Table of Contents

I Introduction

MIDI Keyboard Macros is a sister application to our Multi-Keyboard Macros.
Instead of using multiple keyboards, it allows you to use various MIDI devices as your Macro keyboard trigger, including sliders and knobs

What are Macro Keyboards:
You may have seen some shortcut or macro hardware keyboards.
These are very useful when working with complex applications - for example drawing applications, 3D applications, video editing, audio or music application where your hand (or even both hands) are not always near the keyboard. Nothing is more distracting than to take your hands from the controller, mouse, music keyboard or stylus just to type CTRL+U which for most of us require two hands.

Usual macro keyboards are expensive, need custom drivers and are often tied to a certain macro application - which may suit you or not. Not to mention the elephant in the room - which happened far too many times for the author: most of such keyboards require special drivers and would become obsolete when the developer no longer decide to support new operating systems.

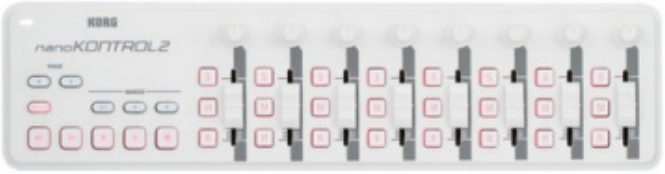
This is what the MIDI Macro application is about:
You can utilize your MIDI keyboards, pads or controllers in any application: graphics, video editing, text processing... but also much more!

1.1 Overview

With Midi keyboard Macros you can turn your MIDI pads, MIDI keyboards or controllers into a most sophisticated Macro keyboard, launchers, text processors and much more.

For example: plug in additional MIDI pad and redefine every single pad to do something else - shortcuts, macros, type text or even simulate mouse click. Great for Photoshop, video editing apps etc...

A full scripting language allows you to process clipboard and do a custom keyboard logic that no other macro keyboard will allow.



MIDI Macros can also understand controllers - such as sliders or knobs and with the Script you can define some interesting logic that combines rotation controllers, sliders or even key velocity.

1.2 Features

Allows you to re-define pad, key or controller across multiple MIDI devices to do something else:

- Macros are tied to a device so multiple devices can trigger different macros.
- define simple shortcuts that are triggered by pressing certain key: for example pressing PAD 4 on MIDI pad will send CTRL+C
- define Macros which are whole sequences of such shortcuts, so it can be CTR+C followed by 3 times right arrow, followed by CTRL+V
- simulate mouse click within macros
- type whole text (signature, greetings etc...) by pressing a single key
- run application
- open folder or file
- open web page
- record keystrokes and then play them back as quick macro (software developers are quite familiar with this type)

New in version 2.0:
Full scripting language:
Every key can now run a **script** - or multiple scripts - or combination of any of the steps from previous version and script.The scripting language can also work with clipboard data.

An Example:
a single macro key key could:

- send CTR+C to capture selected text under cursor
- process the text with full and rich suite of string operations (including string tokenizer, tag extraction and full regex)
- type it changed back to the application.

all with pressing just a single key

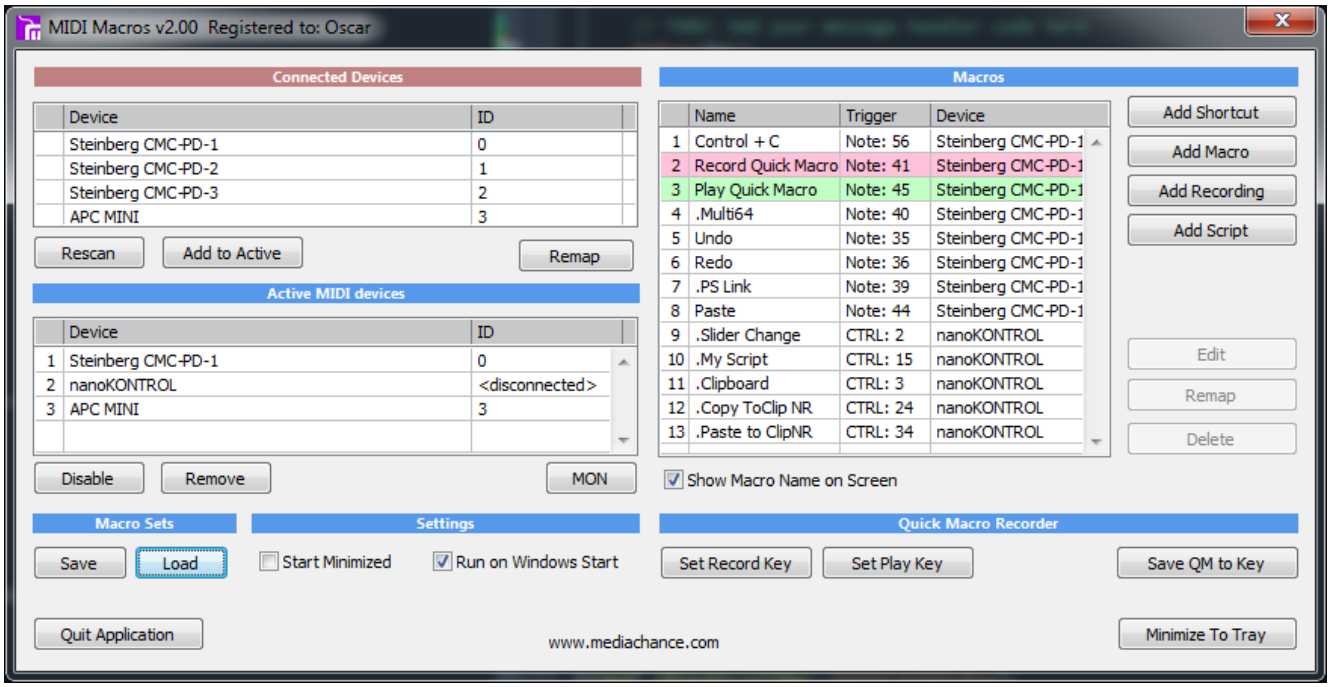
An instant text processor where only the sky is the limit.

Scripts can talk to each other through global variables so another obvious function can be to use some of the keys as modifiers.
If I press 7 then quickly 8 on a numerical keyboard it can do different things than pressing 8 alone.

Of course those are just a few ideas. The script language is incredibly rich and extensive. It can work on arrays, it can load and save text files and more. We can also enhance it with more functionality easily in the future if there is need.

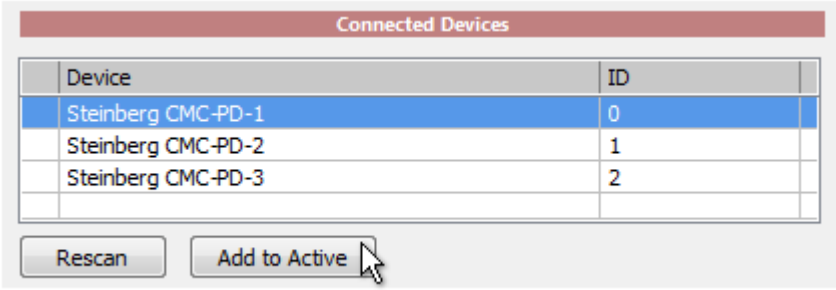
II Basics

This is the software interface



Initial setup:

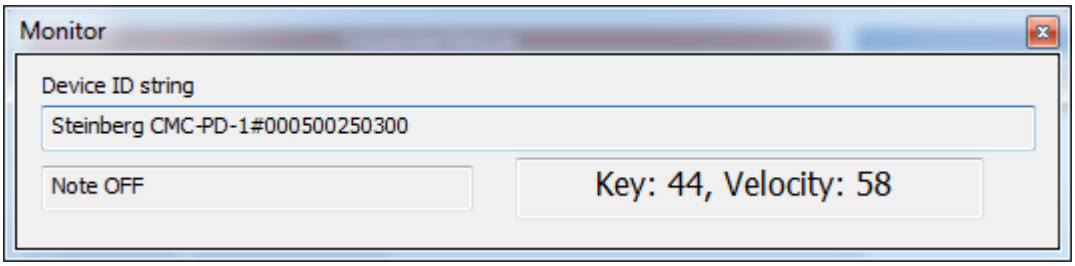
First thing we need to specify which MIDI devices we will use for macros, so plug them in your computer and wait for them to appear in the Connected Devices. Some devices need a driver, you would need to install that first.



Select the device you want to use for Macros and click Add to Active so it will appear in the Active Midi devices list

Some devices may offer multiple interfaces (as in the case above). You generally need to add only the first one in such case.

As soon as the Midi device is added to Active list, it should be detected by the application. You can click on the MON button to open monitor and tap few pads.



If nothing is detected and the device offers multiple interfaces, try another interface.

MIDI Macros works with each device separately - that means a note 35 on MIDI device A can do a different macro than the same note on a different MIDI device B.
(Please note if a keyboard has both pads and keys,it is still a single device and the pads are simply mapped to the keys)

The MIDI device can be used for both your MUSIC application and a macro trigger at the same time, obviously if the keys don't overlap. For example you may map controls and buttons that you normally don't use for playing music to do some other stuff such as Undo/ Redo etc...
However, you may easily disable a MIDI device by selecting it in the Active Midi list and using Disable button.

2.1 Shortkeys

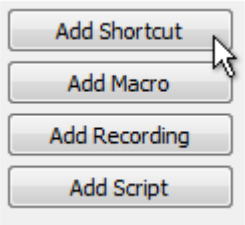
When the initial setup is done, now it is time to add some shortkeys.

It is important to note that MIDI macros will allow you to define shortkeys and macros on per MIDI device basis. That means they will be tied to the particular device. This allows you to add multiple MIDI devices and define the same "note" differently. However these devices should be a different models to be clearly recognizable as separate.

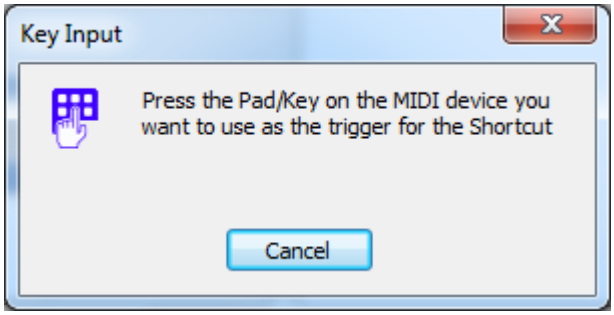
A shortkey or shortcut can be anything, even just a letter.

Sending a shortkey/shortcut is the simplest method.

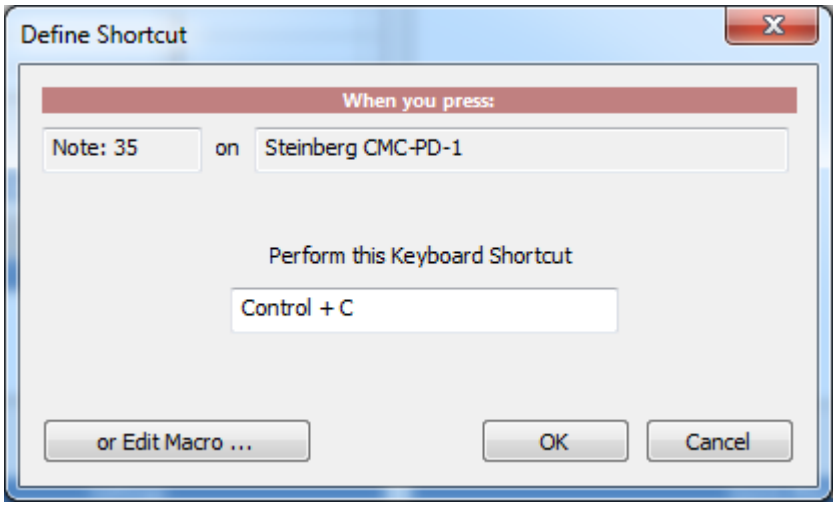
Press Add Shortcut (it could be a good idea in the future if we are clear how to call these -but for now shortcuts and shortkeys are the same thing)



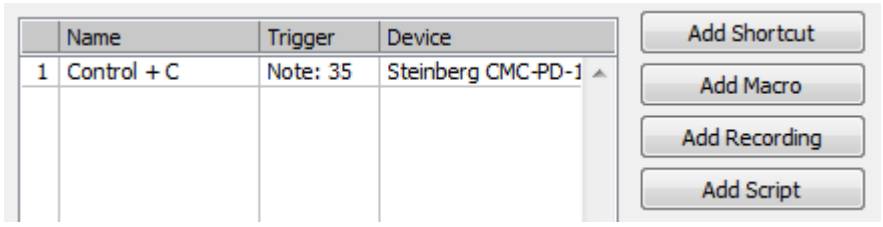
This will follow with a familiar procedure, press the trigger key - that is the key on the keyboard you want to define.



For example I'll tap a first PAD on my MIDI keyboard.
This will get me to the actual definition of the Shortkey (shortcut) - that is what I want my * key to actually do.

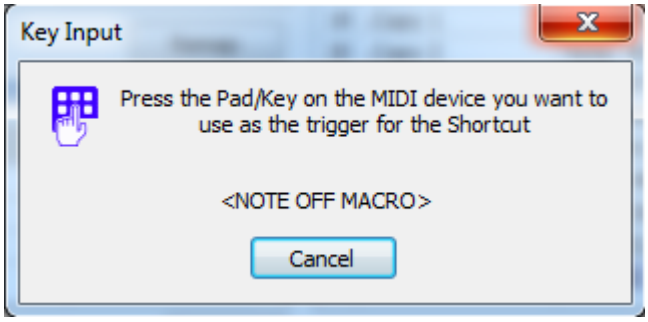


In this case I want it to send copy command - which is CTRL+C. So I Hold that combination. Then press OK. The Shortcut will be added to the list of shortcuts and macros.



Note OFF macro

MIDI macros allows you to capture also the Note OFF

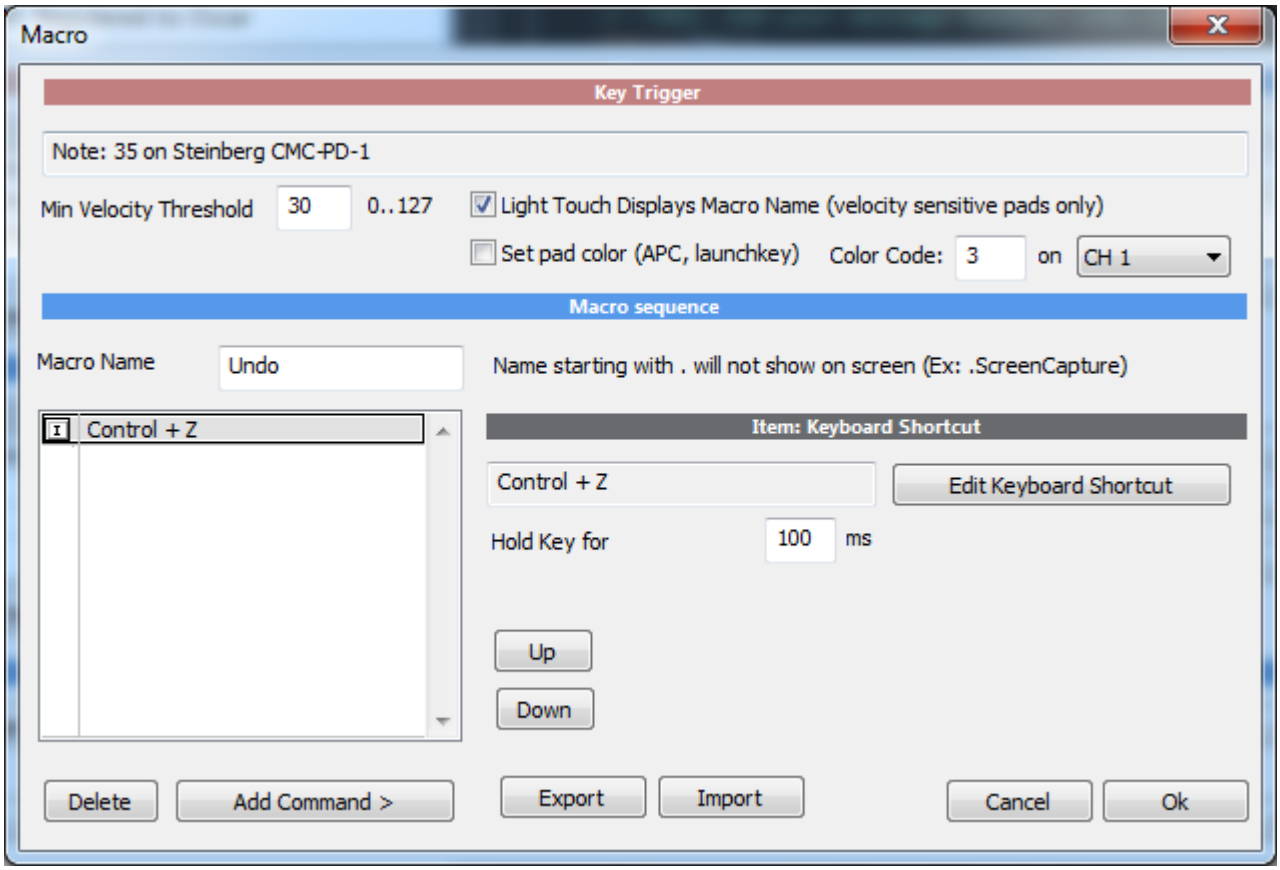


If you hold CTRL during Key Input, the created macro will be for NOTE OFF. This is mostly useful for script if you want to create a NOTE OFF counterpart to Note ON trigger (for example change color of the button on controllers that support it.

2.2 Macros

Shortcut is just a single step - like pressing CTRL+C
Macro is a **sequence** of many of such steps. Shortcut is in fact a Macro with only single step in it.

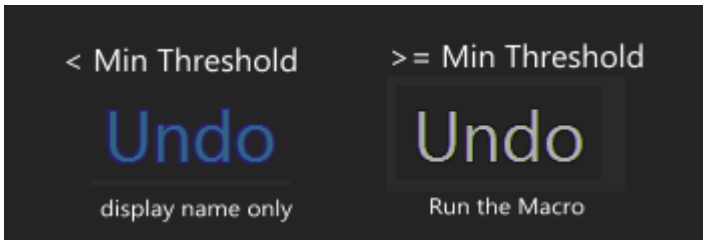
To define macro is a similar procedure to Shortcut except now we press Add Macro.
After setting the trigger key (as in Shortcut) we will get into a bit different window where we can define the steps.



Macro name: this is the name that will appear on the list and also can appear as a label on screen if that option is used. If the macro name starts with dot '.' then the Name display will be suppressed. (This is used for example if the trigger is to capture screen or if a script shows its own labels.

Min Velocity Threshold: Some pads are very sensitive, a default velocity of 30 is automatically used (the key or pad will be ignored if the velocity is less than the Threshold). You can set it to other value. When you are adding control (such as slider or knob) the Threshold will be set to 0

Light Touch Displays Macro Name
A light touch on the pad or key - bellow the Min Velocity Threshold - will only display the Macro name on the screen but not actually run the macro itself until it is triggered with the velocity larger than Min Velocity Threshold. Light Touch doesn't depend on **Show Macro Name on Screen** settings



This is great as a way to locate keys with a light brush of your fingers on the pads.It works on velocity sensitive pads only: Note: many cheaper launchpad type controllers (APC mini, Launchpad MK2) are not velocity sensitive.

Set Pad Color
Launchpad/matrix type controllers (such as APC mini) can have buttons light up in one of the supported colors. You can set the color in Color Code. Each device has different way of setting colors, so please refer to your device programmers manual. Some have 3 colors LEDs, some offer full RGB colors so the number depends on the device.

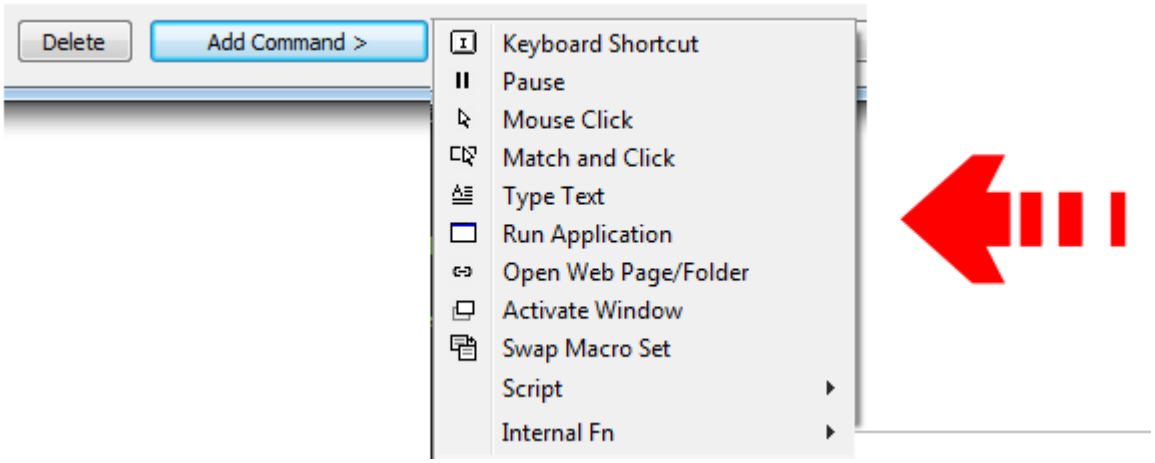
Example Colors: APC Mini
0 - off
1 - green
2 - green blink
3 - red
4 - red blink
5 - yellow
6 - yellow blink

Channel is for setting device channel for the color information - usually CH1 is the default on most devices.

When you add macros for few known devices such as APC mini, the Set Pad Color may be checked on automatically.

Note: do not set it for devices that don't support color on keys/pads since this may then send incorrect message to the device

On the left is a list of Steps - and there is nothing yet.
To add a step, press **Add Command**

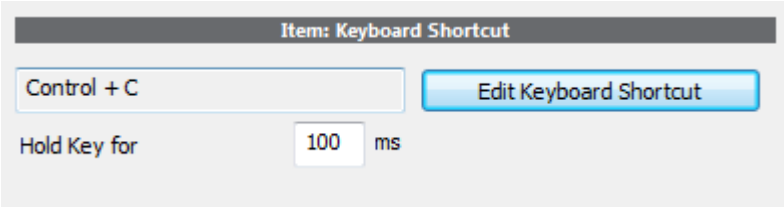


This opens a list of available commands that we can stack together or use single as a single command. Some commands are Internal function commands and have FN prefix. Those will perform only a single function that is related to the app itself such as Record Quick macro 19.

Adding command will also show its available settings on the right side

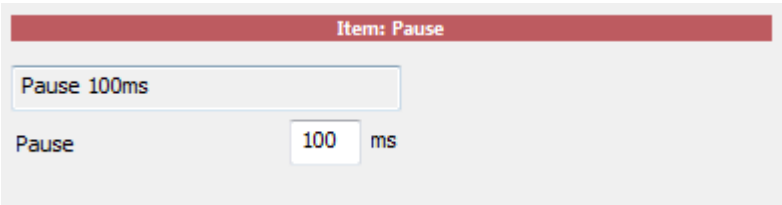
The commands:

Keyboard Shortcut
This is our familiar shortcut 6 (that we also call shortkey to confuse everyone). It will perform one key combination and hold that key for certain amount of time. 100ms is a good number.



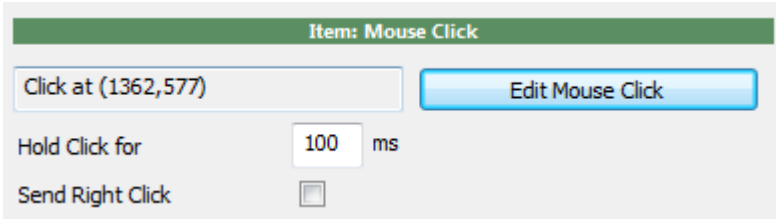
Note: This will trigger a typical simple shortcuts that are used in software: such as CTRL+C. You can simply add multiple keyboard shortcut macros in case of sequential shortcuts.
If you need more complex shortcuts, you need to use script with `SendKeyStroke` command. This allows you to also control hold and release keys, control right and left shift/alt/ctrl and other tricks. For example of holding a key while holding a trigger see example in KEY_OFF macroblock. 113

Pause
This adds a pause of ms. Sometimes application may not accept shortkeys or keys if they are fired too quickly after each other so we can set a pause between commands.

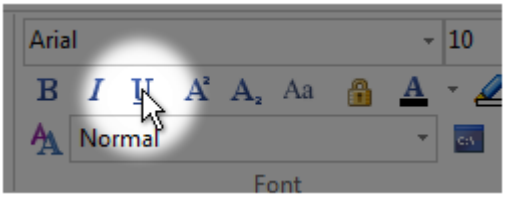


Mouse Click
Simulates mouse click on absolute coordinates on the screen. This can be used in software that doesn't have any shortcut command for a function we want but it has a button on the interface. The said software window has

to be on the same position every time because the coordinates are absolute- so probably best is to work on it maximized.



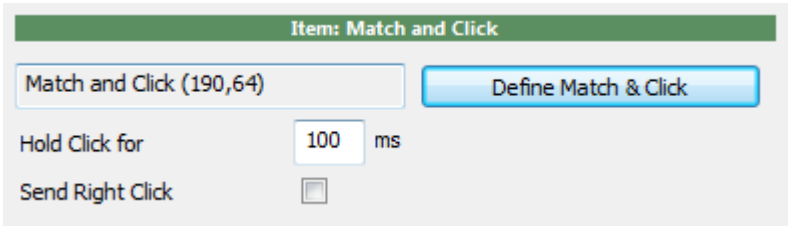
Once you click Edit Mouse click you can visually click on certain part of the host application



Match and Click

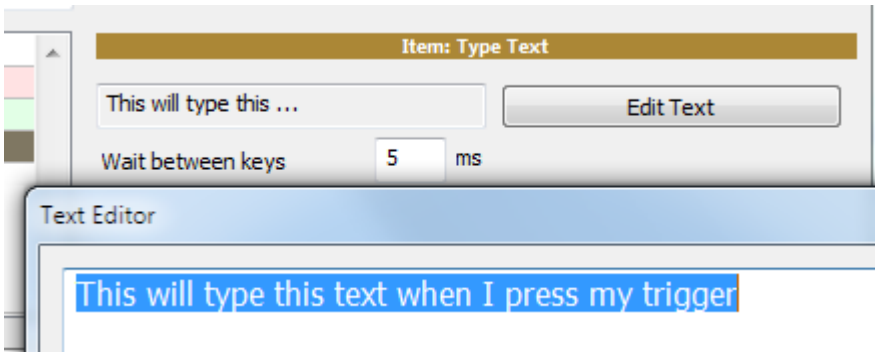
The Mouse Click function requires that the button or item we are clicking on stays on the same position. That is not always the case and on web pages certain items cannot even be assured to be on the exact position even if we maximize browser window. A pattern matching function would locate (match) predefined "Anchor" area on the screen then click on point relative to that area.

This function is for recognizing where are things on the screen - for example a button or form on a web page, or a menu item on a window that may be moved to different place.

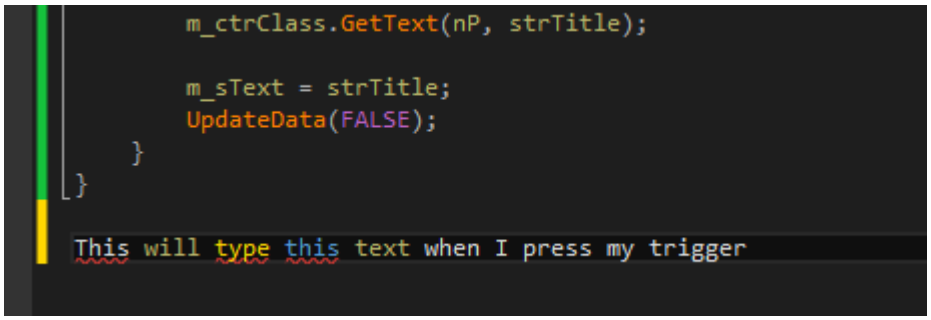


This function is little more complex so it has its own chapter 17.

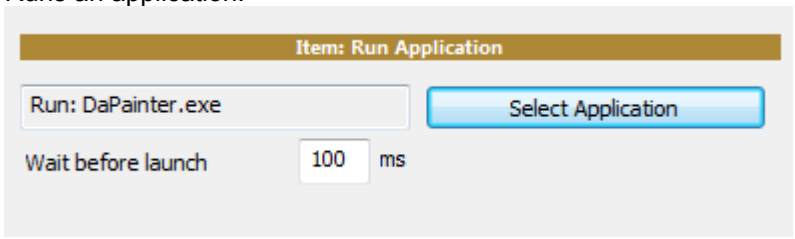
Type Text
Types text in the host application. This could be an address, greetings, footer, recipe for a cookie or whatever you need to type often.



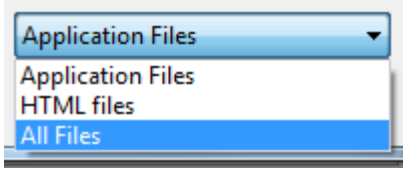
You can specify 'wait' between keys. 5ms will type it pretty fast. Now pressing my trigger in any text application will type that text.



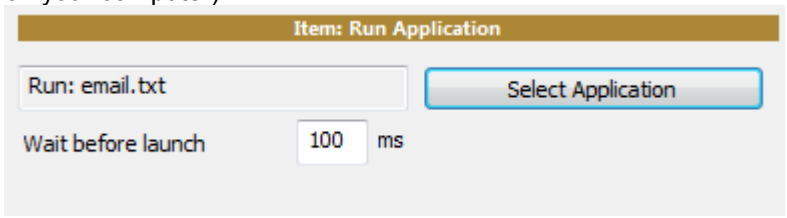
Run Application
Runs an application.



If you want to open a document with its default app, instead of selecting application select **All files** and choose the document file.

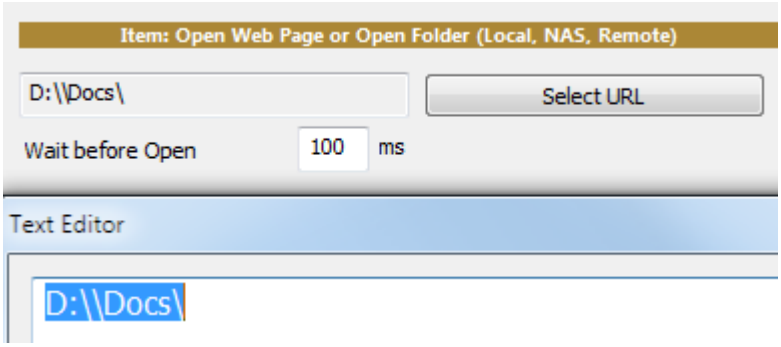


In our case I selected txt file which will be opened with notepad (or whatever else is registered to open txt files on your computer)



Open Web Page / Folder

Opens web page or a folder on local computer, NAS, network... depending on what you specify.



D:\\Docs\ will open folder, \\DISKSTATION\Volume1 will open NAS, 192.168.1.2\Shared files\ will open network location etc...
http:\\www.mediachance.com will open web page

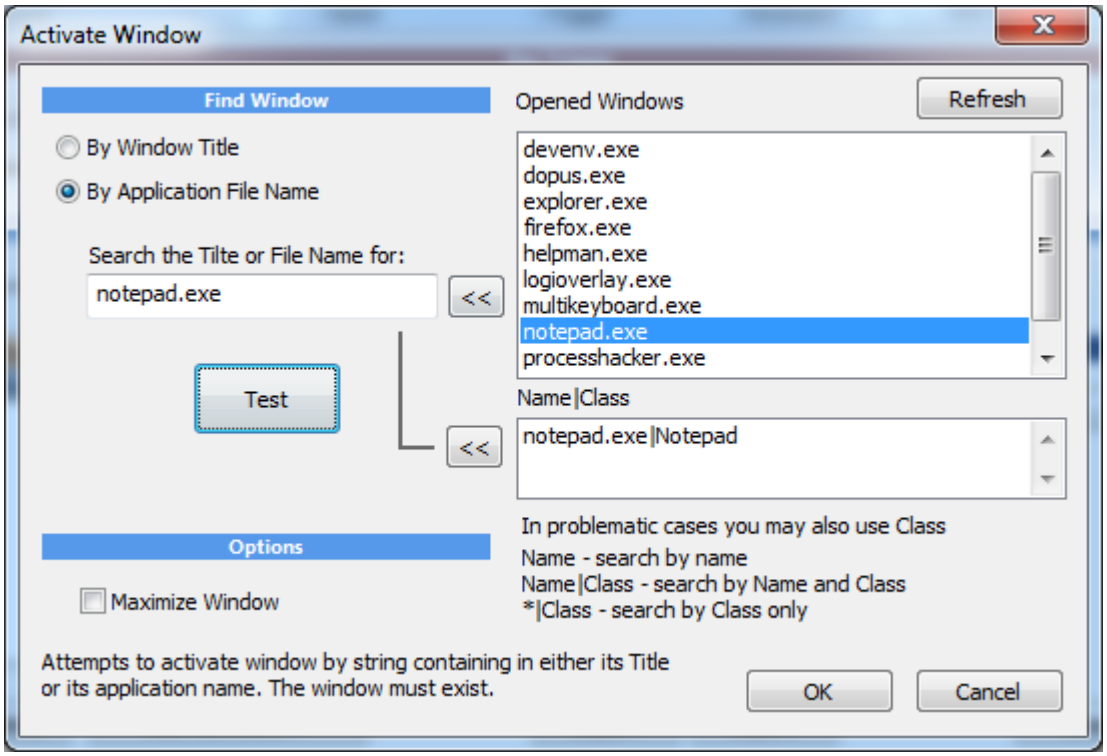
2.3 Activate Window

This will help to find an existing window and then bring it to front. Depending how that application is made, it may not always work or may require consequently sending some keystrokes to set the proper focus to the area we want to affect. Also each new version of windows puts more restrictions how an application can interact with the ones it doesn't own.

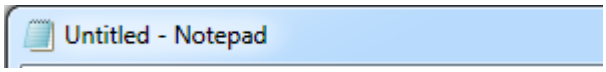
This of course requires such application is already running. This function will not open the application, just search for it and if it is running it will bring it to front. If you are opening an application in the same macro, it is probably no need to add Activate Windows as the app will automatically open in front. But if you need to do it, remember to have some Pause between opening and Activate as the app needs to be fully loaded first.

You can search for a window by its **exe** name or by the **window title**.

Also a third combined possibility: you can search by a **Class**.



Generally you need to type a substring in the "Search the Title..." box. It should be a partial or full name. In case of Window Title, it should be the significant part of the string that will be always present in the title bar.

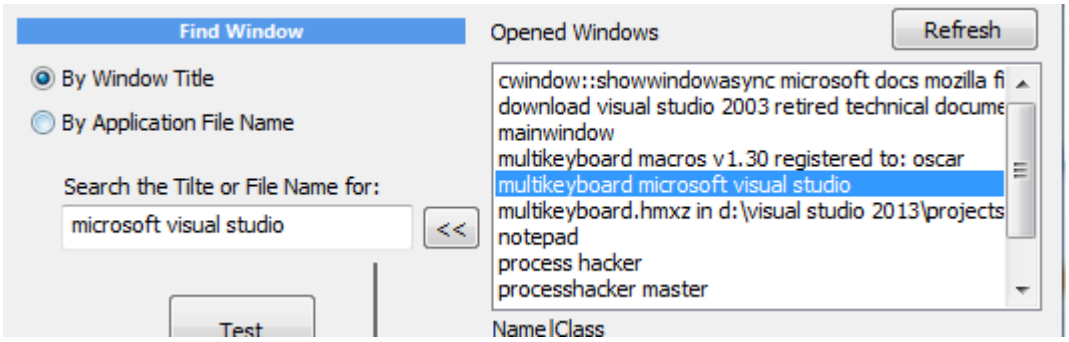


Obviously you want to search the title for "Notepad" and not "Untitled - Notepad" whole title, which will change depending what file the notepad has opened.

When you double click the list of opened windows or use the << button it will type selected item in the box.

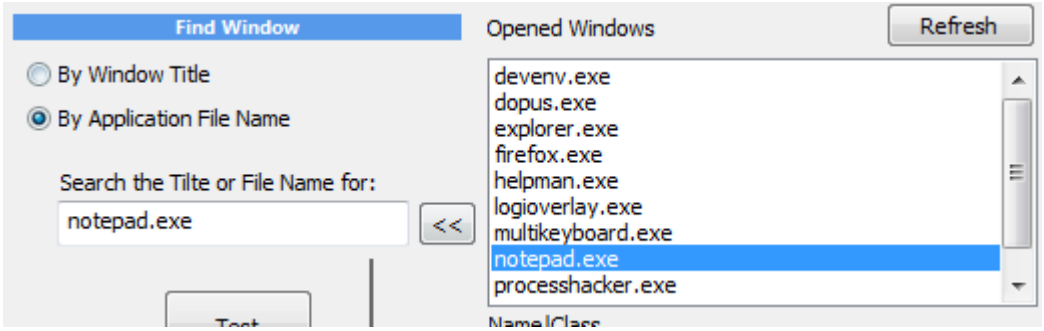
Important Note: in case of Window Title, what will be shown in the Opened Windows are already **processed** strings - with removed spaces, characters, stuff in parenthesis etc... as such if you search for that exact string you will likely not find the window. You need to select only the important part - which is the name of the application. - but it is hard for the MKM to determine that by itself - it needs your eyes.

Searching by **Window Title**
As an example on the right you have **multikeyboard microsoft visual studio**. If you enter the whole text you will not find microsoft visual studio, because the whole window title is actually much longer and has been abbreviated in the list box.
What you need to do is to enter only **microsoft visual studio** which is a string that will be common on all instances.

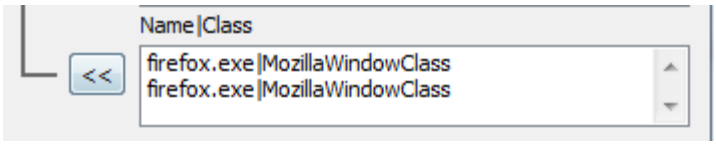


In some cases, such as web browsers, the Windows Title may not always mention the application name at all and may be often just a document title, web page name etc... so it may be hard to find window like that.

Searching **By File Name** is more exact:



When you press Test, the application(s) that satisfy the search string will be brought to front. In case of multiple opened applications that satisfy the search, all of them will be brought to front. And all windows that had been found will be written in the lower box with its class name.



These could be multiple instances of the application with the same class or sometimes a single application that has few hidden windows - and in some cases bringing those hidden windows to front may not be the best idea. This is where you can enter class name after | sign.

For example

firefox.exe|MozillaWindowClass

In this particular example it would not be that practical as every instance of firefox.exe will always have the same *MozillaWindowClass* class, but we can switch now to **By Window Title** and write:

mediachance|MozillaWindowClass

This will brig to front firefox as found by its class, but **only** if its Window title says *mediachance*.
Note: in case of firefox or other apps with tabs - only the active tab window title will be known. This command will not switch the tab in firefox to tab that says mediachance - it has to be currently opened tab. There is really no single mechanism that would work on all apps as everybody develops it a bit differently and there is only so much an app can know about other apps.

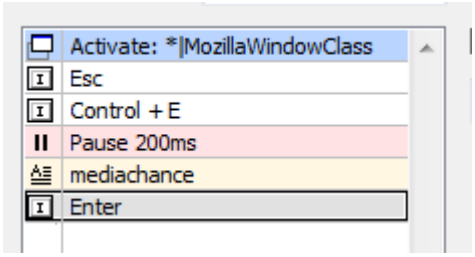
You may also use * instead of title or file name

*|MozillaWindowClass

will bring to front any window that has class *MozillaWindowClass*

Some other thoughts:
Don't expect application that was brought to front being ready to receive keystrokes. Some other things may be in the focus (for example in firefox it would be the actual web page - so you can't send text to it) and it is hard to determine what is selected by default.
It would be good to follow the Activate Window command with another such as appropriate key shortcut that would make sure the application is ready to receive further commands.

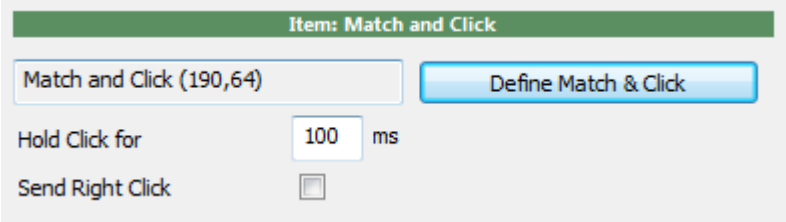
For example we may first send "ESC" to the window which would close any opened menus or windows.
In case of firefox, we may then send CTRL+E that would put focus on its search bar, or CTRL + T which will open new tab and put cursor in the search bar as well. Because this may take some time, a pause will make sure the firefox is ready.
Then we can type something in its search bar "mediachance" and press enter



2.4 Match and Click

The Mouse Click function requires that the button or item we are clicking on stays on the same position. That is not always the case and on web pages certain items cannot even be assured to be on the exact position even if we maximize browser window. A pattern matching function would locate (match) predefined "Anchor" area on the screen then click on point relative to that area.

This function is for recognizing where are things on the screen - for example a button or form on a web page, or a menu item on a window that may be moved to different place.



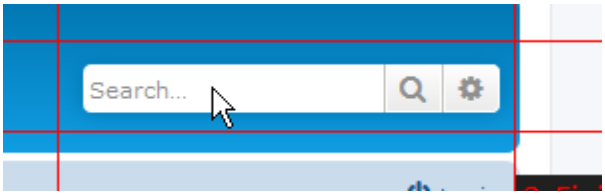
This function require a little setup preparation.

When you click **Define Match and Click**, you will first need to define an area on screen that will be used for matching. It needs to be something that doesn't keep changing (for example a set of buttons, or a text).

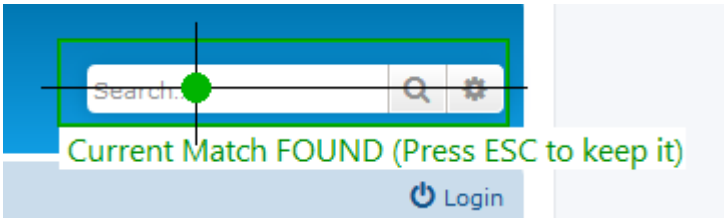
In this example we want to find a search box on particular web page and type something in it.

We would mark the area of interest to be the search box because we assume it won't keep changing.

Then you can define where you need to click relative to the marked area.

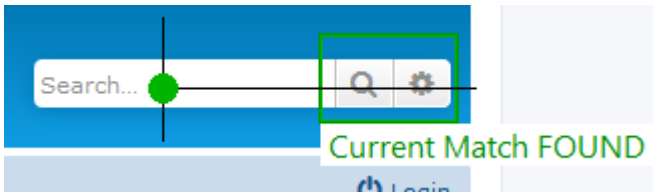


If everything went well, you can then move your window (browser) to different position, then click Define Match & Click again to test previous setting and it should now find the part on the screen that we marked and show where mouse click will be placed even the window is in different position.

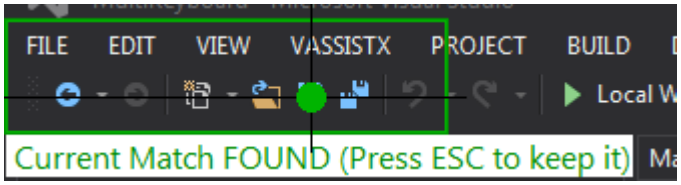


Note: during normal macro operation there is nothing drawn on the screen. If the function succeed and finds the match it will click on the mouse relative position as set. The visual aid is only during Define Match & Click.

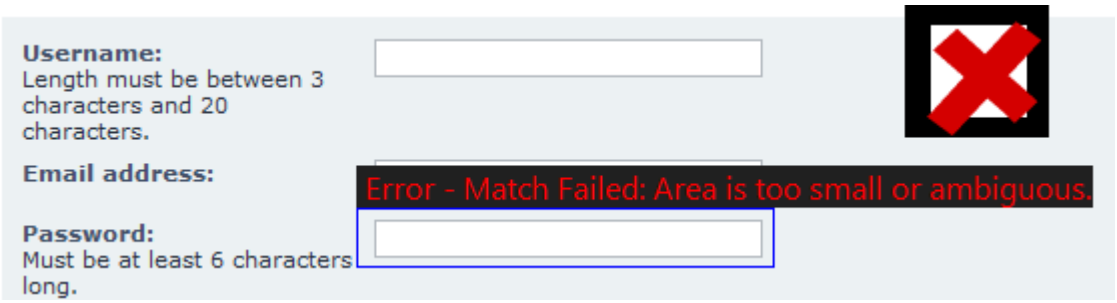
It is important to note that the mouse click and area are not strongly tied up - that is: the mouse click doesn't need to be within the area at all. The area is just to set an anchor - find our position on screen on something recognizable and then click on place relative to that anchor. For example if the size of searchbox changes with the size of the browser window, our match would fail if the search-box is bigger or smaller than the one we defined, so we may search just for for the buttons on right and then have it click on the left side of the matched rectangle.



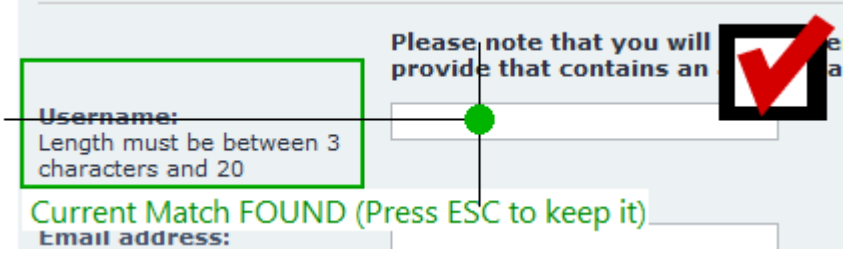
Also the larger the area is, the better chance of a correct match. For example selecting a single small toolbar icon would likely result in incorrect or wrong match, but selecting multiple buttons would ensure more reliable match.



When you are defining the Anchor area, the software will also test if it can find it and if not it will display error and you need to redo the selection.
There are some obvious that would made the match fail. For example selecting ambiguous area like the one below:



In this case there multiple possibilities on the screen for the match and so the function fails. It is better to select area of the text as it is likely the only area like that on the screen, ten position the click relative to it.



2.5 Quick Macro Recorder

Quick Macros are recorded keyboard macros that are temporary.
You may question what that is, but if you are programmer you may be familiar with the concept. Sometimes you need to make lot of repeating text operations on multiple lines. So you record a quick macro of what you need on one line, then play it on the rest of the lines.
This is useful for editing tables, text fields etc...
The way you do a complex tasks is that you cleverly employ a clipboard and word selection in your application to record such macro.

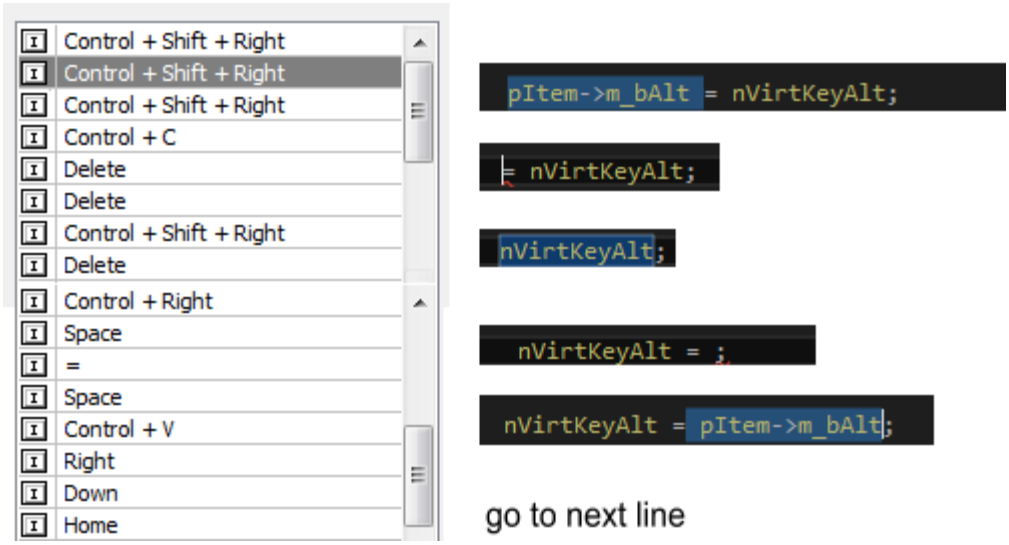
Here is an example:

```
pItem->m_bAlt = nVirtKeyAlt;
pItem->m_bShift = nVirtKeyShift;
pItem->m_bControl = nVirtKeyControl;
pItem->m_bWin = nVirtKeyWin;
```

As it would be a common task in programming, you may need to reverse these lines where what is on the left needs to go to right etc. It is fine for few lines but what if you have 50?

You can record your keystrokes as a simple macro on the first line by using common editing commands such as **select word**, **copy**, **paste** that are used in most editors.
There are some rules and tricks doing it this way but the benefits are big once you master it. It is important that you use only keyboard commands, don't move your cursor by mouse, move it by arrows. Instead of deleting a word letter by letter which would mess up if other line has different length word - use CTR+SHIFT+Right Arrow - which select the next word. Then press Delete. If you need to skip a word, use CTRL-Right Arrow... Tricks like these. Every programmer can tell you these.

Here is a recorded result what I was doing



We ended our one line macro by using Arrow down and pressing Home which simply put cursor to the beginning of next line. And now we are ready to play the macro few times:

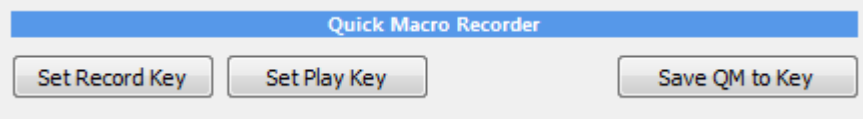
```
nVirtKeyAlt = pItem->m_bAlt;
nVirtKeyShift = pItem->m_bShift;
nVirtKeyControl = pItem->m_bControl;
nVirtKeyWin = pItem->m_bWin;
```

In many editors you can also combine search commands CTRL+F and F3 for example in your macro line to do even crazier stuff.

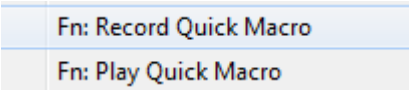
So this is why quick macros are powerful tools once you master the keyboard language.

2.6 Start Quick Macros

In order to start recording quick macro you need to be already in the editor where you perform these. So you can't start by clicking some button on the MultiKeyboard interface because that will bring up the Multikeyboard interface. The only proper way to start recording in application is by a trigger. And for that you need to first define the trigger.



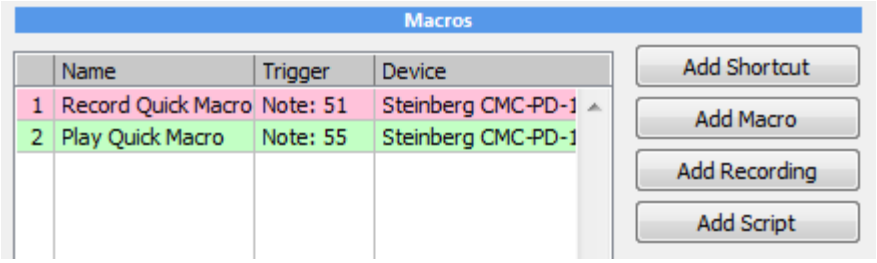
That's what these buttons are on main interface. They do the exact same thing like creating a macro and adding FN: Record Quick Macro command.



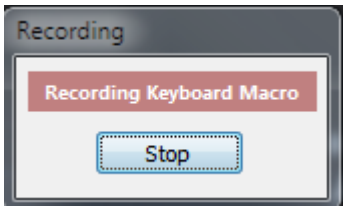
So basically they will ask you which key trigger you want to use for recording quick macro and which for playback.

For example we may use pad 13 and 14 on the MIDI pad board.

The triggers will appear in the list as any other macro - because that's what they are anyway - a trigger that runs internal command.



Once you set both record and play key, you can go to your application and trigger the record. Recording window will appear on top to remind you you are recording macro.

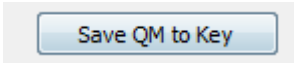


To stop recording, you can press the record trigger again. (or click Stop button)

To play back the macro, place cursor where you want and press Play trigger on your keyboard.

When you actually create a useful recording that can be used later you can export it into a trigger key of your choice.

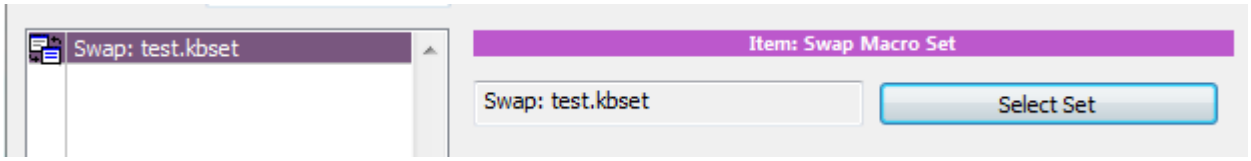
For that there is Save QM to Key



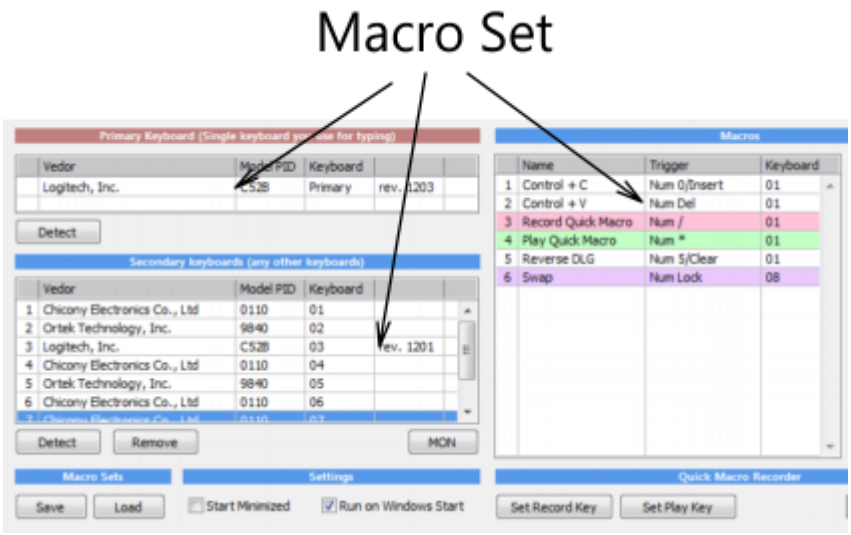
This will save the recorded macro as series of steps in a normal Macro.

2.7 Swap macro Set

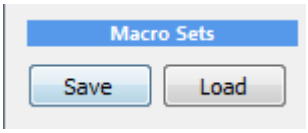
This command will load any previously saved macro set - and it would result in a full macro set replacement (swap).



Macro set is everything you see on the main interface - including keyboards and all macros associated to them.

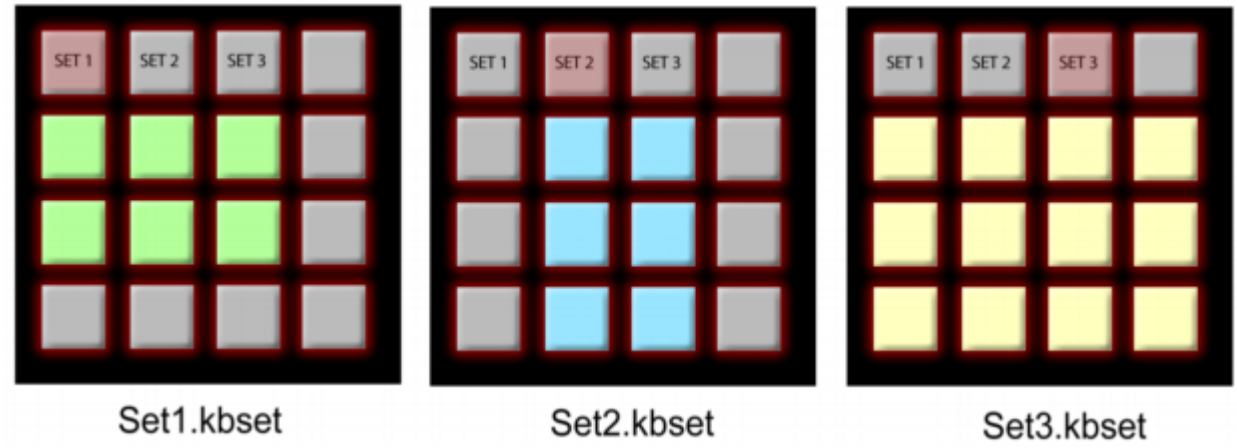


To create Macro Set: Use Save on the Main interface.



This option is great if you need to often load different macros onto one keypad/keyboard during your normal use. For example one macro set for video editing, one for graphics...etc.

In following example with a 4x4 MIDI pad we can define top pads to swap between sets; Set1, Set2, Set3



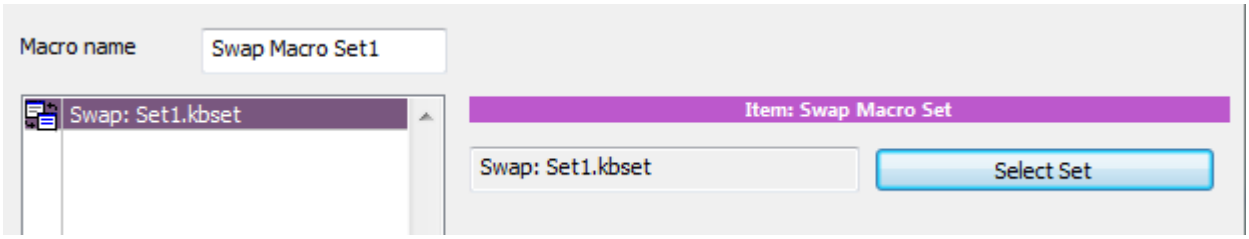
Note: Each set has to also have defined the same Swap buttons as well - otherwise you won't be able to swap back from the new set.

In our example we may go like this to define 3 switchable layouts:

Start with blank command list.

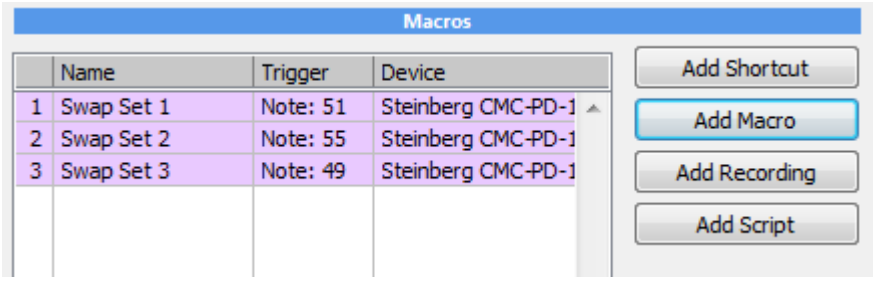
To avoid chicken/egg problem let's first create 3 dummy placeholder sets; Save the currently blank set using Save Macro Set button to 3 different sets: Set1.kbset, Set2.kbset, Set3.kbset

Now use **Add Macro** and choose key "/", Add Command: Swap Macro Set, Select set *Set1.kbset*



Repeat for two additional pads each loading *Set2.kbset* and *Set3.kbset* respectively

Now you see why used placeholders - our 3 swap buttons need to load the sets but we are yet to going to fully define the sets. We should have 3 buttons defined with Swap commands:



Now it would be a good idea to save all this yet again to all three kbsets we saved previously: Set1.kbset, Set2.kbset and Set3.kbset

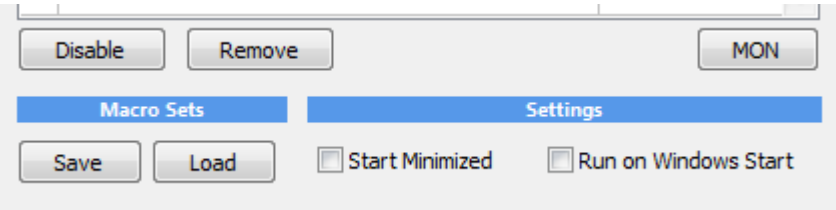
Now you can define all the necessary keys for the Set 1, then save as Set1.kbset.
Load Set2.kbset, define keys for this set and save as Set2.kbset.

Load Set3.kbset, define keys for this set and save as Set3.kbset.

So each of these sets will have three same keys defined to the same action: Swap macro Set1,2,3 but all other keys will be different.
You need to make sure all sets are saved before you go and start defining new set.

Now if you press any of those three buttons, you should see the macro set automatically swap.

2.8 Settings

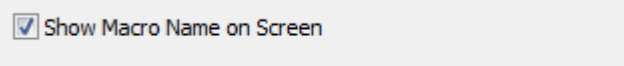


Macro set Save Load
You can save and load entire macro sets - all macros you defined

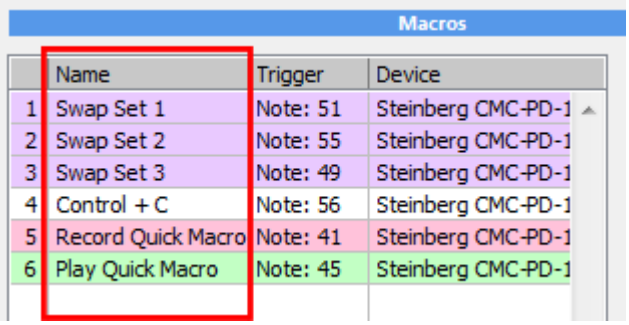
Start Minimized - the app starts in its minimized state in windows tray so you don't see the interface.

Run on Windows Start
Places the app shortcut to the Startup folder of current user so the MultiKeyboard Macros will be loaded when users log in or computer boots.

Show Macro Name on Screen

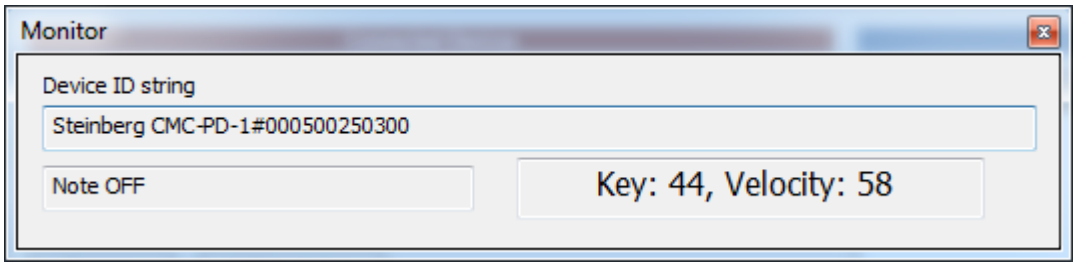


It will briefly display the macro name on the bottom part of screen when triggered.



Note: if you want to suppress only certain macros from displaying its name on screen (for example if it triggers screen capture) prefix the macro name with a dot, Example: .ScreenCapture. Also in script you may use DisplayText 57 to display your own label, in which case you don't want the default macro name to appear briefly.

MON
Monitor functionality, shows pressed controls in a monitor window. It will monitor only the devices that are in Active MIDI devices.



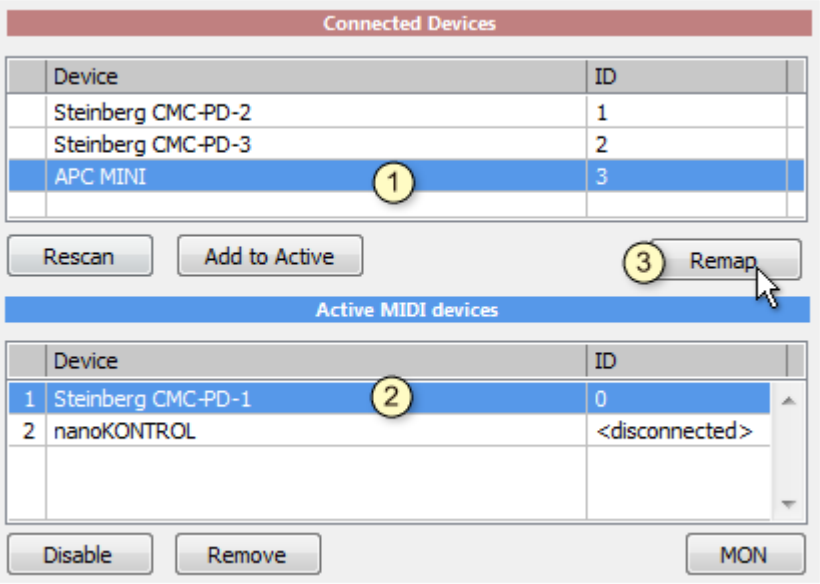
2.9 Remap device

Remap will allow you to remap all existing macros from one device into a new one.

Tip: This may come useful if you update firmware on a device and it will no longer be recognized as the same device due to firmware number change.

Example:
For example you have lots of macros mapped to Steinberg CMC-PD but you would like to use APC MINI instead and free up the Steinberg for something else.

Select the APC MINI in the Connected Devices and the existing Steinberg CMC-PD in the Active MIDI devices. Press Remap.



All the macros will now respond to the new APC mini.

	Name	Trigger	Device
1	Control + C	Note: 56	Steinberg CMC-PD-1
2	Record Quick Macro	Note: 41	Steinberg CMC-PD-1
3	Play Quick Macro	Note: 45	Steinberg CMC-PD-1
4	.Multi64	Note: 40	Steinberg CMC-PD-1
5	Undo	Note: 35	Steinberg CMC-PD-1
6	Redo	Note: 36	Steinberg CMC-PD-1
7	.PS Link	Note: 39	Steinberg CMC-PD-1
8	Paste	Note: 44	Steinberg CMC-PD-1
9	.Slider Change	CTRL: 2	nanoKONTROL
10	.My Script	CTRL: 15	nanoKONTROL
11	.Clipboard	CTRL: 3	nanoKONTROL
12	.Copy ToClip NR	CTRL: 24	nanoKONTROL
13	.Paste to ClipNR	CTRL: 34	nanoKONTROL

	Name	Trigger	Device
1	Control + C	Note: 56	APC MINI
2	Record Quick Macro	Note: 41	APC MINI
3	Play Quick Macro	Note: 45	APC MINI
4	.Multi64	Note: 40	APC MINI
5	Undo	Note: 35	APC MINI
6	Redo	Note: 36	APC MINI
7	.PS Link	Note: 39	APC MINI
8	Paste	Note: 44	APC MINI
9	.Slider Change	CTRL: 2	nanoKONTROL
10	.My Script	CTRL: 15	nanoKONTROL
11	.Clipboard	CTRL: 3	nanoKONTROL
12	.Copy ToClip NR	CTRL: 24	nanoKONTROL
13	.Paste to ClipNR	CTRL: 34	nanoKONTROL

Note: there are some error checks before you can remap:
First, you should save your Macro Set. It is easy to loose your macros if you are not careful.

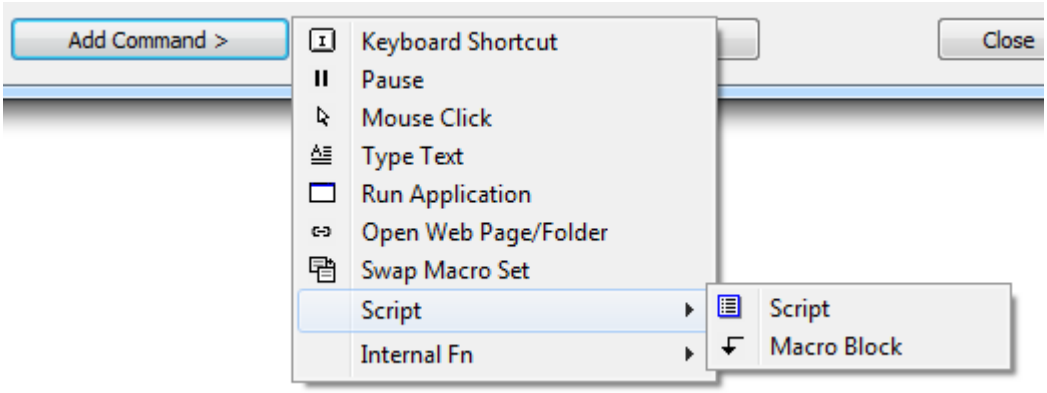
Then the new device we are remapping to can't be in Active MIDI devices yet. (that may be a collision with existing triggers where a same trigger would have two macros etc...)

III Scripting

3.1 Oscar Script

Now we are getting into a much more advanced area!
Till now we could add a sequence of various pre-set steps. Often that functionality may be enough, but what if we want more?.

In version 2.0 we added a scripting command... and that is a big deal!

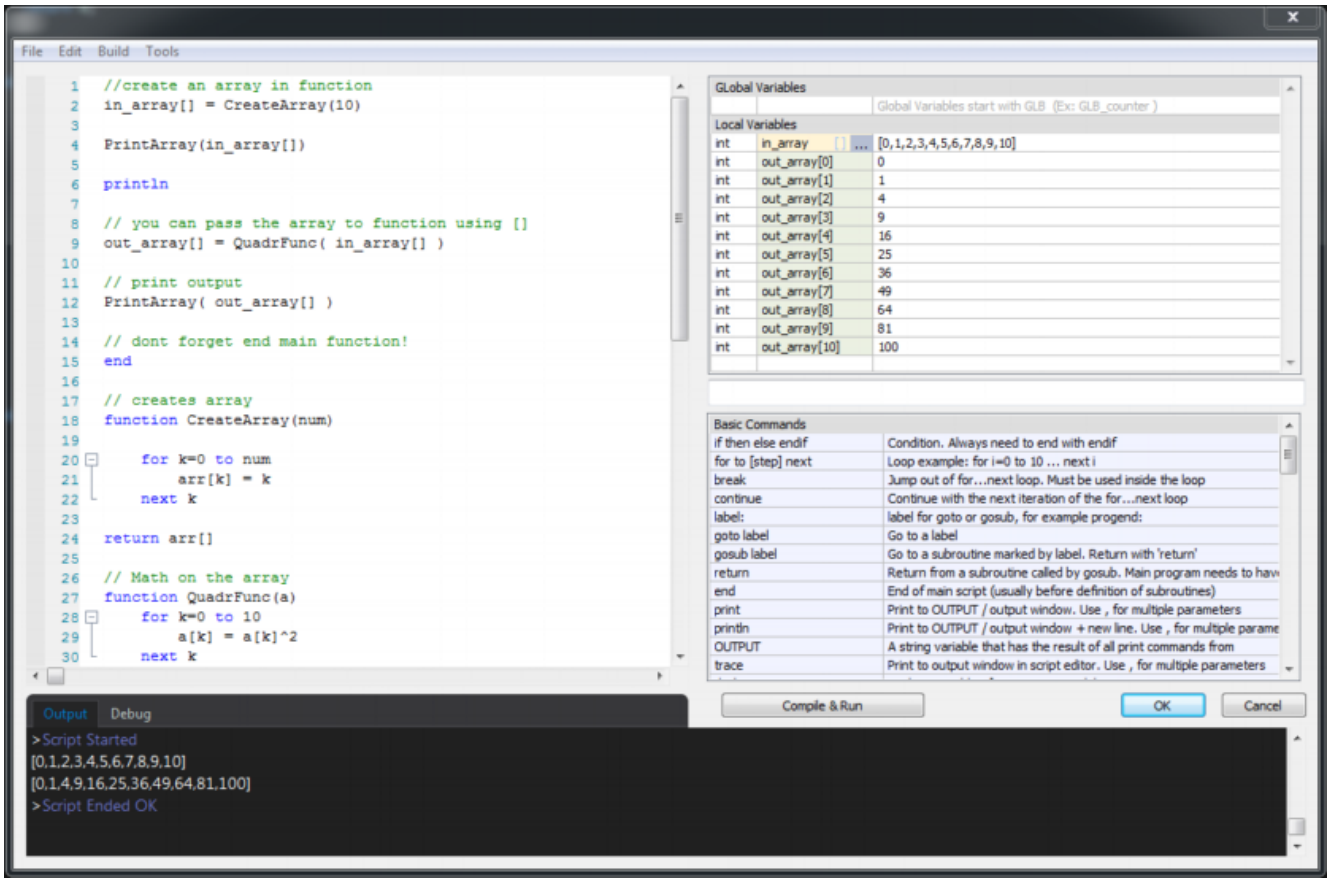


This little block hides an an enormous power - almost like a whole new application itself.

It is important to note that using script is entirely optional. You don't have to if you don't want to, but it can open the door to things that other macro software can only dream of.

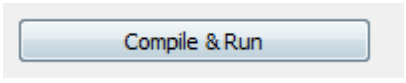
The script itself is quite well suited to process clipboard text and has a large amount of string operation including tag extraction, tokenizer, even regex. A script can grab text from under your cursor, process it in different ways and then type it back or save it to a file. Think of selecting a name in your mailbox, pressing a key and it will instantly format a whole letter. Or just copy a block of text and it will extract names, email addresses etc.
We made few examples, that only scratch the surface - multiple clipboards, XML tag extraction etc.

It can be also used to create a very special logic, where some keys would be modifiers to other keys.



The script editor consist of few parts - the main typing area, on the right is the list of used variables in the script, and list of all commands and output window.

Before script can be accepted it needs to be Run to make sure it has no errors.



or menu Build - Run Script

A script that has an error or has not Run after changes were made cannot be applied.

The List of Variables displays used variables with their values in the script. It is updated every time you Run the script.
You can override a variable during debugging breakpoint by typing new assignment into the edit line bellow the variables.(see more in debug 105 section)

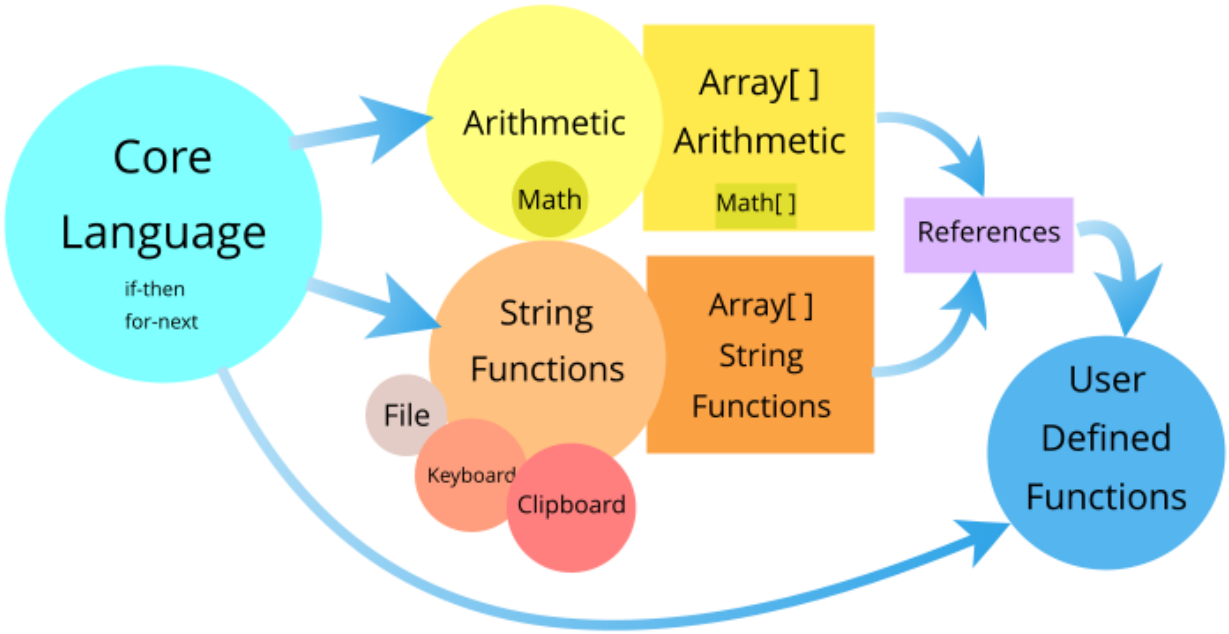
List of commands is a sort of short help file that list all available commands. Double-clicking on a command will insert it into the script.

The richness of the script language may feel almost an overkill for this application - as I was developing it, it soon made detour from a simple few day of work script and became more like a full programming language with many interesting aspects that I was missing in other scripts: we can have user functions, it has rich array arithmetic, local and global variables, even reference variables and pretty great step debugger. I decided to continue working on the script language, enhance it and use it for other applications as well.

3.2 Script basics

We have used various scripts in different Mediachance products, but this is by far our finest script yet!

The language is loosely based on a BASIC for its simplicity and familiarity but it omits many of the old Basic idiosyncrasies and where it was beneficial it borrows syntax and features from modern languages like Java, Lua or C.
If you ever programmed in any of the modern languages you will be right at home.



The core language is kept at minimum to keep it familiar. On top of the core language are arithmetic and string functions for both normal parameters and their array equivalents.

Comments
Comments follow standard C type of comments

```
// This is comment

/*  These are comments too
    ======================
*/
```

Syntax
There is no special character to end a command line (unlike in C where there is ' ; ')

```
a = RND(0, 100)
```

Note: Typing ' ; ' at the line will not return error but neither it signifies end of the command. It is simply ignored - I did that far too many times during testing as an old habit.

Core command set

The core command set consist only of few basic commands, such as `for next, if then,...`. All core commands are written in lowercase.

```
print "Oscar script is alive"
```

After core commands the script has extension (functions) commands that would work on strings, time, clipboard, send text etc.. These use mixed Upper and Lower case and can be extended in future with more commands when needed.

```
newstring = FindNumbers(string)
```

Variables
There is no mandatory declaration, nor type declaration. There are 4 types of variables:

- integer
- float
- string
- reference

Reference will be explained later in more details, the other are self explanatory.

A variable will be auto-assigned a correct type first time it is used.

Variable name is Case Sensitive. Underscores are fine as well as numbers if they are not the first character.

```
nVariable = 123
fVariable = 3.1415
sVariable = "Oscar Script"
```

If you Validate such script you will see in the Variable list:

GLobal Variables		
Local Variables		
float	fVariable	3.141500
int	nVariable	123
string	sVariable	"Oscar Script"

HEX and BIN integer numbers
Binary literal numbers have prefix 0b, hexadecimal literal numbers have prefix 0x

```
nHex = 0xFF
nBin = 0b100000000
```

Local Variables		
int	nBin	256
int	nHex	255

Escape characters in strings
As common in other languages \ character signifies escape characters, for example \t is tab \n is new line etc....

```
sEscapeWrong = "Files\text.txt"
Output: Files     ext.txt
```

In order to have backslash in literal string you need to use \\

```
sEscapeCorrect = "Files\\text.txt"
Output: Files\text.txt
```

Preceding string literals with _R will turn on RAW string option that will ignore any escape characters in the string that follows.

```
sNoEscape = _R"Files\text.txt"
Output: Files\text.txt
```

RAW string option is especially good for RegEx patterns where entering double \\ will make it even more unreadable than it is now.

```
regex = _R"[-+]?(\b[0-9]+\.([0-9]+\b)?|\.[0-9]+\b)"
```

Uninitialized variables
If you use a variable that wasn't yet initialized (assigned any value to it), it isn't an error, but a warning is issued and the script continues assuming integer zero value.

```
c = var1
print c
```

```
>Script Started
>Warning at line 1 : var1 was used without being initialized first. Possible error in this context!
0
>Script Ended OK
>1 Warning(s)
```

Auto re-assigning of type
In some cases the script will automatically re-assign a type if there is a possible loss of data. (we will get warning in Output window)

For example:

```
//we started with 'a' as integer
a = 23
b = 1.23
//script will auto reassign 'a' to FLOAT
a = a + b
c = INT(a)
```

```
>Script Started
>Warning at line 5 : Auto Re-assigning a from int to float to avoid loss of data.
>Script Ended OK
>1 Warning(s)
```

In this case script started with 'a' as integer but then re-assigned it to float to prevent loss of data when we added float number to it.
If we explicitly need to keep integer we can use INT function

Local Variables		
float	a	24.230000
float	b	1.230000
int	c	24

Arrays
Arrays are done the very same way, without declaration.
In fact Oscar Script could have one of the most clever system for arrays. But more about it later.

```
k[6] = 12
sString[1] = "test"
sString[k[6]*1000] = "test 1000"
integer[0] = 1234
integer[-200] = 4325
```

You may notice a peculiar thing on the above listing: one of the index is negative - that is perfectly valid in Oscar Script! Another thing is, we can index arrays any way we wish even non sequentially.

Local Variables		
int	integer[-200]	4325
int	integer[0]	1234
int	k [] ...	[12]
string	sString [] ...	["test","test 1000"]

Multidimensional arrays are done the typical way

```
for y = 0 to 5
    for x = 0 to 5
        array[x][y] = x+y
    next x
next y
```

While there is no limit into dimensions, remember this is a script - so don't go overboard. The bellow is perfectly fine as a syntax, but it makes very little sense:

```
variable[1][34][123][100+2][2][25] = "testing"
```

String Arrays

It is important to mention that any member of array when not assigned value will be an integer zero, even if other members could be strings.

So if you assign

```
string[0] = "zero"
```

You can't just assume the `string[10]` will be also string, unless you actually assign a string to it beforehand.

Therefore if you need array of 10 empty strings you should assign "" to them first

```
for k = 0 to 9
    string[k] = ""
next k
```

#const :Definition of Constant

To define constant use #const keyword on a new line.
Unlike assign operator with variables, there is no '=' .

```
#const IDENTIFIER expression
```

The **#const** will assign a value (integer,float or string) or the result of an expression to a constant during run-time.
Because the constant is defined at runtime, all the parts of the expressions needs to evaluate before the #const definition takes place
In general it is best suited for constants or indexes.

```
#const DEG_MULT 3.1415926/180

a = SIN(90*DEG_MULT)
```

You can't reassign value to a constant.

#define :Definition of Macro

You can create a macro with #define

```
#define IDENTIFIER macro
```

Script will substitute each occurrence of IDENTIFIER in the source with the macro string before it runs.
Because the substitution is done before run-time, the macro doesn't need to evaluate at definition, only where it is actually used.

```
#define MY_PRINT println "Value of a: ", a

a = 10
MY_PRINT
```

`#define` is of course much harder to debug than ordinary code - because you don't see the substitution taken place in your editor. It can have some unwanted effects if its name clashes with other names of functions or variables.

In general `#const` is preferred for defining constants and should be used instead of `#define`

`#define` macro can be multi-line if the last character is space \ backslash followed by immediate new line the macro will also consist of the next line.

Example:

```
#define FORLOOP for i=0 to nM \
                  print i \
                next i
```

```
nM = 5
FORLOOP

Output: 012345
```

3.3 if-then-else-endif

A standard if condition statement that allows identifying if a certain condition is true, and executes a block of code if it is the case.

```
if condition then
    statements
else
    statements
endif
```

The very basic condition is one without else:

```
if a==5 then
    print "a is five"
endif
```

One rule is that each condition has to have `endif` - because we don't have block separators as in C { }, the script needs to know when if -then condition starts and where it ends.

```
if (a == 5) then print a
endif
```

This is enforced even if you put everything in one line - you have to use endif.

```
if a==0 then print a endif
```

The rule is simple: there has to be the same amount of endif than if. If it isn't, you have some logic error. We made this rule, so It is much easier to find problems with nested if-then. Just count the ifs and endifs and they must be the same.

```
if a==5 then
    print "a is five"
else
    print "a is definitely not five"
endif
```

Comparison operators:

```
== != <> < <= > >=
```

Note: that "is equal" is in Oscar Script similar to C equal: ==

```
a == b  // a is equal b
a != b  // a is NOT equal b
```

Boolean operands

```
 a | b  // boolean OR
 a & b  // boolean AND
```

Negation

```
!a  // NOT a - negation of a
```

```
!(a & b) // NAND - NOT (a AND b)
```

Else if - nested if.
We can follow else command with another if, which creates nested if-then command

```
if condition then
     statement
     else if condition then
     statement
     endif
endif
```

You can have many nested `if` blocks etc, just always remember the `endif` rule.

It helps if you write nested if conditions tabbed so it become more obvious what `if` belongs to what `endif`

```
a = 3
if a<1 then print "a<1"
else
    if a<2 then print "a<2"
    else
        if a<3 then print "a<3"
        else
            print "a=3"
        endif
    endif
endif
```

3.4 for-to-next

For - next is your basic loop.

Syntax:

```
for counter = nStart to nEnd
....
next counter
```

It is important to note that on both sides it is inclusive. What you see in the for - to statement will be the numbers the loop will go through, including those numbers.

```
for a = 0 to 5
    print a
next a

Output:
```

```
012345
```

Advanced loop using **'step'** parameter, which specified the value at which a variable is incremented. It can be negative to have the loop decrease the variable instead of increasing it

Syntax:

```
for counter = nStart to nEnd step nStep
....
next counter
```

Example

```
for a = 5 to 1 step -1
    print a
next a

Output:
54321
```

The for - to line is evaluated only once at the beginning. The loop variable is increased (or decreased) every time next command is found.
Changing control variable inside the loop will change how the loop behaves! It is probably a bit risky to use it this way.

```
for a = 0 to 10
    print a
    a = a*2  // this will change the condition
next a

Output:
0137
```

Non linear loops
Even more advanced loop is one with **changing step**
Unlike standard BASIC, Oscar Script allows you to change step within inside the loop by simply assigning it a new value. This creates some new possibilities in creating special non-linear loops.

```
for a = 1 to 256 step 1
    print a,","
    step = a  //this changes the step of the loop inside the loop
next a

Output:
1,2,4,8,16,32,64,128,256,
```

Note: at no point the step can be assigned value of 0 (that would create infinity loop)

step behaves as a hidden variable and can be also used on the right side of equation in the loop:

```
for a = 1 to 256 step 1
    print a,","
    step = a
```

```
    if (step == 4) then
        break
    endif
next a
```

But if you try to use it outside the loop you will get an error.

Break and Continue

Break will exit the loop. In case of nested loops it will exit only the closest loop it is in

```
for k = 0 to 2
    for a = 1 to 10
        if (a == 5) then break
        endif
        print a,"|",k,"  "
    next a
    println "break"
next k

Output:

1|0  2|0  3|0  4|0  break
1|1  2|1  3|1  4|1  break
1|2  2|2  3|2  4|2  break
```

Continue will skip the rest part of the loop and directly do a next loop iteration

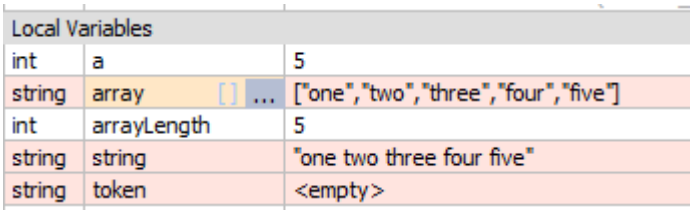
```
for a = 1 to 10

    if (a == 5) then
        print "five,"
        continue  // go back to beginning of loop
    endif

    print a,","

next a

Output:
1,2,3,4,five,6,7,8,9,10,
```

Infinite Loop
While definitely not a good idea, sometimes you may not know the predetermined number of loops you need (for example searching for substring)

you can use either reasonably big number, or even INT_MAX

```
string = "one two three four five"

for a = 0 to INT_MAX
```

```
    token = Tokenize(string," ",a)

    if (token=="") then
        break
    endif

    array[a] = token

next a

arrayLength = a

print "We've got ",arrayLength, " items"
```



Local Variables		
int	a	5
string	array [] ...	["one","two","three","four","five"]
int	arrayLength	5
string	string	"one two three four five"
string	token	<empty>

There is an array version of Tokenize, that will create the array without loop, on just single line. (More about it later) The normal Tokenize was used here for demonstration.

Note: The script will still abort after predetermined **safety** time to avoid infinite loop. The default is set to 5 seconds

3.5 Goto and Gosub

```
goto label
```

Goto statement is used to branch from one part of the code into another that is marked with a label.

label is any name that is at the beginning of a line and ends with :

```
label:
```

You can jump out of the loops, if statements or skip large chunk of code etc... It is usually said to avoid `goto statement` because it makes the code harder to follow.
That may or may not be true, depends how you use it. Sometimes it saves a lot of additional conditional code especially when nested if-then are involved

```
if a>-1 then
....
   if (c>-1) then
     ...
    goto finish
    endif
....
endif
```

```
finish:
println "done"
```

A general idea is to avoid going back to previous lines - that may create infinity loops and it is definitely much harder to follow.

Gosub

```
gosub label
```

Unlike goto statement that simply goes away, `gosub` also remembers where it was and can return with `return` statement creating a basically subroutines in your code
When you are creating subroutines, make sure you mark the end of the main program with `end` statement.

```
test = RND(0,3)

//for every if there has to be endif!
if (test==0) then
    gosub subroutine0
else
    if (test==1) then
        gosub subroutine1
    else
        gosub subroutine2
    endif
endif

println "Finished"

// if we use subroutines, we need 'end' of main program
end

subroutine0:
    DisplayText("We are in Subroutine A")
return

subroutine1:
    DisplayText("We are in Subroutine B")
return

subroutine2:
    DisplayText("We are in Subroutine C")

return
```

Oscar script has also functions which are much more modern way of doing a subroutine jumps. The difference between gosub and function is that all variables inside functions are local while with gosub we share the same variables with the rest of the script..

3.6 Print, Println

`print` command prints to Output Window.

`println` command is same as print but ends the command with new line escape characters ("\r\n")

Printing to Output window makes sense only during Script Editing. It does nothing during normal operation - when called from within a key macro. However, the print has one more trick in its sleeve, called print to OUTPUT.

Syntax:

```
print expression, expression,....
```

The expressions can be variables, strings, or in fact whole "expression"

```
print a,b
print "a=",a,", b=",b
print "Random number: ", RND(0,10)
```

Example:

```
for i = 0 to 5
    print i,","
next i

0,1,2,3,4,5,
```

The same, but using `println`

```
for i = 0 to 5
    println i,","
next i

1,
2,
3,
4,
5,
```

Print to OUTPUT
Print is not just printing to Output Window, that would be a lackluster feature for normal operation. Print commands also add sequentially all print output in the current script into a string variable called `OUTPUT`
This serves as a simple and painless way to format strings that can be then used further in a string operations, clipboard or save to file.

```
//clears OUTPUT in case we used print before this line
OUTPUT = ""
bookid = "0021313"
TAB = "\t"
QT = "\""
println "<?xml version=\"1.0\"?>"
```

```
println "<catalog>"
println TAB,"<book id =",QT,bookid,QT,">"
println TAB,TAB,"<author>Misc, Jones</author>"
println TAB,TAB,"<title>How to compute</title>"
println TAB,"</book>"
println "</catalog>"

SaveString(OUTPUT,"file.xml")
```

Content of the file.xml

```
-<catalog>
  -<book id="0021313">
      <author>Misc, Jones</author>
      <title>How to compute</title>
    </book>
  </catalog>
```

3.7 Conditional operator

```
operand ? expressionYes : expressionNo
```

Conditional operator is a sort of ternary inline 'if' operator that can be used to evaluate two different expressions

if the operand is evaluated as TRUE (>0) then expressionYes is used
if the operand is evaluated as FALSE (==0) then expressionNo is used

For example:

```
b = a>5 ? a*2 : a/2
```

can be written using if then as

```
if a>5 then
    b = a*2
else
    b = a/2
endif
```

The power of course comes from the fact that this operator can be used as any other operator (+,/,*,-...) inside longer expressions, dramatically reducing the need for if -then.

```
a = 32 + (a>5 ? a*2 : a/2) * 4
```

It can be used with string variables as well:

```
string = "my " + (RND(0,10) > 5 ? "car " : "dog ") +"is blue"
```

Because it is an expression it can be nested into itself, making very efficient condition.

Instead of:

```
temp   =  ( a == "R") ? "red" : "green"
result =  ( a == "B" )? "blue" : temp
```

we can write

```
result =  ( a == "B" )? "blue" : ( a == "R") ? "red" : "green"
```

The exact equivalent of the above single line can be described by if-then as:

```
if a=="B" then result = "blue"
else
    if a=="R" then result = "red"
    else
        result = "green"
    endif
```

```
endif
```

Another example:

```
a = ""
type = (TYPE(a)==FLOAT) ? "float" : TYPE(a)==STRING ? "string":
"integer"

print "a is ", type
```

Note: Unlike if-then command, in the Conditional operator both expressions (YES and NO) are processed regardless of the state of the operand and then the correct answer will be used. This is a safer way than using if-then for the same expression, because we will be notified of any error immediately regardless if the operand is yes or no, nut it can also may came up as a surprise.

```
a = 0
b = 1
c = (b>2) ? 1/b : 2/a

Error on line:4 - Division by zero: <int>2 / <int>0
```

3.8 Functions

Oscar Script can also have user functions. Function is declared with syntax:

```
function MyFunction(var1,var2,var3)
...
...
return nret
```

NOTE: When using function, you need to use `end` in your main (also called root) program: The program execution should never get to the function declaration itself.

Example:

```
// calling the function
MyFunction(0)
...
// we need to end main program
end

// Function declaration
function MyFunction(A)
...
return 0
```

Function Parameters
The arguments list the input parameters. They can be from 0 to 9 arguments

```
function Test(nVar1,nVar2)
```

There is no type declaration in Oscar Script and so the function arguments will be assigned the type on run-time depending what you will pass into the function

```
Test(0,"Script")
```

Return Value
Function should return a value using return command. While return value is not mandatory, you should specify return of 0 or `nil` even in function that doesn't return value just to keep warning off.

```
return value
```

Example:

```
//calling function
rnd = RandomFL()
println rnd
end

// function declaration
function RandomFL()
A = RND(0,100)/100.0
return A
```

Local instance

One very important and unique property of functions is that variables (except Global variables) inside functions are local to the function instance only. You may think of function as a whole separate script and can communicate to other parts only through function parameters, return value and global variables.

Example:

```
A = 10
MyFunction()
// A is still 10
println "A outside: ",A
end

function MyFunction()
// A is a local variable declared only within the function
A = 100
println "A inside: ",A
return 0


A inside: 100
A outside: 10
```

Variables declared in main program will not be visible in the functions, unless they are passed through the function arguments
Variables declared in functions will not be visible in other functions or main program unless passed as return value

```
// We declare B in main section of script
B = "Test"
MyFunction()
println B
end

function MyFunction()
// B doesn't exist in this instance !
// We will get warning that B has not been initialized yet
A = B
return 0

Warning at line 9:  B  was used without being initialized first.
```

Recursive Calling
Normally recursive calling should be avoided as it is very hard to debug. However there is a limited number of depth a function can call itself from within itself (or another proxy function) - the depth is set to 10 recursive calls after which an error will be issued and the program will termnate.

```
funct_A()

end

function funct_A()
funct_B()
return
```

```
function funct_B()
funct_A()
return
```

Script Started
Error on line: 10 - Unsafe Nested Recursion - aborting << funct_B ( ) << funct_A ( ) << funct_B ( ) << funct_A ( )
<< funct_B ( ) << funct_A ( ) << funct_B ( ) << funct_A ( ) << funct_B ( ) << funct_A ( )
Script terminated due to Error

Debugging Functions
Debugger 105 would normally not jump inside functions when using step commands, just evaluate them like any normal functions. You can however set break point inside a function if you need to, but be aware that the breakpoint will be deleted as soon as it is reached for the debugger to function properly. Read more in the debugging 105 section.

3.9 Type Conversion

There are few ways how to convert between the common types.

Explicit type conversion for integer and float
Normally we can let the script to worry about figuring out the best way to assign types, but in some cases we may want to explicitly convert the type (for example rounding up numbers)

Syntax	Explanation	Example

`float = FLT(int)`	Explicitly converts integer number into a float.	`intNum = 125`
		`floatNum = FLT(intNum)`
		
		float floatNum 125.000000
		int intNum 125

`int = INT(float)`	Explicitly converts float number into an integer	`a = 4.25`
		`b = INT(a)`
		
		float 4.25 will convert into integer 4

String type conversion
Here it is a little more interesting. There are few ways how you can convert string to number and numbers to string.

Syntax	Explanation	Example

`string = CHAR(ascii_number)`	Converts ASCII number into a string character	`// ASCII 65 is 'A`
		`ch = CHAR(65)`
		
		Local Variables
		string ch "A"

`ascii_number = ASC(string_character)`	Converts string character into an ASCII number. In case if the string has more characters, only the first one will be taken into account	string "A" will convert into integer 65. Sting "ABCD" will also convert to 65
`string = STR(number)`	Converts number (float or integer) into a string. *STR also works on integer or float arrays	integer `123` converts to string `"123"`, float `123.5` converts to string `"123.500000"`
`number = VAL(string)`	Converts number in the string into an integer or float * VAL also works on string arrays	converts `"123"` into integer `123` , converts `"123.5"` into float `123.5`
`string = Format(int, minWidth)`	Converts integer to string and fill the rest with leading 0 to have at least minWidth number of characters *Format also works on integer arrays	`formNum = Format(123, 10)` Local Variables string formNum "0000000123"
Type		

`type = TYPE(variable)`	Test the type of a variable. If it was never assigned it returns 0 (can be tested against `FALSE`)	`a = ""`
		`if TYPE(a)==STRING then`
		`  print "it is string"`
		`endif`

	Otherwise it returns `INTEGER, FLOAT, STRING` with respective values 1,2 or 3 It will also return `REFERENCE` (-1) in case the type is a reference to array.	

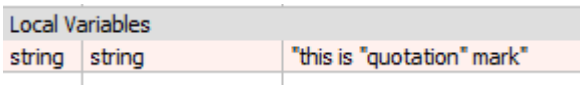
3.10 String Operators

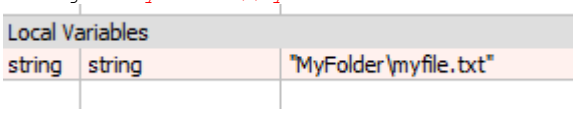
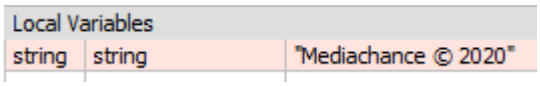
Oscar Script has very large and comprehensible set of string tools. It is especially good with using text clippboard where the script can process strings in clipboard in various ways.

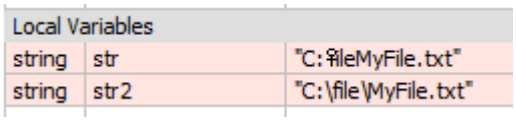
String Syntax:
```
string = "this is string"
```

Be aware of escape characters that are defined by backslash: \

Syntax	Explanation	Example

`\"`	quotation mark inside string, also `\042` 'octal" value can be used	`string = "this is \"quotation\" mark"`
		or
		`string = "this is \042quotation\042 mark"`
		 

`\r\n`	New line inside string	`"line 1\r\nline 2"`
`\\`	Backslash, also `\134` 'octal" value can be used	`string = "MyFolder\\myfile.txt"` 
`\xxx`	A character from ASCII can by typed directly using its 3 number octal value after \ - if you look at most ASCII tables, they will be represented using Decimal, Hex and **Octal** values. This allows you to enter characters for which you don't have key on your keyboard.	`string = "Mediachance \251 2020"` 

`_R`	**Raw** string syntax
	String following immediately the `_R` will be considered RAW string and no escape sequence will be recognized. If you put for example `_R"\"` there will be literally \ written in the string - not a escape sequence ".
	
	This is especially useful for RegEx functions as trying to write regex with escape sequences in place is just recipe for disaster.
	
	As such when using `_R` prefix, it is impossible to write `"` character. You would either need to resort back to standard string or add the character to the string with + operator.
		`str = "C:\file\MyFile.txt"`
		`str2 = _R"C:\file\MyFile.txt"`
		 

Syntax	Explanation	Example
`string = Left(string, nNum)`	returns nNum characters from left	`string = Left("ABCDEFGH", 5)` Output: ABCDE
`string = Right(string, nNum)`	returns nNum characters from right	`string = Right("ABCDEFGH", 5)` Output: DEFGH

`string = Mid(string, nPos, nCount)`	returns nCount characters starting at nPos. nCount of 0 means 'till the end'	`string = Mid("ABCDEFGH", 2, 3)`
		Output: `CDE`
		`string = Mid("ABCDEFGH", 3,0)`

		Output: DEFGH
int = Length(string)	returns string length	intA = Length("ABCDEFGH") Output: 8
string = Trim(string)	removes white-spaces from beginning and end of the string	string = Trim(" abcdef\r\n ") Output: abcdef
string = MakeUpper(string)	returns uppercase of the string	string = MakeUpper("My String") Output: MY STRING
string = MakeLower(string)	returns lowercase of the string	string = MakeLower("My String") Output: my string
char = GetCharAt(string,nPos)	returns a character (string) from string at a position nPos (zero based)	sChar = GetCharAt("ABCDEF",3) Output: D
string = SetCharAt(string,char,nPos)	sets 'char' at position nPos (zero based) and returns the string	string = SetCharAt("ABCDEF","d",3)) Output: ABCdEF
int = Equals(string,string)	No Case compare, returns 1 if two strings say the same, otherwise 0	res = Equals("Hello World","HELLO world") Output: 1
int = Find(string, substring)	returns int position of substring inside the string, -1 if nothing was found. Position is zero based * accepts arrays as string	nPos = Find("My name is Script", " ")) Output: 2
int = ReverseFind(string, substring)	returns int position of substring inside the string but searched from back	nPos = ReverseFind("My name is Script", " ") Output: 10
int = FindOneOf(string,charSet)	returns int position of first character that matches any character in charSet, position is zero based	nA = FindOneOf("New(old)","(){}[]") Output: 3
string = FindNumbers(string)	returns a string with extracted first occurrence of numbers from left	str = FindNumbers("File0019.T222") Output: 0019
string = ReverseFindNumbers(string)	returns a string with extracted first occurrence of numbers from right	str = ReverseFindNumbers ("File009T123En") Output: 123
string = Replace(string, sOld, sNew)	Replace all sOld substrings with sNew inside string	string = "Mon Tue Wed Thu Fri" string = Replace(string, " ", ",") Output: Mon,Tue,Wed,Thu,Fri
string = ReplaceNoCase(string, sOld, sNew)	No Case Sensitive version of Replace. Will replace sOld, regardless of the case.	
string = Insert(string, nPos, sInsert)	Inserts sInsert to string at nPos	str = Insert("ABCFG",3,"de") Output: ABCdeFG
string = Delete(string, nPos, nCount)	Deletes nCount characters from string starting at nPos (zero based) If nPos is -1 then it deletes nCount of strings from the back of the string	str = Delete("ABCDEFG",2,3) Output: ABFG str = Delete("ABCDEFG",-1,1) Output: ABCDEF

`string = Reverse(string)`	returns a string that is a reverse of the original string	`str = Reverse("ABCDEFG")` `Output: GFEDCBA`
`int = IsNumeric(string)`	returns 1 if string is numeric only (integer), otherwise 0. Tests only for integer numbers	`str = IsNumeric("AB192") - NO` `str = IsNumeric("1235") - YES`

`string = Tokenize(string,delimiter,nSkip)`	returns next token in a string separated by delimiter, `nSkip` determines how many of such found tokens to skip before returning the token - essentially it is a zero based occurrence of the token;
	0 will find first token, 1 will skip first and return second....
	
	If nSkip is `REFERENCE` (-1) then the function will return a reference to a string array containing all the tokens.
		`str = Tokenize("Apple, Banana, Car",",",1)`
		`Output: Banana`
		
		`str = Tokenize("Apple ,Banana, Car",",",2)`
		`Output: Car`
		
		`str[] = Tokenize("Apple ,Banana, Car",",",REFERENCE)`
		`Output array: ["Apple ","Banana"," Car"]`

`string = Extract(string,sStartTag,sEndTag,nSkip)`	Extract strings between sStartTag and sEndTag strings. Good for parsing html or xml strings or other structured text that have tags.
	nSkip determines how many of such strings to skip before returning one, if set to 0 then it returns the first of such string.
	if sStartTag = "" it returns string from beginning to the sEnd;
	if sEndTag = "" it returns string from sStart till the end
	The nSkip is ignored if either sStartTag or sEndTag is ""
	The operation will work even if start Tag and end tags are the same
		`string = "<A>oscar</A><B>Banana</B>,<A>Apple</A>"`
		`str = Extract(string,"<A>","</A>",1)`
		
		`Outupt: Apple`

`int = SaveString(string,sFilename)`	Saves string to FileName in Documents: /My Document/Multikeyboard/Files/ returns 0 if failed, 1 if OK	`bOK = SaveString("Test String","filename.txt")`
`string = LoadString(sFilename)`	Loads string from File in Documents: /My Document/Multikeyboard/Files/, return the loaded string or "" if failed	`string = LoadString("filename.txt")`

`string = BASE64(string,ENCODE|DECODE)`	encode/decode string using BASE64, ENCODE =1, DECODE = 0
	
	When DECODE is used, the string will return "" if non BASE64 characters are found.
		`output = BASE64 ("VGhpcyBpcyBhIHRlc3Q=",DECODE)`
		
		`Output: This is a test`

`int = RegexMatch(string,regex)`	Returns 1 if string matches regular expression defined in `regex` otherwise 0
	
	It can be used to test if string matches certain conditions. See some useful Regex strings
		`regex = _R"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"`
		`int = RegexMatch ("oscar@script.com",regex)`
		`Outupt: 1`
		
		`test = "UPPERCASE LETTERS 123"`

	Note: use the `_R` raw string prefix before the string. This will not parse the string for escape sequences and take it exactly as it is written.	`int = RegexMatch(test,_R"[A-Z0-9\s]+")`
		
		`Outupt: 1`

`string[] = RegexSearch(string,regex)`	RegexSearch is described in array Functions 83 as it always returns array	

Some useful RegEx strings for **RegexMatch**

RegexMatch	Regex pattern
is uppercase and numbers	`_R"[A-Z0-9\s]+"`
is lowercase and numbers	`_R"[a-z0-9\s]+"`
is single word only (no numbers)	`_R"^[A-Za-z]+$"`
is integer or float number	`_R"^[-+]?[0-9]*\.?[0-9]+$"`
is float number (will not match integer)	`_R"^[-+]?[0-9]+\.([0-9]+\b)?|\.[0-9]+$"`
is integer number (will not match float)	`_R"^[-+]?\d+$"`

Tokenize example:

```
date = GetDate()

//date is in format MM/DD/YYYY
//extract parts with Tokenize
//into strings
month = Tokenize(date,"/",0)
day = Tokenize(date,"/",1)
year = Tokenize(date,"/",2)

day_as_number = VAL(day)
month_number = VAL(month)

m_str = "JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC"

currentmonth = Tokenize(m_str," ", month_number-1)

print "Today is: ", day_as_number," ",currentmonth
```

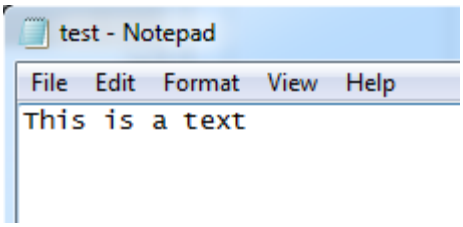
Local Variables		
string	currentmonth	"FEB"
string	date	"02/13/2020"
string	day	"13"
int	day_as_number	13
string	m_str	"JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC"
string	month	"02"
int	month_number	2
string	year	"2020"

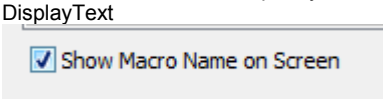
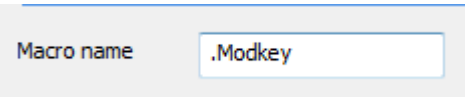
3.11 Clipboard and Key functions

These functions will not be active during Script Editing - their output will be directed only to the Output Window. Once the script is validated and you exit Script Editor into Macro window, these commands are active for testing using the macro trigger.

Syntax	Explanation	Example
`clipboard = GetClipboardText()`	Returns a text in Clipboard	
`SetClipboardText(string)`	Sends text in string to clipboard	

`SendKeyStroke(string)`	Send hardware keystroke to active window such as CTRL C.
	
	Special commands: `HOLD, RELEASE` and `PAUSE`
	
	These special commands can instruct to hold and release key at precise sequence with other keystrokes and also add pause.
	
	`SendKeyStroke("HOLD CTRL")`
	`SendKeyStroke("A")`
	`SendKeyStroke("1")`
	`SendKeyStroke("PAUSE")`
	`SendKeyStroke("RELEASE CTRL")`
	
	Warning: You have to **RELEASE** the key if you use **HOLD** or the key will remain stuck!
	
	SendKeyStroke works on hardware keys not characters. Not all character could be represented by hardware key as some keys can type different character under different locale/country. Hence SendKeyStroke will not accept every character as a parameter.
		`modifiers: SHIFT CTRL WIN ALT`
		`right side modifiers: RSHIFT RCTRL RALT`
		
		`SendKeyStroke("CTRL C")`
		
		`Send Key Stroke to Windows: <CTRL> + C`
		
		Special Keys
		`RETURN DOWN UP LEFT RIGHT DELETE BACKSPACE F1..F12 PAGEDOWN PAGEUP SPACE TAB HOME ESC`
		
		You can also send a direct hex code of the keyboard key inside the string with prefix:`0x`
		
		`SendKeyStroke("SHIFT 0xde")`

`SendText(string)`	Send (type) text to keyboard buffer. The text will be typed in active window as if typed by keyboard. The text is represented as characters (not hardware keys) which is the difference between SendText and SendKeystroke. Where SendKeyStroke is for single key combination, SendText is for typing entire texts.	`SendText("This is a text")` 
`CallMacroBlock(string)`	Calls Named Macro Block outside the Script (see Macroblocks 110)	`CallMacroBlock("finish"))`
`DisplayText(string)`	Display overlay text on bottom part of the screen	`DisplayText("This is overlay")` 

		If Show Macro Name on Screen is used, then the macro Name will flash quickly first before the DisplayText  To disable showing the Macro Name, put a . (dot) in front of the Macro key name. 

Saving and loading variable tables.
Variables can be saved and also loaded back. The file format is a readable/editable XML file. This is a simple way how to save some data or settings that can be recalled later.

Syntax	Explanation	Example
`SaveVarTable(file)`	Save current local and global variables into a XML file	`SaveVarTable("test.xml")`
`LoadVarTable(file)`	Loads local and global variables from XML file	`LoadVarTable("test.xml")`

Format of the file:

```
<?xml version="1.0"?>
<variables>
    <global>
    </global>
    <local>
        <var name="someInteger">
            <type>INTEGER</type>
            <value>46</value>
        </var>
        <var name="someFloat">
            <type>FLOAT</type>
            <value>123.456001</value>
        </var>
        <var name="someString">
            <type>STRING</type>
            <value>This is a string&#13;&#10;this is a new
line</value>
        </var>
    </local>
</variables>
```

Loading variable table in script

```
LoadVarTable("test.xml")
```

Local Variables		
float	someFloat	123.456001
int	someInteger	46
string	someString	"This is a string\nthis is a new line"

3.12 Slider Functions

Virtual slider function allows you to manipulate sliders, buttons, toolbar buttons or other on-screen controls on host applications.

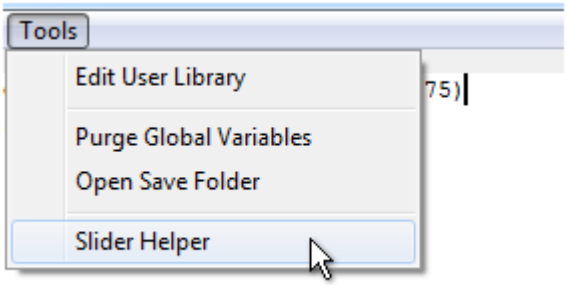
Obviously, for seamless control it is best if you can first find a shortkey for the function you want to control; for example in Photoshop brush size can be changed by sending shortkey [or].
However shortkeys or shortcuts are not always available for all functions that are visible on screen in most apps and this is where you can utilize the Slider Function as a sort of "plan B".

It is important to note that it works with absolute coordinates so the application window (and the control object you are manipulating) has to be always in the same position. The best is to use Activate window 14 step with Maximize Window function before the script to bring the desired application to front and maximize it.

Syntax	Explanation	Example
`nSliderID = DefineSlider(nX,nY,nW,nH)`	Define **horizontal** slider on the screen. The function returns the ID of the defined slider that you use in the SetSliderPos	`See slider helper`
`nSliderID = DefineSliderY(nX,nY,nW,nH)`	Define **vertical** slider on the screen. The function returns the ID of the defined slider that you use in the SetSliderPos	`See slider helper`
`SetSliderPos(nSliderID, fPos)`	"click" on the slider at fPos (0...100) percentage.	`SetSliderPos(nSliderID, 83)`
`SetSliderPosDL(nSliderID, fPos, nDelay)`	same as above, but can introduce larger delay (milliseconds) in the mouse click in case the host application fails to register the click	`SetSliderPosDL(nSliderID, 83,20)`

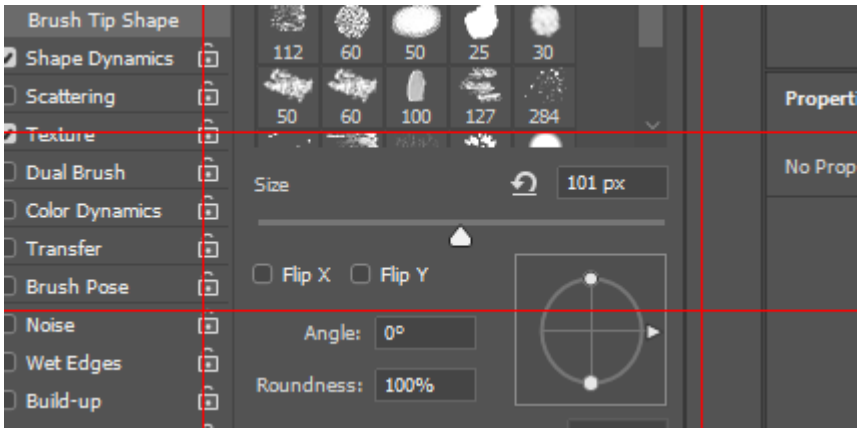
There are two steps, first is to **define** the slider coordinates as it is presented on screen.

To help you define the slider position, we added Slider Helper function:

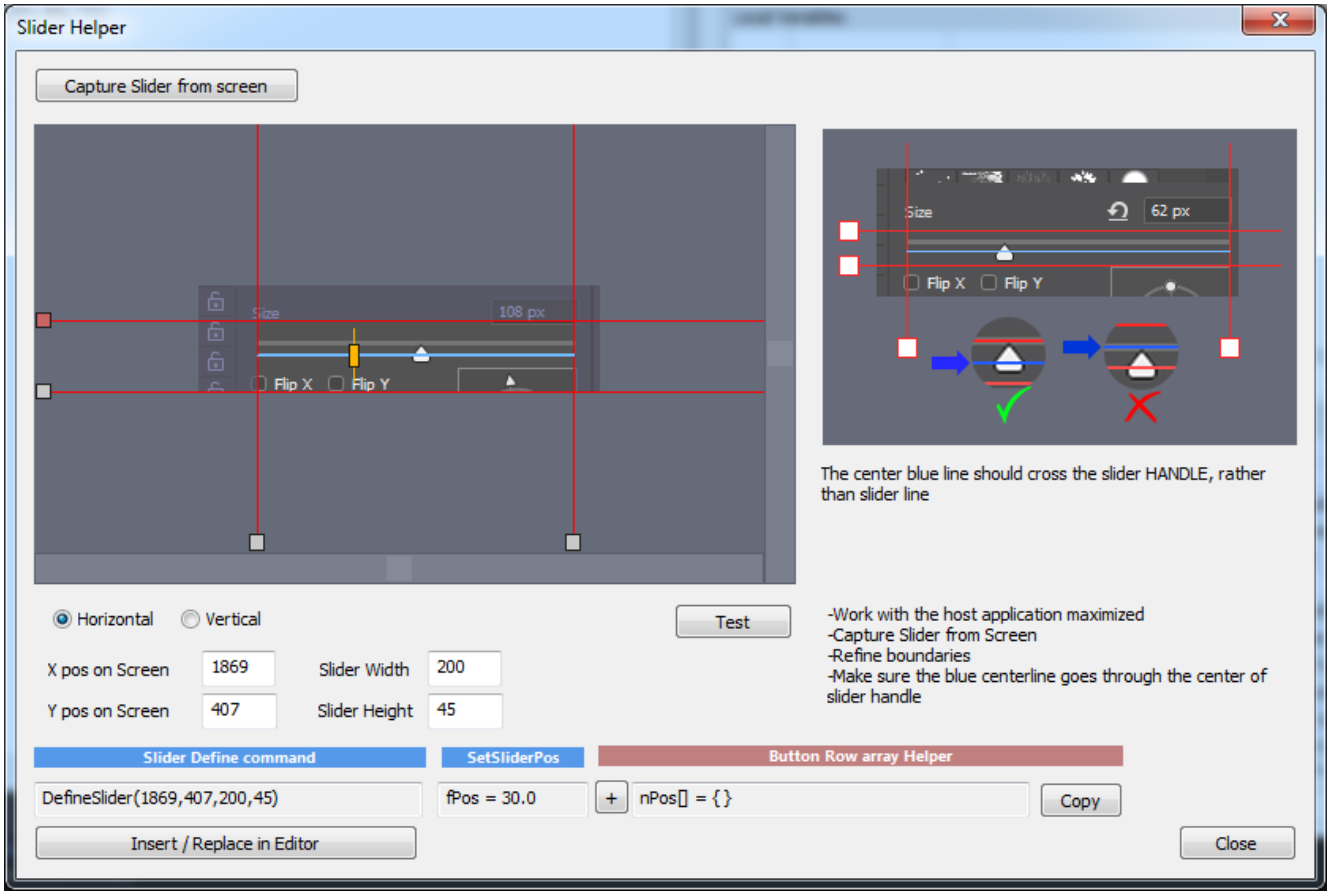


This will allow you to capture a portion of the screen around the slider and precisely define the slider dimensions, then create automatically DefineSlider command.

1. Maximize the host application (for example Photoshop)
2. Set it so the slider you want to control is always visible - for example Brush Size by opening the panel
3. Click Capture Slider from screen
4. Draw rectangle around the slider - draw a bigger rectangle than the slider as you can crop to exact dimensions in the next step



5. Refine the slider sides and center line using the white tabs.



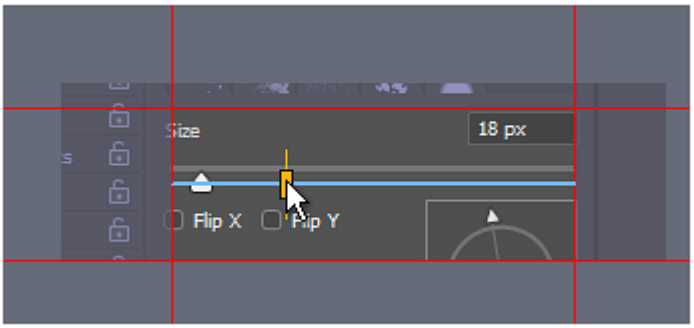
The **blue** center line determines where the function will "click" on the slider. It is important so it goes through active part of the slider - which usually is the handle of the slider. A slider may have other embellishments that are not actually active, for example the slider line on the above slider as captured from Photoshop is in fact on the edge of where the slider is active - which starts at the line and goes down the height of the slider handle. Making the blue center line cross the handle is a safe way to be sure we are clicking on the active part.

Move the vertical red lines so it **tightly** crop the sides of the slider. Move the horizontal lines so the blue center line crosses the handle of the slider (the actual height of slider doesn't matter, the function will "click" along the blue line)

You can press **Test** button which will hide the Slider Helper and simulate clicks on the slider on the screen (make sure you have the host window opened bellow) - one at minimum, one in middle and one at maximum. If one of the edge ones (minimum or maximum) doesn't register, you need to crop that side tighter.

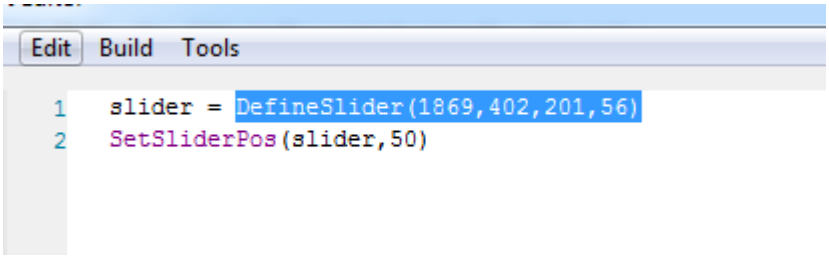
The Slider Define command will show the actual define command line for this situation.
When you press **Inser/Replace in Editor**, this line will be entered in the editor at cursor place.

You can also move the virtual slider handle



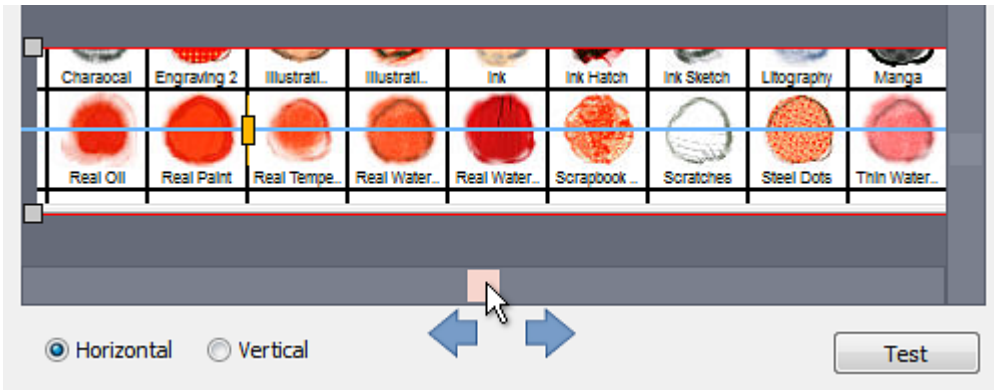
Which will change the SetSliderPos entry - this is only for your information to see what value of the fPos correspond to the actual slider position.

Note: If you select the DefineSlider in the text editor and then call Slider Helper the actual numbers will be used in the dialog and you can refine them, or re-capture the slider and overwrite with new data.



Horizontal and Vertical sliders
Sliders can be horizontal or vertical. They differ in the DefineSlider vs DefineSliderY, otherwise everything is the same.

Scroll
When you capture area larger than the preview you can scroll around with the scroll buttons.



Note:
The Capture slider image is only for initial setting up the slider - the image is not going to be stored anywhere after you press OK so when you call Slider Helper again there will be just a generic image of a slider.

To use the slider

Once the slider is defined, you can then use SetSliderPos. You can of course define many sliders and the way they are recognized is by nSliderID value the DefineSlider returns when called. Each time you use DefineSlider in your script it will create a new slider and return a new slider ID (that starts from 0 and then increments)

```
nSliderID = DefineSlider(1869,402,201,56)
```

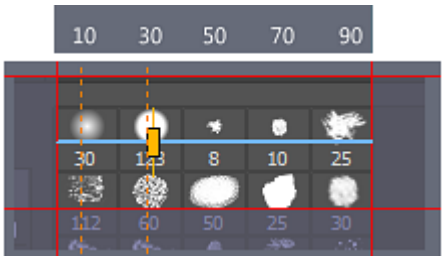
then you can use the nSliderID in your script to adjust that particular virtual slider.

```
SetSliderPos(nSliderID,50)
```

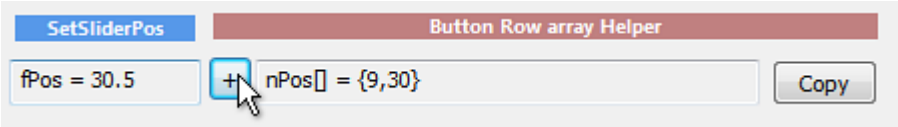
The values of the slider are 0...100 and this is a float numbers, so you can use a finer step such as 35.5, The range basically represents a percentage of the slider width so 50 would be slider at the middle.

Using it for buttons, row of buttons etc...

You can easily use this function to click on buttons or a row of buttons as well not just on slider.
Capture the button coordinates or the button row, set the left and right crop and make sure the blue line goes through the center the of button(s). Then use SetSliderPos with fPos = 50 and it will click in the middle of the button. If you use row of buttons, you can move the yellow slider around and see what fPos value represents with each button middle:



Clicking + button will add the value to the Button Row array helper line that you can then copy to clipboard with Copy Button



For example we can set a different brush in photoshop by simply setting a different value to the "slider" defined around the brush buttons.
It is important to note that if you using a hardware controller such as knob or slider, as you moving the knob or slider you are essentially calling the script multiple times in rapid succession - which on buttons can be interpreted as a double-click.
Particularly the example above with Photoshop Brushes - the brushes also respond to double click and open Brush name. You need to create more smarter script to avoid clicking on the same brush twice if you want to use knobs.

Delay Version

The SetSliderPos is set for a reasonably fluid speed on sliders, but on some software this may be too fast and the software will not register the mouse movement or click, particularly on some buttons.
Therefore there is a second version that allows to fine tune the delay between mouse clicks (particularly the delay between clicking mouse down and releasing it)

```
SetSliderPosDL(nSliderID,fPos, nDelay)
```

The delay is in milliseconds and you should try to slow it down by trying 20 ms and higher. The maximum is 200ms. If it doesn't work even with 200 ms delay then something else may be amiss.

Example:

```
SetSliderPosDL(nSliderID,50, 20)
```

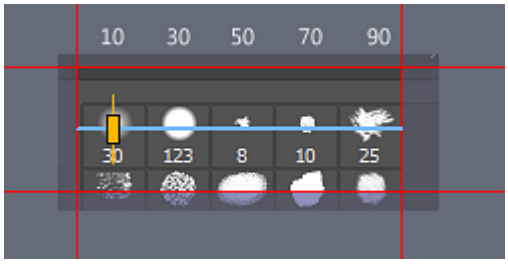
Real-Life Examples

Example 1:
We will map a rotary controller or slider (such as on APC mini) to Photoshop Brush size. Obviously DefineSlider is set for the particular screen situation with your photoshop and cannot be just blindly copied from here.

```
// MIDIvalue is in range 0...127 we remap it to 0..100
velocity = (MIDIvalue/127.0)*100.0

nBrushSize  = DefineSlider(1870,392,198,74)

SetSliderPos(nBrushSize,velocity)
```

Note: Because the action is performed by mouse clicking at the slider on screen you have to be careful not to draw with the mouse on the same time you are moving the hardware slider.

Example 2:
Using hardware slider to change the brush type in Photoshop.



Script A

```
// in range 0...127 we remap it to 0..100
velocity = (MIDIvalue/127.0)*100.0

// we have 5 brushes in row, so 100/5 = 20

nBrush = INT(velocity/20)
```

```
// position of the brushes as we determined from Slider helper

nPos[] = {10,30,50,70,90}

// make sure we have max index 4
nBrush = MIN(nBrush,4)

nBrushID  = DefineSlider(1867,265,158,58)

SetSliderPos(nBrushID,nPos[nBrush])
```

If you try this script, you will realize it has one problem: As mentioned above those particular brush buttons in Photoshop also respond to double-click. If we use hardware slider with a script to "click" on these brushes, moving the hardware slider will call the script in rapid succession. Photoshop will interpret our attempt as double click and open Brush name.

The script above is working in general, just not in the case where the buttons responds to double click... and that is precisely our case. You may think that introducing pause step before or after script may work and in some way that is correct, however introducing delay will also make the hardware slider seems sluggish and easily skip the beat. Moving it too fast may not even register any changes as we will most of the time spend in the Pause so we need to do it some other way....

What we need to do is to avoid calling the same nBrush index twice in a row. To do so we need to use Global Variable.

Here is revised script.
Note, we increased our position index from 1 because global values always start from 0 when not yet defined - this way our script will start ready armed

Script B:

```
// in range 0...127 we remap it to 0..100
velocity = (MIDIvalue/127.0)*100.0

// we have 5 brushes in a row, so 100/5 = 20

// our brush index will be from 1... 5
nBrush = INT(velocity/20) + 1

// we start our brushes index from index 1 not 0 on purpose because
//GLB_OldBrush is by default 0 if this script never ran before

// position of the brushes as we determined from Slider helper
// the first 0 is dummy to start the real values from index 1
nPos[] = {0,10,30,50,70,90}

// make sure we have max index 5 and min index 1
// see the way MAX and MIN functions are used here for that purpose

nBrush = MIN(nBrush,5)
nBrush = MAX(nBrush,1)

nBrushID  = DefineSlider(1867,265,158,58)
```

```
// if it is not the same as last time, then click on it
if nBrush!= GLB_OldBrush then
    SetSliderPos(nBrushID,nPos[nBrush])
endif

// remember this brush for next time
GLB_OldBrush = nBrush
```

The above script works well for the desired task.

3.13 Math & Constants

Syntax	Explanation	Example
`nMin = MIN(nNum, nNum)` `nMax = MAX(nNum, nNum)`	finds minimum and maximum of two numbers	`fNum = 100.0` `fMin = MIN(255.0, fNum)`

`nrand = RND(nMax)`
`nrand = RND(nMin, nMax)`
	Generates random number between nMin (or 0 if nMin is omitted) and less than nMax
	
	`nMin>= nrand < nMax`
	
	(nMax value will not be generated)
		`nrand = RND(10)`
		
		`nrand = RND(-10,10)`

`nAbsoluteVal = ABS(nNumber)`	returns absolute value of number	`nAbs = ABS(-3)`
`fRes = COS(nFloat)` `fRes = SIN(nFloat)`	Sine and cosine	

`nNumber = hex2dec(string)`	Converts hex number in string into decimal number. The hex string can have prefix 0x or not. Large hex number will result in negative integer if it reaches INT_MAX	`nNumber = hex2dec("0xFF")`
		`255`
		`nNumber = hex2dec("AB4EA")`
		`701674`

`string = dec2hex(nNumber)`	Converts integer number to hex in string. There will be no prefix	`string = dec2hex(2000)` `7d0`

Syntax	Explanation	Example

`INT_MAX`	maximum positive number integer can have. After this value the integer sign will flip.	`a = INT_MAX`
		`b = a+1`
		
		Local Variables
		int a 2147483647
		int b -2147483648

`SETTIMEOUT_MS`	Sets a safety timeout in milliseconds. After that time the script will simply abort to avoid infinite loops	`SETTIMEOUT_MS = 3000`
		
		`sets timeout to 3 seconds`

`OUTPUT`	Special variable that receives output of all print commands within the current script.	`OUTPUT = "" // clears any previous print`
		`println "This is a test of OUTPUT variable"`
		
		`for i = 1 to 5`
		`  print i`
		`next i`
		`// save the output to file`
		`SaveString(OUTPUT,"numbers.txt")`

`step`	can be used within a loop to assign a new step of the for -next loop	`for i = 1 to 256`
		`  print i , ","`
		`  step = step*2`
		`next i`
		
		`1,3,7,15,31,63,127,255,`

ENCODE DECODE	Used for Base64 53 command. Encode is value 1, Decode value 0	output = BASE64 ("VGhpcyBpcyBhIHRlc3Q=",DECODE)
true false TRUE FALSE nil NULL	Boolean helper constants true is 1, false is 0 nil is value 0 NULL is empty string	a = 12 > 7 b = a==true? "YES": "NO"
M_PI	3.141593	a = SIN (M_PI)

INTEGER, FLOAT, STRING, REFERENCE	Constants that are returned by TYPE(variable)command.
	They have integer value of 1,2 and 3
	REFERENCE has value of -1
		if TYPE(a)==STRING then
		  print "'a' it is string"
		endif

MIDIvalue	Carries value or key velocity of the trigger controller or key/pad Values of 0-127 Can be also overwritten in script, in case of testing in editor	DisplayText(STR(MIDIvalue)) inside editor the value is 100, you can overwrite it in the script or directly in the editor variable window
MIDInote	Carries value or midi note key if the trigger is a MIDI control (slider, etc) not a key then the MIDInote is 1000+CC to distinguish it from note message	
NOTE_ON NOTE_OFF	Helper constants for SendMIDI function. NOTE_ON is 0x90 NOTE_OFF is 0x80	SendMIDI(NOTE_ON,1, MIDInote,3)

Boolean operators & | !
Apart to the obvious <,>, != operators we can use AND, OR and NOT
When these operators are used on values in range 0-1 they are boolean operators

AND	OR	NOT
BOOL1 & BOOL2	BOOL1 | BOOL2	!BOOL
AND operator 0 & 0 = 0 1 & 0 = 0 0 & 1 = 0 1 & 1 = 1	OR operator 0 | 0 = 0 1 | 0 = 1 0 | 1 = 1 1 | 1 = 1	NOT operator !0 = 1 !1 = 1

You can derive other logic

```
NAND = !(a & b)

XOR = (!(a & b)) & (!(!a & !b))

or much simpler using != (is not equal)
XOR = (a!=b)
```

Bitwise operators & | ~
Using the operators on integer numbers will compare each bits of the number and give us a result that will be a different number

Bitwise AND	Bitwise OR	Bitwise NOT
INTEGER1 & INTEGER2	INTEGER1 | INTEGER2	~INTEGER

a = 0b101010 //42
b = 0b010101 //21
c = a & b //000000 = 0
	a = 0b101010 //42
	b = 0b010101 //21
	c = a | b //111111 = 63
		// bitwise NOT is inverse of all bits in the integer (making positive number negative)
		a = 103 // binary:
		0000000001100111
		b = ~a; // binary:
		1111111110011000 = -104

How does the system knows when | and & it is boolean operator and when it is bitwise? It doesn't; a bitwise | and & operation on 0 and 1 are simply behaving as Boolean operations.

Left and Right Shift

Bitwise left shift << and bitwise right shift >> operators will shift the integer number to the left or to the right

```
a = 0b100     //  00100 = 4
b = a << 2    //  10000 = 16
c = b >> 1    //  01000 = 8
```

Example: set and clear bit in integer using bitwise operators

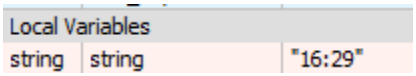
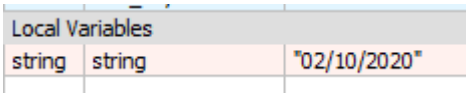
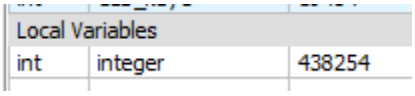
Set bit	Clear bit	Test bit

a = 0b01111 //15
// we want to set bit 5
// bit 5 is 10000
bit = 5
// Set bit formula
a = a | (1 << (bit-1))
// result is 11111 = 31
	a = 0b11111 //31
	// we want to clear bit 3
	// bit 3 is 00100
	bit = 3
	// Clear bit formula
	a = a & ~(1 << (bit -1))
	// result is 11011 = 27
		// checking bit
		bit = 5
		// Checking bit formula
		b = (a >> (bit-1)) & 1
		
		print "Number: ",a," Bit ",bit," is ",b

Example 2
Convert decimal number to binary using bitshift:

```
// input number
a = -30000

print "Dec: ",a," = Bin: "
strbin = ""
for i = 32 to 1 step -1
    b = ( a >> (i-1)) & 1
    strbin = strbin + STR(b)
next i

println strbin

Script Started
Dec: -30000 = Bin: 11111111111111110001010110100000
Script Ended OK
```

3.14 Time and Date

Syntax	Explanation	Example
`string = GetTime()`	Get time in HH:MM format	
`string = GetDate()`	Get date in MM/DD/YYYY format	
`integer = GetTickCount()`	Number of milliseconds since the app started. Can be used for timing.	

`nMSelapsed = TimeElapsed(nTickCount)`	returns number of elapsed milliseconds between now and nTickCount.
	It is basically `now-startTime` but with various checking for integer roll
		`startTime = GetTickCount()`
		`// longer operation`
		`diff2 = TimeElapsed(startTime)`
		`println "TimeElapsed:  ",diff2,"ms"`

Example:

```
// date is in format MM/DD/YYYY
date = GetDate()

//extract parts with Tokenize
//into strings
month = Tokenize(date,"/",0)
day = Tokenize(date,"/",1)
year = Tokenize(date,"/",2)

day_as_number = VAL(day)
month_as_number = VAL(month)
```

Local Variables		
string	date	"02/13/2020"
string	day	"13"
int	day_as_number	13
string	month	"02"
int	month_as_number	2
string	year	"2020"

3.15 MIDI functions

MIDI related functions and constants

Syntax	Explanation	Example
MIDIvalue	Carries value or key velocity of the trigger controller or key/pad Values of 0-127 Can be also overwritten in script, in case of testing in editor	DisplayText(STR(MIDIvalue)) inside editor the value is 100, you can overwrite it in the script or directly in the editor variable window
MIDInote	Carries value or midi note key if the trigger is a MIDI control (slider, etc) not a key then the MIDInote is 1000+CC to distinguish it from note message	
NOTE_ON NOTE_OFF	Helper constants for SendMIDI function. NOTE_ON is 0x90 NOTE_OFF is 0x80	
SendMIDI (MIDImsg,Channel,ctrlNum,ctrlValue)	Sends MIDI message to the same device that triggered the script Usually used to change color of matrix type buttons within the script In case the MIDInote is of controller the function will subtract the 1000 automatically.	Changes color of current note to 3 (Red on APC MINI) SendMIDI(NOTE_ON,1, MIDInote,3)

3.16 Global Variables, Declaration

Global variables
Normal variables are local - they exist only within the script or within a function. In order for scripts to communicate with each other or remember values, some variables could be set global.
Any variable that you want to be global needs to have prefix GLB

```
localVariable = 1.23
GLB_globalVariable = 1
```

GLobal Variables		
int	GLB_globalVariable	1
Local Variables		
float	localVariable	1.230000

If you exit the script and come back or open another script you will see that the `GLB_globalVariable` is still defined.

Global variable self destruction
Local variables stop existing when the script finish, but not global variables. Since you may create a bunch of global variables during testing, It would be confusing if unused ones will linger still there.
Similarly if you only want to use global variable to communicate across different functions in single script, you would probably want to destroy the global variable at the end of the script.

Any global variable that is assigned value of 0 or empty string will be marked for self destruction.

```
GLB_counter = 0
```

and arrays:

```
GLB_array[] = 0
```

Using self destructed Global variables
You may plan to use 0 or empty string as a valid value in a global variable. In case of numbers this doesn't create any problem because when script finds undefined global variables it automatically assumes them as numerical 0

```
GLB_counter = GLB_counter + 1
```

this line will be valid even if `GLB_counter` is not yet defined, because it will be assumed 0

however this approach would not work for strings. When variable doesn't exist but we refer to it...

```
result = GLB_string
```

GLobal Variables		
int	GLB_string	0 (Global Variable will self destruct)
Local Variables		
int	result	0

...it is assumed numerical zero. but we want a global string!

This can be solved by optional **variable declaration**.

Variable declaration
In some complex cases when multiple scripts are involved with global variables shared between them it may be beneficial to tell in advance to the script the type of variables used so we don't get an error in case they don't exist.

This is done with declare as keyword wit syntax:

```
declare <variable> as STRING INTEGER FLOAT
```

Example:

```
declare GLB_string as STRING
```

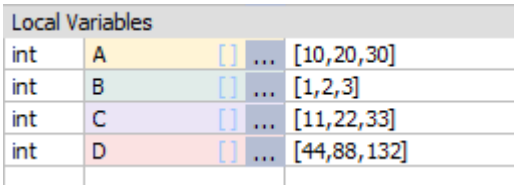
This will make sure the GLB_String will be further assumed as an string regardless if it was defined or what type of variable it was..

Important
The declare doesn't modify the value of the variable. If it was never used before it would be then assumed an empty, if it had some value of the same declared type, that would be still carried over.

3.17 Array Arithmetics

What makes Oscar Script great for arrays is that we can do arithmetic operations with arrays same way as with any other variables.

```
A[0] = 10
A[1] = 20
A[2] = 30

B[0] = 1
B[1] = 2
B[2] = 3

C[] = A[]+B[]
D[] = A[]* B[]+C[]*2
```

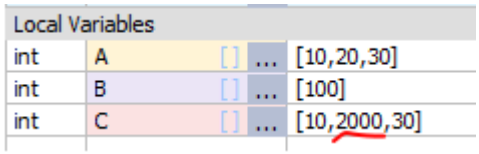


Local Variables			
int	A [] ...	[10,20,30]
int	B [] ...	[1,2,3]
int	C [] ...	[11,22,33]
int	D [] ...	[44,88,132]

The arrays are expected to be the same or overlapping range. If they are not the same range, only the overlapping area will be validated.

Example:

```
A[0] = 10
A[1] = 20
A[2] = 30

B[1] = 100

C[] = A[]*B[]
```



Local Variables			
int	A [] ...	[10,20,30]
int	B [] ...	[100]
int	C [] ...	[10,2000,30]

Arrays are validated regardless if they are sequential or not. An array can have gaps.

```
A[0] = 10
A[10] = 20
A[100] = 30

C[] = A[]*10
```

Local Variables		
int	A [] ...	[10,20,30]
int	C[0]	100
int	C[10]	200
int	C[100]	300

Create Array of certain size
Use the ARRAY command to create or fill Array. See more in Functions 83.

```
A[] = ARRAY(0,5, 100)
```

The parameters are nMin, nMax and fill. The above will create array from 0 to 5 inclusive and fill it up with numerical 100

Fill Array with value
Because assigning ARRAY can only add arrays and never remove them, you can use it to fill existing array with a numbers or strings. You can in fact fill only a certain part by using nMin and nMax smaller than Array bounds.

```
A[] = ARRAY(0,10, 0)
A[] = ARRAY(3,7, 100)
```

Local Variables		
int	A [] ...	[0,0,0,100,100,100,100,100,0,0,0]

Implicit Array initialization
An array can be initialized implicitly using this syntax:

```
A[] = {member,member,...}
```

This will always fill the array starting at index 0
Note this is similar to C++ array initialization.

Example:

```
A[] = {12,13,14}
```

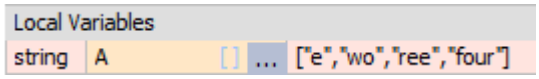
Local Variables		
int	A[0]	12
int	A[1]	13
int	A[2]	14

The implicit array initialization can be also used in array arithmetic operations directly, but it may create less readable code if overused.

```
A[] = {12,13,14}*2

if A[]=={24,26,28} then
    println "is Equal"
endif
```

You can also use implicit array initialization in functions, but the script will let you to use only one implicit array argument per function, the rest of the arguments need to be assigned to variables before you call the function.

```
rArray[] = {1,2,3,4}
A[] = Right({"one","two","three","four"},rArray[])
```



Delete Array
Assigning empty implicit array will remove all array members.
In general it is not necessary to call this function on local members as they will be removed regardless - but you can use it if you want to clean-up a global array inside your script.

```
A[]={}
```

Example:

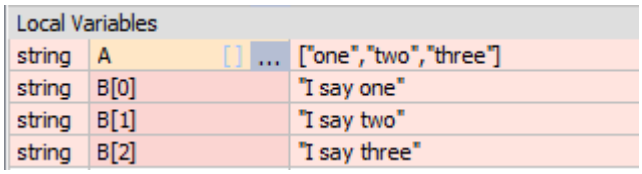
```
A[] = {12,13,14}
print A[]
A[]={}
print A[]
```

[12,13,14]
Warning at line 4 : The Array A[] was used without being initialized first. Possible error in this context!

String Arithmetic

The only arithmetic operation that woks with string is +

```
A[0] = "one"
A[1] = "two"
A[2] = "three"

B[] = "I say " +A[]
```



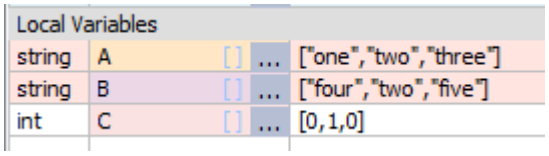
Boolean Arithmetic

Boolean operators will return an integer array with 0 or 1 depending on the condition.
You can compare both strings and numbers where appropriate (just not strings with numbers)

Boolean operator for numbers: == != < > <= >= | &
Boolean operator for strings: == != < > <= >=

```
A[0] = "one"
A[1] = "two"
A[2] = "three"

B[0] = "four"
B[1] = "two"
B[2] = "five"
```

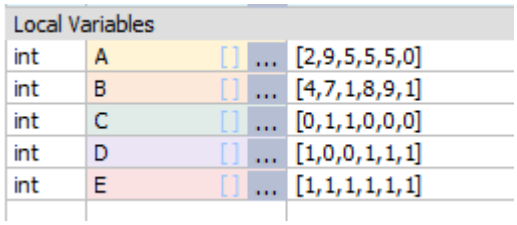
```
C[] = A[] == B[]
```



Local Variables			
string	A	[] ...	["one","two","three"]
string	B	[] ...	["four","two","five"]
int	C	[] ...	[0,1,0]

With numbers:

```
A[] = RND(ARRAY(0,5,0),10)
B[] = RND(ARRAY(0,5,0),10)

C[] = A[] >= B[]
D[] = A[] < B[]
E[] = C[] | D[]
```



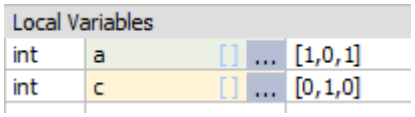
Local Variables			
int	A	[] ...	[2,9,5,5,5,0]
int	B	[] ...	[4,7,1,8,9,1]
int	C	[] ...	[0,1,1,0,0,0]
int	D	[] ...	[1,0,0,1,1,1]
int	E	[] ...	[1,1,1,1,1,1]

The | (or) and & (and) operator are both boolean and bitwise operators. Make sure when you want to use them as boolean operators that you correctly use ( ) or to split them to lines to make sure they apply to other boolean operators as in the example before (C[] and D[] are always in 0..1 range)

```
//This is correct Boolean expression because each "is larger"
produces only BOOLEAN results
correct_bool[] = (A[] > B[]) & (B[] > 5)

//This would be incorrect as BOOLEAN because if B[] has values > 1
//the result of bitwise & and integer number value would be vastly
different
//than what we expect
wrong_bool[] = (A[] > B[]) & B[] > 5
```

Additionally you can use ! (NOT) operator in front of boolean expression

```
a[0] = 1
a[1] = 0
a[2] = 1

c[] = !a[]
```



Local Variables			
int	a	[] ...	[1,0,1]
int	c	[] ...	[0,1,0]

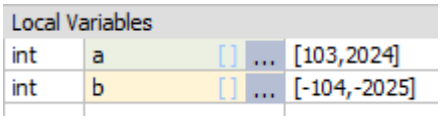
Bitwise oerators & | ~

As with the normal counterparts, you can use bitwise (binary) operators with integer arrays
If the integer number in the array item has only values 0 and 1 then it is equal to BOOLean operators, but for every other number the result is an integer number.

For clarity we used arrays with only single element.

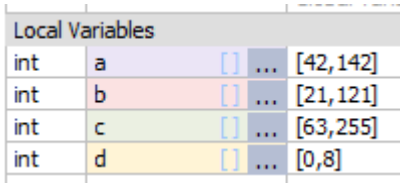
Bitwise negation using ~:

```
a[0] = 103   // binary:  0000000001100111
a[1] = 2024  // binary:  0000011111101000

b[] = ~a[]
//              binary:  1111111110011000 b[0] = -104
//              binary:  1111100000010111 b[1] = -2025
```



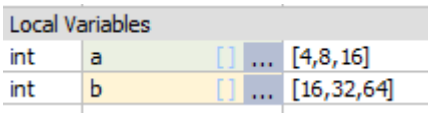
Bitwise OR and AND

```
a[0] = 42          //00101010
a[1] = 142         //10001110
b[0] = 21          //00010101
b[1] = 121         //01111001
c[] = a[] | b[]
d[] = a[] & b[]
```



Bitshift operators
In bitshift operators only the left side can be array - the right side should be integer

```
a[0] = 4
a[1] = 8
a[2] = 16

b[] = a[] << 2
```



Arrays in if-then condition

Arrays could be used in if-then condition. In such case the condition is TRUE only if all BOOLEAN results in the array are TRUE as well.

```
A[] = RND(ARRAY(0,5,0),10)

res = "NO"

if A[]<8 then
    print "All A values are smaller than 8"
    res = "YES"
endif
```

Local Variables		
int	A [] ...	[2,6,1,3,0,1]
string	res	"YES"

Unsupported Operations inside arrays

while using arithmetic with different types (strings with numbers for example) will produce error in any normal variables:

```
a = "apple"
b = a + 12
```

```
>Script Started
>Error on line: 2 - Type mismatch: Adding string and numerical value: <string>'apple' + <int>12
>Script terminated due to Error
```

Inside array this is dropped to only a warning and the operation will proceed with only items that are supported. This allows for processing arrays with multiple types without causing error where only the correct type will continue the operation.

```
a[0] = "apple"
a[1] = 24

b[] = a[] + 12
```

```
>Script Started
>Warning at line 4 : Unsupported operation inside array: mixing strings and numbers: (1 times), Ex: <string>'apple' '+' <int>12
>Script Ended OK
>1 Warning(s)
```

Local Variables		
string	a [] ...	["apple",24]
string	b[0]	"apple"
int	b[1]	36

or

```
a[0] = "apple"
a[1] = 24

b[] = "green "+ a[]
```

Local Variables		
string	a [] ...	["apple",24]
string	b[0]	"green apple"
int	b[1]	24

3.18 Array Conditional Operator

```
ArrayOperand[] ? ArrayExpressionYes[] : ArrayExpressionNo[]
```

We already had conditional operator with normal variables, but now it is time for arrays

for every item in ArrayOperand:
if the operand is evaluated as TRUE (>0) then the result of expressionYes for that item is used
if the operand is evaluated as FALSE (==0) then the result of expressionNo for that item is used

in case of operand being a string an empty string evaluates as FALSE otherwise it is TRUE.

See example:

```
a[0] = "apple"
a[1] = "car"
a[2] = "house"

b[0] = "MIXER"
b[1] = "AIRPORT"
b[2] = "KEYBOARD"

condition[0] = 0
condition[1] = 1
condition[2] = 0


result[] = condition[] ? a[] : b[]
```

Local Variables			
string	a [] ...	["apple","car","house"]
string	b [] ...	["MIXER","AIRPORT","KEYBOARD"]
int	condition [] ...	[0,1,0]
string	result [] ...	["MIXER","car","KEYBOARD"]

In the previous example we used a condition variable for clarity, but that is only one way to use it.
Of course you could put a true array condition:

```
a[0] = "apple"
a[1] = "car"
a[2] = "keyboard"

b[0] = "SOCK"
b[1] = "AIRPORT"
b[2] = "HOUSE"


result[] = Length(b[])>Length(a[]) ? b[] : a[]
```

Local Variables			
string	a	[] ...	["apple","car","keyboard"]
string	b	[] ...	["SOCK","AIRPORT","HOUSE"]
string	result	[] ...	["apple","AIRPORT","keyboard"]

In this example the result will have the longest strings from each array parts.

It is important to distinguish between

```
Operand ? ArrayYes[] : ArrayNo[]
```

and

```
Operand[] ? ArrayYes[] : ArrayNo[]
```

The first will return the whole ArrayYES or the ArrayNO depending on the normal number Operand
the second will process each individual members in the array depending on the members of the operand array.
The result may be a mixture of members from ArrayYES and ArrayNO

It is easy to make this mistake especially if you try to cram everything on one line like I did.

```
Operand ? ArrayYes[] : ArrayNo[]

condition = RND(0,2)
result[] = condition ? b[] : a[]
```

This returned either a[] or b[] array depending on the randomized number. The condition is a **normal** value.

Local Variables			
string	a	[] ...	["apple","car","keyboard"]
string	b	[] ...	["SOCK","AIRPORT","HOUSE"]
int	condition		0
string	result	[] ...	["apple","car","keyboard"]

```
Operand[] ? ArrayYes[] : ArrayNo[]

condition[] = RND(ARRAY(0,2,0),2)
result[] = condition[] ? b[] : a[]
```

This processed each member of the array separately according to the condition array and returned a mixture of a[] and b[]

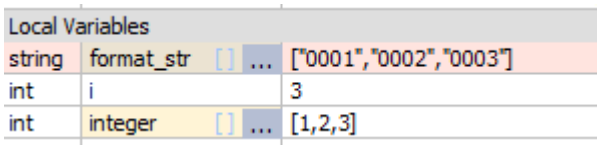
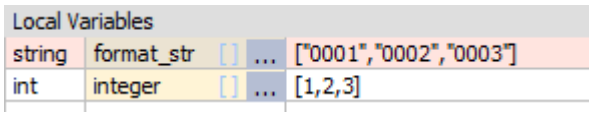
Local Variables			
string	a	[] ...	["apple","car","keyboard"]
string	b	[] ...	["SOCK","AIRPORT","HOUSE"]
int	condition	[] ...	[0,1,1]
string	result	[] ...	["apple","AIRPORT","HOUSE"]

3.19 Array Functions

In Oscar Script nearly all functions work on arrays directly.
So instead of looping over the array and calling a function on its data we can process it all at once using the array [] syntax.

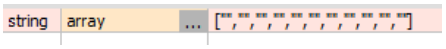
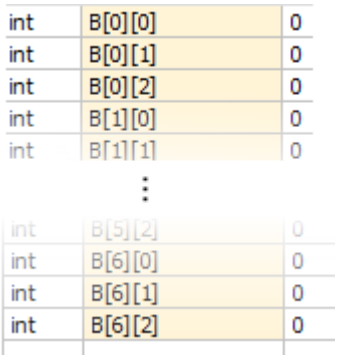
Normal way, without Array Operators	The clever way With Array Operators

`integer[0] = 1`
`integer[1] = 2`
`integer[2] = 3`
 	
`for i = 0 to 2`
`  format_str[i] =`
`Format(integer[i],4)`
`next i`
	`integer[0] = 1`
	`integer[1] = 2`
	`integer[2] = 3`
	
	`format_str[] = Format(integer[],4)`

Basic Array helper functions

Syntax	Explanation	Example

`array[] = ARRAY(nMin,nMax,fill)`
 	
`or`
 	
`array[] = ARRAY(nMin,nMax,fill[])`
	creates an array that starts from index nMin and ends at index nMax inclusive
	'fill' is a value the array will be filled with. It allows you to create 2500 items max at one time.
	Fill can be also array fill[], in which case the returned array would be multidimensional.
	
	Be aware that this function will return error if more than 2500 item are created regardless of dimensions. This is a safety precaution.
	
	ARRAY doesn't destroy previous existing arrays only adds to them.
	
	`A[] = ARRAY(0,5, 0)`
	`A[] = ARRAY(6,10, 0)`
	
	will ultimately create A[ ] from 0 to 10
		`array[] = ARRAY(0,10,"")`
		 
		two dimensional array
		
		`B[] = ARRAY(0,6, ARRAY(0,2,0))`
		 

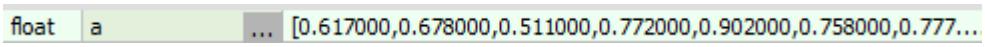
`nMin = FIRST(array[])`
`nMax = LAST(array[])`
	returns first and last index of an array. Assumes the array is sequential.
	
	Error handling:
	If array has not yet been initialized or
		`m_str = "JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC"`
		
		`array[] = Tokenize(m_str," ", REFERENCE)`

	has no members, the return of FIRST will be 0 and return of LAST will be -1
	
	`nMin = FIRST(array)`
	`nMax = LAST(array)`
	
	Local Variables
	int array Variable is not yet initialized
	int nMax -1
	int nMin 0
		`for i = FIRST(array[]) to LAST(array[])`
		`  println array[i]`
		`next i`

By creating array directly we can nest other commands on top of it to make quick function in one single line ! Of course it is a bit harder to understand when written like that.

```
a[] = FLT(RND(ARRAY(0,9,500),1000)) /1000.0
```

This will create an array a[] with 10 items [0...9] and fill it with random float number 0.5-1.0



Syntax	Explanation	Example

`float[] = FLT(int[])`	Explicitly converts integer array into a float array	`value[0] = 1`
		`value[2] = 2`
		`value[3] = 3`
		
		`float[] = FLT(value[])`
		`Output:`
		`[1.000000,2.000000,3.000000]`

`int[] = INT(float[])`	Explicitly converts float array into an integer array	`value[0] = 1.12`
		`value[2] = 2.21`
		`value[3] = 3.34`
		
		`int[] = INT(value[])`
		`Output: [1,2,3]`

`string[] = STR(number[])`	Converts number (float or integer) array into a string array.	`value[0] = 1`
		`value[2] = 2`
		`value[3] = 3`
		`strArray[] = STR(value[])`
		
		We can nest the fuctions to force conversion to float in one line:
		`strArray[] = STR(FLT(value[]))`

`number[] = VAL(string[])`	Converts numbers in the string array into an integer or float array.	`string[0] = "1.34"`
		`string[1] = "2.25"`
		`string[2] = "4.55"`
		`value[] = VAL(string[])`
		`Output:`
		`[1.340000,2.250000,4.550000]`

`string[] = CHAR(ascii_number[])`	Converts ASCII number into a string character	

`ascii_number[] = ASC(string_character[])`	Converts string character into an ASCII number. In case if the string has more characters, only the first one will be taken into account	`string[0] = "interdum"`
		`string[1] = "tempus"`
		`string[2] = "consequat"`
		
		`ascii[] = ASC(GetCharAt(string[],0))`
		
		`Output: [105,116,99]`

`string[] = Format(int[], minWidth)`	Converts integer array to string array while filling leading 0 to have at least minWidth number of characters	`value[0] = 1`
		`value[2] = 2`
		`value[3] = 3`
		`str[] = Format(value[],3)`
		
		`Output: ["001","002","003"]`

String and Numerical Functions
In many functions,**more than one parameter** can be an array.

There is always single parameter that is a **master** parameter that actually determines if the function is an array function or just ordinary function as from previous pages 53.
The **master** parameter would be written in this document as:

`string``[]`

If any other parameters can be **optional** arrays then it is written in this document:

`nNum~[]`

which means the parameter can be **either** normal parameter or an array.

For example function written here:

`string[] = Left(`**`string`**`[], nNum~[])`

means it will use first parameter as an array and the second can be optionally a number or an array of numbers.

If you put master parameter as ordinary (non array) parameter then the **normal function** will be assumed. With multiple arrays used in one function it is assumed they both have the same range and use the same indexes otherwise only the overlapping range will have the correct answer.

Operation using only master parameter as an array	Operation with both parameters as array

`string[0] = "interdum"`
`string[1] = "tempus"`
`string[2] = "consequat"`
 	
`strRes[] = Left(string[],4)`
	`string[0] = "interdum"`
	`string[1] = "tempus"`
	`string[2] = "consequat"`
	
	`left[0] = 1`
	`left[1] = 2`
	`left[2] = 3`
	
	`strRes[] = Left(string[],left[])`

Local Variables string strRes ... ["inte","temp","cons"] string string ... ["interdum","tempus","consequat"]	Local Variables int left ... [1,2,3] string strRes ... ["i","te","con"] string string ... ["interdum","tempus","consequat"]

Syntax	Explanation	Example
`nMin[] = MIN(`**`nNum`**`[], nNum~[])` `nMax[] = MAX(`**`nNum`**`[], nNum~[])`	finds minimum and maximum of two arrays or an array and a number	`array[] = RND(ARRAY(0,10,0),100)` `nMin[] = MIN(array[], 50)`

`nrand[] = RND(nMin[],nMax~[])`	Generates random number between the values stored in **nMin**[..] array and less than value of nMax or values inside nMax[..] array.
	
	range is:
	
	`nMin>= nrand < nMax`
	
	(nMax value will not be generated)
		`nMin = 0`
		`nMax = 100`
		`random[] = ARRAY(0,10,nMin)`
		`random[] = RND(random[],nMax)`

`nAbsoluteVal[] = ABS(nNumber[])`	returns absolute value of number	`array[] = RND(ARRAY(0,10,-100),100)` `abs[] = ABS(array[])`
`fRes[] = COS(nFloat[])` `fRes[] = SIN(nFloat[])`	Sine and cosine on array	

Syntax	Explanation	Example

`string[] = Left(string[], nNum~[])`	returns nNum characters from left
	
	nNum can be just integer, or it can be an array itself
		`string[0] = "interdum"`
		`string[1] = "tempus"`
		`string[2] = "consequat"`
		
		`strRes[] = Left(string[],4)`
		
		Output: `["inte","temp","cons"]`

`string[] = Right(string[], nNum~[])`	returns nNum characters from right nNum can be just integer, or it can be an array itself	`strRes[] = Right(string[],4)` **Output:** `["rdum","mpus","quat"]`

`string[] = Mid(string[], nPos~[], nCount~[])`	returns nCount characters starting at nPos. nCount of 0 means 'till the end'
	
	nPos and nCount can optionally be arrays as well.
		`string[0] = "interdum"`
		`string[1] = "tempus"`
		`string[2] = "consequat"`
		
		`nC[0] = 1`
		`nC[1] = 2`
		`nC[2] = 3`
		
		`strRes[] = Mid(string[],4,nC[])`
		
		Output: `["r","us","equ"]`

`int[] = Length(string[])`	returns string length	`string[0] = "interdum"`
		`string[1] = "tempus"`
		`string[2] = "consequat"`
		
		`iRes[] = Length(string[])`
		
		Output: `[8,6,9]`

`string[] = Trim(string[])`	removes white-spaces from beginning and end of the string	`string[0] = " interdum "`
		`string[1] = "  tempus  "`
		`string[2] = "    consequat  "`
		
		`string[] = Trim(string[])`

`string[] = MakeUpper(string[])`	returns uppercase of the string	`string_o[] = MakeUpper(string[])`
`string[] = MakeLower(string[])`	returns lowercase of the string	

`char[] = GetCharAt(string[],nPos~[])`	returns a character (string) from string at a position nPos (zero based)	
`str[] = SetCharAt (string[],char~[],nPos~[])`	sets 'char' at position nPos (zero based) and returns the string	
`int[] = Equals(string[],string~[])`	No Case compare, returns integer array with values: 1 if two string pairs are the same, otherwise 0	

`int[] = Find(string[], substring~[])`	returns int array of position of substring inside the string, -1 if nothing was found. Position is zero based	`array[0] = "test me"`
		`array[1] = "Summer"`
		`array[2] = "Domestic"`
		`//using array`
		`find[] = Find(array[],"e")`
		
		Output: `[1,4,3]`

`int[] = ReverseFind(string[], substring~[])`	returns int position of substring inside the string but searched from back	
`int[] = FindOneOf(string[],charSet)`	returns int position of first character that matches any character in charSet, position is zero based	
`string[] = FindNumbers(string[])`	returns a string with extracted first occurrence of numbers from left	see example in Replace
`string[] = ReverseFindNumbers(string[])`	returns a string array with extracted first occurrence of numbers from right from each string	

`string[] = Replace(string[], sOld~[], sNew~[])`	Replace all sOld substrings with sNew inside each string in array	`str[1] = "file0123.txt"`
		`str[2] = "file653.txt"`
		`str[3] = "file12643.txt"`
		
		`sNums[] = FindNumbers(str[])`
		`nNums[] = VAL(sNums[])`
		`sNewNums[] = Format(nNums[],6)`
		`str2[] = Replace(str[],sNums[],sNewNums[])`
		
		Output: **`["file000123.txt","file000653.txt","file012643.txt"]`**

`string[] = ReplaceNoCase(string[], sOld~[], sNew~[])`	No Case Sensitive version of Replace. Will replace sOld, regardless of the case.	
`string[] = Insert(string[], nPos~[], sInsert~[])`	Inserts sInsert to each item string at nPos	
`string[] = Delete(string[], nPos~[], nCount~[])`	Deletes nCount characters from string starting at nPos (zero based)	
`string[] = Reverse(string[])`	returns an array of strings where each item is a reverse of the original string	

`int[] = IsNumeric(string[])`	returns array, 1 if string is numeric only (integer), otherwise 0. Tests only for integer numbers	`str[1] = "12345"`
		`str[2] ="hello"`
		`str[3] =" hello 123"`
		
		`isnum[] = IsNumeric(str[])`

`str[] = Tokenize(string,delimiter,REFERENCE)`	This function doesn't have any array as parameters but will return an array of all tokens specified by delimiter.	Using Tokenize to fill string array from a string: `m_str = "JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC"`

	Array is returned when REFERENCE (-1) is used in place of nSkip parameter You can use FIRST(str[]) and LAST(str[]) to get bounds of the returned array.	monthArr[] = Tokenize(m_str," ", REFERENCE) Using Tokenize and VAL to quickly create an array of integers string = "1,23,45,32,56,78,45" value[] = VAL (Tokenize(string,"," ,REFERENCE))
Extract has two modes: Mode A - string is Master str[] = Extract (**string**[],sStartTag,sEndTag,nSkip) Mode B - start Tag is Master str[] = Extract(string,**sStartTag**[],sEndTag~[],nSkip~[])	Extract strings between sStartTag and sEndTag strings. Good for parsing html or xml strings or other structured text that have tags. There are two modes: **Mode A** when string is array, it will extract the same tag from all items in array **Mode B** will extract multiple tags from a single string into array. nSkip determines how many of found tags to skip before returning one, if set to 0 then it returns the first occurrence . if sStartTag = "" it returns string from beginning to the sEnd; if sEndTag ="" it returns string from sStart till the end The nSkip is ignored if either sStartTag or sEndTag is "" The operation will work even if start Tag and end tags are the same You can use FIRST(str[]) and LAST(str[]) to get bounds of the returned array.	m_str = "<f1>this is first<e> and <f2>this is second<e> tag" tagS[0]="<f1>" tagS[1]="<f2>" array[] = Extract(m_str,tagS[],"<e>",0) for i = FIRST(array[]) to LAST(array[]) println array[i] next i Output: this is first this is second
int[] = RegexMatch(string[],regex)	Returns integer array with members having value of 1 if string member matches regular expression in the pattern otherwise 0 Note: use the _R raw string prefix before the string. This will not parse the string for escape sequences and take it exactly as it is written. The pattern is a Regex syntax. It is beyond the scope of this document to deal with regex syntax. There are few more examples in the String Functions 53	Test if strings are emails: emails[0] = "oscar@script.com" emails[1] = "bambus@script" emails[2] = "I.am.fish@fish.org" int[] = RegexMatch(emails[],_R"^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$") Output:[1,0,1]
str[] = RegexSearch(string,regex)	Search sub-string that matches the regular expression in regex Will always return **string array**.	string = "Saturday and Sunday is fine but not Monday" result[] = RegexSearch(string,_R"\w+day")

	If no match will be found there will be only one member of string array with empty string at 0.
	
	`string[0] = ""`
	
	You can use `FIRST(str[])` and `LAST(str[])` to get bounds of the returned array.
	
	If correct syntax is used, this can quickly search through a string and extract the corresponding matches saving you writing a lot of code. The syntax is rather complex but there are numerous sites with examples. Beware that regex for Match and Search may differ!
		`for i = FIRST(result[]) to LAST(result[])`
		`  println result[i]`
		`next i`
		
		`Output:`
		`Saturday`
		`Sunday`
		`Monday`

`RegexSearch` **samples**

There are many resources on the web about RegEx syntax. Remember use raw string option `_R` in front of the regex literal so you don't have to deal with escape characters and can use regex strings directly as written.

Extracts all e-mail addresses from text

```
string = "My email is test.cs@strawbery.org and other is
bambus@perfect-a.org"
regex = _R"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}\b"
result[] = RegexSearch(string,regex)
print result[]

["test.cs@strawbery.org","bambus@perfect-a.org"]
```

Extracts text between <TAG> and </TAG>

```
string = "This is test <TAG>and this text is</TAG> between
<TAG>tags</TAG>"
regex = _R"<TAG\b[^>]*>(.*?)</TAG>"
result[] = RegexSearch(string,regex)
print result[]

["<TAG>and this text is</TAG>","<TAG>tags</TAG>"]
```

Extracts all numbers from text

```
string = "Can it find 123.4 and 234 or it cant?"
regex = _R"[-+]?([0-9]*\.[0-9]+|[0-9]+)"
result[] = RegexSearch(string,regex)
print result[]

["123.4","234"]
```

Extract float numbers but not integers

```
string = " 123 , 990 and 123.5 or -34.6"
regex = _R"[-+]?(\b[0-9]+\.([0-9]+\b)?|\.[0-9]+\b)"
```

```
result[] = RegexSearch(string,regex)
print result[]

["123.5","-34.6"]
```

Extract hexadecimal numbers format

```
string = " hex 0xFFAA number and 0xAA3FB2 number 123"
regex = _R"\b0[xX][0-9a-fA-F]+\b"
result[] = RegexSearch(string,regex)
print result[]

["0xFFAA","0xAA3FB2"]
```

Extract uppercase letters

```
string = "Give Me ONLY uppercase leTTers"

regex = _R"[A-Z]+"
result[] = RegexSearch(string,regex)
println "Any Uppercase: ", result[]

regex2 = _R"\b[A-Z]"
result2[] = RegexSearch(string,regex2)
println "Single Uppercase At the start of word: ", result2[]

Any Uppercase: ["G","M","ONLY","TT"]
Single Uppercase At the start of word: ["G","M","O"]
```

Unsupported type

if you apply function to an array with unsupported type, the data will be turned into the output type and processed that way. This allows for hybrid arrays with multiple types inside to be processed without errors.

```
data[0] = "John"
data[1] = "Crichton"

data2[0] = 23325
data2[1] = 7657

newdata[] = MakeUpper(data[])
newdata2[] = MakeUpper(data2[])
```

Local Variables			
string	data [] ...	["John","Crichton"]
int	data2 [] ...	[23325,7657]
string	newdata [] ...	["JOHN","CRICHTON"]
string	newdata2 [] ...	["23325","7657"]

The newdata2 has now strings instead of integers.

3.20 References to Array

This is a more technical/advanced topic.
References are in the Oscar Script for some advanced tricks but if you feel overwhelmed you don't have to use References at all or even think about them and skip this entire chapter.

You can just simply use arrays, array functions and arithmetic's on arrays straight forward as described before and everything will work

All you need to do is to use [ ] syntax and work with arrays as with any other variables.
A simple example would be:

```
A[] = A[] * B[] + 45
```

or

```
A[] = ABS(A[] -100)
```

That's in a nutshell all you need to know about arrays if you don't want to know anything about references.

But if you want to know more about references, go on, read the rest of this chapter:

Ok, so this is an array:

```
array[0] = 1
array[1] = 2
array[2] = 3
```

This syntax describes the array as an object and that object represents existing data

```
array[]
```

And we can use it this way in arithmetic operation as with normal numbers.. We can however assign a normal variable to that very same array[ ] :

```
pReference = array[]
```

In this case the variable will became a reference.

Local Variables		
int	array[0]	1
int	array[1]	2
int	array[2]	3
*ref	pReference	->array[...]

The reference variable doesn't have any value, nor copies the array, it only points back to that array. It can be used as a substitute to the array. In fact you can actually name it the same as the array which will then became super confusing.

See the example:

```
array[0] = 1
array[1] = 2
array[2] = 3
```

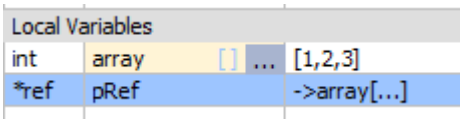
```
// this is reference - it references the array
pRef = array[]

println pRef
// it is exactly the same as calling
// println array[]

// Output [1,2,3]

// if we change the original data in any way
array[4] = 100
array[0] = 10

// the print of reference will obviously reflect that
println pRef

// Output [10,2,3,100]
```

Reference: Copy or not to copy data.

It is important to understand what syntax copies data and what not. So far we didn't copy any data at all in the above example, just referenced it.

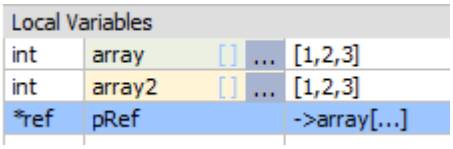
1. This syntax will create a reference variable that points to an existing array data - no actual copy of data will be done, we will get one more "reference" variable and that's it

```
pRef = array[]
```



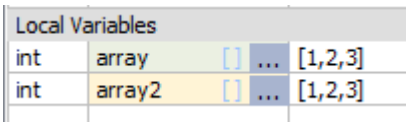
2. This syntax will **copy the data** that are referenced by pRef (which is the array[ ]) to a new array array2[ ], so we will have two copies of the same data now.

```
array2[] = pRef
```



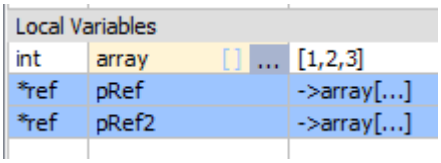
3. This syntax will create a copy of one array into another directly. It is basically a combination of 1. and 2. omitting the reference variable and it is basic array arithmetic.

```
array2[] = array[]
```



4. This syntax will create a duplicate reference - you will have two reference variables pointing both at the same data. Syntactically correct but not very useful and misleading.

```
pRef2 = pRef
```



Reference restrictions
There is a certain limit what you can reference.

You can reference only existing data. So referencing existing array[] variable directly is fine, but referencing output of a Function or array arithmetic operation may not be always possible:

```
ref = array[] + 2
```

if the `ref` doesn't already reference some other array that can take the result data of the expression you will get an error. If you think about it, when you trying to create a new reference to a result of expression - where would the new data be stored unless the reference already points to some array? .

However if the reference variable is already assigned to an existing array (even to the same array that is in expression)

```
ref = array[]
ref = array[] + 2
```

This will work fine. So can be reference used as a part of the expression:

```
ref = array[]
ref = ref + 2
```

Why do we even need references ?

Reference in Functions

The absolutely main use of a reference is as arguments and return arguments in functions

A function written as:

```
function Test(a)
    a = a *10
return a
```

may look like a simple expression that work on normal numbers - but wait, I was just showing you that references looks like normal variables as well...so what if we pass an array to the function?

If you somehow read all the above gobbledygook you will know that the variable 'a' will actually then assume role of a reference to an array and as such the expression will correctly validate for arrays, even it is not written `a[] = a[] *10`

Thanks to the references the function will actually accept both normal variable:

```
var = Test(var)
```

and array:

```
array[] = Test(array[])
```

without us doing anything special about it. Pure magic!

Pointing to some other data in the middle of calculation

Because reference is just a pointer to the actual data, reassigning reference can then point to other data in a whim, without the need to create copy or making the program less readable with if/then.

Note: There is usually not a big need to save memory or time on copy operations, especially in script as we are never working with data that is too big. Not to mention that all this is written in 2020 and not 1980s.
But it is here left as an example of possible reference usage.

```
a[0] = 2.34
a[1] = 35.6
a[2] = 17.0

b[0] = 1
b[1] = 2
b[2] = 3

test = RND(0,10)

pReference = test<5 ? a[]: b[]
result[] = pReference * 2 + pReference*3
```

The above will evaluate the expression with either a[] or b[] depending on the random generator

Local Variables		
float	a [] ...	[2.340000,35.599998,17.000000]
int	b [] ...	[1,2,3]
*ref	pReference	->a[...]
float	result [] ...	[14.039999,213.599991,102.000000]
int	test	0

Local Variables		
float	a [] ...	[2.340000,35.599998,17.000000]
int	b [] ...	[1,2,3]
*ref	pReference	->b[...]
int	result [] ...	[6,12,18]
int	test	7

Of course the above can be achieved many other ways:

```
temp[] = test<5 ? a[]: b[]
result[] = temp[] * 2 + temp[]*3
```

In which case we created an useless temporary array

or

```
result[] = (test<5 ? a[]: b[]) * 2 + (test<5 ? a[]: b[])*3
```

In which case we didn't create temporary array but made it very hard to understand just two days later

or

```
if test<5 then
    result[] = a[] * 2 + a[]*3
else
    result[] = b[] * 2 + b[]*3
endif
```

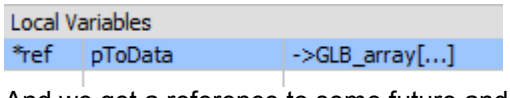
In which case we made it, I don't know, 'iffy"

Reference to a non existing Global data

The references can actually point to a Global array that doesn't yet exist. This may be a bit confusing, but with global variables, a certain variable may be created by another script at a much later time. It is almost same as in fuction, when the argument will be filled at some other time.

We can type:

```
pToData = GLB_array[]
```

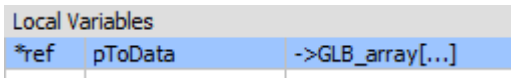


And we get a reference to some future and not yet existing array GLB_array and nothing else. We don't even know what type it is. Now any time later if we create the array GLB_array the reference will then point to its data.

You may question why this even exist? Well, without jumping too much ahead, any operations performed on the reference will be still valid regardless if the data exist or not.

So if we type:

```
pToData = GLB_array[]
pToData = pToData * 2
print pToData
```

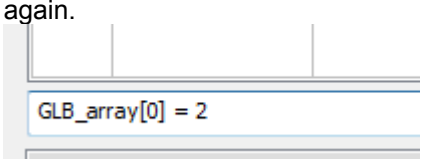


We still get no actual data taking place, nor any array was created, but the program works without error!

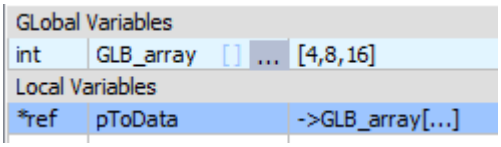
Now if we or other script creates the global variable that we are referencing:

```
GLB_array[0] = 2
GLB_array[1] = 4
GLB_array[2] = 8
```

You can for example copy each line into the edit box bellow variables and press enter. Then run the script again.



Boom, as a magic, the program now works with some actual data:

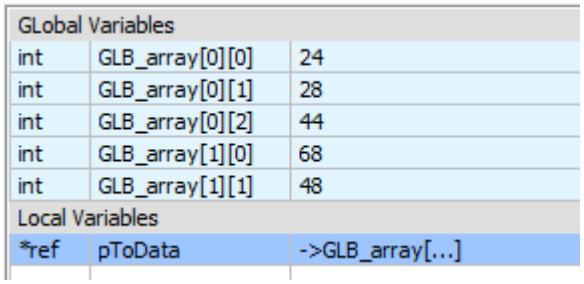


But that is not all.

What if we define instead our array as multidimensional data :

```
GLB_array[0][0] = 12
GLB_array[0][1] = 14
GLB_array[0][2] = 22
GLB_array[1][0] = 34
GLB_array[1][1] = 24
```

Well, the script doesn't care about that either and simply process it same way:



Reference as an Array
Before you ask, yes you can have a reference to an array in an array configuration

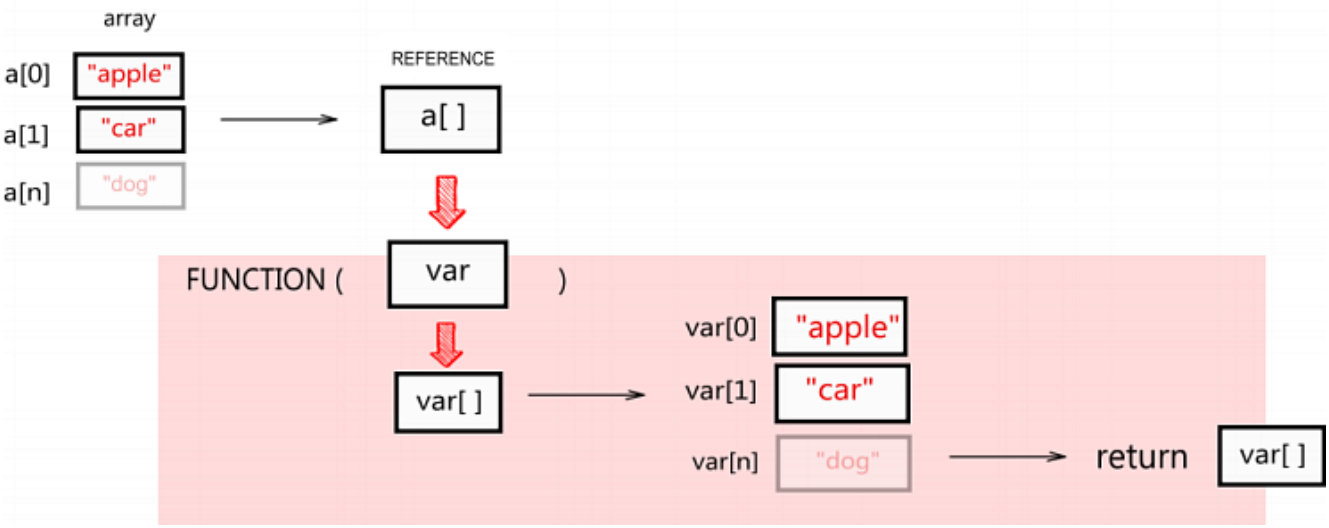
```
a[0] = 1
a[1] = 2
a[2] = 3

b[0] = 10
b[1] = 20
b[2] = 30

p[0] = a[]
p[1] = b[]

println p[0]
println p[1]
```

```
println p[0]+p[1]
println p[]

Script Started
[1,2,3]
[10,20,30]
[11,22,33]
[<*ref>a[],<*ref>b[]]
Script Ended OK
```

I assume there could be some clever way how to utilize it, but so far I think it only makes everything even more confusing....

3.21 Using Arrays in user functions

The syntax arry[] allows passing arrays to user defined functions same way as any other variables. This mechanism why it works is explained more in the References 91 chapter.




```
a[0] = "apple"
a[1] = "car"
a[2] = "dog"

MyFunction(a[])
```

The function is defined just as any other function would be - nothing special needs to be there.

```
function MyFunction(var)
    print var
return var
```

if we put a break point and stop inside the function, we will see there will be not only the array but also a reference 91 variable (and both will be named the same as it is defined in function). This is a crucial point that allows function to work with both array and normal variables the same way.

```
 9  function MyFunction(var)
10      print var
11  return var
```

Local Variables		
*ref	var	->var[...]
string	var [] ...	["apple","car","dog"]

Inside the function we can then use the new array as any other array by using the name from the function definition:

```
var[n] = ....
```

You are working on a **local copy** of the array inside the function. If you want the changes to return back to main program you need to return the var[] (or reference 91 to the var) from the function and then assign it in main program to some array variable.

So at the end of function you can use:

```
return var[]
```
or
```
return var
```

Note: As shown above we have two things going on: the array var[] and a reference variable var that points to the data of var[]. So you can in fact type return both ways.
However when you use only reference variable inside the function and also return the reference variable, then the function will work with both normal numbers and arrays alike. It is explained in the reference chapter 91 in details why it is like that.

Receive array from function
To receive the array data returned from the function into an array variable we can use syntax:

```
result[] = MyFunc( a[] )
```

Or:

```
a[] = MyFunc( a[] )
```

which will update the same array that was used for input.

Example:
This example demonstrate using arrays in user functions

```
//create an array in function
in_array[] = CreateArray(10)

PrintArray(in_array[])

println

// you can pass the array to function using []
out_array[] = QuadrFunc( in_array[] )

// print output
PrintArray( out_array[] )

// dont forget end in main function!
end

// creates array
```

```
function CreateArray(num)

    for k=0 to num
        arr[k] = k
    next k

return arr[]

// here is the clever part
// when written using straight syntax (not array[]) it will work
// for normal numbers AND arrays alike
// in one case 'a' is just a normal variable
// in other case it is a reference to array
function QuadrFunc(a)
     a = a ^ 2
return a

function PrintArray(a)
    print a
return nil
```

Local Variables		
int	in_array [] ...	[0,1,2,3,4,5,6,7,8,9,10]
int	out_array [] ...	[0,1,4,9,16,25,36,49,64,81,100]

3.22 Multidimensional Hybrid Arrays

Arrays in Oscar Script are **not your ordinary arrays** you may have seen in other programming languages. The arrays can be multidimensional, but also hybrid and non sequential - and the arithmetic still work on them!

Let me define array:

```
data[0] = 20
data[1] = 40
```

Local Variables		
int	data[0]	20
int	data[1]	40

Nothing weird about it. Now let me just add some other data to it:

```
data[0][1] = 100
data[0][2] = 200
data[1][1] = 300
data[1][2] = 400
```

Local Variables		
int	data[0]	20
int	data[0][1]	100
int	data[0][2]	200
int	data[1]	40
int	data[1][1]	300
int	data[1][2]	400

We now created hybrid multi-dimensional array. It is still same array, but it is both one dimensional and two dimensional at the same time. Now lets add more:

```
data[2][1][0] = 1000
data[2][2][0] = 2000
data[2][3][0] = 3000
data[2][1][1] = 4000
```

Our data array still holds them all even if they are now in three different dimensions.

Local Variables		
int	data[0]	20
int	data[0][1]	100
int	data[0][2]	200
int	data[1]	40
int	data[1][1]	300
int	data[1][2]	400
int	data[2][1][0]	1000
int	data[2][1][1]	4000
int	data[2][2][0]	2000
int	data[2][3][0]	3000

if we type:

```
data[] = data[] * 2
```

All dimensions will be updated! And it doesn't even matter if we have gaps in the arrays either.

Local Variables		
int	data[0]	40
int	data[0][1]	200
int	data[0][2]	400
int	data[1]	80
int	data[1][1]	600
int	data[1][2]	800
int	data[2][1][0]	2000
int	data[2][1][1]	8000
int	data[2][2][0]	4000
int	data[2][3][0]	6000

Now lets add:

```
other[2][2][0] = -1
other[1] = -1
other[100] = -1
```

And then a an operation involving both arrays:

```
data[] = data[]*other[]
```

Local Variables		
int	data[0]	20
int	data[0][1]	100
int	data[0][2]	200
int	data[1]	-40
int	data[1][1]	300
int	data[1][2]	400
int	data[2][1][0]	1000
int	data[2][1][1]	4000
int	data[2][2][0]	-2000
int	data[2][3][0]	3000
int	other ...	[-1,-1,-1]

Only the parts that overlapped in our two arrays were updated with the arithmetic!
This overlapping arithmetic however depends on the order written.
If we used

```
data[] = other[]*data[]
```

we will make the other[] array significant and the result will have only 3 members.

Array can have multiple types inside
Array members can be of different type (unlike most other languages)

```
data[0] = 20
data[1] = 3.1415
```

Local Variables		
int	data[0]	20
float	data[1]	3.141500

First item is integer, second is float. This will in fact continue with arithmetic operations if there is no loss of data, the script will keep the first one integer.

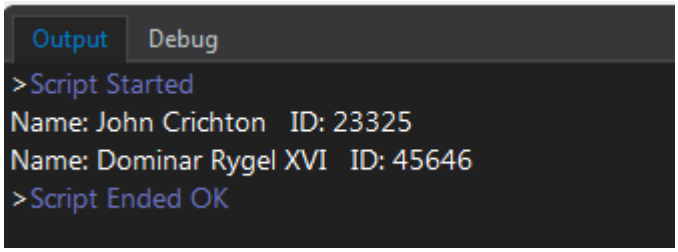
```
result[] = data[]*2
```

Local Variables		
float	data [] ...	[20,3.141500]
int	result[0]	40
float	result[1]	6.283000

However if we multiply the array by a float number then both members will became float.

This can have advantage in creating mixed data arrays (structures) without much of any effort.

```
#const FIRST_N 1
#const LAST_N 2
#const ID 3

data[0][FIRST_N] = "John"
data[0][LAST_N] = "Crichton"
data[0][ID] = 23325

data[1][FIRST_N] = "Dominar"
data[1][LAST_N] = "Rygel XVI"
data[1][ID] = 45646


for i = 0 to 1
    println "Name: ",data[i][FIRST_N]," ",data[i][LAST_N],"    ID:
",data[i][ID]
next i
```



When using arithmetic with multi-type array, only the parts that give correct answer will be processed. For example multiplying array will multiply only its numerical parts and leave string parts untouched.

However when using functions then all data will revert to the output type the best way it can and be processed that way.

On the above:

```
newdata[] = VAL(data[])
```

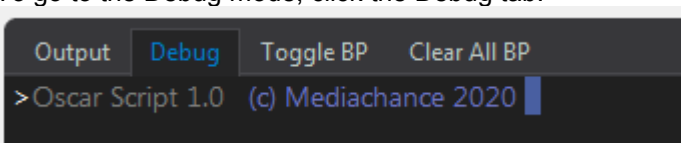
will produce:

Local Variables		
int	FIRST_N	1
int	ID	3
int	LAST_N	2
int	data [] ...	["John",23325,"Crichton","Rygel XVI","Dominar",45646]
int	newdata[0][1]	0
int	newdata[0][2]	0
int	newdata[0][3]	23325
int	newdata[1][1]	0
int	newdata[1][2]	0
int	newdata[1][3]	45646

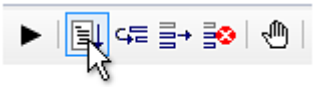
3.23 Debugging, trace

Oscar Script has quite comprehensive way of debugging.

To go to the Debug mode, click the Debug tab:

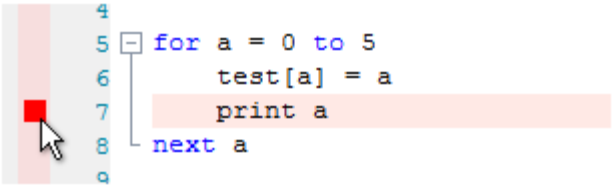


or press the Debug Mode toolbar button or Debug Mode in Menu Build



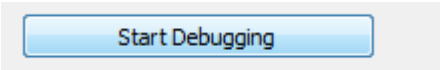
This will slightly change the look of the editor and add few buttons.

Break Points
The editor left marker bar can be used to add or remove **Break Points** with mouse. Break point on current line can be also toggled with the button on the tab bar Toggle BP.

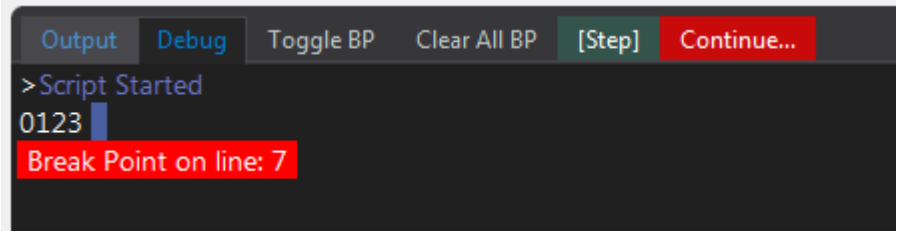


Break point is where the execution will stop and we will get to see the current variables at that point. Break point will stop the line **before** it is being executed.

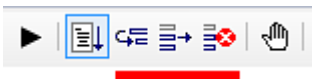
The Compile & Run button also changed into:



(If you don't set any break points the script will simply run its course)
Once we press that button the program will run but then stop at the break point. At that moment two additional buttons will appear: Step and Continue



The same buttons will be enabled in the toolbar along with Terminate button



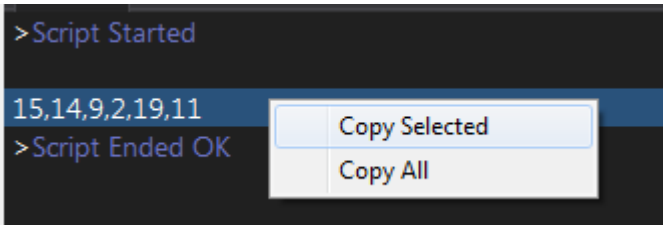
Continue will run the script from the break point till it finish or find another break point. In the case of break point in a loop we will stop next loop cycle.

The [Step] will go into a step debugging, that means the program will advance one command then stop again.

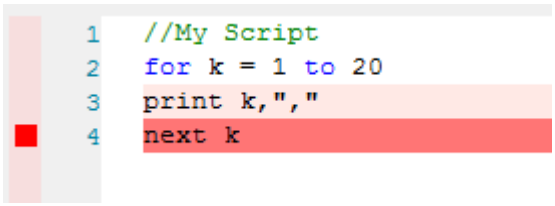
During the Debugging you can add or remove break points... but if you try to change something in the editor, the debugging session will stop as the program will need to restart from beginning to update the changes you have made.

Debugging inside Functions
Debugger normally does not jump inside functions when using step commands, just evaluate them like any normal functions. You can however set break point inside a function if you need to, but be aware that the break point will be deleted as soon as it is reached so the debugger can function properly. Once you return from the function in the step debugger the line that was calling the function will need to be executed again. This may in some rare cases produce wrong results (for example an IF statement with a global value in the condition and calling a function after 'then' that changes that global value). This is usually rare.

Copy strings from output.
You can select a line, then right click to open menu.



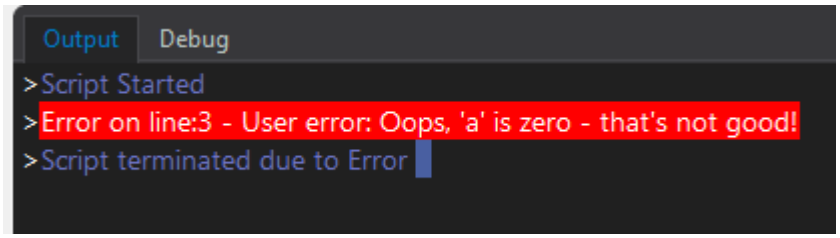
Changing Variables mid Debugging
When you are debugging, you can change variables during breakpoint and so change the outcome of next step.



Click on the variable in the list of variables, then put cursor in the edit box and change the value. Press Enter.

`throw` statement
Throws user defined error and terminates the script. This may be used for debugging parameters if they go out of desired bounds

```
if a<1 then throw "Oops, 'a' is zero - that's not good!"
endif
```

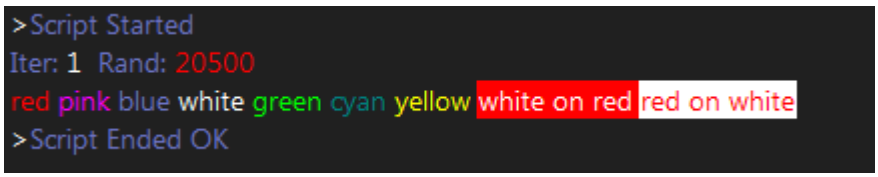


`trace` statement
Trace statement has similar syntax to `println` statement and it is used to display messages in the Output Window. Unlike `print` or `println` trace command doesn't change the `OUTPUT` string

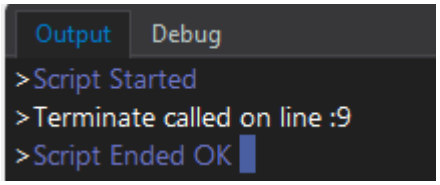
```
trace "Counter is ", counter
```

Trace statement can have color tags <..> such as <R> that would change color of the output text.

```
trace "<R>red <P>pink <B>blue <W>white <G>green <C>cyan <Y>yellow
<E>white on red <!>red on white"
```

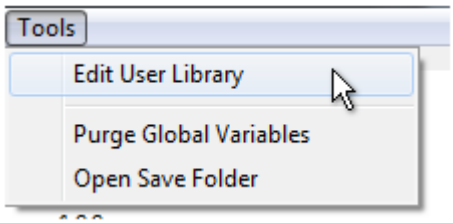


`terminate`
Terminate can be used to exit Script, but a bit more verbally than with `end`
While end is meant to be a quiet exit, the terminate will write to the output window the line where it terminates so it can be used for debugging to quickly determine where a complex script ended.



3.24 User Library Functions

You can place your custom functions you use often into an User Library.
These functions will be available for every script.



```
// User Library functions
// ***************************************************
// DONT use GOTO or GOSUB inside User Library functions!

// use this part to test the functions
// while still in script editor
// this part will be never called by the main script.

a = TestLibrary(1)


end
//********** FUNCTIONS START HERE ***************

function TestLibrary(a)
    a = a*10
    println "Hello From library, fParam * 10 = ",a
    DisplayText("Hello")
return a
```

User Library is a whole script that you can run and debug. While its main body will be never called outside the editing window (so you can and should freely use it to test the functions and even leave any code there), the functions themselves can be accessed by any other script.

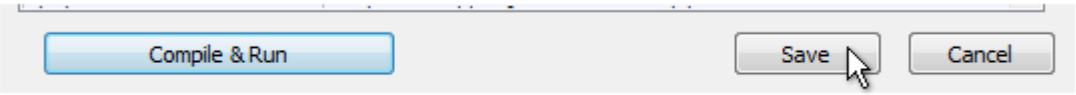
Make sure you test the user library functions well for various parameters to avoid errors.

Important:
During testing you have to call the functions you just wrote from inside the main body of User Library script (where it says: `use this part to test the functions`) otherwise the function will not be tested and so you wouldn't even know if it runs well or not until you call it in other script.

Don't use `goto` or `gosub` in the Library Functions.

You can call other Library Functions from inside the Library Functions - if you need to, although from speed perspective, it is far better to put everything in one function, even if you may need to duplicate code.

When everything looks good close the User Library by clicking click Save (You can Save only when there are no errors)



Then back in main script you can test the function you just created:

```
k = TestLibrary(5)

Hello From library, fParam * 10 = 10
```

If you read the paragraph about References 91, you will know that the function as is written will also work with arrays.

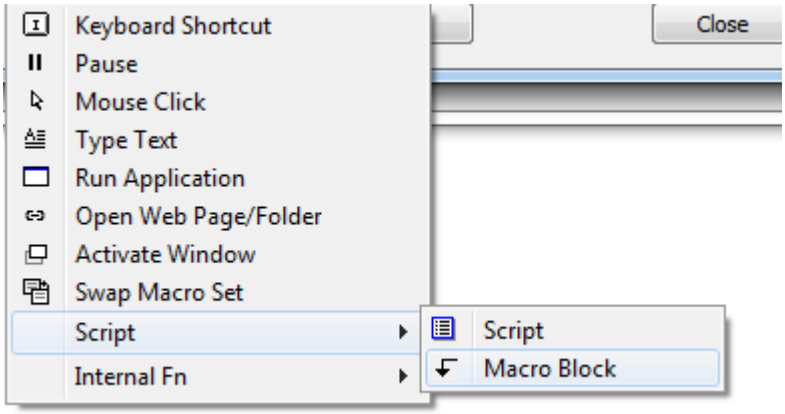
```
k[] = TestLibrary(A[])
```

Notes:
It is harder to debug User Library functions than normal functions (as normal functions are in the same code listing as the rest of your program) so it is better to put only well working and well debugged functions into user library to avoid un-necessary errors.

The `#define` can be used in Library functions as it the macros are substituted before run-time, however `#const` can be used only if it is within each function limits (as it is evaluated on run-time)

3.25 Macroblocks

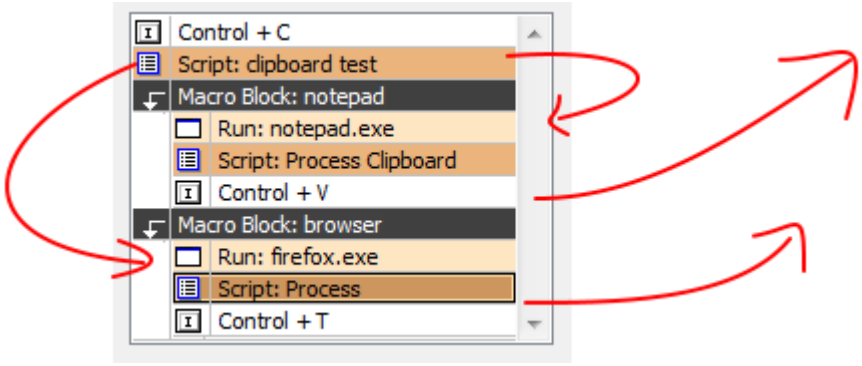
Macro blocks are special subroutines inside the Macro steps window that allow for structuring the steps and also various scripts in certain way from within the main script.
Everything after a Macro Block will be skipped normally, but can be accessible from inside script as a sort of GOTO command using CallMacroBlock.



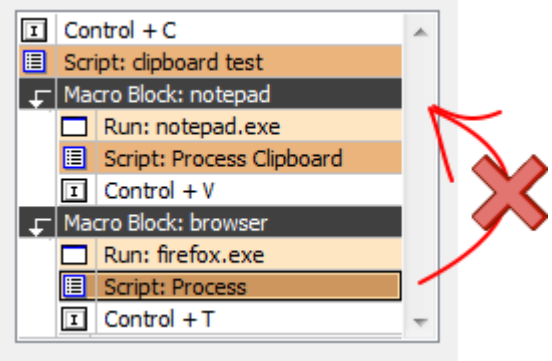
To call Macro block from within the script use `CallMacroBlock(string) such as:`

```
sClip = GetClipboardText()

 if sClip=="notepad" then
    CallMacroBlock("notepad")
 endif
```

The script will exit and the Macro block "notepad" will be called. The macro block will stop itself on the another Macro Block object (browser)



`Important:`
To avoid infinite loops which may lock up your keyboard, Script can call only Macro Blocks that are bellow the script. So a script within a macro block, cannot call macro block that is above itself



A special macroblock named KEY_OFF is used for key off (release) triggers.

3.26 Note OFF

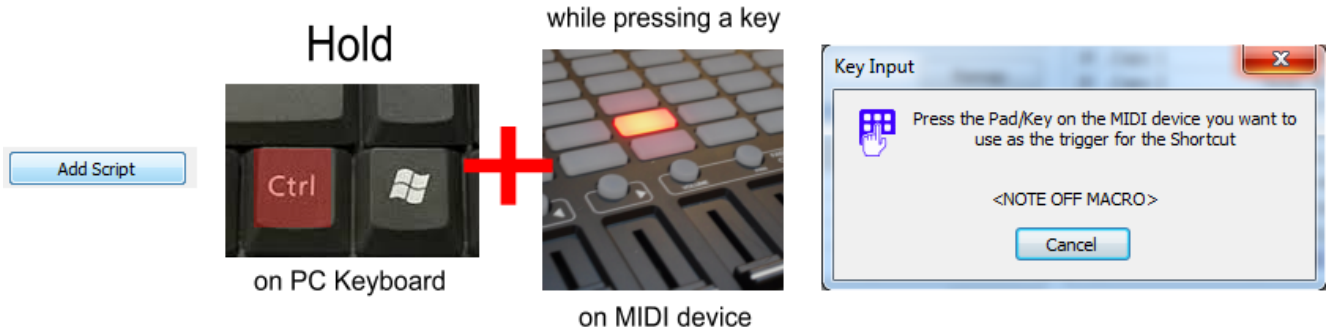
Normally keys are triggered as soon as you press them down. This will be enough for 99% of situations.

However for advanced scripting MIDI macros allows you to also capture NOTE OFF triggers (when you release button).

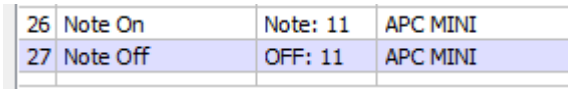
This can be done two different ways in MIDI macro:

A separate Macro for Key OFF

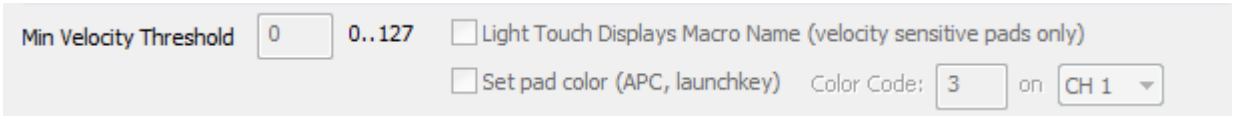
You can capture NOTE OFF message when you hold CTRL during setting up the MACRO trigger.



This will create **a separate OFF macro** - so you will have two macros for the same key; one for Note ON the other for OFF trigger.



In the OFF macro you can't set the extra key properties such as pad color or velocity threshold (OFF has always velocity 0). These properties apply only in the ON macro.



If you want to change color of the pad during ON or OFF dynamically you can do it in the script

An example for APC MINI on Note:

```
//set color to yellow
SendMIDI(NOTE_ON,1,MIDInote,5)
```

on Note OFF:

```
//set color to red
SendMIDI(NOTE_ON,1,MIDInote,3)
```

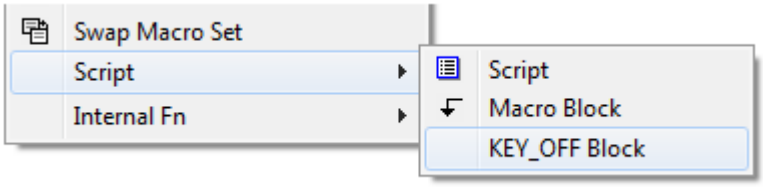
3.27 KEY_OFF Macroblock

Another way to trigger OFF messages is to use a special macro block within a single MACRO.

This will create only a single macro - this macro will be shared for ON and OFF triggers.

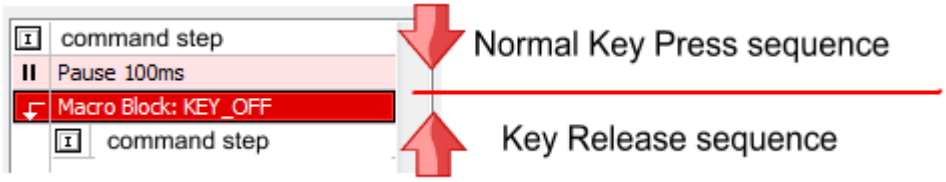
Note: The separate OFF macro as described above has priority over KEY_OFF block - if you have defined separate macro for OFF then the KEY_OFF Block in your normal macro will be not called.

As described in a Macro block section: Macro blocks 110 are special subroutines inside the Macro steps that allow for structuring the steps.



While normal MACRO BLOCKS can be called from within script as a sort of external subroutines, the KEY_OFF block (which is in fact a MACRO BLOCK that is named "KEY_OFF") is used to dedicate part of the macro steps for Key Off (release key) commands.

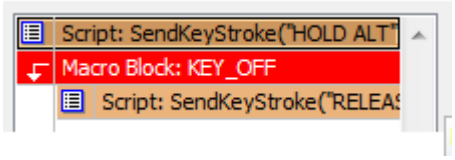
The macro block function as a stop. The normal sequence of steps will be executed on Key Press until a Macro block then return. The sequences after KEY_OFF macro block will be executed when the key is released



Example:

For example in Photoshop when you are using Brush tool, holding ALT key will change the tool to eye dropper (color pick) and when you release the alt it will go back to brush.
If we want to map this functionality to some MIDI key by simply sending keystroke ALT, it would not work because the macro will hold ALT then release it shortly afterward.

However with KEY_OFF block (or Separate OFF MACRO) and small script using HOLD and RELEASE options in SendKeyStroke we can mimic this function easily.
Here is example with KEY_OFF Block



We will need one script before KEY_OFF block and one script after.

The first script:

```
SendKeyStroke("HOLD ALT")
DisplayText("ALT ON")
```

The Key OFF script after KEY_OFF macro block:

```
SendKeyStroke("RELEASE ALT")
DisplayText("ALT OFF")
```

This will do exactly what we expect the original ALT key in Photoshop to do. While the trigger note/key is held the ALT will be hold as well, when trigger key is released, the ALT will be also released.
The DisplayText is just for show.

Note: KEY OFF Queue

Once the OFF triggers are defined in either of the two ways, they will use message Queue - that is even if OFF message occurs while the ON macro is still executing (for example some long script on the key press or use of delays), the commands defined for the key OFF trigger will be added to Queue and will be executed once the ON commands are finished. This way the OFF sequence will be always triggered, but it may not be right away if long scripts are used.

3.28 Script Examples

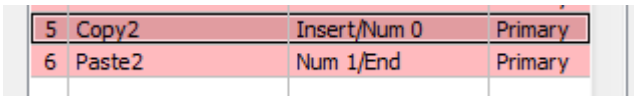
3.28.1 Clipboard example


```
//Working with clipboard

// get what is in the clipboard now
prevclipboard = GetClipboardText()

// send "copy" keystroke to windows so it will capture
// selected text in whatever app we are
//(in the Script Editor SendText and SendKeyStroke
// is disabled - obviously - or we will have mayhem
// so just copy something to clipboard manually when testing)
SendKeyStroke("CTRL C")

// get the text from clipboard to another string
clipboard = GetClipboardText()

trace "This is now in Clipboard: ",clipboard

// process the clipboard however you want
// and what I want is to make every second letter capital
// making it hard to read :)

newstring = ""
// length of the string
k = Length(clipboard)

for i = 0 to k

    // get one char at a time
    char = Mid(clipboard,i,1)

    // i % 2 is modulus = remainer of i divided by 2
    // so basically it flipflops between 0 and 1
    if (i % 2==0) then
        char = MakeUpper(char)
    else
        char = MakeLower(char)
    endif
    // and make a new string out of it
    newstring = newstring+char
next i
```

```
// set back what was in clipboard previously
SetClipboardText(prevclipboard)

//send new string to the windows as text
//so it will replace the selected text with the new one
SendText(newstring)

trace "\a01,13 mischief managed "
```

3.28.2 Secondary Clipboard

Task: We want to have a secondary text clipboard
CTR+C and CTRL+V would work with normal windows clipboard
and we want to assign two other keys that would work as a Copy 2 and Paste 2 - a secondary clipboard, so we are able to copy and paste two things independently



5	Copy2	Insert/Num 0	Primary
6	Paste2	Num 1/End	Primary

COPY 2 button


```
// remember what is in the clipboard now
previous = GetClipboardText()

// send COPY to app
SendKeyStroke("CTRL C")

// grab the clipboard
GLB_clipboard2 = GetClipboardText()

// put the previous one back
SetClipboardText(previous)
```

PASTE 2 button


```
//declare variable just in case it was not yet assigned
//so we don't get warnings
declare GLB_clipboard2 as STRING

if GLB_clipboard2!="" then
    SendText(GLB_clipboard2)
endif
```

Similarly you can extend it to more than one alternative clipboards, or a FIFO clipboard (see next example)

3.28.3 FIFO Clipboard

Task: make FIFO clipboard
Pressing "CopyStack" key will copy item into stack. So we can select multiple items and copy them to stack one after another.
Pressing "PasteStack" key will type the text that is on the top of the stack and then roll to previous item so we can "paste" items one after another that are in the stack

Button Stack Copy:


```
// send COPY to app
SendKeyStroke("CTRL C")

GLB_FIFO[GLB_fifo_counter] = GetClipboardText()

DisplayText("Copy "+STR(GLB_fifo_counter))

GLB_fifo_counter = GLB_fifo_counter+1
```

Button Stack Paste:


```
GLB_fifo_counter=GLB_fifo_counter-1

// stack is at the bottom
if GLB_fifo_counter<0 then
    GLB_fifo_counter = 0
    DisplayText("Stack Empty")
    terminate
endif

// avoid errors - test if the array item is already defined as
STRING
if TYPE(GLB_FIFO[GLB_fifo_counter])==STRING then
    DisplayText("Paste "+STR(GLB_fifo_counter))
// type the text
    SendText(GLB_FIFO[GLB_fifo_counter])
endif
```

3.28.4 Slider Clipboard

Task: Multiple text clipboards
MIDI slider (or knob) will select the clipboard bin
two additional buttons Copy and Paste will copy / paste text to the selected bin

Slider/Knob (select clipboard)


```
// index of clipboard (clipboard 0...4)
GLB_clipbNr = MIDIvalue/26
// display clipboard name
str = "Clipboard["+ STR(GLB_clipbNr+1)+"]: "

// get a partial line from clipboard if it exist to display on
screen
if TYPE(GLB_MClipboard[GLB_clipbNr]) > 0 then
    line = GLB_MClipboard[GLB_clipbNr]
    line = Left(line,120)
else
    line = "<empty>"
endif

// display on screen
DisplayText(str+line)
```

Button Copy


```
//COPY

SendKeyStroke("CTRL C")
SendKeyStroke("PAUSE")

// get the new clipboard text
GLB_MClipboard[GLB_clipbNr] = GetClipboardText();

DisplayText("Copy to Clipboard["+STR(GLB_clipbNr+1)+"]")
```

Button Paste


```
// test if clipboard exist
// if so set clipboard and paste it
if TYPE(GLB_MClipboard[GLB_clipbNr])>0 then
    SetClipboardText(GLB_MClipboard[GLB_clipbNr])
    SendKeyStroke("CTRL V")
    DisplayText("Paste ["+STR(GLB_clipbNr+1)+"]")
else
    DisplayText("Clipboard ["+STR(GLB_clipbNr+1)+"]: <empty>")
endif
```

3.28.5 XML Tags Extract

Task: parse XML document and find tags we are interested in


```
//Example of String parsing using XML tags

//Imagine we receive following string which is in XML
//and we need to extract name and associated product IDs

//It can come to the script from a clipboard
// string = GetClipboardText()
//...but for this example we just define it directly here:

string = "<LastName>Holden</LastName>\
<FirstName>James</FirstName>\
<Company>Rocinante Consulting LLC</Company>\
<ManuelOrderPrice>0</ManuelOrderPrice>\
<ShippingVatPct>0</ShippingVatPct>\
<ProdId>11302-42-0</ProdId>\
<PurchaseItemKey><Key>826724</Key>\
</PurchaseItemKey>\
<ProdId>12342-23-1</ProdId>\
<PurchaseItemKey><Key>225664</Key>\
</PurchaseItemKey>"

// a super simple way to extract a single element:
sFname = Extract(string,"<FirstName>","</FirstName>",0)
```

```
sLname = Extract(string,"<LastName>","</LastName>",0)

sFullName = sFname+" "+sLname

//we can have multiple elements with the same ProdId tag
//but we don't know yet how many
//let's try a really big number of such elements to test
for k=0 to 100
    // extract new element in each loop
    // - see the k used as nSkip in Extract
    sId = Extract(string,"<ProdId>","</ProdId>", k )

    if (sId=="") then
        // no more elements to extract
        // exit loop
        break
    endif
    // valid element, so add it to array
    sProdId[k] = sId
next k
//this is number of elements found
nNumProducts = k

// now print it all
println "Found ",nNumProducts," ID's for ",sFullName
for k = 0 to nNumProducts-1
    println "ID",k+1,": ", sProdId[k]
next k

// now we can either send it to clipbard, send it as text
// to current app, save it as a file ...

SetClipboardText(OUTPUT)
```

Output:

```
>Script Started
Found 2 ID's for James Holden
ID1: 11302-42-0
ID2: 12342-23-1
>Script Ended OK
```

3.28.6 BASE64 example

BASE64 example, Encode/Decode Secret Text:

Script A: Encodes string in clipboard to BASE64, obfuscating it by encoding it multiple times in a loop


```
//Encode to BASE64 multiple times

// get the text from clipboard
clipboard = GetClipboardText()

trace "Text in clipboard: ",clipboard

nDifficulty = 3

for i = 0 to nDifficulty
    clipboard = BASE64(clipboard,ENCODE)
next i

SendText(clipboard)
```

Script B: Decodes string in clipboard from Base64. Determines number of times it has been encoded


```
//BASE64 "Secret" decoding example

// get the text from clipboard
clipboard = GetClipboardText()

 nc = Length(clipboard)

 if nc==0 then
    DisplayText("No String in Clipboard")
    terminate
 endif

// loop sufficiently enough
for i = 0 to 10
test = BASE64(clipboard,DECODE)
// if non BASE64 characters are found
// then it returns ""
```

```
if (test=="") then
    if i>0 then
        // it failed this iteration, but we are > 0
        // it means previous iteration have succeeded
        SetClipboardText(clipboard)
        DisplayText(clipboard)
        break
    else
        // it failed to decode on first try
        // must be garbage, not BASE64
        DisplayText("Failed to decode, no Base64")
        terminate
    endif
 endif
clipboard = test
next i
```

3.28.7 Mod key Example

Implementing modifier key:
Pressing KEY (Num 9) will do one action
Pressing MODKEY (Num 8) and then KEY (Num9) one after another within 1 sec will do different action

5	.Action	Num 9/Page Up	Primary
6	.Modkey	Num 8/Up	Primary

This of course makes sense only if we have more than one action KEY defined, otherwise we don't need to bother with modifier key, just define 2 keys


```
//Script on MODKEY
GLB_ModKeyTime = GetTickCount()
DisplayText("Mod Key")
```

Now script on the action key:


```
//Script on ACTIONKEY
timeElapsed = TimeElapsed(GLB_ModKeyTime)

if timeElapsed>1000 then
// too long, we assume the modifier was not pressed or it was too
long time ago
    goto NoModKey
endif

// mod key was pressed;

DisplayText("Mod Key 1 Action")
// do the MOD action here

// don't forget to end so we don't go to NoModKey
end

NoModKey:
DisplayText("No Mod Key Action")

// do the NO MOD action here
```

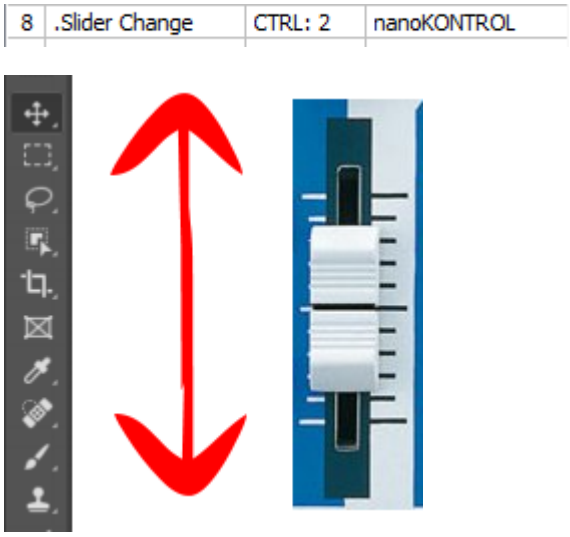
3.28.8 Recursion

This is a classic example of calculating permutations of letters in a word.
It is using recursion and while such algorithms are not encouraged in script, we used it for testing purpose.


```
// RECURSION example
// The recursion depth is set at 10 for security reasons
// after which error would be issued
// so the maximum length for permutation in this example
// would be 9 letters, and that would be 362880 permutations
// 9 letters would take probably around 30 minutes anyway
// so don't ry it

str = "OSCAR"
// with 5 letters it is only 120 permutations
// 6 letters is 720 permutations etc....

p_count = permutate("", str , 0)
println "Total: ",p_count," premutations"

end

//recursive permutation function
//formula was taken from somewhere on "internets"
function permutate(candidate,remaining, count)

g = Length(remaining)

if g == 0 then
    count = count+1
    println " Permutation: ",Format(count,3)," = ",candidate
endif

rl = Length(remaining)-1

for i = 0 to rl
    newCandidate = candidate + GetCharAt(remaining,i)
    newRemaining = Left(remaining,i) + Mid(remaining,i+1,0)
    count = permutate (newCandidate, newRemaining,count)
next i

return count
```

3.28.9 MIDI Slider to Photoshop

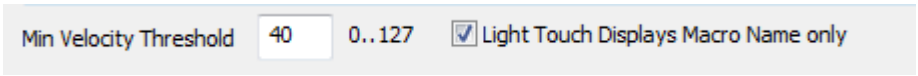
Change Photoshop tools with Midi Slider. This uses the `MIDIvalue` variable to obtain value of the control.




```
//Photoshop example with a MIDI slider

// this is triggered by a MIDI CONTROL slider

// lazy way of adding arrays
str = "Move,Marque,Laso,Crop,Brush,Clone"
keys = "V,M,L,C,B,S"
name[] = Tokenize(str,",",REFERENCE)
keystroke[] = Tokenize(keys,",",REFERENCE)

// how many items we have?
nMax = LAST(name[])+1
// what is the slider step per item?
nstep = INT(127/nMax)

// reverse so it goes from top to bottom
item = (127-MIDIvalue)/nstep

DisplayText(name[item])
SendKeyStroke(keystroke[item])
```

3.28.10 Touch Label Display

This is actually **already implemented** within the application itself as **Light Touch displays Macro Name only**:

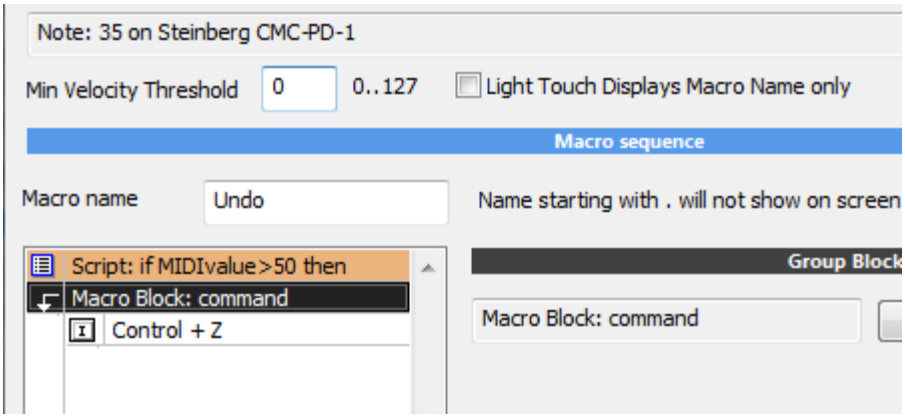


When pads on the controller are touched only very lightly (below the Min Velocity Threshold) a macro name will appear on the screen (in blue), but the actual macro will be run only when a more forceful tap pressure is used. This is great in case of large amount of pads as in launchpad type controllers.

This uses the fact that many pads are very sensitive - barely touching them will create a velocity that is larger than 0

The following script does essentially the exact same thing - only more complicated way. It was added as an example during development but then the author decided it is a functionality that is worth having permanently.

It is left here as an example of using Macro Block, or in case if you want to extend the functionality.



One way to approach this is to use Macro Block.

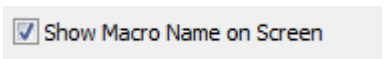
First we will set the Min Velocity Threshold to 0 so any velocity will trigger the macro

The next item is a simple script:

```
if MIDIvalue>50 then
    CallMacroBlock("command")
endif
```

Then it follows by a Macro Block with label "command" and finally the actual macro set that is executed when higher velocity is used.

How does it work:
When the pad is touched only lightly the macro will be triggered and the Macro name will automatically appear on the screen as defined in the Macro name box.
(if Show Macro Name on Screen is set to on)



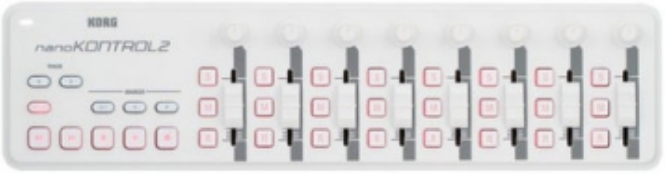
The Macro Block step works as a stop for the list of commands. It cannot be bypassed by itself, only by the script using CallMacroBlock.

So if the velocity is less than 50 the script will simply end and the whole macro list will end there as well as it cannot go past Macro Block.
When the velocity is larger than 50 the script calls the macro block and the steps listed here will be executed.

IV MIDI devices

MIDI Keyboard Macros work with most USB MIDI devices, class compliant or not. Obviously class compliant devices are refereed these days as they don't need drivers.

You can plug more than one, but each needs to be either different model or be able to change the name of the device in firmware (for example some novation launchpads). This is in order so the software can recognize with certainty which MIDI device is which.

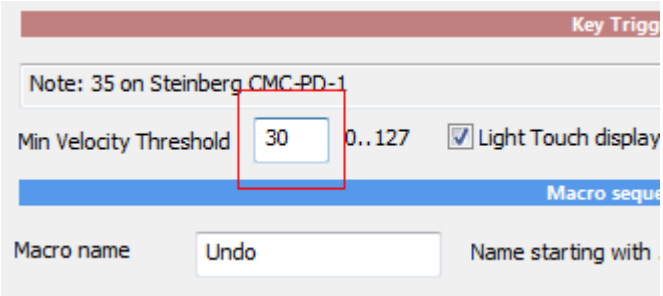


MIDI devices could be pads - which would work pretty much like a typical Macro board, keyboards - which is unusual way to launch macro, but actually in reality may work pretty well due to muscle memory or even controllers with sliders and knobs.

Keyboards and Pads
Keyboards and pads are straightforward to use, pressing on keyboard or pad will trigger the action. In addition a `MIDIvalue` variable in script will have the velocity value of the key/pad which you can use in some logic (for example skipping a word by sending CTRL+RIGHT when velocity is bellow 100 and selecting a word by sending SHIFT+CTRL+RIGHT when velocity is higher)

```
if MIDIvalue <100 then
    SendKeyStroke("CTRL RIGHT")
else
    SendKeyStroke("CTRL SHIFT RIGHT")
endif
```

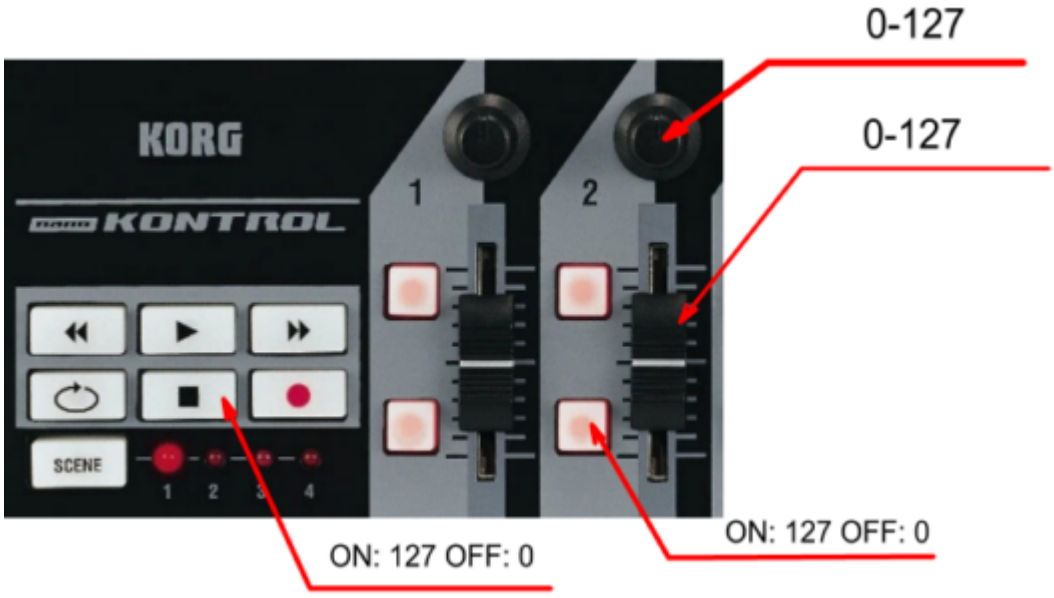
The Macro has also settings for Min Velocity Threshold. Many pads are very sensitive and it takes only a brush of fingers to trigger them. By default when creating new macro the value will be set to 30 which means all velocity bellow will be ignored. This may still be low for some sensitive pads, in that case you may increase the value. For controllers such as sliders the threshold will be set to 0 during setup as it is expected you want to use the whole range.



Light Touch displays the Name only:
This works together with Min Velocity Threshold. Any velocity below the threshold will displays only the Macro Name but not actually run the macro itself. This is great for matrix type keypads with large amount of keys.

Controllers

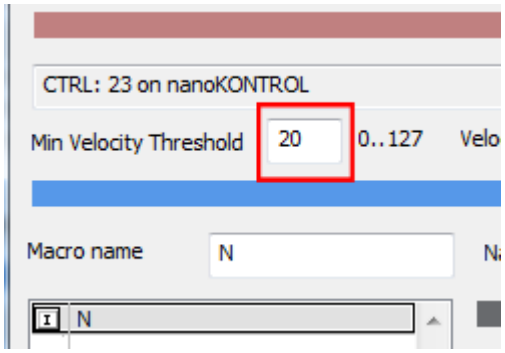
There is difference between a keyboard or pad and controller. The keyboard will trigger action when key is pressed (with an optional velocity number that can be accessed via script). Controller will send a value.
Note: on a midi keyboard the MOD wheel is such a controller.



In case of slider or knob every time you move it just a little it will trigger the action with a value that can be accessed via script.

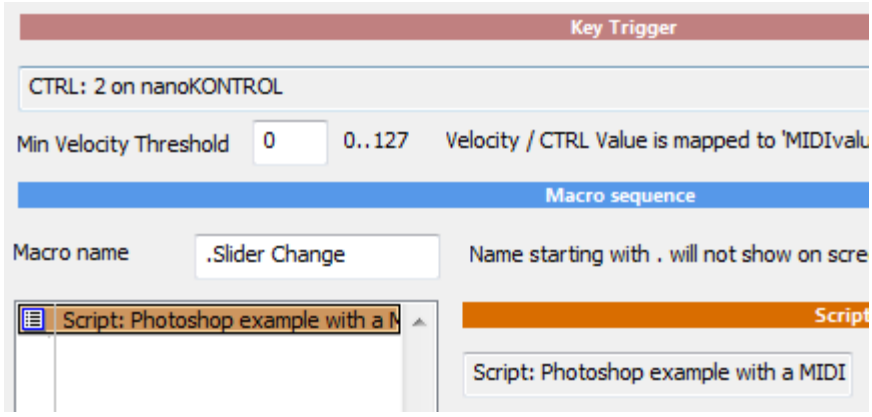
In case of control buttons (for example on the image nano kontrol - the transport buttons or the buttons near sliders) they will trigger action with value 127 when pressed and 0 when released.
This allows you to do some interesting logic.

In case of push control buttons, if the Min Velocity Threshold is set to > 0 the control will trigger the action only once when pressed, as the release action will be ignored. So you can basically use them as any other Pads or Keyboard keys.



If you set the Threshold to 0 then the macro will be triggered twice: on Push down and on Release and you can in script use `MIDIvalue` to determine if the value is 0 or 127 and then decide on the course of action.

In case of sliders and knobs the Threshold should be set to 0 to have whole range.



In case of sliders and knobs - the most logical way to use them is in script, The variable `MIDIvalue` will carry the controller value. You can of course use it any way you want, just remember any change in the value will trigger the action.

For example we can use the slider or MOD wheel to switch into 4 different tools in Photoshop depending on the position of the slider/wheel or knob. This surprisingly works pretty well.

```
if MIDIvalue < 32 then
    DisplayText("Brush Tool")
    SendKeyStroke("B")
else
    if MIDIvalue < 64 then
        DisplayText("Laso Tool")
        SendKeyStroke("L")
    else
        if MIDIvalue < 92 then
            DisplayText("Marque Tool")
            SendKeyStroke("M")
        else
            DisplayText("Move Tool")
            SendKeyStroke("V")
        endif
```

```
    endif
endif
```

Note: Put a . (dot) in front of Macro name so it won't display the macro name on screen before the DisplayText.

You can also use the Slider just to set a Global Variable that will be then used on a button action

```
GLB_Slider2 = MIDIvalue
DisplayText("Param: "+ STR(GLB_Slider2))
```

The GLB_Slider2 can be then used when you press some pad or key to do different action.

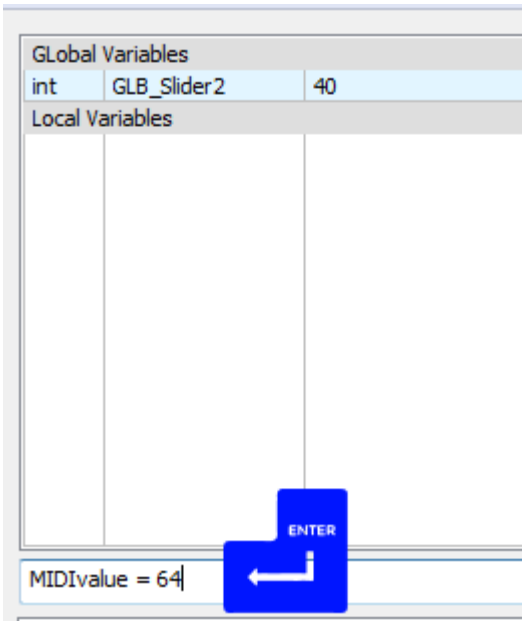
MIDIvalue during Script Editing

During the script editing the MIDI devices are disabled (that is when you have the Script Editor open, no MIDI pads, keys or controls are triggered, until you close the editor).
By default in Script Editor the MIDIvalue is set to 100
If you want to test it for other values you can temporary assign new value to it in script:

```
MIDIvalue = RND(127)
DisplayText("Param: "+ STR(MIDIvalue))
```

(remember to remove the assign line after testing is done)

or you can add new value in the variable window edit box directly and press Enter, then run script again:



V Limitations

The macros are tied to a device. That is if you plug two devices that each trigger for example a note 35, each of the devices can trigger different macros.

In order for the MIDI keyboard Macros to recognize different devices at the same time they need to be different models or have different revision/firmware number

Alternatively some MIDI devices allows to change the MIDI device name in its drivers, so you can have two of the same plugged in, but each will display as a different device.

The reason for this is that two exact same device will have the exact same ID so it is impossible to distinguish them from each other.

MIDI channels: Midi channels are ignored. The macro will be triggered regardless of the midi channel. Adding midi channel to the mix will only increase complexity and lead to a mouse hunt situation when it would be hard to determine why a MIDI device is not triggering data. Especially when setting MIDI channels on many keyboards and pads is not obvious.