cursors).

This control parameter can only be used in the `on init` callback.

When allowing automation for `ui_xy` cursors, use `set_control_par_arr()` command instead of `set_control_par()`.

$CONTROL_PAR_AUTOMATION_NAME

Sets or returns an automation name to a UI widget when used with `set_control_par_str()`.

`$CONTROL_PAR_LABEL` can be used to set the automation value string.

When assigning automation names to `ui_xy` cursors, use `set_control_par_str_arr()` command instead of `set_control_par_str()`.

$CONTROL_PAR_AUTOMATION_ID

Sets or returns an automation ID to a UI widget, in range **0 ... 2047**. Can only be used in the init callback.

Automation IDs can only be assigned to automatable widgets (`ui_knob`, `ui_slider`, `ui_switch`, `ui_xy` cursors).

When assigning automation IDs to `ui_xy` cursors, use `set_control_par_arr()` command instead of `set_control_par()`.

Key Modifiers

`$CONTROL_PAR_KEY_SHIFT`
Returns **1** when the shift key was pressed (**0** otherwise) while clicking the UI widget. `ui_menu` and `ui_value_edit` are not supported. The basic [Shift] modifier functionality on `ui_slider` and `ui_knob` is preserved.

`$CONTROL_PAR_KEY_ALT`
Returns **1** if the [Alt] (Windows) or [Opt] (macOS) key was pressed (**0** otherwise) while clicking the UI widget. `ui_menu` and `ui_value_edit` are not supported.

`$CONTROL_PAR_KEY_CONTROL`
Returns **1** if the [Ctrl] (Windows) or [Cmd] (macOS) key was pressed (**0** otherwise) while clicking the UI widget. `ui_menu` and `ui_value_edit` are not supported. The basic [Ctrl]/[Cmd] modifier functionality on `ui_slider` and `ui_knob` is preserved.

Specific

Tables

`$NI_CONTROL_PAR_IDX`

Returns the index of the ui_table column that triggered the on ui_control callback.

Tables and Waveforms

`$CONTROL_PAR_BAR_COLOR`

Sets or returns the color of the step bar in ui_table and ui_value_edit.

Colors are set using a hex value in the following format:

`0ff0000h { red }`

The **0** at the start is just to let Kontakt know the value is a number. The **h** or **H** at the end is to indicate that it is a hexadecimal value.

`$CONTROL_PAR_ZERO_LINE_COLOR`

Sets or returns the color of the middle line in ui_table.

Menus

`$CONTROL_PAR_NUM_ITEMS`

Returns the number of menu entries in a specific ui_menu.

Only works with `get_control_par()`.

`$CONTROL_PAR_SELECTED_ITEM_IDX`

Returns the index of the currently selected menu entry.

Only works with `get_control_par()`.

Mouse Area

`$CONTROL_PAR_DND_ACCEPT_AUDIO`
`$CONTROL_PAR_DND_ACCEPT_MIDI`
`$CONTROL_PAR_DND_ACCEPT_ARRAY`

Enables ui_mouse_area to accept audio, MIDI or NKA array files.

These can be set to one of the following values:

`$NI_DND_ACCEPT_NONE`

`$NI_DND_ACCEPT_ONE`

`$NI_DND_ACCEPT_MULTIPLE`

$CONTROL_PAR_RECEIVE_DRAG_EVENTS

Configures whether on ui_control callback of ui_mouse_area gets triggered just for the drop event (when set to **0**) or also for drag events (when set to **1**).

The UI callback has two built-in variables:

$NI_MOUSE_EVENT_TYPE	Specifies the event type that triggered the callback and can have one of the following values: $NI_MOUSE_EVENT_TYPE_DND_DROP $NI_MOUSE_EVENT_TYPE_DND_DRAG
$NI_MOUSE_OVER_CONTROL	**1**: The mouse has entered ui_mouse_area on a drag event **0**: The mouse has left ui_mouse_area on a drag event

Example

```
on ui_control ($aMouseArea)
    if ($NI_MOUSE_EVENT_TYPE = $NI_MOUSE_EVENT_TYPE_DROP)
        message(num_elements(!NI_DND_ITEMS_AUDIO))
    end if

    if ($NI_MOUSE_EVENT_TYPE = $NI_MOUSE_EVENT_TYPE_DRAG)
        message(num_elements(!NI_DND_ITEMS_AUDIO))
        message($MOUSE_OVER_CONTROL)
    end if
end on
```

Labels

$CONTROL_PAR_DND_BEHAVIOUR

Sets or returns the drag and drop behavior for ui_label. Using a value of **1** sets the label as a drag and drop area, allowing the user to export the MIDI object currently held in memory by a simple drag and drop action. For more information on MIDI handling in KSP, refer to MIDI Object Commands.

$CONTROL_PAR_MIDI_EXPORT_AREA_IDX

Assigns one of 512 available MIDI object export areas to be drag and drop exported via a particular ui_label. For more information on MIDI handling in KSP, refer to MIDI Object Commands.

Value Edit

$CONTROL_PAR_SHOW_ARROWS

Hides the arrows of ui_value_edit.

0: Arrows are hidden

1: Arrows are shown

Level Meters

$CONTROL_PAR_BG_COLOR

Sets or returns the background color of ui_level_meter.

$CONTROL_PAR_OFF_COLOR

Sets the second background color of ui_level_meter.

$CONTROL_PAR_ON_COLOR

Sets the main level meter color of ui_level_meter.

$CONTROL_PAR_OVERLOAD_COLOR

Sets the color of ui_level_meter overload section.

$CONTROL_PAR_PEAK_COLOR

Sets the color of the little bar showing the current peak level.

$CONTROL_PAR_VERTICAL

Aligns ui_level_meter vertically (**1**) or horizontally (**0**, default).

$CONTROL_PAR_RANGE_MIN
$CONTROL_PAR_RANGE_MAX

Sets the minimum and maximum display range of ui_level_meter, with default range **0 ... 1000000**.

If the minimum values is smaller than the maximum value, the display is inverted.

File Selector

$CONTROL_PAR_BASEPATH

Sets or returns the basepath of ui_file_selector. This control parameter can be used in any callback.

Be careful with the number of subfolders in the basepath, as it might take too long to scan the filesystem.

$CONTROL_PAR_COLUMN_WIDTH

Sets or returns the width of ui_file_selector columns. This control par can only be used in on init callback.

$CONTROL_PAR_FILEPATH

Sets or returns the actual path (full path of the file) currently selected in ui_file_selector. The file path must be a subpath of the instrument's basepath. This control parameter is useful for recalling the prior state of the file selector upon loading the instrument. Can only be used in on init callback.

$CONTROL_PAR_FILE_TYPE

Sets or returns the file type for ui_file_selector. Can only be used in on init callback.
The following file types are available:
`$NI_FILE_TYPE_MIDI`
`$NI_FILE_TYPE_AUDIO`
`$NI_FILE_TYPE_ARRAY`

Instrument Icon and Wallpaper

$INST_ICON_ID

The ID of the instrument icon.
It's possible to hide the instrument icon:
`set_control_par($INST_ICON_ID, $CONTROL_PAR_HIDE, $HIDE_WHOLE_CONTROL)`
It's also possible to load a different picture file for the instrument icon:
`set_control_par_str($INST_ICON_ID, $CONTROL_PAR_PICTURE, <file-name>)`

$INST_WALLPAPER_ID
The ID of the instrument wallpaper. It is used in a similar way as `$INST_ICON_ID`: `set_control_par_str($INST_WALLPAPER_ID, $CONTROL_PAR_PICTURE, <file_name>)` This command can only be used in on init callback. Note that a wallpaper set via the script replaces the one set in Instrument Options dialog, and it will not be checked in the Content Missing dialog when loading the wallpaper from the resource container. This command only supports wallpapers that are located within the resource container. If you use it in different script slots, then the last script slot in which wallpaper was set will be the one that is loaded.

Waveform

Waveform Flag Constants	
To be used with `attach_zone()`. You can combine flag constants using the bitwise `.or.`.	
`$UI_WAVEFORM_USE_SLICES`	Display the zone's slice markers.
`$UI_WAVEFORM_USE_TABLE`	Display a per-slice table. **Note:** this only works if the slice markers are also active.
`$UI_WAVEFORM_TABLE_IS_BIPOLAR`	Make the table bipolar.
`$UI_WAVEFORM_USE_MIDI_DRAG`	Display a MIDI drag and drop icon. **Note:** this only works if the slice markers are also active.

Waveform Property Constants	
To be used with get_ui_wf_property() and set_ui_wf_property().	
`$UI_WF_PROP_PLAY_CURSOR`	Sets or returns the play cursor position, in microseconds.

Waveform Property Constants	
`$UI_WF_PROP_FLAGS`	Used to set new flag constants after the `attach_zone()` command is used.
`$UI_WF_PROP_TABLE_VAL`	Sets or returns the value of the indexed slice's table.
`$UI_WF_PROP_TABLE_IDX_HIGHLIGHT`	Highlights the indexed slice within the ui_waveform widget.
`$UI_WF_PROP_MIDI_DRAG_START_NOTE`	Defines the start note for the MIDI drag and drop function.

`$CONTROL_PAR_WF_VIS_MODE`
Changes the way the waveform is drawn. Valid values: `$NI_WF_VIS_MODE_1` (Default) `$NI_WF_VIS_MODE_2` (X-ray) `$NI_WF_VIS_MODE_3` (X-ray filled)

`$CONTROL_PAR_BG_COLOR`
Sets or returns the background color of ui_waveform.

`$CONTROL_PAR_WAVE_COLOR`
Sets or returns the color of the waveform drawn in ui_waveform.

`$CONTROL_PAR_WAVE_CURSOR_COLOR`
Sets or returns the color of the playback cursor in ui_waveform.

`$CONTROL_PAR_SLICEMARKERS_COLOR`
Sets or returns the color of the slice markers in ui_waveform.

`$CONTROL_PAR_BG_ALPHA`
Sets or returns the alpha channel (opacity) of the background of ui_waveform. Range: 0 (fully transparent) to 255 (fully opaque).

Wavetable

`$CONTROL_PAR_WT_ZONE`
Attaches a zone to ui_wavetable, taking the zone ID as the argument.

`$CONTROL_PAR_WT_VIS_MODE`
Sets or returns the visualization mode of ui_wavetable. Can be set to the following values: `$NI_WT_VIS_2D` (2D, oscilloscope-style visualization, only showing the current wavetable position) `$NI_WT_VIS_3D` (3D visualization displaying the whole wavetable as well as the the current position)

$CONTROL_PAR_PARALLAX_X
Sets or returns the X-axis parallax of ui_wavetable (only applicable to 3D mode). Range: **-1000000 ... 1000000**

$CONTROL_PAR_PARALLAX_Y
Sets or returns the Y-axis parallax of ui_wavetable (only applicable to 3D mode). Range: **-1000000 ... 1000000**

$CONTROL_PAR_WAVE_COLOR
Sets or returns the color of the waveform drawn in 2D visualization mode, or the current waveform drawn in 3D visualization mode of ui_wavetable.

$CONTROL_PAR_WAVE_ALPHA
Sets or returns the alpha channel (opacity) of the waveform drawn in 2D visualization mode, or the current waveform drawn in 3D visualization mode of ui_wavetable. Range: **0** (fully transparent) to **255** (fully opaque).

$CONTROL_PAR_WAVETABLE_COLOR
Sets or returns the color of the background waveforms in 3D visualization mode of ui_wavetable.

$CONTROL_PAR_WAVETABLE_ALPHA
Sets or returns the alpha channel (opacity) of the background waveforms in 3D visualization mode of ui_wavetable. Range: **0** (fully transparent) to **255** (fully opaque).

$CONTROL_PAR_BG_COLOR
Sets or returns the background color of ui_wavetable.

$CONTROL_PAR_BG_ALPHA
Sets or returns the alpha channel (opacity) of ui_wavetable. Range: **0** (fully transparent) to **255** (fully opaque).

Additional Color and Alpha Parameters	
To be paired with the above control parameters in order to create gradient effects. If not explicitly set, they inherit the value of their match from above, resulting in no gradient.	
$CONTROL_PAR_WAVE_END_COLOR	Sets or returns the color for the end of the gradient applied to the waveform (2D) or current waveform (3D).
$CONTROL_PAR_WAVE_END_ALPHA	Sets or returns the alpha channel (opacity) for the end of the gradient applied to the waveform (2D) or current waveform (3D).
$CONTROL_PAR_WAVETABLE_END_COLOR	Sets or returns the color for the end of the gradient applied to the background waveforms (3D).

Additional Color and Alpha Parameters	
`$CONTROL_PAR_WAVETABLE_END_ALPHA`	Sets or returns the alpha channel (opacity) for the end of the gradient applied to the background waveforms (3D).

Slider

`$CONTROL_PAR_MOUSE_BEHAVIOUR`
A value from **-5000** to **5000**, setting the move direction of ui_slider and its sensitivity. Settings are relative to the size of the slider picture. Negative values give a vertical slider behavior, positive values give a horizontal behavior.

XY Pad

`$CONTROL_PAR_MOUSE_BEHAVIOUR_X`
Mouse behavior, i.e. the drag scale, of the X axis of all ui_xy cursors.

`$CONTROL_PAR_MOUSE_BEHAVIOUR_Y`
Mouse behavior, i.e. the drag scale, of the Y axis of all ui_xy cursors.

`$CONTROL_PAR_MOUSE_MODE`
Sets the way ui_xy responds to mouse clicks and drags. **0:** Clicks anywhere other than on a cursor are ignored. Clicking on a cursor and dragging, sets new values respecting the usual `$CONTROL_PAR_MOUSE_BEHAVIOUR` settings. **1:** Clicks anywhere on the XY pad are registered but don't change the values. Clicking anywhere and dragging, sets new values; the cursor moves parallel to the mouse cursor with distances scaled based on the `$CONTROL_PAR_MOUSE_BEHAVIOUR` settings. **2:** Clicks anywhere on the XY pad are registered and immediately change the values, with the cursor immediately matching the mouse cursor. Clicking anywhere and dragging has a similar effect; the `$CONTROL_PAR_MOUSE_BEHAVIOUR` settings are ignored; cursor always follows mouse cursor one-to-one.

`$CONTROL_PAR_ACTIVE_INDEX`
Sets and gets the index of the active ui_xy cursor. Only relevant in multi-cursor setups. The `$CONTROL_PAR_MOUSE_MODE` setting will influence how this parameter behaves: **0** and **1**: The active cursor can only be changed manually by setting this control parameter. Inactive cursors don't receive any clicks. **2**: Active cursor is set automatically based on the last clicked cursor. Setting it manually within on ui_control callback of ui_xy can result in unexpected results, but using it in other callbacks is fully encouraged and makes sense in many scenarios. The returned value is **-1** when not clicking on any cursor. The index can only ever be an even number (with the exception of the **-1** value) that matches the index of the X axis of the cursor in the main array representing the XY control, e.g. the first cursor has an index of **0**, the second one has an index of **2**, etc.

$CONTROL_PAR_CURSOR_PICTURE

Sets the cursor image. Each cursor can have its own image set using the `set_control_par_str_arr()` command.

Using `$CONTROL_PAR_PICTURE` on ui_xy UI ID itself will set the background image of the control.

The cursor images can have up to **6** frames, corresponding to the following states. Frame selection is automatic, exactly like with `ui_button` and ui_switch.

1: Inactive

2: Active

3: Inactive pressed

4: Active pressed

5: Inactive mouse over

6: Active mouse over

$HIDE_PART_CURSOR

When used with set_control_par_arr(), this can be used to hide specific ui_xy cursors. Below is a simple example:

```
if ($hide = 1)
    set_control_par_arr($id, $CONTROL_PAR_HIDE, $HIDE_PART_CURSOR, $index)
else
    set_control_par_arr($id, $CONTROL_PAR_HIDE, $HIDE_PART_NOTHING, $index)
end if
```

The index should be an even number that matches the index of the X axis of the cursor in the main array representing the XY control, so the first cursor has an index of **0**, the second has an index of **2**, and so on.

$NI_CONTROL_PAR_IDX

Returns the index of the cursor that triggered the on ui_control callback of a ui_xy widget. Note that indices are always even numbers starting from **0**, so the first cursor has an index of **0**, the second has an index of **2**, and so on.

$NI_MOUSE_EVENT_TYPE

Returns the type of mouse event that triggered the on ui_controlcallback of a ui_xy widget. Can only be used within the on ui_control callback.

The following mouse event types are available:

`$NI_MOUSE_EVENT_TYPE_LEFT_BUTTON_DOWN`

`$NI_MOUSE_EVENT_TYPE_LEFT_BUTTON_UP`

`$NI_MOUSE_EVENT_TYPE_DRAG`

23. Engine Parameters

Instrument, Source and Amplifier Module

$ENGINE_PAR_VOLUME
Instrument, group or bus volume.

$ENGINE_PAR_PAN
Instrument, group or bus panorama.

$ENGINE_PAR_PHASE_INVERT
Group or instrument bus phase invert.

$ENGINE_PAR_LR_SWAP
Group or instrument bus left-right channel swap.

$ENGINE_PAR_TUNE
Instrument or group tuning.

Source Module

`$ENGINE_PAR_SOURCE_MODE` (only works with get_engine_par()!)

```
    $NI_SOURCE_MODE_SAMPLER
    $NI_SOURCE_MODE_DFD
    $NI_SOURCE_MODE_TONE_MACHINE
    $NI_SOURCE_MODE_TIME_MACHINE_1
    $NI_SOURCE_MODE_TIME_MACHINE_2
    $NI_SOURCE_MODE_TIME_MACHINE_PRO
    $NI_SOURCE_MODE_BEAT_MACHINE
    $NI_SOURCE_MODE_MP60_MACHINE
    $NI_SOURCE_MODE_S1200_MACHINE
    $NI_SOURCE_MODE_WAVETABLE
$ENGINE_PAR_TRACKING
$ENGINE_PAR_HQI_MODE
    $NI_HQI_MODE_STANDARD
    $NI_HQI_MODE_HIGH
    $NI_HQI_MODE_PERFECT
$ENGINE_PAR_SMOOTH
$ENGINE_PAR_FORMANT
$ENGINE_PAR_SPEED
$ENGINE_PAR_GRAIN_LENGTH
$ENGINE_PAR_SLICE_ATTACK
$ENGINE_PAR_SLICE_RELEASE
$ENGINE_PAR_TRANSIENT_SIZE
$ENGINE_PAR_ENVELOPE_ORDER
$ENGINE_PAR_FORMANT_SHIFT
$ENGINE_PAR_SPEED_UNIT
$ENGINE_PAR_TM_LEGATO
$ENGINE_PAR_TMPRO_KEEP_FORMANTS
$ENGINE_PAR_WT_POSITION
$ENGINE_PAR_WT_FORM
$ENGINE_PAR_WT_PHASE
$ENGINE_PAR_WT_PHASE_RAND
$ENGINE_PAR_WT_QUALITY
    $NI_WT_QUALITY_LOFI
    $NI_WT_QUALITY_MEDIUM
    $NI_WT_QUALITY_HIGH
    $NI_WT_QUALITY_BEST
$ENGINE_PAR_WT_FORM_MODE
    $NI_WT_FORM_LINEAR
    $NI_WT_FORM_SYNC1
    $NI_WT_FORM_SYNC2
    $NI_WT_FORM_SYNC3
    $NI_WT_FORM_SYNC4
    $NI_WT_FORM_SYNC5
```

Source Module
`$NI_WT_FORM_SYNC6`
`$NI_WT_FORM_BENDP`
`$NI_WT_FORM_BENDM`
`$NI_WT_FORM_BENDMP`
`$NI_WT_FORM_BEND2P`
`$NI_WT_FORM_BEND2M`
`$NI_WT_FORM_BEND2MP`
`$NI_WT_FORM_EXP`
`$NI_WT_FORM_LOG`
`$NI_WT_FORM_LOGEXP`
`$NI_WT_FORM_PWM`
`$NI_WT_FORM_ASYMP`
`$NI_WT_FORM_ASYMM`
`$NI_WT_FORM_ASYMMP`
`$NI_WT_FORM_ASYM2P`
`$NI_WT_FORM_ASYM2M`
`$NI_WT_FORM_ASYM2MP`
`$NI_WT_FORM_2BLINDS`
`$NI_WT_FORM_4BLINDS`
`$NI_WT_FORM_6BLINDS`
`$NI_WT_FORM_8BLINDS`
`$NI_WT_FORM_FLIP`
`$NI_WT_FORM_FOLD`
`$NI_WT_FORM_MIRROR`
`$NI_WT_FORM_SATURATE`
`$NI_WT_FORM_SEESAW`
`$NI_WT_FORM_QUANTIZE`
`$NI_WT_FORM_WRAP`
`$ENGINE_PAR_WT_INHARMONIC_MODE`
`$ENGINE_PAR_WT_INHARMONIC`
`$ENGINE_PAR_S1200_FILTER_MODE`
`$NI_S1200_FILTER_NONE`
`$NI_S1200_FILTER_HIGH`
`$NI_S1200_FILTER_HIGH_MID`
`$NI_S1200_FILTER_LOW_MID`
`$NI_S1200_FILTER_LOW`

`$ENGINE_PAR_POST_FX_SLOT`
Sets the number of Post Amp group effects.

`$ENGINE_PAR_OUTPUT_CHANNEL`
Designates the output for the group or bus. **0 ... 63:** Routes to one of Kontakt’s available outputs. This bypasses the instrument insert effects. **-1:** Routes to the instrument output (default). **-2:** Routes to the instrument output with the instrument insert effects bypassed. `$NI_BUS_OFFSET + 0 ... 15`: Routes to one of the buses. Buses cannot be routed to other buses.

Filter and EQ

$ENGINE_PAR_CUTOFF

Cutoff frequency of all filters.

$ENGINE_PAR_RESONANCE

Resonance of all filters.

$ENGINE_PAR_EFFECT_BYPASS

Bypass button of all filters and EQs.

$ENGINE_PAR_GAIN

Gain control for the Ladder and Daft filter types.

$ENGINE_PAR_FILTER_LADDER_HQ

High Quality mode for the Ladder filter types.

$ENGINE_PAR_BANDWIDTH

Bandwidth control, found on the following filter types: SV Par. LP/HP, SV Par. BP/BP, SV Ser. LP/HP.

3x2 Versatile

```
$ENGINE_PAR_FILTER_SHIFTB
$ENGINE_PAR_FILTER_SHIFTC
$ENGINE_PAR_FILTER_RESB
$ENGINE_PAR_FILTER_RESC
$ENGINE_PAR_FILTER_TYPEA
$ENGINE_PAR_FILTER_TYPEB
$ENGINE_PAR_FILTER_TYPEC
$ENGINE_PAR_FILTER_BYPA
$ENGINE_PAR_FILTER_BYPB
$ENGINE_PAR_FILTER_BYPC
$ENGINE_PAR_FILTER_GAIN
```

Formant Filters

```
$ENGINE_PAR_FORMANT_TALK
$ENGINE_PAR_FORMANT_SHARP
$ENGINE_PAR_FORMANT_SIZE
```

Simple Filter

```
$ENGINE_PAR_LP_CUTOFF
$ENGINE_PAR_HP_CUTOFF
```

EQ
$ENGINE_PAR_FREQ1
$ENGINE_PAR_BW1
$ENGINE_PAR_GAIN1
$ENGINE_PAR_FREQ2
$ENGINE_PAR_BW2
$ENGINE_PAR_GAIN2
$ENGINE_PAR_FREQ3
$ENGINE_PAR_BW3
$ENGINE_PAR_GAIN3

Solid G-EQ
$ENGINE_PAR_SEQ_HP
$ENGINE_PAR_SEQ_HP_FREQ
$ENGINE_PAR_SEQ_LF_GAIN
$ENGINE_PAR_SEQ_LF_FREQ
$ENGINE_PAR_SEQ_LF_BELL
$ENGINE_PAR_SEQ_LMF_GAIN
$ENGINE_PAR_SEQ_LMF_FREQ
$ENGINE_PAR_SEQ_LMF_Q
$ENGINE_PAR_SEQ_HMF_GAIN
$ENGINE_PAR_SEQ_HMF_FREQ
$ENGINE_PAR_SEQ_HMF_Q
$ENGINE_PAR_SEQ_HF_GAIN
$ENGINE_PAR_SEQ_HF_FREQ
$ENGINE_PAR_SEQ_HF_BELL
$ENGINE_PAR_SEQ_LP
$ENGINE_PAR_SEQ_LP_FREQ

Insert Effects

`$ENGINE_PAR_EFFECT_BYPASS`
Bypass button of all insert effects.

`$ENGINE_PAR_INSERT_EFFECT_OUTPUT_GAIN`
Output gain of all insert effects. **Note:** This engine parameter will affect the output gain of all **filters and EQs**, even if they don't have the Output parameter visible on their module panels!

Dynamics

Compressor

```
$ENGINE_PAR_THRESHOLD
$ENGINE_PAR_RATIO
$ENGINE_PAR_COMP_ATTACK
$ENGINE_PAR_COMP_DECAY
$ENGINE_PAR_COMP_LINK
$ENGINE_PAR_COMP_TYPE
    $NI_COMP_TYPE_CLASSIC
    $NI_COMP_TYPE_ENHANCED
    $NI_COMP_TYPE_PRO
```

Feedback Compressor

```
$ENGINE_PAR_FCOMP_INPUT
$ENGINE_PAR_FCOMP_RATIO
$ENGINE_PAR_FCOMP_ATTACK
$ENGINE_PAR_FCOMP_RELEASE
$ENGINE_PAR_FCOMP_MAKEUP
$ENGINE_PAR_FCOMP_MIX
$ENGINE_PAR_FCOMP_HQ_MODE
$ENGINE_PAR_FCOMP_LINK
```

Limiter

```
$ENGINE_PAR_LIM_IN_GAIN
$ENGINE_PAR_LIM_RELEASE
```

Solid Bus Comp
$ENGINE_PAR_SCOMP_THRESHOLD
$ENGINE_PAR_SCOMP_RATIO
$ENGINE_PAR_SCOMP_ATTACK
$ENGINE_PAR_SCOMP_RELEASE
$ENGINE_PAR_SCOMP_MAKEUP
$ENGINE_PAR_SCOMP_MIX
$ENGINE_PAR_SCOMP_LINK

Supercharger GT
$ENGINE_PAR_SUPERGT_TRIM
$ENGINE_PAR_SUPERGT_HPF_MODE
$NI_SUPERGT_HPF_MODE_OFF
$NI_SUPERGT_HPF_MODE_100
$NI_SUPERGT_HPF_MODE_300
$ENGINE_PAR_SUPERGT_SATURATION
$ENGINE_PAR_SUPERGT_SAT_MODE
$NI_SUPERGT_SAT_MODE_MILD
$NI_SUPERGT_SAT_MODE_MODERATE
$NI_SUPERGT_SAT_MODE_HOT
$ENGINE_PAR_SUPERGT_COMPRESS
$ENGINE_PAR_SUPERGT_ATTACK
$ENGINE_PAR_SUPERGT_RELEASE
$ENGINE_PAR_SUPERGT_CHARACTER
$ENGINE_PAR_SUPERGT_CHAR_MODE
$NI_SUPERGT_CHAR_MODE_FAT
$NI_SUPERGT_CHAR_MODE_WARM
$NI_SUPERGT_CHAR_MODE_BRIGHT
$ENGINE_PAR_SUPERGT_MIX
$ENGINE_PAR_SUPERGT_CHANNEL_LINK_MODE
$NI_SUPERGT_CHANNEL_LINK_MODE_STEREO
$NI_SUPERGT_CHANNEL_LINK_MODE_DUAL_MONO
$NI_SUPERGT_CHANNEL_LINK_MODE_MS

Transient Master
$ENGINE_PAR_TR_INPUT
$ENGINE_PAR_TR_ATTACK
$ENGINE_PAR_TR_SUSTAIN
$ENGINE_PAR_TR_SMOOTH

Transparent Limiter
$ENGINE_PAR_TRANSLIM_THRESHOLD
$ENGINE_PAR_TRANSLIM_RELEASE
$ENGINE_PAR_TRANSLIM_CEILING

Amps

ACBox
$ENGINE_PAR_AC_NORMALVOLUME
$ENGINE_PAR_AC_BRILLIANTVOLUME
$ENGINE_PAR_AC_BASS
$ENGINE_PAR_AC_TREBLE
$ENGINE_PAR_AC_TONECUT
$ENGINE_PAR_AC_TREMOLOSPEED
$ENGINE_PAR_AC_TREMOLODEPTH
$ENGINE_PAR_AC_MONO

Bass Invader
$ENGINE_PAR_BASSINVADER_VOLUME
$ENGINE_PAR_BASSINVADER_TREBLE
$ENGINE_PAR_BASSINVADER_HI_MID
$ENGINE_PAR_BASSINVADER_LO_MID
$ENGINE_PAR_BASSINVADER_BASS
$ENGINE_PAR_BASSINVADER_BOOST
$ENGINE_PAR_BASSINVADER_MASTER
$ENGINE_PAR_BASSINVADER_LO_CUT
$ENGINE_PAR_BASSINVADER_MID_CONTOUR
$ENGINE_PAR_BASSINVADER_HI_BOOST

Bass Pro
$ENGINE_PAR_BASSPRO_GAIN
$ENGINE_PAR_BASSPRO_BASS
$ENGINE_PAR_BASSPRO_MID
$ENGINE_PAR_BASSPRO_MIDFREQ
$ENGINE_PAR_BASSPRO_TREBLE
$ENGINE_PAR_BASSPRO_DRIVE
$ENGINE_PAR_BASSPRO_MASTER
$ENGINE_PAR_BASSPRO_GEQ_40
$ENGINE_PAR_BASSPRO_GEQ_90
$ENGINE_PAR_BASSPRO_GEQ_180
$ENGINE_PAR_BASSPRO_GEQ_300
$ENGINE_PAR_BASSPRO_GEQ_500
$ENGINE_PAR_BASSPRO_GEQ_1K
$ENGINE_PAR_BASSPRO_GEQ_2K
$ENGINE_PAR_BASSPRO_GEQ_4K
$ENGINE_PAR_BASSPRO_GEQ_10K
$ENGINE_PAR_BASSPRO_GEQ_VOLUME
$ENGINE_PAR_BASSPRO_ULTRALO
$ENGINE_PAR_BASSPRO_ULTRAHI
$ENGINE_PAR_BASSPRO_BRIGHT
$ENGINE_PAR_BASSPRO_GEQ
$ENGINE_PAR_BASSPRO_MONO

Cabinet
$ENGINE_PAR_CB_SIZE
$ENGINE_PAR_CB_AIR
$ENGINE_PAR_CB_TREBLE
$ENGINE_PAR_CB_BASS
$ENGINE_PAR_CABINET_TYPE

EP Preamps

```
$ENGINE_PAR_EPP_DRIVE_MODE
    $NI_EPP_DRIVE_MODE_BYPASS
    $NI_EPP_DRIVE_MODE_DE_TUBE
    $NI_EPP_DRIVE_MODE_US_TUBE
    $NI_EPP_DRIVE_MODE_TRANSISTOR
    $NI_EPP_DRIVE_MODE_TAPE
$ENGINE_PAR_EPP_DRIVE
$ENGINE_PAR_EPP_EQ_MODE
    $NI_EPP_EQ_MODE_BYPASS
    $NI_EPP_EQ_MODE_PASSIVE
    $NI_EPP_EQ_MODE_70S
    $NI_EPP_EQ_MODE_80S
    $NI_EPP_EQ_MODE_E_GRAND
$ENGINE_PAR_EPP_EQ_BASS
$ENGINE_PAR_EPP_EQ_MID
$ENGINE_PAR_EPP_EQ_TREBLE
$ENGINE_PAR_EPP_PASSIVE_BASS
$ENGINE_PAR_EPP_TREMOLO_MODE
    $NI_EPP_TREMOLO_MODE_BYPASS
    $NI_EPP_TREMOLO_MODE_70S
    $NI_EPP_TREMOLO_MODE_80S
    $NI_EPP_TREMOLO_MODE_SYNTH
    $NI_EPP_TREMOLO_MODE_GUITAR
    $NI_EPP_TREMOLO_MODE_E_GRAND
$ENGINE_PAR_EPP_TREMOLO_WAVE
    $NI_EPP_TREMOLO_WAVE_SINE
    $NI_EPP_TREMOLO_WAVE_TRIANGLE
    $NI_EPP_TREMOLO_WAVE_SQUARE
    $NI_EPP_TREMOLO_WAVE_SAW_DOWN
    $NI_EPP_TREMOLO_WAVE_SAW_UP
$ENGINE_PAR_EPP_TREMOLO_RATE
$ENGINE_PAR_EPP_TREMOLO_RATE_UNIT
$ENGINE_PAR_EPP_TREMOLO_AMOUNT
$ENGINE_PAR_EPP_TREMOLO_WIDTH
$ENGINE_PAR_EPP_MONO
```

HotSolo
$ENGINE_PAR_HS_PRENORMAL
$ENGINE_PAR_HS_PREOVERDRIVE
$ENGINE_PAR_HS_BASS
$ENGINE_PAR_HS_MID
$ENGINE_PAR_HS_TREBLE
$ENGINE_PAR_HS_MASTER
$ENGINE_PAR_HS_PRESENCE
$ENGINE_PAR_HS_DEPTH
$ENGINE_PAR_HS_OVERDRIVE
$ENGINE_PAR_HS_MONO

Jump
$ENGINE_PAR_JMP_PREAMP
$ENGINE_PAR_JMP_BASS
$ENGINE_PAR_JMP_MID
$ENGINE_PAR_JMP_TREBLE
$ENGINE_PAR_JMP_MASTER
$ENGINE_PAR_JMP_PRESENCE
$ENGINE_PAR_JMP_HIGAIN
$ENGINE_PAR_JMP_MONO

Twang
$ENGINE_PAR_TW_VOLUME
$ENGINE_PAR_TW_TREBLE
$ENGINE_PAR_TW_MID
$ENGINE_PAR_TW_BASS
$ENGINE_PAR_TW_BRIGHT
$ENGINE_PAR_TW_MONO

Van51
$ENGINE_PAR_V5_PREGAINRHYTHM
$ENGINE_PAR_V5_PREGAINLEAD
$ENGINE_PAR_V5_BASS
$ENGINE_PAR_V5_MID
$ENGINE_PAR_V5_TREBLE
$ENGINE_PAR_V5_POSTGAIN
$ENGINE_PAR_V5_RESONANCE
$ENGINE_PAR_V5_PRESENCE
$ENGINE_PAR_V5_LEADCHANNEL
$ENGINE_PAR_V5_HIGAIN
$ENGINE_PAR_V5_BRIGHT
$ENGINE_PAR_V5_CRUNCH
$ENGINE_PAR_V5_MONO

Stomps

Big Fuzz
$ENGINE_PAR_BIGFUZZ_SUSTAIN
$ENGINE_PAR_BIGFUZZ_TONE
$ENGINE_PAR_BIGFUZZ_BASS
$ENGINE_PAR_BIGFUZZ_TREBLE
$ENGINE_PAR_BIGFUZZ_MONO

Cat
$ENGINE_PAR_CT_VOLUME
$ENGINE_PAR_CT_DISTORTION
$ENGINE_PAR_CT_FILTER
$ENGINE_PAR_CT_BASS
$ENGINE_PAR_CT_BALLS
$ENGINE_PAR_CT_TREBLE
$ENGINE_PAR_CT_TONE
$ENGINE_PAR_CT_MONO

Cry Wah
$ENGINE_PAR_CW_MONO
$ENGINE_PAR_CW_PEDAL

Dirt

```
$ENGINE_PAR_DIRT_DRIVEA
$ENGINE_PAR_DIRT_AMOUNTA
$ENGINE_PAR_DIRT_BIASA
$ENGINE_PAR_DIRT_TILTA
$ENGINE_PAR_DIRT_MODEA
    $NI_DIRT_MODE_I
    $NI_DIRT_MODE_II
    $NI_DIRT_MODE_III
$ENGINE_PAR_DIRT_SAFETYA
$ENGINE_PAR_DIRT_DRIVEB
$ENGINE_PAR_DIRT_AMOUNTB
$ENGINE_PAR_DIRT_BIASB
$ENGINE_PAR_DIRT_TILTB
$ENGINE_PAR_DIRT_MODEB
    $NI_DIRT_MODE_I
    $NI_DIRT_MODE_II
    $NI_DIRT_MODE_III
$ENGINE_PAR_DIRT_SAFETYB
$ENGINE_PAR_DIRT_ROUTING
    $NI_DIRT_ROUTING_ATOB
    $NI_DIRT_ROUTING_BTOA
    $NI_DIRT_ROUTING_PARALLEL
$ENGINE_PAR_DIRT_MIX
$ENGINE_PAR_DIRT_BLEND
```

Distortion

```
$ENGINE_PAR_DRIVE
$ENGINE_PAR_DAMPING
$ENGINE_PAR_DISTORTION_TYPE
    $NI_DISTORTION_TYPE_TUBE
    $NI_DISTORTION_TYPE_TRANS
```

DStortion

```
$ENGINE_PAR_DS_VOLUME
$ENGINE_PAR_DS_TONE
$ENGINE_PAR_DS_DRIVE
$ENGINE_PAR_DS_BASS
$ENGINE_PAR_DS_MID
$ENGINE_PAR_DS_TREBLE
$ENGINE_PAR_DS_MONO
```

Fuzz
$ENGINE_PAR_FUZZ_AMOUNT
$ENGINE_PAR_FUZZ_BASS
$ENGINE_PAR_FUZZ_TREBLE
$ENGINE_PAR_FUZZ_MONO

Saturation
$ENGINE_PAR_SHAPE
$ENGINE_PAR_SHAPE_TYPE
$NI_SHAPE_TYPE_CLASSIC
$NI_SHAPE_TYPE_ENHANCED
$NI_SHAPE_TYPE_DRUMS

Skreamer
$ENGINE_PAR_SK_TONE
$ENGINE_PAR_SK_DRIVE
$ENGINE_PAR_SK_BASS
$ENGINE_PAR_SK_BRIGHT
$ENGINE_PAR_SK_MIX

Lo-Fi

Bite
$ENGINE_PAR_BITE_FREQUENCY
$ENGINE_PAR_BITE_BITS
$ENGINE_PAR_BITE_JITTER
$ENGINE_PAR_BITE_CRUNCH
$ENGINE_PAR_BITE_DITHER
$ENGINE_PAR_BITE_EXPAND
$ENGINE_PAR_BITE_SATURATE
$ENGINE_PAR_BITE_PREFILTER
$ENGINE_PAR_BITE_POSTFILTER
$ENGINE_PAR_BITE_DC_QUANT
$ENGINE_PAR_BITE_HPF
$NI_BITE_HPF_MODE_5
$NI_BITE_HPF_MODE_100
$NI_BITE_HPF_MODE_200
$ENGINE_PAR_BITE_MIX

Lo-Fi
$ENGINE_PAR_BITS
$ENGINE_PAR_FREQUENCY
$ENGINE_PAR_NOISELEVEL
$ENGINE_PAR_NOISECOLOR

Tape

Tape Saturator
$ENGINE_PAR_TP_GAIN
$ENGINE_PAR_TP_WARMTH
$ENGINE_PAR_TP_HF_ROLLOFF
$ENGINE_PAR_TP_QUALITY

Wow/Flutter
$ENGINE_PAR_WOWFLUTTER_WOW
$ENGINE_PAR_WOWFLUTTER_FLUTTER
$ENGINE_PAR_WOWFLUTTER_SCRAPE
$ENGINE_PAR_WOWFLUTTER_SPEED
$ENGINE_PAR_WOWFLUTTER_STEREO
$ENGINE_PAR_WOWFLUTTER_AGE
$ENGINE_PAR_WOWFLUTTER_SATURATION
$ENGINE_PAR_WOWFLUTTER_MIX
$ENGINE_PAR_WOWFLUTTER_GATE

Modulation

Choral
$ENGINE_PAR_CHORAL_RATE
$ENGINE_PAR_CHORAL_MODE
$NI_CHORAL_MODE_SYNTH
$NI_CHORAL_MODE_ENSEMBLE
$NI_CHORAL_MODE_DIMENSION
$NI_CHORAL_MODE_UNIVERSAL
$ENGINE_PAR_CHORAL_AMOUNT
$ENGINE_PAR_CHORAL_VOICES
$ENGINE_PAR_CHORAL_DELAY
$ENGINE_PAR_CHORAL_WIDTH
$ENGINE_PAR_CHORAL_FEEDBACK
$ENGINE_PAR_CHORAL_SCATTER
$ENGINE_PAR_CHORAL_INVERT_PHASE
$ENGINE_PAR_CHORAL_MIX

Flair
`$ENGINE_PAR_FLAIR_MODE`
`$NI_FLAIR_MODE_STANDARD`
`$NI_FLAIR_MODE_THRU_ZERO`
`$NI_FLAIR_MODE_SCAN`
`$ENGINE_PAR_FLAIR_CHORD` (**0 ... 23**)
`$ENGINE_PAR_FLAIR_INVERT_PHASE`
`$ENGINE_PAR_FLAIR_RATE`
`$ENGINE_PAR_FLAIR_RATE_UNIT`
`$ENGINE_PAR_FLAIR_FEEDBACK`
`$ENGINE_PAR_FLAIR_AMOUNT`
`$ENGINE_PAR_FLAIR_WIDTH`
`$ENGINE_PAR_FLAIR_PITCH` (**0 ... 96**)
`$ENGINE_PAR_FLAIR_DAMPING`
`$ENGINE_PAR_FLAIR_VOICES`
`$ENGINE_PAR_FLAIR_DETUNE`
`$ENGINE_PAR_FLAIR_MIX`
`$ENGINE_PAR_FLAIR_OFFSET`
`$ENGINE_PAR_FLAIR_SCANMODE`
`$NI_FLAIR_SCANMODE_TRIANGLE`
`$NI_FLAIR_SCANMODE_SAW_UP`
`$NI_FLAIR_SCANMODE_SAW_DOWN`

Freak
`$ENGINE_PAR_FREAK_MODE`
`$NI_FREAK_MODE_RADIO`
`$NI_FREAK_MODE_OSCILLATOR`
`$NI_FREAK_MODE_SIDECHAIN`
`$ENGINE_PAR_FREAK_TYPE`
`$ENGINE_PAR_FREAK_HARMONICS`
`$ENGINE_PAR_FREAK_FEEDBACK`
`$ENGINE_PAR_FREAK_MIX`
`$ENGINE_PAR_FREAK_CARRIER`
`$ENGINE_PAR_FREAK_TUNING`
`$ENGINE_PAR_FREAK_WIDTH`
`$ENGINE_PAR_FREAK_DEMOD`
`$ENGINE_PAR_FREAK_GATE`
`$ENGINE_PAR_FREAK_FREQUENCY`
`$ENGINE_PAR_FREAK_ANTIFOLD`
`$ENGINE_PAR_FREAK_STEREO`
`$ENGINE_PAR_FREAK_WIDE_RANGE`
`$ENGINE_PAR_FREAK_CONTOUR`
`$ENGINE_PAR_FREAK_RELEASE`
`$ENGINE_PAR_FREAK_BP_FREQ`
`$ENGINE_PAR_FREAK_BP_FILTER`

Phasis
`$ENGINE_PAR_PHASIS_RATE`
`$ENGINE_PAR_PHASIS_RATE_UNIT`
`$ENGINE_PAR_PHASIS_ULTRA`
`$ENGINE_PAR_PHASIS_AMOUNT`
`$ENGINE_PAR_PHASIS_CENTER`
`$ENGINE_PAR_PHASIS_STEREO`
`$ENGINE_PAR_PHASIS_SPREAD`
`$ENGINE_PAR_PHASIS_FEEDBACK`
`$ENGINE_PAR_PHASIS_MOD_MIX`
`$ENGINE_PAR_PHASIS_NOTCHES` (**2 ... 12**)
`$ENGINE_PAR_PHASIS_INVERT_PHASE`
`$ENGINE_PAR_PHASIS_INVERT_MOD_MIX`
`$ENGINE_PAR_PHASIS_MIX`

Ring Modulator
`$ENGINE_PAR_RINGMOD_RING`
`$ENGINE_PAR_RINGMOD_FM`
`$ENGINE_PAR_RINGMOD_FREQUENCY`
`$ENGINE_PAR_RINGMOD_EDGE`
`$ENGINE_PAR_RINGMOD_LFO_RATE`
`$ENGINE_PAR_RINGMOD_LFO_RATE_UNIT`
`$ENGINE_PAR_RINGMOD_LFO_AMOUNT`
`$ENGINE_PAR_RINGMOD_LFO_WAVE`
`$NI_RINGMOD_LFO_WAVE_SINE`
`$NI_RINGMOD_LFO_WAVE_SQUARE`
`$ENGINE_PAR_RINGMOD_FAST_MODE`

Rotator
`$ENGINE_PAR_RT_SPEED`
`$ENGINE_PAR_RT_BALANCE`
`$ENGINE_PAR_RT_ACCEL_HI`
`$ENGINE_PAR_RT_ACCEL_LO`
`$ENGINE_PAR_RT_DISTANCE`
`$ENGINE_PAR_RT_MIX`

Vibrato/Chorus
`$ENGINE_PAR_VC_BLEND`
`$ENGINE_PAR_VC_RATE`
`$ENGINE_PAR_VC_WIDTH`
`$ENGINE_PAR_VC_MIX`
`$ENGINE_PAR_VC_DEPTH`
`$NI_VC_DEPTH_1`
`$NI_VC_DEPTH_2`
`$NI_VC_DEPTH_3`
`$NI_VC_DEPTH_4`
`$NI_VC_DEPTH_5`
`$NI_VC_DEPTH_6`
`$ENGINE_PAR_VC_COLOR`
`$NI_VC_COLOR_TYPE_A`
`$NI_VC_COLOR_TYPE_B`
`$NI_VC_COLOR_TYPE_C`

Spatial

Stereo Modeller
$ENGINE_PAR_STEREO
$ENGINE_PAR_STEREO_PAN
$ENGINE_PAR_STEREO_PSEUDO

Stereo Tune
$ENGINE_PAR_STEREOTUNE_SPLIT
$ENGINE_PAR_STEREOTUNE_DRIFT
$ENGINE_PAR_STEREOTUNE_SPREAD
$ENGINE_PAR_STEREOTUNE_MIX

Surround Panner
$ENGINE_PAR_SP_OFFSET_DISTANCE
$ENGINE_PAR_SP_OFFSET_AZIMUTH
$ENGINE_PAR_SP_OFFSET_X
$ENGINE_PAR_SP_OFFSET_Y
$ENGINE_PAR_SP_LFE_VOLUME
$ENGINE_PAR_SP_SIZE
$ENGINE_PAR_SP_DIVERGENCE

Utilities

AET Filter
$ENGINE_PAR_EXP_FILTER_MORPH
$ENGINE_PAR_EXP_FILTER_AMOUNT

Inverter
$ENGINE_PAR_PHASE_INVERT
$ENGINE_PAR_LR_SWAP

Send Levels
$ENGINE_PAR_SENDLEVEL_0
$ENGINE_PAR_SENDLEVEL_1
$ENGINE_PAR_SENDLEVEL_2
$ENGINE_PAR_SENDLEVEL_3
$ENGINE_PAR_SENDLEVEL_4
$ENGINE_PAR_SENDLEVEL_5
$ENGINE_PAR_SENDLEVEL_6
$ENGINE_PAR_SENDLEVEL_7

Send Effects

`$ENGINE_PAR_SEND_EFFECT_BYPASS`
Bypass button of all send effects.

`$ENGINE_PAR_SEND_EFFECT_DRY_LEVEL`
Dry amount of send effects when used in an **Instrument Bus, Insert or Main FX** chain.

`$ENGINE_PAR_SEND_EFFECT_OUTPUT_GAIN`
When used with send effects, this controls either: • **Wet** amount of send effects when used in an **Instrument Bus, Insert or Main FX** chain • **Return** amount of send effects when used in a **Send FX** chain

Delay

PsycheDelay

`$ENGINE_PAR_PSYDL_TIME`
`$ENGINE_PAR_PSYDL_TIME_UNIT`
`$ENGINE_PAR_PSYDL_FEEDBACK`
`$ENGINE_PAR_PSYDL_CROSS_FEEDBACK`
`$ENGINE_PAR_PSYDL_LR_OFFSET`
`$ENGINE_PAR_PSYDL_PITCH`
`$ENGINE_PAR_PSYDL_DETUNE`
`$ENGINE_PAR_PSYDL_REVERSE`
`$ENGINE_PAR_PSYDL_REVERSE_STEREO`
`$ENGINE_PAR_PSYDL_DETUNE_STEREO`

Replika Delay
$ENGINE_PAR_RDL_TYPE
$NI_REPLIKA_TYPE_MODERN
$NI_REPLIKA_TYPE_TAPE
$NI_REPLIKA_TYPE_VINTAGE
$NI_REPLIKA_TYPE_DIFFUSION
$NI_REPLIKA_TYPE_ANALOGUE
$ENGINE_PAR_RDL_TIME
$ENGINE_PAR_RDL_TIME_UNIT
$ENGINE_PAR_RDL_FEEDBACK
$ENGINE_PAR_RDL_LOWCUT
$ENGINE_PAR_RDL_HIGHCUT
$ENGINE_PAR_RDL_SATURATION
$ENGINE_PAR_RDL_TAPEAGE
$ENGINE_PAR_RDL_FLUTTER
$ENGINE_PAR_RDL_QUALITY
$ENGINE_PAR_RDL_DEPTH
$ENGINE_PAR_RDL_RATE
$ENGINE_PAR_RDL_TYPE
$ENGINE_PAR_RDL_STEREO
$ENGINE_PAR_RDL_NOISE
$ENGINE_PAR_RDL_PINGPONG
$ENGINE_PAR_RDL_AMOUNT
$ENGINE_PAR_RDL_SIZE
$ENGINE_PAR_RDL_DENSE
$ENGINE_PAR_RDL_MODULATION
$ENGINE_PAR_RDL_BBDTYPE

Twin Delay
$ENGINE_PAR_TDL_PREDELAY_L
$ENGINE_PAR_TDL_PREDELAY_L_UNIT
$ENGINE_PAR_TDL_TIME_L
$ENGINE_PAR_TDL_TIME_L_UNIT
$ENGINE_PAR_TDL_FEEDBACK_L
$ENGINE_PAR_TDL_LEVEL_L
$ENGINE_PAR_TDL_PREDELAY_R
$ENGINE_PAR_TDL_PREDELAY_R_UNIT
$ENGINE_PAR_TDL_TIME_R
$ENGINE_PAR_TDL_TIME_R_UNIT
$ENGINE_PAR_TDL_FEEDBACK_R
$ENGINE_PAR_TDL_LEVEL_R
$ENGINE_PAR_TDL_CROSS_FEEDBACK
$ENGINE_PAR_TDL_WIDTH

Legacy Delay
$ENGINE_PAR_DL_TIME
$ENGINE_PAR_DL_TIME_UNIT
$ENGINE_PAR_DL_DAMPING
$ENGINE_PAR_DL_PAN
$ENGINE_PAR_DL_FEEDBACK

Reverb

Convolution
$ENGINE_PAR_IRC_PREDELAY
$ENGINE_PAR_IRC_LENGTH_RATIO_ER
$ENGINE_PAR_IRC_FREQ_LOWPASS_ER
$ENGINE_PAR_IRC_FREQ_HIGHPASS_ER
$ENGINE_PAR_IRC_LENGTH_RATIO_LR
$ENGINE_PAR_IRC_FREQ_LOWPASS_LR
$ENGINE_PAR_IRC_FREQ_HIGHPASS_LR
$ENGINE_PAR_IRC_ER_LR_BOUNDARY
$ENGINE_PAR_IRC_AUTO_GAIN
$ENGINE_PAR_IRC_REVERSE

Plate Reverb
$ENGINE_PAR_PR_DECAY
$ENGINE_PAR_PR_LOWSHELF
$ENGINE_PAR_PR_HIDAMP
$ENGINE_PAR_PR_PREDELAY
$ENGINE_PAR_PR_STEREO

Raum
$ENGINE_PAR_RAUM_TYPE
$NI_RAUM_TYPE_GROUNDED
$NI_RAUM_TYPE_AIRY
$NI_RAUM_TYPE_COSMIC
$ENGINE_PAR_RAUM_PREDELAY
$ENGINE_PAR_RAUM_PREDELAY_UNIT
$ENGINE_PAR_RAUM_FEEDBACK
$ENGINE_PAR_RAUM_LOWSHELF
$ENGINE_PAR_RAUM_HIGHCUT
$ENGINE_PAR_RAUM_DECAY
$ENGINE_PAR_RAUM_MOD
$ENGINE_PAR_RAUM_REVERB
$ENGINE_PAR_RAUM_DIFFUSION
$ENGINE_PAR_RAUM_SIZE
$ENGINE_PAR_RAUM_DAMPING
$ENGINE_PAR_RAUM_RATE
$ENGINE_PAR_RAUM_FREEZE
$ENGINE_PAR_RAUM_SPARSE

Reverb
$ENGINE_PAR_RV2_TYPE
$NI_REVERB2_TYPE_ROOM
$NI_REVERB2_TYPE_HALL
$ENGINE_PAR_RV2_TIME
$ENGINE_PAR_RV2_SIZE
$ENGINE_PAR_RV2_DAMPING
$ENGINE_PAR_RV2_MOD
$ENGINE_PAR_RV2_DIFF
$ENGINE_PAR_RV2_PREDELAY
$ENGINE_PAR_RV2_HIGHCUT
$ENGINE_PAR_RV2_LOWSHELF
$ENGINE_PAR_RV2_STEREO

Legacy Reverb
$ENGINE_PAR_RV_PREDELAY
$ENGINE_PAR_RV_SIZE
$ENGINE_PAR_RV_COLOUR
$ENGINE_PAR_RV_STEREO
$ENGINE_PAR_RV_DAMPING

Modulation

Legacy Chorus
$ENGINE_PAR_CH_DEPTH
$ENGINE_PAR_CH_SPEED
$ENGINE_PAR_CH_SPEED_UNIT
$ENGINE_PAR_CH_PHASE

Legacy Flanger
$ENGINE_PAR_FL_DEPTH
$ENGINE_PAR_FL_SPEED
$ENGINE_PAR_FL_SPEED_UNIT
$ENGINE_PAR_FL_PHASE
$ENGINE_PAR_FL_FEEDBACK
$ENGINE_PAR_FL_COLOR

Legacy Phaser
$ENGINE_PAR_PH_DEPTH
$ENGINE_PAR_PH_SPEED
$ENGINE_PAR_PH_SPEED_UNIT
$ENGINE_PAR_PH_PHASE
$ENGINE_PAR_PH_FEEDBACK

Utilities

Gainer
$ENGINE_PAR_GN_GAIN

Modulation

$ENGINE_PAR_MOD_TARGET_INTENSITY
The intensity slider of a modulation assignment. This controls the modulation amount in the positive range only. In order to apply a negative modulation amount, Invert button needs to be toggled (see below).

$ENGINE_PAR_MOD_TARGET_MP_INTENSITY
The intensity slider of a modulation assignment, including both negative and positive ranges (bipolar) - no modulation is at engine parameter value of **500000**, maximum modulation amount is at value **1000000**, maximum inverted modulation amount is at value **0**.

$MOD_TARGET_INVERT_SOURCE
The Invert button of a modulation assignment.

$ENGINE_PAR_INTMOD_BYPASS
The bypass button of an internal modulator, e.g. AHDSR envelope, LFO, Step Modulator...

$ENGINE_PAR_INTMOD_RETRIGGER
The Retrigger button of an internal modulator. This restarts the envelope or the phase of an LFO or the Step Modulator every time a note is received.

AHDSR
$ENGINE_PAR_ATK_CURVE
$ENGINE_PAR_ATTACK
$ENGINE_PAR_ATTACK_UNIT
$ENGINE_PAR_HOLD
$ENGINE_PAR_HOLD_UNIT
$ENGINE_PAR_DECAY
$ENGINE_PAR_DECAY_UNIT
$ENGINE_PAR_SUSTAIN
$ENGINE_PAR_RELEASE
$ENGINE_PAR_RELEASE_UNIT
$ENGINE_PAR_ENV_AHD

DBD
$ENGINE_PAR_DECAY1
$ENGINE_PAR_DECAY1_UNIT
$ENGINE_PAR_BREAK
$ENGINE_PAR_DECAY2
$ENGINE_PAR_DECAY2_UNIT
$ENGINE_PAR_ENV_DBD_EASY

Flexible Envelope
`$ENGINE_PAR_FLEXENV_LOOP`
`$ENGINE_PAR_FLEXENV_ONESHOT`
`$ENGINE_PAR_FLEXENV_NUM_STAGES` (**3 ... 31**)
`$ENGINE_PAR_FLEXENV_LOOP_START` (**1 ... 30**)
`$ENGINE_PAR_FLEXENV_LOOP_END` (**2 ... 31**)
`$ENGINE_PAR_FLEXENV_STAGE_TIME` (**1000000** is one second)
`$ENGINE_PAR_FLEXENV_STAGE_LEVEL`
`$ENGINE_PAR_FLEXENV_STAGE_SLOPE`

Envelope Follower
`$ENGINE_PAR_ENVF_ATTACK`
`$ENGINE_PAR_ENVF_RELEASE`
`$ENGINE_PAR_ENVF_GAIN_BOOST`
`$ENGINE_PAR_ENVF_ADAPTION`

LFO
For all LFOs:
`$ENGINE_PAR_INTMOD_FREQUENCY`
`$ENGINE_PAR_INTMOD_FREQUENCY_UNIT`
`$ENGINE_PAR_LFO_DELAY`
`$ENGINE_PAR_LFO_DELAY_UNIT`
`$ENGINE_PAR_LFO_PHASE`
For Rectangle and Multi LFOs:
`$ENGINE_PAR_INTMOD_PULSEWIDTH`
For Multi LFOs:
`$ENGINE_PAR_LFO_SINE`
`$ENGINE_PAR_LFO_RECT`
`$ENGINE_PAR_LFO_TRI`
`$ENGINE_PAR_LFO_SAW`
`$ENGINE_PAR_LFO_RAND`
`$ENGINE_PAR_LFO_NORMALIZE`

Step Modulator
`$ENGINE_PAR_INTMOD_FREQUENCY`
`$ENGINE_PAR_INTMOD_FREQUENCY_UNIT`
`$ENGINE_PAR_STEPSEQ_STEPS`
`$ENGINE_PAR_STEPSEQ_ONESHOT`
`$ENGINE_PAR_STEPSEQ_STEP_VALUE`

Glide
`$ENGINE_PAR_GLIDE_COEF`
`$ENGINE_PAR_GLIDE_COEF_UNIT`

Module Types and Subtypes

`$ENGINE_PAR_EFFECT_TYPE`

Used to query the type of an effect loaded in Group FX, Instrument Bus FX, Insert FX or Main FX chains. Can be any of the following:
`$EFFECT_TYPE_NONE` (Empty slot)
`$EFFECT_TYPE_FILTER`
`$EFFECT_TYPE_SOLID_GEQ`
`$EFFECT_TYPE_COMPRESSOR`
`$EFFECT_TYPE_FB_COMP`
`$EFFECT_TYPE_LIMITER`
`$EFFECT_TYPE_BUS_COMP`
`$EFFECT_TYPE_SUPERGT`
`$EFFECT_TYPE_TRANS_MASTER`
`$EFFECT_TYPE_TRANSLIM`
`$EFFECT_TYPE_ACBOX`
`$EFFECT_TYPE_BASSINVADER`
`$EFFECT_TYPE_BASSPRO`
`$EFFECT_TYPE_CABINET`
`$EFFECT_TYPE_EP_PREAMPS`
`$EFFECT_TYPE_HOTSOLO`
`$EFFECT_TYPE_JUMP`
`$EFFECT_TYPE_TWANG`
`$EFFECT_TYPE_VAN51`
`$EFFECT_TYPE_BIGFUZZ`
`$EFFECT_TYPE_CAT`
`$EFFECT_TYPE_CRYWAH`
`$EFFECT_TYPE_DIRT`
`$EFFECT_TYPE_DISTORTION`
`$EFFECT_TYPE_DSTORTION`
`$EFFECT_TYPE_FUZZ`
`$EFFECT_TYPE_SHAPER` (Saturation)
`$EFFECT_TYPE_SKREAMER`
`$EFFECT_TYPE_BITE`
`$EFFECT_TYPE_LOFI`
`$EFFECT_TYPE_TAPE_SAT`
`$EFFECT_TYPE_WOWFLUTTER`
`$EFFECT_TYPE_PSYCHEDELAY`
`$EFFECT_TYPE_REPLIKA`
`$EFFECT_TYPE_TWINDELAY`
`$EFFECT_TYPE_DELAY`
`$EFFECT_TYPE_IRC` (Convolution)
`$EFFECT_TYPE_PLATEREVERB`
`$EFFECT_TYPE_RAUM`
`$EFFECT_TYPE_REVERB2`

$ENGINE_PAR_EFFECT_TYPE
`$EFFECT_TYPE_REVERB`
`$EFFECT_TYPE_CHORAL`
`$EFFECT_TYPE_FLAIR`
`$EFFECT_TYPE_FREAK`
`$EFFECT_TYPE_PHASIS`
`$EFFECT_TYPE_RINGMOD`
`$EFFECT_TYPE_ROTATOR`
`$EFFECT_TYPE_VIBRATO_CHORUS`
`$EFFECT_TYPE_CHORUS`
`$EFFECT_TYPE_FLANGER`
`$EFFECT_TYPE_PHASER`
`$EFFECT_TYPE_STEREO` (Stereo Modeller)
`$EFFECT_TYPE_STEREO_TUNE`
`$EFFECT_TYPE_SURROUND_PANNER`
`$EFFECT_TYPE_AET_FILTER` (Group FX chain only!)
`$EFFECT_TYPE_GAINER`
`$EFFECT_TYPE_INVERTER`
`$EFFECT_TYPE_SEND_LEVELS`

$ENGINE_PAR_SEND_EFFECT_TYPE
Used to query the type of an effect loaded in Send FX chain. Can be any of the following:
`$EFFECT_TYPE_NONE` (Empty slot)
`$EFFECT_TYPE_PSYCHEDELAY`
`$EFFECT_TYPE_REPLIKA`
`$EFFECT_TYPE_TWINDELAY`
`$EFFECT_TYPE_DELAY`
`$EFFECT_TYPE_IRC` (Convolution)
`$EFFECT_TYPE_PLATEREVERB`
`$EFFECT_TYPE_RAUM`
`$EFFECT_TYPE_REVERB2`
`$EFFECT_TYPE_REVERB`
`$EFFECT_TYPE_CHORUS`
`$EFFECT_TYPE_FLANGER`
`$EFFECT_TYPE_PHASER`
`$EFFECT_TYPE_GAINER`

`$ENGINE_PAR_EFFECT_SUBTYPE`

Used to query the filter or EQ subtype. Can be any of the following:
`$FILTER_TYPE_LP1POLE`
`$FILTER_TYPE_HP1POLE`
`$FILTER_TYPE_BP2POLE`
`$FILTER_TYPE_LP2POLE`
`$FILTER_TYPE_HP2POLE`
`$FILTER_TYPE_LP4POLE`
`$FILTER_TYPE_HP4POLE`
`$FILTER_TYPE_BP4POLE`
`$FILTER_TYPE_BR4POLE`
`$FILTER_TYPE_LP6POLE`
`$FILTER_TYPE_PHASER`
`$FILTER_TYPE_VOWELA`
`$FILTER_TYPE_VOWELB`
`$FILTER_TYPE_PRO52`
`$FILTER_TYPE_LADDER`
`$FILTER_TYPE_VERSATILE`
`$FILTER_TYPE_EQ1BAND`
`$FILTER_TYPE_EQ2BAND`
`$FILTER_TYPE_EQ3BAND`
`$FILTER_TYPE_DAFT_LP`
`$FILTER_TYPE_SV_LP1`
`$FILTER_TYPE_SV_LP2`
`$FILTER_TYPE_SV_LP4`
`$FILTER_TYPE_SV_LP6`
`$FILTER_TYPE_LDR_LP1`
`$FILTER_TYPE_LDR_LP2`
`$FILTER_TYPE_LDR_LP3`
`$FILTER_TYPE_LDR_LP4`
`$FILTER_TYPE_AR_LP2`
`$FILTER_TYPE_AR_LP4`
`$FILTER_TYPE_AR_LP24`
`$FILTER_TYPE_SV_HP1`
`$FILTER_TYPE_SV_HP2`
`$FILTER_TYPE_SV_HP4`
`$FILTER_TYPE_SV_HP6`
`$FILTER_TYPE_LDR_HP1`
`$FILTER_TYPE_LDR_HP2`
`$FILTER_TYPE_LDR_HP3`
`$FILTER_TYPE_LDR_HP4`
`$FILTER_TYPE_AR_HP2`
`$FILTER_TYPE_AR_HP4`
`$FILTER_TYPE_AR_HP24`
`$FILTER_TYPE_DAFT_HP`

$ENGINE_PAR_EFFECT_SUBTYPE

`$FILTER_TYPE_SV_BP2`
`$FILTER_TYPE_SV_BP4`
`$FILTER_TYPE_SV_BP6`
`$FILTER_TYPE_LDR_BP2`
`$FILTER_TYPE_LDR_BP4`
`$FILTER_TYPE_AR_BP2`
`$FILTER_TYPE_AR_BP4`
`$FILTER_TYPE_AR_BP24`
`$FILTER_TYPE_SV_NOTCH4`
`$FILTER_TYPE_SV_NOTCH6`
`$FILTER_TYPE_LDR_PEAK`
`$FILTER_TYPE_LDR_NOTCH`
`$FILTER_TYPE_SV_PAR_LPHP`
`$FILTER_TYPE_SV_PAR_BPBP`
`$FILTER_TYPE_SV_SER_LPHP`
`$FILTER_TYPE_FORMANT_1`
`$FILTER_TYPE_FORMANT_2`
`$FILTER_TYPE_SIMPLE_LPHP`

Note that the Solid G-EQ is not treated as a filter/EQ subtype, but as an effect.

$ENGINE_PAR_INTMOD_TYPE

Used to query the type of an internal modulator, can be any of the following:

`$INTMOD_TYPE_NONE`
`$INTMOD_TYPE_LFO`
`$INTMOD_TYPE_ENVELOPE`
`$INTMOD_TYPE_STEPMOD`
`$INTMOD_TYPE_ENV_FOLLOW`
`$INTMOD_TYPE_GLIDE`

$ENGINE_PAR_INTMOD_SUBTYPE

Used to query the subtype of envelopes and LFOs. Can be any of the following:

`$ENV_TYPE_AHDSR`
`$ENV_TYPE_FLEX`
`$ENV_TYPE_DBD`
`$LFO_TYPE_SINE`
`$LFO_TYPE_RECTANGLE`
`$LFO_TYPE_TRIANGLE`
`$LFO_TYPE_SAWTOOTH`
`$LFO_TYPE_RANDOM`
`$LFO_TYPE_MULTI`

Group Start Options Query

Group Start Options Constants

```
$ENGINE_PAR_START_CRITERIA_MODE
    $START_CRITERIA_NONE
    $START_CRITERIA_ON_KEY
    $START_CRITERIA_ON_CONTROLLER
    $START_CRITERIA_CYCLE_ROUND_ROBIN
    $START_CRITERIA_CYCLE_RANDOM
    $START_CRITERIA_SLICE_TRIGGER
$ENGINE_PAR_START_CRITERIA_KEY_MIN
$ENGINE_PAR_START_CRITERIA_KEY_MAX
$ENGINE_PAR_START_CRITERIA_CONTROLLER
$ENGINE_PAR_START_CRITERIA_CC_MIN
$ENGINE_PAR_START_CRITERIA_CC_MAX
$ENGINE_PAR_START_CRITERIA_CYCLE_CLASS
$ENGINE_PAR_START_CRITERIA_ZONE_IDX
$ENGINE_PAR_START_CRITERIA_SLICE_IDX
$ENGINE_PAR_START_CRITERIA_SEQ_ONLY
$ENGINE_PAR_START_CRITERIA_NEXT_CRIT
    $START_CRITERIA_AND_NEXT
    $START_CRITERIA_AND_NOT_NEXT
    $START_CRITERIA_OR_NEXT
```

24. Zone Parameters

Zone Parameters

These set the parameters for the user zones via KSP in the same manner and ranges as available on the mapping editor. They can be set with `set_zone_par()` and retrieved with `get_zone_par()` commands. When the user zones are declared, all these parameters are set to **0** by default.

`$ZONE_PAR_HIGH_KEY`
Sets the high key for the zone. Range: **0 ... 127**.

`$ZONE_PAR_LOW_KEY`
Sets the low key of the zone. Range: **0 ... 127**.

`$ZONE_PAR_HIGH_VELO`
Sets the maximum velocity response of the zone. Range: **1 ... 127**.

`$ZONE_PAR_LOW_VELO`
Sets the minimum velocity response of the zone. Range: **1 ... 127**.

`$ZONE_PAR_ROOT_KEY`
Sets the root key of the zone. Range: **0 ... 127**.

`$ZONE_PAR_FADE_LOW_KEY`
Optionally use this parameter to create zone crossfades. The value is set in the form of a distance to the `$ZONE_PAR_LOW_KEY`. Range: `$ZONE_PAR_HIGH_KEY - $ZONE_PAR_LOW_KEY + 1`

`$ZONE_PAR_FADE_HIGH_KEY`
Optionally use this parameter to create zone crossfades. The value is set in the form of a distance to the `$ZONE_PAR_HIGH_KEY`. Range: `$ZONE_PAR_HIGH_KEY - $ZONE_PAR_LOW_KEY + 1`

`$ZONE_PAR_FADE_LOW_VELO`
Optionally use this parameter to create zone crossfades. The value is set in the form of a distance to the $`$ZONE_PAR_LOW_VELO`. Range: `$ZONE_PAR_HIGH_VELO - $ZONE_PAR_LOW_VELO + 1`

$ZONE_PAR_FADE_HIGH_VELO

Optionally use this parameter to create zone crossfades. The value is set in the form of a distance to the `$ZONE_PAR_HIGH_VELO`.

Range: `$ZONE_PAR_HIGH_VELO - $ZONE_PAR_LOW_VELO + 1`

$ZONE_PAR_VOLUME

Sets the volume of the zone. Range: **-3600 ... 3600** .

$ZONE_PAR_PAN

Sets the panning of the zone. Range: **-1000 ... 1000** .

$ZONE_PAR_TUNE

Sets the tuning of the zone. Range: **-3600 ... 3600** .

$ZONE_PAR_GROUP

Sets the group of the user zone. By default a user zone is placed in group **0**.

$ZONE_PAR_SAMPLE_START

Sets the sample start value of the sample attached to the zone.

$ZONE_PAR_SAMPLE_END

Sets the sample end value of the sample attached to the zone.

$ZONE_PAR_SAMPLE_MOD_RANGE

Sets the sample start modulation range of the sample attached to the zone. This is the **S.Mod** parameter in Kontakt's Wave Editor.

$ZONE_PAR_SAMPLE_RATE

Gets the sample rate of the sample attached to the zone. This zone parameter cannot be used with `set_zone_par()`.

$ZONE_PAR_SELECTED

Returns **1** if the zone is selected in Kontakt's Mapping Editor, or **0** if it's not selected. This zone parameter cannot be used with `set_zone_par()`.

Examples

```
on init
    message("")

    set_num_user_zones(4)

    set_zone_par(%NI_USER_ZONE_IDS[0], $ZONE_PAR_GROUP, 0)
    set_zone_par(%NI_USER_ZONE_IDS[1], $ZONE_PAR_GROUP, 1)
    set_zone_par(%NI_USER_ZONE_IDS[2], $ZONE_PAR_GROUP, 2)
    set_zone_par(%NI_USER_ZONE_IDS[3], $ZONE_PAR_GROUP, 3)
end on
```

Creates 4 user zones in an instrument and assigns each to a separate group. Make sure that you have enough empty groups defined in the instrument when testing this example!

Loop Parameters

`$LOOP_PAR_MODE`

The loop playback mode of the selected loop within the zone.

0: Loop off

1: Loop until end

2: Loop until end, bidirectional

3: Loop until release

4: Loop until release, bidirectional

`$LOOP_PAR_START`

The starting point in samples of the selected loop within the zone. If this parameter is not set, the loop will start at the beginning of the sample.

`$LOOP_PAR_LENGTH`

The loop length in samples of the selected loop within the zone. If this parameter is not set, the loop length will correspond to the entire sample.

`$LOOP_PAR_XFADE`

The crossfade duration in microseconds for the selected loop within the zone.

`$LOOP_PAR_COUNT`

The number of times the selected loop within the zone will repeat. If this parameter is not set (or is set to **0**), the loop will play indefinitely.

`$LOOP_PAR_TUNING`

Sets the tuning offset inside the loop area for the selected loop within the zone. This offset is applied on the first repeat of the loop, and for all successive repeats (as defined by `$LOOP_PAR_COUNT`).

Examples

```
on init
    message("")

    set_num_user_zones(1)

    set_sample(%NI_USER_ZONE_IDS[0], "path/to/sample.wav")
end on

on ui_control ($SampleLoopOn)
    wait_async(set_loop_par(%NI_USER_ZONE_IDS[0], 0, $LOOP_PAR_MODE, $SampleLoopOn))
end on
```

Enable or disable the loop of a sample loaded into a user zone.

Sample Parameters

`$NI_FILE_NAME`
The file name of a zone's sample (corresponds to the zone name).

`$NI_FILE_FULL_PATH`
The full path of a zone's sample (same result as without the parameter).

`$NI_FILE_FULL_PATH_OS`
The full OS path of a zone's sample (uses backslashes on Windows).

`$NI_FILE_EXTENSION`
The file extension of a zone's sample (without the dot).

25. Advanced Concepts

Preprocessor & System Scripts

```
SET_CONDITION(<condition-symbol>)
```

Define a symbol to be used as a condition.

```
RESET_CONDITION(<condition-symbol>)
```

Delete a condition definition.

```
USE_CODE_IF(<condition-symbol>)
…
END_USE_CODE
```

Interpret code when `<condition-symbol>` is defined.

```
USE_CODE_IF_NOT(<condition-symbol>)
…
END_USE_CODE
```

Interpret code when `<condition-symbol>` is not defined.

```
NO_SYS_SCRIPT_GROUP_START
```

Condition; if defined with `SET_CONDITION()`, the system script which handles Group Start Options will be bypassed.

```
NO_SYS_SCRIPT_PEDAL
```

Condition; if defined with `SET_CONDITION()`, the system script which sustains notes when CC# 64 is received will be bypassed.

```
NO_SYS_SCRIPT_RLS_TRIG
```

Condition; if defined with `SET_CONDITION()`, the system script which triggers samples upon key release is bypassed.

```
reset_rls_trig_counter(<note>)
```

Resets the release trigger counter (used by the release trigger system script).

```
will_never_terminate(<event-id>)
```

Tells the script engine that this event will never be finished (used by the release trigger system script).

Examples

A preprocessor is used to exclude code elements from interpreting. `<condition-symbol>` refers to a symbolic name which consists of alphanumeric symbols, started by a letter. For example, you could write:

```
on note
    { do something general }
    $var := 5

    {do some conditional code}
    USE_CODE_IF_NOT(dont_do_sequencer)
    while ($NOTE_HELD = 1)
        play_note($EVENT_NOTE, $EVENT_VELOCITY, 0, $DURATION_SIXTEENTH)
        wait($DURATION_EIGHTH)
    end while
    END_USE_CODE
end on
```

What's happening here?

Only if the symbol `dont_do_sequencer` is NOT defined, the code between `USE_` and `END_USE` will be processed. If the symbol were to be found, the code would not be passed on to the parser; it is as if the code was never written. Therefore it does not utilize any CPU resource.

All commands will be interpreted **before** the script is running, i.e., by using `USE_CODE_`, the code might get stalled before it is passed to the script engine. This means, `SET_CONDITION` and `RESET_CONDITION` are not actually true commands: they cannot be utilized in `if ... else` control statements; also a `wait()` statement before those commands is useless. Each `SET_CONDITION` and `RESET_CONDITION` will be executed before something else happens.

All defined symbols are passed on to the following script slots, i.e. if script slot 3 contains conditional code, you can turn it on or off in script 1 or 2.

You can use conditional code to bypass system scripts. If you define one of those symbols with `SET_CONDITION()`, the corresponding part of the system scripts will be bypassed. For clarity, those definitions should always take place in `on init` callback.

```
on init
    { we want to do our own release triggering }
    SET_CONDITION(NO_SYS_SCRIPT_RLS_TRIG)
end on

on release
    { do something custom here }
end on
```

If we want to do our own release triggering logic, this is the first step to do so.

PGS

It is possible to send and receive values from one script to another, circumventing the usual left-to-right processing order, by using the Program Global Storage (PGS) commands. PGS is shared memory that can be read or written by any script.

PGS commands

```
pgs_create_key(<key-id>, <size>)
pgs_key_exists(<key-id>)
pgs_set_key_val(<key-id>, <index>, <value>)
pgs_get_key_val(<key-id>, <index>)
```

`<key-id>` is similar to a variable name; it can only contain letters and numbers and must not start with a number. It also cannot be longer than **64** characters. It is a good idea to always write them in capitals to emphasize their unique status.

Here's an example, insert this script into any slot:

```
on init
    declare ui_button $Just_Do_It

    pgs_create_key(FIRST_KEY, 1)  { defines a key with 1 element }
    pgs_create_key(NEXT_KEY, 128) { defines a key with 128 elements }
end on

on ui_control($Just_Do_It)
    { writes 70 into the first and only memory location of FIRST_KEY }
    pgs_set_key_val(FIRST_KEY, 0, 70)

    { writes 50 into the first and 60 into the last memory location of NEXT_KEY }
    pgs_set_key_val(NEXT_KEY, 0, 50)
    pgs_set_key_val(NEXT_KEY, 127, 60)
end on
```

Then, insert the following script into any other slot:

```
on init
    declare ui_knob $First (0, 100, 1)
    declare ui_table %Next[128] (5, 2, 100)
end on

on pgs_changed
    { checks if FIRST_KEY and NEXT_KEY have been declared }
    if (pgs_key_exists(FIRST_KEY) and _pgs_key_exists(NEXT_KEY))
        $First := pgs_get_key_val(FIRST_KEY, 0)      { in this case 70 }
        %Next[0] := pgs_get_key_val(NEXT_KEY, 0)     { in this case 50 }
        %Next[127] := pgs_get_key_val(NEXT_KEY, 127) { in this case 60 }
    end if
end on
```

As illustrated above, there is also a callback that is executed whenever a `pgs_set_key_val()` command has been executed.

on pgs_changed

Callback type, executed whenever any `pgs_set_key_val()` is executed in any script

It is possible to have as many keys as you want, however each key can only have up to **256** elements.

The basic handling for PGS strings is the same as for normal PGS keys; there's only one difference: PGS strings keys aren't arrays like the standard PGS keys you already know – they resemble normal string variables.

PGS strings commands

```
pgs_create_str_key(<key-id>)
pgs_str_key_exists(<key-id>)
pgs_set_str_key_val(<key-id>, <string>)
<string> := pgs_get_str_key_val(<key-id>)
```

Resource Container

Introduction

The resource container is a useful tool for library developers. It is a dedicated location to store scripts, graphics, .nka files, Creator Tools GUI Designer performance views and impulse response files that can be referenced by any NKI or a group of NKIs that are linked to the container. Another benefit is that you can create a resource container monolith file containing all the scripts, graphics etc, so that you can easily move them around or send them to other team members. When loading an NKI, the resource container is treated like a sample, so if it is not found it will appear in the Content Missing dialog.

Setup

To create a resource container for your library, open up the instrument Options dialog for the Instrument you're working on, and click the **Create** button beside the area labeled Resource Container. After creating a new resource container file, Kontakt checks if there is already a `Resources` folder available. If there isn't, Kontakt will ask to create it for you. If you do this, you will find `Resources` and `Data` folders next to the NKR file you have just created.

The `Resources` folder is the place where you can store the files that an Instrument can use which are not samples. As you can see, Kontakt has already created several subfolders for you: `ir_samples`, `pictures` (for GUI graphics and wallpapers), `data` (for .nka files), `performance_view` (for Creator Tools' GUI Designer) and `scripts`. The only thing to do now is to move your files into the right folders and you are ready to go.

Working with the Resource Container

Let's say you're creating a new library: after setting up the resource container as described above, you can tell all of the NKIs that are part of your library to use this particular resource container. Just open up the Instrument Options dialog and use the **Browse** function (click on the open folder button to the left of **Create** button).

As long as the `Resources` folder exist besides the NKR file (which is the resource container monolith), Kontakt will read all files directly from this folder structure.

For loading scripts from the `scripts` subfolder, use the "Apply from… → Resources folder" function within the Script Editor.

Now let's say you want to send your current working status to another team member. Open up the Instrument Options dialog, click the **Repack** button, which will quickly fully update your existing NKR file. Be aware that this will completely overwrite your monolith, it won't be matched in any way. Now Kontakt will do the following:

- Check the `ir_samples` subfolder for any .wav, .aif, .aiff or .ncw files and put them into the monolith.
- Check the `pictures` folder for any .png files that also have a .txt file of the same filename next to them. All of these will be packed into the monolith. Note that wallpapers also need a .txt file, or they will be ignored.
- Check the `scripts` subfolder for any .txtfiles which will then be put into the monolith.
- Check the `data` subfolder for any .nka files which will then be put into the monolith.
- Check the `performance_view` folder for any .nckp files containing performance views created by Creator Tools.

After that, rename your `Resources` folder and reopen your Instrument. Now that the `Resources` folder is not present anymore, Kontakt will automatically read from the NKR monolith file. If everything is still working as expected, you can send your library (instruments, along with samples and the NKR monolith) to your team members. This is also how your library should be deployed to the market - you are not supposed to release your library with the `Resources` folder present!

To continue your work, simply rename the Resources folder back to `Resources`.

Remarks

- The resource container will be checked in the Content Missing dialog.
- When you save your Instrument as a monolith file, the resource container will not be integrated into the monolith. The path to the resource container will be saved as an absolute path, which will only work locally on your machine - on other computers Content Missing dialog will show up upon loading such Instrument, since the path to the resource container is most likely not going to be valid anymore.

Changing FX from KSP

Introduction

Prior to Kontakt 5.5, the infrastructure to get information about the content of effect slots was already in place via engine parameter variables like `$ENGINE_PAR_EFFECT_TYPE` and built-in constants like `$EFFECT_TYPE_FILTER` (refer to Module Types and Subtypes).

Starting with Kontakt 5.5, it is also possible to change the loaded effects with the same set of built-in constants.

Example

```
on init
    wait_async(set_engine_par($ENGINE_PAR_EFFECT_TYPE, $EFFECT_TYPE_FILTER, 0, 0,
-1))
    wait_async(set_engine_par($ENGINE_PAR_EFFECT_SUBTYPE, $FILTER_TYPE_LDR_LP4, 0,
0, -1))
end on
```

Inserts a 4-pole lowpass ladder filter into the first Group FX slot of the first group in the Instrument.

on async_complete callback

Changing the effect slot contents is an asynchronous operation. This means, one cannot reliably access the newly instantiated effect immediately after instantiation. To resolve this, the command returns an `$NI_ASYNC_ID` and triggers the `on async_complete` callback. In addition, `wait_async()` command can also be used to streamline this process (and is recommended).

Default Filter Type

Filters are somewhat special as they are effect types that feature subtypes. Since one can now instantiate a new filter from KSP without explicitly selecting its subtype, there is a need for a predefined default filter subtype. This is SV LP4.

Implications on Modulation and Automation assignments

When changing the contents of effect slots through KSP, it is expected that the handling of assigned automation and modulation is identical to performing the same action using Kontakt's GUI.

- When changing a slot's effect type or removing it entirely, all modulation and automation assignments are also removed. Specifically to modulators, if the removed assignments are the only ones belonging to a certain internal modulator (i.e., if the modulator is not assigned to other targets as well), the modulator itself is also removed.
- When changing a slot's effect subtype (only applies to filters), everything is left unchanged. It is accepted that in certain cases, one may end up with "orphaned" modulation assignments as it is the case right now; e.g., when having modulation assigned to a parameter that is no longer available, like Resonance or Gain.

Changing Modulator Subtypes

Using the same commands described above, one can also change the subtype of internal modulators. Specifically, one could switch between envelope types (AHDSR, Flex and DBD), or LFO types (Rectangle, Triangle, Sawtooth, Random, Multi, Multi Digital). A modulator cannot be inserted or removed. Its type (LFO, Envelope, Step Modulator, Envelope Follower and Glide) cannot be changed either.

Special Cases

There are two effect types which cannot be set from KSP: **Surround Panner** and **AET Filter**.

The Advanced Engine Tab

The Advanced Engine tab can be a useful tool for debugging and measuring the performance of your instruments and scripts.

While the Engine tab (a sub-tab of the Monitor tab in Kontakt's Side Pane) can provide a useful display of performance statistics, the advanced version gives higher accuracy to things like CPU usage, and also displays information on multiple instances of Kontakt when it is used as a plug-in.

Displaying the Advanced Engine Tab

As mentioned earlier, the Engine tab is a sub section of the Monitor tab, which can be found in Kontakt's Side Pane.

- To access the Advanced Engine tab, hold the [Alt] (Windows) or [Opt] (macOS) key while clicking on the Engine tab.
- To return to the main Engine tab, just click on the Engine tab again with no keys held.

Instance Overview

If you are running multiple instances of Kontakt as a plug-in in a DAW or host, each instance will be given an entry in this section. If you are using Kontakt standalone, only the current instance will be displayed.

There are five performance statistics you can view here:

- **CPU**: displays the current CPU load in percent (at a higher resolution than the other CPU readouts in Kontakt) as well as the highest recorded peak CPU level (displayed in parentheses). You can reset the high peak by re-initializing the Kontakt instance by clicking on the "!" button in the top right of Kontakt's interface.
- **Voices**: displays the total number of voices currently in use by the current Kontakt instance.
- **Voices killed**: displays the total number of voices that have been killed due to CPU overload (displayed on the left) and DFD overload (displayed on the right).
- **Process buffer**: displays the current audio buffer size in samples.
- **Events**: displays the total number of events currently in the event queue. While a voice is the equivalent to a sample being played back, an event is more closely related to MIDI note messages being processed by the engine. For example, a single event could produce 3 voices, if there are 3 samples mapped to a single note. Additionally, if you are holding a MIDI key event though the triggered sample has finished playback, the voice will terminate, but the event will remain in the queue. As such, this display can be useful for tracking down events that are hanging, as these are not always audible in the way that hanging voices would be.

Total

The lower section displays the total performance statistics for all Kontakt instances currently loaded. It has the following parameters:

- **Voices** and **Voices killed**: like the displays in the Instance Overview, but a total for all instances.
- **DFD load**: if you are playing Instruments that use DFD mode, this measures their hard disk access. It is essentially a more accurate version of the Disk meter in Kontakt’s Header.
- **DFD memory**: a measurement of how much RAM is being used to process the DFD stream.
- **DFD requests**: the total number of requests made by Kontakt to read data from the hard disk.

26. Multi Script

General Information

The multi script utilizes the same KSP syntax as the instrument scripts. Here are the main differences:

- The multi script works on a pure MIDI event basis, i.e., you're working with raw MIDI data.
- There are only four callback types available: `on init`, `on persistence_changed`, `on midi_in`, and the various `on ui_control` callbacks.
- Every MIDI event triggers the `on midi_in` callback.
- There are various built-in variables for the respective MIDI bytes.

The new multi script tab is accessed by clicking on the **KSP** button in the multi header.

Just as instrument scripts are saved with the instrument, multi scripts are saved with the multi. In relation to GUIs, everything is identical with the instrument script. The scripts are stored in a folder called `Multiscripts`, which resides next to the already existing `Scripts` folder inside the `Presets` folder in the factory data:

macOS: `/Library/Application Support/Native Instruments/Kontakt 7/Presets/Multiscripts`

Windows: `C:\Program Files\Common Files\Native Instruments\Kontakt 7\Presets\Multiscripts`

It is very important to understand the different internal structure of the event processing in the multi script, as opposed to the instrument script.

On the instrument level, you can retrieve the event IDs of notes only, i.e., `$EVENT_ID` only works in the `on note` and `on release` callbacks. On the multi level, any incoming MIDI event has a unique ID which can be retrieved with `$EVENT_ID`. This means, `$EVENT_ID` can be a note event, a controller message, a Program Change command etc.

This brings us to the usage of `change_note()`, `change_velo()` etc. commands. Since `$EVENT_ID` does not necessarily refer to a note event, these commands will not work in the multi script either. However, it is possible to modify aspects of the incoming event using the set_event_par() command.

And most important of all, remember that the multi script is nothing more than a MIDI processor, whereas the instrument script is an event processor. A note event in the instrument script is bound to a voice, whereas MIDI events from the multi script are translated into note events on the instrument level. This simply means that `play_note()`, `note_off()` etc. don't work in the multi script.

It is beneficial to be familiar with the basic structure of MIDI messages when working with the multi script.

ignore_midi

`ignore_midi`
Ignores the event which triggered the callback.

Remarks

- Like `ignore_event()`, `ignore_midi` is a very "strong" command. Keep in mind that `ignore_midi` will ignore all incoming events.
- If you just want to change the MIDI channel and/or any of the MIDI bytes of incoming events, you can also use `set_event_par()`.

Example

```
on midi_in
    if ($MIDI_COMMAND = $MIDI_COMMAND_NOTE_ON and $MIDI_BYTE_2 > 0)
        ignore_midi
    end if

    if ($MIDI_COMMAND = $MIDI_COMMAND_NOTE_OFF or ($MIDI_COMMAND =
$MIDI_COMMAND_NOTE_ON and $MIDI_BYTE_2 = 0))
        ignore_midi
    end if
end on
```

Ignoring Note On and Note Off messages. Note that some keyboards use a Note On command with a velocity of 0 to designate a Note Off command.

See Also

ignore_event()

on midi_in

`on midi_in`
MIDI callback, triggered by every incoming MIDI event.

Example

```
on midi_in
    select ($MIDI_COMMAND)
        case $MIDI_COMMAND_NOTE_ON
            if ($MIDI_BYTE_2 > 0)
                message("Note On")
            else
                message("Note Off")
            end if
        case $MIDI_COMMAND_NOTE_OFF
            message("Note Off")
        case $MIDI_COMMAND_CC
            message("Controller")
        case $MIDI_COMMAND_PITCH_BEND
            message("Pitch Bend")
        case $MIDI_COMMAND_MONO_AT
            message("Channel Pressure")
        case $MIDI_COMMAND_POLY_AT
            message("Poly Pressure")
        case $MIDI_COMMAND_PROGRAM_CHANGE
            message("Program Change")
    end select
end on
```

Monitoring various incoming MIDI messages.

See Also

ignore_midi

set_midi()

`set_midi(<channel>, <command>, <byte-1>, <byte-2>)`

Create any type of MIDI event.

Remarks

- If you simply want to change the MIDI channel and/or any of the MIDI bytes, you can use `set_event_par()` command.

Example

```
on midi_in
    if ($MIDI_COMMAND = $MIDI_COMMAND_NOTE_ON and $MIDI_BYTE_2 > 0)
        set_midi($MIDI_CHANNEL, $MIDI_COMMAND_NOTE_ON, $MIDI_BYTE_1 + 4,
$MIDI_BYTE_2)
        set_midi($MIDI_CHANNEL, $MIDI_COMMAND_NOTE_ON, $MIDI_BYTE_1 + 7,
$MIDI_BYTE_2)
    end if

    if ($MIDI_COMMAND = $MIDI_COMMAND_NOTE_OFF or ($MIDI_COMMAND =
$MIDI_COMMAND_NOTE_ON and $MIDI_BYTE_2 = 0))
        set_midi($MIDI_CHANNEL, $MIDI_COMMAND_NOTE_ON, $MIDI_BYTE_1 + 4, 0)
        set_midi($MIDI_CHANNEL, $MIDI_COMMAND_NOTE_ON, $MIDI_BYTE_1 + 7, 0)
    end if
end on
```

A simple harmonizer – note that you also have to supply the correct Note Off commands!

See Also

set_event_par()

Events and MIDI: `$EVENT_PAR_MIDI_CHANNEL`, `$EVENT_PAR_MIDI_COMMAND`, `$EVENT_PAR_MIDI_BYTE_1`, `$EVENT_PAR_MIDI_BYTE_2`

Multi Script Command Arguments

`$MIDI_CHANNEL`
The MIDI channel of the received MIDI event. Since Kontakt can handle four different MIDI ports, this number is in **0 ... 63** range (4 ports, 16 MIDI channels).

`$MIDI_COMMAND`
The command type, i.e Note, CC, Program Change etc. of the received MIDI event. There are various constants for this variable (see below).

`$MIDI_BYTE_1` `$MIDI_BYTE_2`
The two MIDI bytes of the message, can be **0 ... 127**.

`$MIDI_COMMAND_NOTE_ON`
`$MIDI_BYTE_1`: note number `$MIDI_BYTE_2`: velocity Note: a velocity value of **0** is equivalent to a Note Off command

`$MIDI_COMMAND_NOTE_OFF`
`$MIDI_BYTE_1`: note number `$MIDI_BYTE_2`: release velocity

`$MIDI_COMMAND_POLY_AT`
`$MIDI_BYTE_1`: note number `$MIDI_BYTE_2`: polyphonic key pressure value

`$MIDI_COMMAND_CC`
`$MIDI_BYTE_1`: controller number `$MIDI_BYTE_2`: controller value

`$MIDI_COMMAND_PROGRAM_CHANGE`
`$MIDI_BYTE_1`: program number `$MIDI_BYTE_2`: not used

`$MIDI_COMMAND_MONO_AT`
`$MIDI_BYTE_1`: channel pressure value `$MIDI_BYTE_2`: not used

$MIDI_COMMAND_PITCH_BEND
$MIDI_BYTE_1: LSB value $MIDI_BYTE_2: MSB value

$MIDI_COMMAND_RPN **$MIDI_COMMAND_NRPN**
$MIDI_BYTE_1: address $MIDI_BYTE_2: value

Event Parameter Constants
Event parameters which can be used with `get_event_par()` and `set_event_par()`: $EVENT_PAR_MIDI_CHANNEL $EVENT_PAR_MIDI_COMMAND $EVENT_PAR_MIDI_BYTE_1 $EVENT_PAR_MIDI_BYTE_2

27. Additional Resources

Whether you are starting out with KSP or a seasoned veteran, find a variety of resources, from KSP scripts to tutorials and beyond, below.

⚠ The websites linked below are owned and operated by third parties. The links are provided for your information and convenience only. Native Instruments has no control over the contents of any of the linked websites and is not responsible for these websites or their content or availability.

- SublimeKSP: A Sublime Text plugin for working with and compiling KSP code, including many added features.
- Koala: A library of additional KSP functions to be used in conjunction with SublimeKSP.
- Resources for Kontakt Builders: An experimental repository with a large variety of script examples for KSP and Lua (Creator Tools). It also includes links to additional resources for the larger Kontakt environment, from GUI tools to sample manipulation.
- Discord: Become a member in the Kontakt developer Discord and connect with fellow KSP developers.
- NI Scripting Forum: Become a part of the discussion on the new NI community forum.
- VS Code extension: KSP language support for Visual Studio Code.

28. Version History

The following changelog provides a version history of all new and improved features released in Kontakt.

Kontakt 7.8

New Features

- New callback type, on ui_controls, which allows defining behavior of all UI widgets by their ID in a single callback.
- Engine parameters for new effects: EP Preamps, Big Fuzz, Fuzz, Stereo Tune.

Improved Features

- `$EVENT_PAR_SOURCE` now works in multi scripts.

Kontakt 7.6

New Features

- Support for **NKS2** parameters and navigation definition through new commands and control parameters: see NKS2 SDK documentation.
- New control parameter, `$CONTROL_PAR_TYPE` , which returns the type of UI widget for the specified UI ID.
- New control parameter, `$CONTROL_PAR_CUSTOM_ID`, which allows assigning a custom value to every UI control.
- New internal constant, `$NI_UI_ID`, which returns the UI ID of the widget that triggered the UI callback.
- Engine parameter for Raum Predelay tempo sync, `$ENGINE_PAR_RAUM_PREDELAY_UNIT`.
- Engine parameter constants for 20 new Wavetable Form modes.

Improved Features

- get_event_mark() now correctly returns the state of the exact queried event mark. Previously the command didn't work correctly when multiple event marks were assigned to an event ID, and only a single event mark was queried.
- get_event_par_arr() used with `$EVENT_PAR_MOD_VALUE_(EX_)ID` always returned **`0`** in case an event held an active voice. Now it works identically regardless of voices being played or not.
- set_event_par() can now set `$EVENT_PAR_MIDI_CHANNEL` for events in instrument scripts.

Kontakt 7.5

New Features

- Engine parameters for the following new effects: **Bite**, **Dirt**, **Freak**, **Raum**, **Twin Delay**, **Vibrato/Chorus**, **Wow/Flutter**.
- Engine parameter for getting the Source module playback mode, `$ENGINE_PAR_SOURCE_MODE`.
- `continue` statement is now available.
- Real arrays can now be stored to and loaded from NKA files.
- Zone selection state can now be queried with `$ZONE_PAR_SELECTED` zone parameter, and indices of all selected zones can be collected with `get_sel_zones_idx()` command.
- New command for setting the color of script generated events displayed in Mapping Editor, `set_map_editor_event_color()`.

Improved Features

- Boolean `not` operator now correctly inverts a boolean expression that consists of only literals or constants.
- `exit` statement now correctly exits just the function in which it was called, instead of collapsing the whole function call stack (this happened only if `exit` was the first statement in a function).
- TMPro polyphony can now be set to **0** via `set_voice_limit()` command (previously the minimum value was **1**).

Kontakt 7.3

Improved Features

- `sort()` command now supports two additional arguments which specify the array bounds within which values will be sorted.
- `in_range()` command now also works with real expressions.

Kontakt 7.2

New Features

- Engine parameters for new Solid EQ parameters: `$ENGINE_PAR_SEQ_LP`, `$ENGINE_PAR_SEQ_HP`, `$ENGINE_PAR_SEQ_LP_FREQ`, `$ENGINE_PAR_HP_FREQ`.
- New engine parameters for several Source module parameters: `$ENGINE_PAR_TRACKING`, `$ENGINE_PAR_HQI_MODE`, `$ENGINE_PAR_S1200_FILTER_MODE`, `$ENGINE_PAR_TMPRO_KEEP_FORMANTS`.
- New command to retrieve filename of the last loaded MIDI file, `mf_get_last_filename()`.
- New constant which allows determining if Kontakt is running as a plugin or in standalone: `$NI_KONTAKT_IS_STANDALONE`.

Improved Features

- Major performance improvement of `set_sample()`, `set_zone_par()` and `set_loop_par()` commands (for more info, see descriptions of mentioned commands).
- `search()` command now supports two additional arguments which specify the array bounds that will be searched through.
- `$ENGINE_PAR_PHASE_INVERT` and `$ENGINE_PAR_LR_SWAP` can now also be pointed at instrument buses.
- Easier resource container updating via single click on Repack button in Instrument Options → Instrument tab, provided the resource container has already been assigned to the instrument.

Kontakt 7.1

New Features

- New commands to retrieve group, modulator and modulation target indices: `get_group_idx()`, `get_mod_idx()`, `get_target_idx()`. These commands will return `$NI_NOT_FOUND` when the queried object is not found. Introduction of these commands also deprecates the usage of `find_group()`, `find_mod()` and `find_target()`!
- Engine parameter for Convolution Auto Gain: `$ENGINE_PAR_IRC_AUTO_GAIN`
- Engine parameter for the Legato button across various Tone/Time Machine modes (`$ENGINE_PAR_TM_LEGATO`)
- Engine parameters for the new Flexible Envelope parameters: `$ENGINE_PAR_FLEXENV_LOOP`, `$ENGINE_PAR_FLEXENV_ONESHOT`
- Filter type constants for the new SV 6-pole filters: `$FILTER_TYPE_SV_LP6`, `$FILTER_TYPE_SV_BP6`, `$FILTER_TYPE_SV_HP6`, `$FILTER_TYPE_SV_NOTCH6`
- New command for querying the status of a zone: `get_zone_status()`. Introduction of this command also deprecates the usage of `is_zone_empty()`!
- New zone-related commands: `get_num_zones()` and `get_zone_id()`
- New zone parameters to get the sample rate of the sample used by the zone (`$ZONE_PAR_SAMPLE_RATE`) and get or set the zone BPM (`$ZONE_PAR_BPM`)
- XY pad values can now be retrieved and set using UI IDs (`get_control_par_real_arr(), set_control_par_real_arr()`)
- New real number math commands: `cbrt()`, `log2()`, `log10()`, `exp2()`
- New basic operators dealing with the negative sign: `signbit()`, `sgn()`
- New boolean operator: `xor`
- New bitwise operator: `.xor.`
- New, shorter commands for integer/real value conversion: `int()`, `real()`. Introduction of these commands also deprecates the usage of `real_to_int()` and `int_to_real()`

Improved Features

- Added mode **2** to `change_pan()` and `change_vol()`, which retains the zone pan/volume adjustments on top of the absolute offset added by `change_pan()` or `change_vol()`, unlike mode **0** which disregarded zone pan/volume adjustments entirely
- The `set_zone_par()` command now works on all zones when using snapshot modes **2** and **3**, from any callback type
- The `mod` basic operator now also works with real numbers, providing remainder after regular floating-point division

Kontakt 7

New Features

- Engine parameter for adjusting LFO phase, `$ENGINE_PAR_LFO_PHASE`
- Engine parameters for adjusting step modulator parameters: `$ENGINE_PAR_STEPSEQ_NUM_STEPS`, `$ENGINE_PAR_STEPSEQ_ONESHOT`, `$ENGINE_PAR_STEPSEQ_STEP_VALUE`
- Engine parameter for bipolar adjustment of modulation amount, `$ENGINE_PAR_MOD_TARGET_MP_INTENSITY`
- Engine parameters for the following new effects: **PsycheDelay**, **Ring Modulator**
- ui_mouse_area now responds to `$CONTROL_PAR_KEY_CONTROL`, `$CONTROL_PAR_KEY_SHIFT`, `$CONTROL_PAR_KEY_ALT` control parameters

Improved Features

- Increased number of user zones to **1024**
- `$EVENT_PAR_MOD_VALUE_ID` can now be retrieved by using `get_event_par_arr()`

Kontakt 6.7.0

New Features

- New snapshot types for saving only the persistent KSP variables with snapshots
- Engine parameters for the following new effects: **Bass Invader**, **Bass Pro**

Kontakt 6.6.0

New Features

- New functionality enabling "from script" modulation assignments in Kontakt with new built-in variable to be used with `set_event_par_arr()`: `$EVENT_PAR_MOD_VALUE_ID`
- New command for redirecting the audio signal of an event to an output or bus: `redirect_output()`
- New built-in variables to be used with `redirect_output()`:
  `$OUTPUT_TYPE_DEFAULT`, `$OUTPUT_TYPE_MASTER_OUT`, `$OUTPUT_TYPE_AUX_OUT`, `$OUTPUT_TYPE_BUS_OUT`
- The ui_level_meter can be attached to gain reduction data (i.e. from compressor and limiter effects)
- New built-in variables for setting the display range of gain reduction meters: `$CONTROL_PAR_RANGE_MIN`, `$CONTROL_PAR_RANGE_MAX`

Improved Features

- `$CONTROL_PAR_BASEPATH` can be set from anywhere in the script, updating the file selector even if the path did not change

Kontakt 6.5.0

New Features

- New command for checking the bit-mark of an event: `get_event_mark()`
- New built-in variable for checking if Kontakt is loaded without GUI engine (i.e. on Maschine+): `$NI_Kontakt_IS_HEADLESS`
- Added engine parameters for many existing effects:
  - Solid Bus Comp: Link button (`$ENGINE_PAR_SCOMP_LINK`)
  - Classic Compressor: Stereo Link button (`$ENGINE_PAR_COMP_LINK`) and compressor type menu (`$ENGINE_PAR_COMP_TYPE` -> `$NI_COMP_TYPE_CLASSIC`, `$NI_COMP_TYPE_ENHANCED`, `$NI_COMP_TYPE_PRO`)
  - Stereo Modeller: Pseudo Stereo button (`$ENGINE_PAR_STEREO_PSEUDO`)
  - Convolution: Reverse button (`$ENGINE_PAR_IRC_REVERSE`), early reflection/late reflection divider (`$ENGINE_PAR_IRC_ER_LR_BOUNDARY`)
  - Group FX: Post Amp FX slider (`$ENGINE_PAR_POST_FX_SLOT`)
  - Envelope Follower (`$ENGINE_PAR_ENVF_ATTACK`, `$ENGINE_PAR_ENVF_RELEASE`, `$ENGINE_PAR_ENVF_GAIN_BOOST`, `$ENGINE_PAR_ENVF_ADAPTION`)
  - Various others (`$ENGINE_PAR_ENV_AHD`, `$ENGINE_PAR_ENV_DBD_EASY`, `$ENGINE_PAR_LFO_NORMALIZE`)

Improved Features

- The maximum number of UI controls has been increased to **999** per UI control type (except for the UI file selector)

Kontakt 6.4.0

New Features

- New Main Effects signal processing module
- New engine parameters for new **Supercharger GT** and **Transparent Limiter** effects
- New constants for the `<generic>` argument when setting and getting engine parameters: `$NI_SEND_BUS`, `$NI_INSERT_BUS`, `$NI_MAIN_BUS`
- New constant that defines which area should be used when dragging from a specific label: `$CONTROL_PAR_MIDI_EXPORT_AREA_IDX`
- New command that defines the number of MIDI object export areas: `mf_set_num_export_areas()`
- New command to manage the usage of the new additional export areas: `mf_copy_export_area()`
- New engine parameters for Inverter and Amplifier parameters for Phase Invert and L/R swap: `$ENGINE_PAR_PHASE_INVERT`, `$ENGINE_PAR_LR_SWAP`
- New constant allows up to **16** custom event parameters to be assigned: `$EVENT_PAR_CUSTOM`

Improved Features

- The number of maximum MIDI object export areas has been increased to **512**

Kontakt 6.3.0

New Features

- New constant for handling release velocity: `$EVENT_PAR_REL_VELOCITY`
- New constant for hiding the value display of ui_table: `$HIDE_PART_VALUE`

Kontakt 6.2.0

New Features

- New **Choral**, **Flair** and **Phasis** modulation effects.
- New UI widget: `ui_mouse_area`
- New type of zones accessible from KSP: `set_num_user_zones()`, `set_sample()`, `set_zone_par()`, `set_loop_par()`
- All zone parameters can now be read from KSP: `get_sample()`, `get_zone_par()`, `get_loop_par()`
- New function to check whether a sample is loaded for a zone: `is_zone_empty()`
- New MIR functions to detect pitch, RMS, peak level and loudness of samples, and to classify samples based on their audio characteristics.
- New command to make handling asynchronous operations more convenient: `wait_async()`

Improved Features

- `purge_group()` now returns an async ID, allowing for reliable tracking of the operation's completion.

Kontakt 6.1.0

New Features

- New engine parameter for the Retrigger button on internal modulators (`$ENGINE_PAR_INTMOD_RETRIGGER`)
- New waveform visualization modes (`$CONTROL_PAR_WF_VIS_MODE` with `$NI_WF_VIS_MODE_1, $NI_WF_VIS_MODE_2` and `$NI_WF_VIS_MODE_3` as values)
- New Wavetable Mode (`$ENGINE_PAR_WT_INHARMONIC_MODE`)
- New UI widget (`ui_panel`) and related control parameter (`$CONTROL_PAR_PARENT_PANEL`)
- New user interface command (`load_performance_view()`) to load performance views created in Creator Tools

Kontakt 6.0.2

New Features

- Engine parameters for the new Kontakt 6 effects: **Replika**, **(Galois) Reverb**, **Plate Reverb**.
- Engine parameters for the new Wavetable sampler mode.
- New UI widget: `ui_wavetable` including new commands and built-in variables.
- New commands for variable watching through Creator Tools: `watch_var()` and `watch_array_idx()`
- New control parameter allows deactivating text position shifts when clicking on buttons and switches: `$CONTROL_PAR_DISABLE_TEXT_SHIFTING`
- New command enables use of custom dynamic fonts: `get_font_id()`
- New control parameters allow granular control over font types for different button and menu states: `$CONTROL_PAR_FONT_TYPE_ON`, `$CONTROL_PAR_FONT_TYPE_OFF_PRESSED`, `$CONTROL_PAR_FONT_TYPE_ON_PRESSED`, `$CONTROL_PAR_FONT_TYPE_OFF_HOVER` and `$CONTROL_PAR_FONT_TYPE_ON_HOVER`
- New command allowing for quickly disabling emission of messages, warnings or watched variable events to both the Kontakt status bar and Creator Tools: `disable_logging()` with one of the following as the: `$NI_LOG_MESSAGE`, `$NI_LOG_WARNING`, `$NI_LOG_WATCHING`

Improved Features

- New built-in variable and related built-in constants for the XY Pad allow identification of the mouse events that trigger its callback: `$NI_MOUSE_EVENT_TYPE`, `$NI_MOUSE_EVENT_TYPE_LEFT_BUTTON_DOWN`, `$NI_MOUSE_EVENT_TYPE_LEFT_BUTTON_UP` and `$NI_MOUSE_EVENT_TYPE_DRAG`
- `$CONTROL_PAR_TEXTPOS_Y` is now allowed on value edit controls.

Kontakt 5.8.0

Improved Features

- It is now possible to have up to **three** file selectors per script slot.
- The maximum number of controls per type has now been raised to **512**.
- The maximum size for an array has now been raised to **1000000**.

Kontakt 5.7.0

New Features

- New control parameter for all UI widgets: `$CONTROL_PAR_Z_LAYER`
- Waveform styling options: `$CONTROL_PAR_WAVE_COLOR`, `$CONTROL_PAR_BG_COLOR`, `$CONTROL_PAR_WAVE_CURSOR_COLOR`, `$CONTROL_PAR_SLICEMARKERS_COLOR`, `$CONTROL_PAR_BG_ALPHA`
- Engine parameter variables for new effects: **ACBox**, **Cat**, **DStortion**, **HotSolo**, **Van51**.
- Added engine parameter variables for various on-off effect parameters.
- Added engine parameter variables for setting the subtype for the **Distortion** and **Saturator** effects: `$ENGINE_PAR_DISTORTION_TYPE`, `$ENGINE_PAR_SHAPE_TYPE`

Improved Features

- ui_waveform now accepts `$HIDE_PART_BG` as a `hide_part()` and `$CONTROL_PAR_HIDE` constant.

Kontakt 5.6.8

New Features

- New built-in control parameters: `$NI_CONTROL_PAR_IDX`, `$HIDE_PART_CURSOR`

Kontakt 5.6.5

New Features

- New UI widget: `ui_xy`, including new built-in variables: `$CONTROL_PAR_CURSOR_PICTURE`, `$CONTROL_PAR_MOUSE_MODE`, `$CONTROL_PAR_ACTIVE_INDEX`, `$CONTROL_PAR_MOUSE_BEHAVIOUR_X`, `$CONTROL_PAR_MOUSE_BEHAVIOUR_Y`
- New UI commands: `set_control_par_arr()` and `set_control_par_str_arr()`

Kontakt 5.6.0

New Features

- Support for real numbers, including new `~realVariable` and `?realArray[]` variable types.
- Additional mathematical commands for real numbers.
- New constants: `~NI_MATH_PI` and `~NI_MATH_E`
- New UI commands: `set_ui_color()` and `set_ui_width_px()`
- New control parameter for setting automation IDs via KSP: `$CONTROL_PAR_AUTOMATION_ID`

Kontakt 5.5.0

New Features

- New engine parameter variables and built-in constants for controlling the unit parameter of time-related parameters, e. g, `$ENGINE_PAR_DL_TIME_UNIT`, `$NI_SYNC_UNIT_8TH`
- It is now possible to change FX from KSP by using engine parameter variables for effect type, e. g. `set_engine_par($ENGINE_PAR_EFFECT_TYPE,$EFFECT_TYPE_FILTER, 0, 0, -1)`. For more information, refer to Changing FX from KSP.
- It is now possible to set Time Machine Pro voice settings: `set_voice_limit()`, `get_voice_limit()`, `$NI_VL_TMPRO_STANDARD`, `$NI_VL_TMRPO_HQ`

Kontakt 5.4.2

Improvements

- Various corrections to the KSP reference manual.

Kontakt 5.4.1

New Features

- New callback type: `on persistence_changed`
- New command: `set_snapshot_type()`
- New command: `make_instr_persistent()`
- New key color constants and command: `get_key_color()`
- Ability to set the pressed state of Kontakt's keyboard: `set_key_pressed()`, `set_key_pressed_support()`, `get_key_triggerstate()`
- Ability to specify key names and ranges: `set_key_name()`, `get_key_name()`, `set_keyrange()`, `remove_keyrange()`
- Ability to specify key types: `set_key_type()`, `get_key_type()`

Improved Features

- `Data` folder in resource container, additional mode **2** for `load_array()`
- `load_array_str()` can now be used in more callback types.

Kontakt 5.3.0

New Features

- Added engine parameters for the new **Simple LP/HP** filter.

Kontakt 5.2.0

New Features

- New commands to insert and remove MIDI events.

Improved Features

- Updated MIDI file handling.

Kontakt 5.1.1

New Features

- Added engine parameters for the new **Feedback Compressor** effect.

Kontakt 5.1.0

New Features

- New commands: `load_array_str()`, `save_array_str()`
- Added engine parameters for the new **Jump Amp** effect.

Improvements

- Various corrections and improvements to the KSP reference manual.

Kontakt 5.0.2

New Features

- New engine parameters for Time Machine Pro in HQ Mode):
  `$ENGINE_PAR_ENVELOPE_ORDER`, `$ENGINE_PAR_FORMANT_SHIFT`

Kontakt 5.0.1

New Features

- Added effect type and sub-type constants for the new Kontakt 5 effects.

Kontakt 5

New Features

- MIDI file support including many new commands: `load_midi_file()`, `save_midi_file()`, `mf_get_num_tracks()`, `mf_get_first()`, `mf_get_next()`, `mf_get_next_at()`, `mf_get_last()`, `mf_get_prev()`, `mf_get_prev_at()`, `mf_get_channel()`, `mf_get_command()`, `mf_get_byte_one()`, `mf_get_byte_two()`, `mf_get_pos()`, `mf_get_track_idx()`, `mf_set_channel()`, `mf_set_command()`, `mf_set_byte_one()`, `mf_set_byte_two()`, `mf_set_pos()`
- New UI widget: `ui_text_edit`
- New UI widget: `ui_level_meter`, including new commands and control parameters: `attach_level_meter()`, `$CONTROL_PAR_BG_COLOR`, `$CONTROL_PAR_OFF_COLOR`, `$CONTROL_PAR_ON_COLOR`, `$CONTROL_PAR_OVERLOAD_COLOR`, `$CONTROL_PAR_PEAK_COLOR`, `$CONTROL_PAR_VERTICAL`
- New UI widget: `ui_file_selector`, including new commands and control parameters: `fs_get_filename()`, `fs_navigate()`, `$CONTROL_PAR_BASEPATH`, `$CONTROL_PAR_COLUMN_WIDTH`, `$CONTROL_PAR_FILEPATH`, `$CONTROL_PAR_FILE_TYPE`
- New commands for dynamic dropdown menus: `get_menu_item_value()`, `get_menu_item_str()`, `get_menu_item_visibility()`, `set_menu_item_value()`, `set_menu_item_str()`, `set_menu_item_visibility()`, `$CONTROL_PAR_SELECTED_ITEM_IDX`, `$CONTROL_PAR_NUM_ITEMS`
- New callback type: `on async_complete`, including new built-in variables: `$NI_ASYNC_ID`, `$NI_ASYNC_EXIT_STATUS`, `$NI_CB_TYPE_ASYNC_OUT`
- New internal constant for Kontakt's new instrument bus effect chains: `$NI_BUS_OFFSET`
- New engine parameters for new Kontakt 5 effects.
- New commands: `wait_ticks()`, `stop_wait()`

Improved Features

- Support for string arrays added to `load_array()` and `save_array()`
- PGS support for strings: `pgs_create_str_key()`, `pgs_str_key_exists()`, `pgs_set_str_key_val()`, `pgs_get_str_key_val()`
- The maximum height of `set_ui_height_px()` is now **540** pixels.

Kontakt 4.2.0

New Features

- Resource Container, a helpful tool for creating instrument libraries.
- New ID to set wallpapers via script: `$INST_WALLPAPER_ID`
- New key color: `$KEY_COLOR_BLACK`
- New callback type: `on listener`
- New commands for this callback: `set_listener()`, `change_listener_par()`
- New commands for saving or loading arrays: `save_array()`, `load_array()`
- New command to check the purge status of a group: `get_purge_state()`
- New built-in variable: `$NI_SONG_POSITION`
- New control parameter: `$CONTROL_PAR_ALLOW_AUTOMATION`

Improved Features

- The script editor is now much more efficient, especially with large scripts.
- New UI widget limit: **256** (per widget type and script slot).
- Event parameters can now be used without affecting the system scripts.

Kontakt 4.1.2

New Features

- New UI widget: `ui_waveform`
- New commands for this UI widget: `set_ui_wf_property()`, `get_ui_wf_property()`, `attach_zone()`
- New control parameters to be used with these commands: `$UI_WAVEFORM_USE_SLICES`, `$UI_WAVEFORM_USE_TABLE`, `$UI_WAVEFORM_TABLE_IS_BIPOLAR`, `$UI_WAVEFORM_USE_MIDI_DRAG`, `$UI_WF_PROP_PLAY_CURSOR`, `$UI_WF_PROP_FLAGS`, `$UI_WF_PROP_TABLE_VAL`, `$UI_WF_PROP_TABLE_IDX_HIGHLIGHT`, `$UI_WF_PROP_MIDI_DRAG_START_NOTE`
- New event parameter: `$EVENT_PAR_PLAY_POS`

Kontakt 4.1.1

Improved Features

- The built-in variables `$SIGNATURE_NUM` and `$SIGNATURE_DENOM` don't reset to 4/4 if the host's transport is stopped

Kontakt 4.1.0

New Features

- Implementation of user-defined functions: `function`
- New control parameter: `$CONTROL_PAR_AUTOMATION_NAME`
- New command: `delete_event_mark()`
- Support for polyphonic aftertouch: `on poly_at`, `%POLY_AT[]`, `$POLY_AT_NUM`
- New command: `get_event_ids()`
- New control parameters: `$CONTROL_PAR_KEY_SHIFT`, `$CONTROL_PAR_KEY_ALT`, `$CONTROL_PAR_KEY_CONTROL`

Improved Features

- The built-in variable `$MIDI_CHANNEL` is now also supported in the instrument script.
- The sample offset parameter in `play_note()` now also works in DFD mode, according to the **S. Mod** value set for the respective zone in Kontakt's Wave Editor
- KSP reference corrections for the modulation engine parameters

Kontakt 4.0.2

New Features

- New engine parameter to set the group output channel: `$ENGINE_PAR_OUTPUT_CHANNEL`
- New built-in variable: `$NUM_OUTPUT_CHANNELS`
- New function: `output_channel_name()`
- New built-in variable: `$CURRENT_SCRIPT_SLOT`
- New built-in variable: `$EVENT_PAR_SOURCE`

Improved Features

- The `load_ir_sample()` command now also accepts single file names for loading IR samples into Kontakt's Convolution effect, i.e. without a path designation. In this case the sample is expected to reside in the folder called `ir_samples` inside the user folder.

Kontakt 4

New Features

- Multi script
- New ID-based user interface system: `set_control_par()`, `get_control_par()` and `get_ui_id()`
- Pixel-exact positioning and resizing of UI widgets.
- Skinning of UI widgets.
- New UI widgets: `ui_switch` and `ui_slider`.
- Assign colors to Kontakt's keyboard by using `set_key_color()`
- New timing variable: `$KSP_TIMER` (in microseconds).
- New path variable: `$GET_FOLDER_FACTORY_DIR`
- New hide constants: `$HIDE_PART_NOTHING` and `$HIDE_WHOLE_CONTROL`
- Scripts can be linked to text files.

Improved Features

- New array size limit: **32768**
- Retrieve and set event parameters for tuning, volume and pan of an event: `$EVENT_PAR_TUNE`, `$EVENT_PAR_VOL` and `$EVENT_PAR_PAN`
- Larger performance view size: `set_ui_height()`, `set_script_title()`
- Beginning underscores from Kontakt 2/3 commands like `_set_engine_par()` can now be omitted, i.e. you can write `set_engine_par()` instead.

Kontakt 3.5.0

New Features

- Retrieve the status of a particular event: `event_status()`
- Hide specific parts of UI controls: `hide_part()`

Improved Features

- Support for channel aftertouch: `$VCC_MONO_AT`
- New array size limit: **2048**

Kontakt 3

New Features

- Offset for wallpaper graphic: `_set_skin_offset()`
- Program Global Storage (PGS) for inter-script communication: `_pgs_create_key()`, `_pgs_key_exists()`, `_pgs_set_key_val()`, `_pgs_get_key_val()`
- New callback type: `on _pgs_changed`
- Addressing modulators by name: `find_mod()` and `find_target()`
- Change the number of displayed steps in a column: `set_table_steps_shown()`
- Info tags for UI controls: `set_control_help()`

Improved Features

- All five performance views can now be displayed together.

Kontakt 2.2.0

New Features

- New callback type: `on ui_update`
- New built-in variables for group-based scripting: `$REF_GROUP_IDX` and `%GROUPS_SELECTED`
- Ability to create custom group start options: `NO_SYS_SCRIPT_GROUP_START`, along with various Group Start Options variables.
- Retrieving the release trigger state of a group: `$ENGINE_PAR_RELEASE_TRIGGER`
- Default values for knobs: `set_knob_defval()`

Kontakt 2.1.1

New Features

- Assign unit marks to knobs: `set_knob_unit()`
- Assign text strings to knobs: `set_knob_label()`
- Retrieve the knob display: `_get_engine_par_disp()`

Kontakt 2.1.0

New Features

- String arrays (**!** prefix) and string variables (**@** prefix)
- Engine parameters: `_set_engine_par()`
- Loading IR samples: `_load_ir_sample()`
- Performance View: `make_perfview`
- RPN/NRPN implementation: `on rpn`, `on nrpn`, `$RPN_ADDRESS $RPN_VALUE`, `msb()`, `lsb()`, `set_rpn()` and `set_nrpn()`
- Event parameters: `set_event_par()`
- New built-in variables: `$NUM_GROUPS`, `$NUM_ZONES`, `$VCC_PITCH_BEND`, `$PLAYED_VOICES_TOTAL`, `$PLAYED_VOICES_INST`

Improved Features

- It is now possible to name UI widgets with `set_text()`
- Moving and hiding UI widgets: `move_control()`
- MIDI CCs generated by `set_controller()` can now also be used for automation as well as modulation.

Kontakt 2

Initial release.