SOUNDSQUARES

0423

COMMAND AND SCRIPTING REFERENCE

APRIL 2023

CONTENTS

SoundSquares uses a hybrid commandline/scripting language to store presets and customise its functionality. As well as using the host's regular storage, patch import and export is achieved using text files.

The language is configured as a series of commands with each command starting on a new line. The syntax is structured that each command starts with a verb followed by options consisting of recipients and parameters.

SoundSquares scripts run on an internal virtual machine with instructions stored in a simplified assembler-style language format. Anyone familiar with coding ASM for Z80 6502 or i86 will find it reasonably straight-forward and readable, although for a beginner it may be a little difficult to approach.

This document sets-out the commands that both sides of the system accept - the patch side and the scripting side. Both are related, with the caveat that whilst patches and macros files can contain and run ASM scripting commands, only scripting files can be used to load collections of functions for later use. Furthermore, scripts are capable of composing and triggering patch and macro commands ON-THE-FLY.

The Command-Line-Interface panel (CLI) accepts all commands, and offers additional functionality and tracer-feedback to help in constructing and debugging patches, macros, and scripts.

Assuming a known blank state, patches work by issuing the commands required to reconstruct a given setup. This is where things like the number of ROOMS, TARGETS, SOURCES, and GROUPS are determined, along with their positions on the stage, levels, phase, mute/solo status, colours, and so-on.

All aspects of the current patch configuration are setup by patching commands, including DSP functionality.

The format is broadly, as follows :

Lines starting with # or ; are comments :

```
##############################
# start of script
# leave some information here
```

Commands are structured into verbs, recipient nodenames, and parameters, and are separated by a single space character :

```
[VERB] [RECIPIENT] [PARAMETERS]
```

or in other words :

```
[DO_SOMETHING] [TO_THIS] [USING_THIS]
```

where

[DO_SOMETHING]	is a keyword/verb describing the action
[TO_THIS]	is the name of a node (or nodes) which are the recipient(s) of the verb
[USING_THIS]	is the set of parameters informing the verb

for example:

```
POS S1 X:100 Y:100
```

is the command to position the node called S1 at coordinates of X:100 and Y:100

```
POS      : VERB
S1       : RECIPIENT
X:100    : PARAMETER
Y:100    : PARAMETER
```

In addition to the recipient being a single node (SOURCE/TARGET/GROUP/ROOM), recipients can be multiple. Consider the following command :

```
COLOUR S1 S2 S3 RED:1.0 GREEN:0.0 BLUE:0.0
```

This translates as “set the RGB colour of nodes named S1 S2 and S3”.

Node names can be combined into a grouped recipient string, with a single space character separating the node names.

As well as using node names to identify recipients, numerical bracketings can be used as follows :

()	Round braces contain SOURCE indeces
[]	Square braces contain GROUP indeces
{}	Curly braces contain TARGET indeces
<>	Angle braces contain ROOM indeces

such that :

```
COlOUR (0 1 2) RED:1.0 GREEN:0.0 BLUE:0.0
```

will set SOURCES 0, 1, and 2 to the specified colour, and :

```
MUTE [4 5 6]
```

will mute GROUPS 4, 5, and 6

Furthermore, multiple recipient groups can be used in a command string :

```
MUTE (0 1 2) [4 5 6]
```

will mute SOURCES 0, 1, and 2 as well as GROUPS 4, 5, and 6

In addition to referencing recipients by their names and numerical indeces, bracketing also enables the use of the wildcard ‘...’ meaning ALL, such that

```
MUTE (...)
```

will mute all SOURCES ...

and, as you might suspect ...

```
MUTE (...) [...]
```

will mute all SOURCES and all GROUPS.

On top of the named, indexed and wildcard references, SVM also understands a collection of named groupings. Using the SOLO command as example, we already know that :

```
SOLO NODE_NAME
```

will solo the named node, and :

```
SOLO (12)
```

will solo the indexed node.

The following multi-select reference names work as follows :

```
MARQUEE         INCLUDES EVERYTHING ‘UNDER’ THE MARQUEE
           SUCH THAT:
                NODE’S LAYER IS VISIBLE
                NODE’S LAYER IS NOT LOCKED
                NODE IS NOT HIDDEN
                NODE IS LOCATED WITH THE MARQUEE

SOURCES         ONLY SOURCES UNDER MARQUEE
GROUPS          ONLY GROUPS UNDER MARQUEE
TARGETS         ONLY TARGETS UNDER MARQUEE

SELECTION       ALL CURRENTLY SELECTED NODES

SYNDICATION     ALL NODES IN CURRENT SYNDICATION SET

(...)           ALL SOURCES REGARLESS HIDE & LOCK STATUS
[...]           ALL SOURCES REGARLESS HIDE & LOCK STATUS
{...}           ALL SOURCES REGARLESS HIDE & LOCK STATUS
<...>           ALL SOURCES REGARLESS HIDE & LOCK STATUS

ALL_SOURCES     SYNONYM FOR (...)    -> IF NOT LOCKED OR HIDDEN
ALL_GROUPS      SYNONYM FOR [...]    -> IF NOT LOCKED OR HIDDEN
ALL_TARGETS     SYNONYM FOR {...}    -> IF NOT LOCKED OR HIDDEN
ALL_ROOMS       SYNONYM FOR <...>    -> IF NOT LOCKED OR HIDDEN
```

The SoundSquares Virtual Machine (SVM) consists of a simulated CPU capable of executing a variety of commands using an assembler-like syntax similar to Z80 6502 6800 and 8080 style of early microcomputer.

It can be called from the CLI, and also from SCRIPTS, and is capable of addressing the patch system using the same commands as patch files. Furthermore, the SVM is capable of remixing these commands on-the-fly and can programmatically control the SoundSquares STAGE, NODES, and DSP settings : anything that can be done with patching, can also be done with scripting.

When scripts are loaded from files, they can be used to extend the built-in functionality of the CLI, such that named entry points can be called directly from the CLI as if they are built-in commands. Calling a script function triggers the script-level thead to run until completion, or until terminated via the CLI.

In addition to directly running a “root” script thread, SVM can operate in a pseudo multi-threaded manner, which whilst not multi-tasking in the true sense, does offer the possibility that multiple scripts can run “at the same time”, isolated from each other in variable scope, and capable of communicating with each other.

The ASM code is sectioned into LABELS which can be arbitrarily called or jumped to, and form the entry-points for script access from the CLI. Scripting LABELS can therefore be used to build higher-level functional constructs, which can in-turn trigger each-other in order to create a logical flow of action controlling items on the stage, dsp-parameter, panel-positions and so-on.

The SoundSquares MACRO-PANEL is a place where cutomised interface elements can be drawn by using specific marco-panel instructions, and when combined with ASM code enables a situation such that additional mini-interfaces for controlling scripts can be constructed.

Unlike a regular CPU, SVM’s vCPU is composed without registers!

Instead, any variable declaration is treated as if it is a register, whether that is a discrete variable, or a pointer to a node’s parameter, So rather than jumping straight-in and being able to instantly manipulate registers, they must first be declared.

# VARIABLE/REGISTER DECLARATION :

```
INT       [NAME]  [INT]           -> create an integer register
FLOAT     [NAME]  [FLOAT]         -> create a float register
STRING    [NAME]  [STRING]        -> create a string register

DEL       [NAME]                  -> delete a register
```

such that:

```
INT MY_INT 123
```

creates a variable/register called “MY_INT” of type integer and value 123.

and :

```
FLOAT MY_FLOAT 14.764
```

creates a variable/register called “MY_FLOAT” of type float and value 14.764

Naturally, then :

```
STRING MY_STRING HERE’S SOME TEXT
```

creates the variable/register “MY_STRING” of type string and ...... right ?

The command DEL will remove the register from the current scope.

To list the current set of registers in the CLI, use :

```
LIST_VARS
```

or, for short :

```
LV
```

A read-out of registers/variables will be printed to the CLI output.

SVM ASM uses two types of labels which operate both as entry-points and as jump-points within a script :

```
     >>LABEL_NAME                   INTENDED FOR ENTRY-POINTS
```

and

```
     ->SUB_LABEL_NAME               INTENDED FOR SCRIPTING JUMP-POINTS
```

However, there is no real semantic difference between the two - they simply serve to offer a little organisational clarity when writing scripts.

To view which script labels are currently in memory and accessible from the CLI, use the command :

```
     LIST_LABELS
```

or, for short :

```
     LL
```

A read-out of entry-points and script labels will be printed to the CLI output.

NOTE : labels starting with “_” are hidden from the print-out.

When a script is loaded using the command LOAD_SCRIPT [FILENAME], if it contains a label “>>MAIN” then this is automatically called as a scripting entry-point to initiate some action.

When a script executes, it runs from the plugin’s GUI thread from a timer. Everytime the timer is called, each script (or thread) is called, and runs until it reaches an end-point, or signals a jump-repeat. In order for the script not to cripple the GUI with a huge list of calls, and thus cause the GUI thread to hang, it is important to know how flow control actively facilitates this.

When a label is called, the script will run from there until it reaches either a return, or end, command, as follows :

```
     CALL      [LABEL_NAME]    -> run script from entry-point

     RTN                       -> return program-counter to
                               -> continue execution from line after
                               -> the CALL command

     END                       -> end current execution
```

BUT if a script is required to keep running over time, it will also need to be able to jump, and that jump will trigger a small delay in the script so that the plugin GUI can continue to operate even though the command-flow continues. To do so, use the JUMP command :

```
     JUMP      [LABEL_NAME]
```

In addition to the JUMP command, another way of preventing the script from crippling the GUI is to use a PAUSE command, to temporarily suspend the script whilst it waits for a certain amount of milliseconds :

```
     PAUSE     [MS]
```

And for the sake of old-time-completeness, there is the no-operation command, which simply triggers a short delay, and is a synonym for pausing for 1 millisecond :

```
     NOP
```

In order to combat the potential for scripts to become very slow, by having to JUMP (pause) all the time, and assuming they have a valid end-point in the form of RTN, END, JUMP, PAUSE or NOP, the GOTO command can be used to trigger the equivalent of a JUMP command, but without causing a delay to the script.

```
     GOTO      [LABEL_NAME]
```

Any script or thread which reaches the recursion depth limit will automatically be terminated for the sake of protecting the main GUI thread.

The vCPU contains a stack 1024 variables deep. Unlike a traditional CPU stack, which runs in bytes, SVM's stack contains full variables, so an INT or a FLOAT or a STRING can be pushed and popped in-full from the stack using the simple commands :

```
PUSH       [A]            -> push a variable onto the stack
POP        [A]            -> pull a variable from the stack
```

*where [A] is the NAMED REGISTER/VARIABLE

To see the current state of the stack, use the following command :

```
PRINT_STACK
```

or, for short :

```
PS
```

A read-out of stack variables will be printed to the CLI output.

In addition to pushing and popping named variables, immediate values can also be used, as follows :

```
PUSHI      [TYPE] [i]     -> push an immediate onto the stack
POPI       [A]            -> pop from stack new variable
```

*where [TYPE] is either INT, FLOAT, or STRING
and [i] is the IMMEDIATE value

Values can also be moved between variables using

```
MOV        [A] [B]        -> move value from B into A
MOVI       [A] [i]        -> move immediate into A
SWAP       [A] [B]        -> swap values of A and B
```

*where [A] and [B] represent any two named variables of the same type
and [i] represents an immediate value.

```
-> FOR MOV :
-> If the type of [B] does not match the type of [A]
-> [A]'s type is reassigned to [B]'s type

-> to 'cast' a value from float to int, it can be
-> resolved to an immediate and used with MOVI
-> see below for PARAMETER RESOLUTION
```

The vCPU is capable of running multiple isolated threads, meaning that multiple scripts can be run “simultaneously”. Excluding the root scripting thread, 64 discrete threads can be run :

```
DISPATCH   [ENTRY-POINT]  [THREAD NAME]
```

Starts a thread runing from the named entry-point, with the optional thread name for management purposes. If no thread name is provided, the thread gets called the same as the entry-point.

```
READY      [ENTRY-POINT]  [THREAD NAME]
```

Configures a thread ready to run from the named entry-point, but instead of running it, puts it into suspended status to be later resumed :

```
SUSPEND    [THREAD NAME]
RESUME     [THREAD NAME]
```

- do exactly what you’d expect, in pausing and unpausing a named thread.

To kill a thread and stop it dead in its tracks, use the following :

```
TERMINATE  [THREAD NAME]
```

To see the current list of threads, use the following command :

```
THREAD_INFO
```

or, for short :

```
TI
```

To communicate between threads, the TELL command is used. It triggers the command payload to be executed in the context of the thread it names.

Note that there are no timing guarantees, other than to say that when the targeted thread is next called, or resumed, the command will have been executed.

```
TELL       [THREAD NAME]  [COMMAND STRING]
```

for example :

```
TELL RUNNER MOVI VAR_1 1.234
```

will set the variable VAR_1 to 1.234 in thread named RUNNER

When using scripting to programmatically construct patches, register values can be resolved as immediates by using the $ prefix. Consider the following script :

```
>>MAIN
# space nodes at interval 100 ON X from -450 onwards
# start positions
FLOAT X -450.0
FLOAT Y 200.0
FLOAT X_INCREMENT 100.0
# recipient node base-name
STRING _S S
# start and end values for loop
INT INDEX 0
INT COUNT $STAGE.NUM_SOURCES
# variable for constructing dynamic node name
STRING NAME

# start a loop to position the nodes
->LOOP_POINT

# build node name concatenating index as string onto name
MOV NAME _S
ADDI NAME $INDEX
# NAME now equals S0, S1, S2, etc

# do the positioning action
POS $NAME X:$_X Y:$_Y

# increment index counter and setup next node position
INC INDEX
ADD X X_INCREMENT

# loop if we've not run out of nodes
CMP_LTE INDEX COUNT GOTO LOOP_POINT

# exit at end of script
END
```

Using $STAGE.NUM_SOURCES dereferences the number of sources into register COUNT. $INDEX is used as an immediate value to build the name of each node targeted by the action, and in the action, $X and $Y are used to get the register contents (position variables in this case) into the command.

Note : when resolving register values into ASM commands, they are treated as immediate values, hence the following comparison operations equivolate :

```
CMP_LTE INDEX COUNT GOTO LOOP_POINT
CMP_LTEI INDEX $COUNT GOTO LOOP_POINT
```

They both perform the comparison (Less Than Equal) - the first by looking at the variable itself, and the second by treating it as an immediate since it is resolved by the lexing/parsing stage prior to reaching the execution engine.

Whilst superficially, scripts and threading might look identical, there are a few important differences, primarily around speed of execution.

The scripting engine gets called by a timer in the gui thread. Every time the timer is called, if there are commands waiting, they are then executed.

-> A script executes at the rate of one command per timer call.
-> A thread can execute upto 1000 commands per timer call.

In addition to using the CALL command to trigger a script or thread to move its program pointer to a new label, the command parsing engine will automatically attempt to a command to a label if the command is not found as part of the built-in lexicon.

consider the following :

```
->MY_LABEL                  -> LABEL NAME
INT A 10                    -> SET INTEGER VARIABLE A AS 10
INT B 20                    -> SET INTEGER VARIABLE B AS 10
ADD A B                     -> ADD B TO A
MOVE S1 X:$A                -> MOVE SOURCE S1 BY A IN X
RTN                         -> RETURN FROM FUNCTION
```

Cannonically this function label would be run by using : CALL MY_LABEL
BUT can also be called direct, using : MY_LABEL

Further extending this usage, such ‘bypass-calls’ can also make use of auto-pushed immediates, and can therefore behave more like a regular scripting language. Consider the following:

```
->MY_LABEL                  -> LABEL NAME
POPI B                      -> POP VALUE FOR B
POPI A                      -> POP VALUE FOR A
ADD A B                     -> ADD B TO A
MOVE S1 X:$A                -> MOVE SOURCE S1 BY A IN X
RTN                         -> RETURN FROM FUNCTION
```

This function emands that values for A and B are on the stack, otherwise they will be populated as ZERO-VALUE INTS as a result of their POPI commands. To run requires :

```
PUSHI INT 20                -> PUSH VALUE FOR A
PUSHI INT 10                -> PUSH VALUE FOR B
CALL MY_LABEL               -> CALL THE FUNCTION
```

Using a bypass-call, the same can be achieved by issuing :

```
MY_LABEL 20 10
```

In the above example, the bypass-call automatically PUSHIs the INT values 20 and 10 onto the stack before the call, and assumes the function will correctly POP, so as to avoid stack overflow.

Variable types are resolved to INT, FLOAT, AND STRING on the basis of :

All characters are DIGITS	-> INT
First character is DIGIT and there's also a "."	-> FLOAT
otherwise	-> STRING

In addition to the lexer/parser resolving variables from the scope of script-call it also resolves values associated with nodes and stage items.

See the ASM section for more details on how this all comes-together ... but for now, know that, for example :

```
FLOAT F $SOURCE_1.X
results in a float variable being created
using the immediate value of SOURCE_1.X
```

Node parameters that can be accessed this way are lised below, and further interface elements accessible this way are listed under the section on POINTERS and ALIASES

		SOURCE	TARGET	GROUP	ROOM
X		X	X	X	X
Y		X	X	X	X
LEVEL		X	X	X	X
SIZE		X			
DB		X	X	X	X
MUTE		X	X	X	X
SOLO		X	X	X	X
VISIBLE		X	X	X	
PHASE		X	X	X	X
RED		X	X	X	X
GREEN		X	X	X	X
BLUE		X	X	X	X
ROTATES				X	
ROTATION				X	
MOVES				X	
DX				X	
DY				X	
RESPONSE			X		
W					X
H					X

Casting variables from STRING to FLOAT or INT and back again, is a simple matter of overwriting any existing variable as if defining the variable for the first time, and resolving the required variable name into it:

```
FLOAT F 4.56    -> make a FLOAT with value 4.56
STRING S $F     -> S = “4.56”
INT I $S        -> I = 5
```

The parser/lexer is also capable of doubly-resolving strings contaning parameter names, by using $$ as follows :

```
      STRING RM <0>
      # string contains name of first room

      # resolve the .X and .Y of the room into registers X and Y
      FLOAT X $$RM.X
      FLOAT Y $$RM.Y
```

Let's assume the first room in the patch is called "ROOM_1"

At the command FLOAT X $$RM.X :

The first $ of the string $$RM.X resolves into :

```
      $ROOM_1.X
```

Which in turn is resolved into an immediate value containing the X coordinate of ROOM_1

To the execution engine the commands are transformed into :

```
      FLOAT X -400.00
      FLOAT Y -350.00
```

This resolution syntax can also be used to combine strings with ints and floats to create a pseudo-pointer-style approach to programatic scripting, which can also be used in flow control.

Names of commands can also be parsed on-the-fly from strings, such as :

```
      INT A 10
      INT B 5
      INT C 0
      STRING ACTION_1 ADD
      STRING ACTION_2 SUB
      MOV C A
      $ACTION_1 C B                     -> ADD C B
      $ACTION_2 A B                     -> SUB A B
      END
```

The result of this is :

```
      A = 5
      B = 5
      C = 15
```

Whilst the SVM resolves and dereferences both scripting variables and node parameters into function calls, it can sometimes be useful to assign pointers as script variables, such that an ASM command has an immediate impact on the node parameter without the need to call a full patching command. This is achieved by aliasing using :

```
ALIAS	[NAME]	[NODE.PARAMETER]
```

Which creates (or overwrites) a register/variable pointing directly to the parameter, such that the following script snippets perform the same function :

```
# SVM redirection :
FLOAT _X 100
FLOAT _Y 200
POS (0) X:$_X Y:_Y

# POINTER indirection :
ALIAS _X (0).X
ALIAS _Y (0).Y
MOVI _X 100
MOVI _Y 200
```

Both scripts position source node 0 at X/Y coordinates 100/200.

The first example relies on the parsing engine to dereference script variables when formulating a POS command, and the second example directly controls the X and Y variables of the node itself.

As with non-pointer registers/variables, aliased registers/variables can access all other ASM functions. Internally the SVM uses pointers inside registers/variables such that a non-pointer register/variable uses a pointer to its own inner parameter.

Aliased registers/variables can therefore also be pushed/popped from/to the stack ... moved, swapped, compared ... and so on.

Currently, only INT and FLOAT pointed registers are available, and their type and indirection is automatically handled by the parsing/lexing engine.

The following tables shows which node parameters can be used in pointers :

INT parameters :

		SOURCE	TARGET	GROUP	ROOM
MUTE		X	X	X	X
SOLO		X	X	X	X
VISIBLE		X	X	X	
ROTATES				X	
MOVES				X	
VIRTUAL		X	X		

FLOAT parameters :

		SOURCE	TARGET	GROUP	ROOM
X		X	X	X	X
Y		X	X	X	X
W					X
H					X
LEVEL		X	X	X	X
PHASE		X	X	X	X
SIZE		X			
RESPONSE			X		
ROTATION				X	
DX				X	
DY				X	
RED		X	X	X	X
GREEN		X	X	X	X
BLUE		X	X	X	X

-> NOTE : THIS WILL GROW AS SOUNDSQUARES DEVELOPS MOVING-FORWARDS ...

STAGE.

[INT]
STAGE.NUM_SOURCES
STAGE.NUM_GROUPS
STAGE.NUM_TARGETS
STAGE.NUM_ROOMS
STAGE.ROTATION
STAGE.MOVEMENT
STAGE.SHOW_ROOMS
STAGE.SHOW_TARGETS
STAGE.SHOW_SOURCES
STAGE.SHOW_GROUPS
STAGE.SHOW_CABLES
STAGE.SHOW_XHAIRS
STAGE.SHOW_ANNOTATIONS
STAGE.LOCK_STAGE
STAGE.LOCK_ROOMS
STAGE.LOCK_TARGETS
STAGE.LOCK_GROUPS
STAGE.LOCK_SOURCES
STAGE.LOCK_ANNOTATIONS
STAGE.SOURCE_CLIPPING
STAGE.TARGET_CLIPPING

[FLOAT]
STAGE.MOUSE_X
STAGE.MOUSE_Y
STAGE.OFFSET_X
STAGE.OFFSET_Y

GLOBAL.

[INT]
GLOBAL.MOUSE_X
GLOBAL.MOUSE_Y
GLOBAL.GUI_W
GLOBAL.GUI_H
GLOBAL.DSP_MODE
GLOBAL.OVERSAMPLE
GLOBAL.OVERSAMPLING_FILTER
GLOBAL.SYNDICATION_MODE
GLOBAL.XFTIME
//GLOBAL.VERSION_STRING

[FLOAT]
GLOBAL.RESPONSE
GLOBAL.MASTER_LEVEL

PANEL.

[INT]

[FLOAT]

SECTION INCOMPLETE

TO DO >>>

[INT]	SOURCE.DELAY.ENABLED
[FLOAT]	SOURCE.DELAY.TIME
[INT]	SOURCE.FILTERS.ENABLED
[FLOAT]	SOURCE.FILTER[0].FREQ
[FLOAT]	SOURCE.FILTER[0].GAIN
[FLOAT]	SOURCE.FILTER[0].BW
[INT]	SOURCE.FILTER[0].ENABLED
[INT]	SOURCE.DYNAMICS.ENABLED
[INT]	SOURCE.DYNAMICS.MODE
[FLOAT]	SOURCE.DYNAMICS.ATTACK
[FLOAT]	SOURCE.DYNAMICS.RELEASE
[FLOAT]	SOURCE.DYNAMICS.THRESHOLD
[FLOAT]	SOURCE.DYNAMICS.RATIO
[FLOAT]	SOURCE.DYNAMICS.MAKEUP
[FLOAT]	SOURCE.DYNAMICS.KNEE
[INT]	SOURCE.DYNAMICS.SIDECHAIN
[INT]	SOURCE.DYNAMICS.SIDECHAIN_LOCATION

MARQUEE/SELECTION/SYNDICATION

[INT]

MARQUEE.NUM_ITEMS	COUNT
MARQUEE.ITEMS[n].INDEX	NODE INDEX
MARQUEE.ITEMS[n].TYPE	TYPE OF
SELECTION.NUM_ITEMS	COUNT
SELECTION.ITEM[n].INDEX	NODE INDEX
SELECTION.ITEM[n].TYPE	TYPE OF
SYNDICATION.NUM_ITEMS	COUNT
SYNDICATION.ITEM[n].INDEX	NODE INDEX
SYNDICATION.ITEM[n].TYPE	TYPE OF

[STRING]

MARQUEE.ITEMS[n].NAME	NAME OF
SELECTION.ITEM[n].NAME	NAME OF
SYNDICATION.ITEM[n].NAME	NAME OF

SERIAL.

[INT]	SERIAL.IS_CONNECTED
[INT]	SERIAL.PORT_ID
[STRING]	SERIAL.PORT_NAME
[STRING]	SERIAL.PORTS[n].NAME

One-shot commands consisting of just a single verb :

```
    BIG_CLI
         TOGGLES LARGER CLI_TEXT AT THE BOTTOM OF THE SCREEN

    CLS
         CLEARS THE CLI TEXT DISPLAY

    CLEANUP
         SANITISE NAMING TO PREVENT CLI MISFIRES

    COPY_DYNAMICS
         COPY CURRENT DYNAMICS FROM PANEL TO CLIPBOARD

    COPY_FILTER
         COPY CURRENT FILTER FROM PANEL TO CLIPBOARD

    DRAW_IN_MACRO
         MOVE THE DRAWING FOCUS TO THE MACRO PANEL

    DRAW_WITH_STAGE
         MOVE THE DRAWING FOCUS TO (ON-TOP-OF) THE STAGE

    FLAT_ZOOM_IN
         STRAIGHT ZOOM, TAKING NO ACCOUNT OF STAGE DISPLACEMENT

    FLAT_ZOOM_OUT
         STRAIGHT ZOOM, TAKING NO ACCOUNT OF STAGE DISPLACEMENT

    FULLSCREEN
         ENTER/LEAVE FULLSCREEN MODE ON SCREEN WHERE MOUSE IS

    HARD_RESET
         ANIHILATE THE PATCH IN A GLOBAL RESET

    LIST_CABLES
         PRINT LIST OF CABLES

    LIST_LABEL      (ALIAS : LL)
         PRINT CURRENTLY LOADED SCRIPT LABELS

    LITS_VARS       (ALIAS : LV)
         PRINT SCRIPT VARIABLES FOR CURRENT CONTEXT

    LOAD_MACRO_FILE
         LOAD PATCH USING FILE OPEN DIALOG

    NEXTPAGE
         MOVE TO THE NEXT GUI PAGE
```

NO_PANELS
    HIDE ALL PANELS EXCEPT TOOLBOX AND MASTER METERS

PRINT_SELECTED
    PRINTS LIST OF CURRENT SELECTED NODE

PRINT_STACK   (ALIAS : PS)
    PRINTS CURRENT STACK STATUS

PROTECT_THREAD
    PREVENT THIS THREAD FROM BEING NUKED BY TERMINATE COMMAND

RESET_ALL_NAMES
    SOURCES = S1, S2, S3 : GROUPS = G, TARGETS = T, ROOMS = R

PASTE_DYNAMICS
    PASTE CLIPBOARD DYNAMICS TO THE SELECTED NODE’S DYNAMICS

PASTE_FILTER
    PASTE CLIPBOARD FILTER TO THE SELECTED NODE’S FILTERS

PREVIOUSPAGE
    MOVE TO THE PREVIOUS GUI PAGE

RESET_DE
    RESET ALL DELAY SETTINGS

RESET_DY
    RESET ALL DYNAMICS SETTINGS

RESET_F
    RESET ALL FILTER SETTINGS

RESET_FF
    RESET ALL FEEDBACK SETTINGS

RESET_PANELS
    PUT ALL PANELS INTO DEFAULT STATE

RESET_SELECT
    REMOVE ALL ITEMS FROM ALL SELECTIONS

RESET_TOOLBOX
    PUT TOOLBOX INTO DEFAULT STATE, DEFEATING ALL INDICATORS
    -> ALSO DISMISSES MASTER ROTATION AND MOVEMENT

RESET_V
    RESET ALL VIRTUALISATION SETTINGS

```
SAMPLE_RATE     (ALIAS : SR)
     PRINT THE CURRENT SAMPLE RATE

SAVE_NOW
     SAVE SETTINGS TO CURRENTLY LOADED PATCH FILE

SAVE_PATCH
     SAVE SETTINGS TO PATCH FILE USING DAVE FILE DIALOG


STOP_MACRO
     PREVENTS FURTHER PROCESSING OF PATCH OR MACRO FILE

STEALTH
     HIDE ALL PANELS AND GUI DECORATIONS

SYSTEM_INFO     (ALIAS : SI)
     PRINTS THE CPU-SPEC AND WINDOWS LAYOUT

TERMINATE
     END ALL CURRENTLY RUNNING (UNPROTECTED) THREADS

THREAD_INFO     (ALIAS : TI)
     PRINTS CURRENT THREADING CONFIGURATION

UNDO
     UNDO LAST COMMAND (ONLY POSITIONS, LEVELS, COLOURS)

UNPROTECT_THREAD
     REMOVE PROTECTION FROM THIS THREAD

WRAP
     ENCAPSULATE CURRENT SELECTION IN MARQUEE

ZOOM_IN
     ZOOM IN A STEP (*1.1 MAGNIFICATION)

ZOOM_OUT
     ZOOM OUT A STEP (/1.1 MAGNIFICATION)
```

The following common named parameters are used for a variety of commands, in the form of NAME:VALUE pairs :

NAME	COMMON USAGE
X:	NODE X POSITION
Y:	NODE Y POSITION
W:	ROOM WIDTH
H:	ROOM HEIGHT
I:	INDEX
RED:	NODE COLOUR
GREEN:	NODE COLOUR
BLUE:	NODE COLOUR
R:	RESERVED - CURRENTLY UNUSED
S:	NODE COLOUR USING H:S:L
L:	COLOUR USING H:S:L, ANNOTATION ALPHA
TIME:	DELAY TIME
BAND:	FILTER BAND INDEX
TYPE:	FILTERS AND DYNAMICS PROCESSORS TYPE
G:	FILTER GAIN SETTING
F:	FILTER FREQUENCY SETTING
BW:	FILTER BANDWIDTH SETTING
ATTACK:	DYNAMICS ATTACK TIME
RELEASE:	DYNAMICS RELEASE TIME
THRESHOLD:	COMPRESSOR/GATE/LIMITER THRESHOLD dB
RATIO:	COMPRESSOR RATIO
KNEE:	COMPRESSEOR/GATE/LIMITER KNWW
LIMIT:	LIMITER LIMIT
SIDECHAIN:	DYNAMICS SIDECHAIN CHANNEL
X1:	USED FOR SPECIFYING DYNAMICS CURVE SHAPE
Y1:	
X2:	
Y2:	
X3:	
Y3:	
X4:	
Y4:	
X5:	
Y5:	

NOTE : This is just a guide - there are also other uses of these, and commands which use additional NAME:VALUE parameter schemes - but you’ll encounter these perhaps more ...

The following commands are used to configure the stage, and position and colour nodes :

```
SET        : CONTROL THE NUMBERS OF NODES

     SYNTAX     : SET [ID] [COUNT]
     [ID]  :
          : NUM_SOURCES
          : NUM_TARGETS
          : NUM_GROUPS
          : NUM_ROOMS

     EXAMPLE    : SET NUM_SOURCES 14



RESET      : RESET A NODE TO ‘FACTORY’ SETTINGS

     SYNTAX     : RESET [NODES]
     EXAMPLE    : RESET S1 S2 S3
                : RESET [...]



NAME       : (RE)NAME NODE(S)

     SYNTAX     : NAME [NODES] [NEW_NAME]
     EXAMPLE    : NAME T1 MY_TARGET
                : NAME <0> ROOM_ZERO

     -> FOR MULTIPLE NODES, NAMES ARE INDEXED AND
     -> APPENDED WITH “_i” WHERE i = INDEX
     -> NODE_1 NODE_2 NODE_3 ETC
     -> IF NEW NAME ENDS IN “_” THEN INDEXING STARTS AT 1st NODE
     -> ELSE STARTS AT 2nd NODE
     -> NODE NAMES CANNOT BEGIN WITH NUMBERS
     -> MULTIPLE NODES CANNOT SHARE THE SAME NAME



POS        : ABSOLUTE POSITIONING OF NODES ON THE STAGE

     SYNTAX     : POS [NODES] [VARS]
     EXAMPLE    : POS S1 X:20 Y:-10
                : POS GROUP1 Y:100

     -> THE ORIGIN POINT 0,0 IS IN THE CENTRE OF THE STAGE
     -> WHERE THE GRID X-HAIRS ARE BRIGHTEST AND THIS APPLIES
     -> REGARDLESS OF STAGE X/Y DISPLACEMENT
```

```
MOVE       : RELATIVE MOVE NODES ON THE STAGE

     SYNTAX     : MOVE [NODES] [VARS]
     EXAMPLE    : MOVE S1 X:20 Y:-10
                : MOVE GROUP1 Y:100



SWAP_POS   : SWAP POSITION OF FIRST 2 NAMED NODES

     SYNTAX     : SWAP_POS [A] [B]
     EXAMPLE    : SWAP_POS S1 S2



TRANSPORT  : (ABSOLUTE) MOVE ROOM INCLUDING CONTENTS

     SYNTAX     : TRANSPORT [ROOM] [VARS]
     EXAMPLE    : TRANSPORT ROOM_1 X:-200 Y:-600



ALIGN      : LINES-UP OBJECTS

     SYNTAX     : ALIGN [NODES] [VARS]
     [VARS]     : TOP / MIDDLE / BOTTOM / LEFT / CENTRE / RIGHT

     EXAMPLE    : ALIGN SYNDICATION TOP
                : ALIGN S1 T1 GROUPS CENTRE



SPACING    : DISTRIBUTE NODES EVENLY

     SYNTAX     : SPACING [NODES] [VAR]
     VAR        : HORIZONTAL / VERTICAL



CIRCLE     : ARRANGE NODES IN A CIRCLE USING EXTREMITIES OF
             NODE X/Y POSITIONS AS DELIMITER OF PERIMETER

     SYNTAX     : CIRCLE [NODES]
     EXAMPLE    : CIRCLE (...)

     -> POSITION A NODE AT EACH CORNER OF THE BOUNDING-BOX
     -> OF THE CIRCLE, WITH ALL OTHER NODES INSIDE THE BOUNDING
```

```
OVAL       : ARRANGE NODES IN AN OVAL USING EXTREMITIES OF
                  NODE X/Y POSITIONS AS DELIMITER OF PERIMETER

     SYNTAX     : OVAL [NODES]
     EXAMPLE    : OVAL {...}

     -> POSITION A NODE AT EACH CORNER OF THE BOUNDING-BOX
     -> OF THE OVAL, WITH ALL OTHER NODES INSIDE THE BOUNDING



SNAP       : SNAP NODES TO NEAREST GRID POINT

     SYNTAX     : SNAP [NODES]
     EXAMPLE    : SNAP GROUP_1



EXPAND     : GROW THE POSITIONS OF A SET OF NODE
             AWAY FROM THEIR SHARED CENTRE

     SYNTAX     : EXPAND [NODES]
     EXAMPLE    : EXPAND S1 S2 S3 S4



CONTRACT   : SHRINK THE POSITION OF A SET OF NODES
             TOWARDS THEIR SHARED CENTRE

     SYNTAX     : CONTRACT [NODES]
     EXAMPLE    : CONTRACT {0 1 2 3}



HIDE       : HIDE NODES
UNHIDE     : UNHIDE NODES

     SYNTAX     : HIDE [NODES]
     EXAMPLE    : HIDE SYNDICATION
                : HIDE SOURCE_1


PRESET_COLOUR   : USE ONE OF 20 PREDEFINE COLOURS

     SYNTAX     : PRESET_COLOUR [NODES] 1-20
     EXAMPLE    : PRESET_COLOUR ROOM_1 12

     -> SOURCES TARGETS AND GROUPS SHARE ONE SET OF COLOUR PRESETS
     -> ROOMS HAVE A SEPARATE, DARKER SET
```

```
COLOUR     : SELECT A COLOUR FOR NODES USING WINDOWS COLOUR-PICKER

     SYNTAX     : COLOUR [NODES]
     EXAMPLE    : COLOUR SUBWOOFERS

     -> ALL SELECTED NODES GET THE SAME COLOUR
     -> CRASH ALERT -> THIS SHOULD NOT BE USED IN A PATCH
     -> BUT INSTEAD CALLED FROM A MACRO OR A SCRIPT ...
     -> OR SCRIPTED BUTTON ...
     -> OR A TYPED CLI COMMAND


RGB        : SET SPECIFIED COLOUR VALUES TO NODE

     SYNTAX     : RGB [NODES] RED:VALUE GREEN:VALUE BLUE:VALUE
     EXAMPLE    : RGB SYNDICATION RED:0.8 GREEN:0.5 BLUE:0.4


SPECTRUM   : SPREAD A SPECTRUM OF COLOUR ACROSS NODES

     SYNTAX     : SPECTRUM [NODES] (OPT)
     (OPT)      : S:0.0-1.0 -> COLOUR SATURATION
     EXAMPLE    : SPECTRUM [...] 0.7


REDRAW     : TURN ON/OFF DRAWING OUTPUT RESPONSES

     SYNTAX     : REDRAW (OPT)
     (OPT)      : ON / OFF
     EXAMPLE    : REDRAW ON
                : REDRAW OFF


ROOM_TO_MARQUEE: PLACE A ROOM AT THE MARQUEE

     SYNTAX     : ROOM_TO_MARQUEE [OPT]
     [OPT]      : INDEX (0 to 7) OR NAME OF ROOM
     EXAMPLE    : ROOM_TO_MARQUEE 0
                : TOOM_TO_MARQUEE MY_MIX_ROOM
```

```
LEVEL      : SET THE ABSOLUTE dB LEVEL OF A NODE

     SYNTAX     : LEVEL [NODES] [dB]
     EXAMPLE    : LEVEL S1 -6
                : LEVEL [...] 3



TRIM       : CHANGE THE RELATIVE dB LEVEL OF A NODE

     SYNTAX     : TRIM [NODES] [dB]
     EXAMPLE    : TRIM S1 -7
                : TRIM (...) 2



MUTE       : MUTES OBJECTS
UNMUTE     : UNMUTES OBJECTS

     SYNTAX     : MUTE [NODES]
     EXAMPLE    : MUTE SOURCE_1
                : UNMUTE MARQUEE



SOLO       : SOLOS OBJECTS
UNSOLO     : UNSOLOS OBJECTS USING THE SAME SYNTAX

     SYNTAX     : SOLO [NODES]
     EXAMPLE    : SOLO ROOM_1
                : UNSOLO SYNDICATION



PHASE      : CHANGE THE PHASE OF A NODE

     SYNTAX     : PHASE [NODES] *[-1 .. +1)
     EXAMPLE    : PHASE SOURCE_1 -1
                : PHASE GROUP_1

     -> PHASE INDICATION IS OPTIONAL
     -> IF NOTHING SPECIFIED, PHASE SET TO NEUTRAL/+1/POSITIVE



VIRTUALISE : CHOOSE WHICH VIRTUAL I/O EQUATES TO WHICH SOURCE/TARGET

     SYNTAX     : VIRTUALSISE [SOURCE/TARGET NODE] [VIRTUAL INDEX]
     EXAMPLE    : VIRTUALS TARGET_1 17
```

```
USE_GLOBAL : WHETHER OR NOT TARGET[S] USES LOCAL OR GLOBAL RESPONSE

     SYNTAX     : USE_GLOBAL [TARGETS] (VAR)
     (VAR)      : YES/TRUE/GLOBAL/1 | NO/FALSE/LOCAL/0
     ALIAS      : GLOBALZ
     EXAMPLE    : USE_GLOBAL {9 10} FALSE
                : GLOBALZ {0 1 2 3 4 5 6 7 8} TRUE



SET_ZERO   : SET THE ZEROPOINT RESPONSE DISTANCE FOR TARGET[S]

     SYNTAX     : SET_ZERO (TARGETS) (DISTANCE)
     (TARGETS)  : IF NO TARGET SPECIFIED, GLOBAL RESPONSE IS SET
     EXAMPLE    : SET_ZERO 250                  (GLOBAL)
     EXAMPLE    : SET_ZERO SUBWOOFER 600        (LOCAL)



SET_SIZE   : SET THE SIZE FOR SOURCE

     SYNTAX     : SET_SIZE [SOURCES] (SIZE)
     EXAMPLE    : SET_SIZE SOURCE_7 2.3

     -> THE SIZE PARAMETER FOR SOURCES ACTS AS A MULTIPLIER
     -> OF RESPONSE DISTANCE OF ALL TARGET NODES IT MIXES TO



SET_MASTER : SET MASTER VOLUME ABSOLUTE dB (RANGE -200 to +12)
SET_MASTERV : SET MASTER VOLUME ABSOLUTE VALUE (RANGE 0 to 4)
TRIM_MASTER : TRIM MASTER VOLUME BY dB (RANGE +/-60)

     SYNTAX     : SET_MASTER -6.0
     EXAMPLE    : TRIM_MASTER 3.6
```

```
CABLE       : ESTABLISH A CABLE CONNECTING [SOURCES] TO [TARGETS]

     SYNTAX     : CABLE [SOURCES] [TARGETS] [X:dB]
     EXAMPLE    : CABLE S1 T1 -3
                : CABLE (0 1 2 3) {4 5 6 7} -12

     -> IF DB NOT SPECIFIED LEVEL DETERMINED BY SOURCE->TARGET DISTANCE
     -> WHICH MAINTAINS THE CURRENT MIX VIA LOCATION
     -> IF THIS IS NOT APPLICABLE, DEFAULT LEVEL IS -6DB


UNCABLE     : REMOVE ALL CABLES FROM [NODES]

     SYNTAX     : UNCABLE [NODES]
     EXAMPLE    : UNCABLE SOURCE_1
                : UNCABLE (...)


CABLE_LEVEL : SET ABSOLUTE DB LEVEL OF CABLE

     SYNTAX     : CABLE_LEVEL [SOURCE] [TARGET] [dB]
     EXAMPLE    : CABLE_LEVEL SOURCE_1 TAGRET_1 -22


REMOVE_CABLE : REMOVE SPECIFIED CABLE BETWEEN SOURCE AND TARGET

     SYNTAX     : REMOVE_CABLE [SOURCES] [TARGETS]


REMOVE_CABLES : REMOVE CABLES FROM NAMED NODES

     SYNTAX     : REMOVE_CABLES [SOURCES] [TARGETS]
     EXAMPLE    : REMOVE_CABLES (...) SUB_1 SUB_2


EXCLUDE_CABLE  : SET SPECIFIC CABLE TO EXCLUSION MODE

     SYNTAX     : EXCLUDE_CABLE [SOURCE] [TARGET] (OPTION)
     (OPTION)   : [yes/no | true/false | 1/0]

     -> IF OPTION NOT SPECIFIED, CABLE IS SET TO EXCLUSION MODE


EXCLUDE_CABLES  : EXCLUDE ALL CABLES SOURCES->TARGETS

     -> SAME SYNTAX AS EXCLUDE_CABLE


LIST_CABLES     : PRINT LIST OF CABLES
```

The following commands all trigger toggles in the mix and follow a similar syntax :

```
TOGGLE_MUTE     : APPLIES TO SOURCES TARGETS GROUPS AND ROOMS

     OPTION     : MUTE/YES/ON/1/TRUE | UNMUTE/NO/OFF/0/FALSE]

TOGGLE_SOLO     : APPLIES TO SOURCES TARGETS GROUPS AND ROOMS

     OPTION     : SOLO/YES/ON/1/TRUE | UNSOLO/NO/OFF/0/FALSE]

TOGGLE_PHASE    : APPLIES TO SOURCES TARGETS GROUPS AND ROOMS

     OPTION     : NORMAL/1/FORWARDS | ANTIPHASE/-1/REVERSE]


     EXAMPLE    : TOGGLE_MUTE (0 1 2) TRUE
                : TOGGLE_SOLO GROUP_1 FALSE
                : TOGGLE_PHASE ROOM_1 ANTIPHASE

TOGGLE_HIDE     : APPLIES TO SOURCES TARGETS AND GROUPS NODES
TOGGLE_HIDE_ALL : APPLIES TO EVERYTHING UNDER MARQUEE
                : SYNONYM FOR TOGGLE_HIDE MARQUEE

     EXAMPLE    : TOGGLE_HIDE {...}
                : TOGGLE_HIDE_ALL

TOGGLE_ROTATION : APPLIES TO GROUPS
TOGGLE_MOVEMENT : APPLIES TO GROUPS

     EXAMPLE    : TOGGLE_ROTATION GROUP_1
                : TOGGLE_MOVEMENT GROUP_2

TOGGLE_CABLE_MUTE         [SOURCE] [TARGET] (YES/NO | TRUE/FALSE | 1/0)
TOGGLE_CABLE_SOLO         [SOURCE] [TARGET] (YES/NO | TRUE/FALSE | 1/0)
TOGGLE_CABLE_PHASE        [SOURCE] [TARGET] (-1/1)

TOGGLE_CABLE_USE_GROUP    [SOURCE] [TARGET] [YES/NO | TRUE/FALSE | 1/0]

     -> WHEN A CABLE IS SET TO NOT USE GROUP
     -> GROUP VCA SETTINGS ON THE SOURCE
     -> DO NOT IMPACT THIS CABLE
```

```
GROUPIFY   : ADD SOURCES TO A GROUP NODE

     SYNTAX     : GROUPIFY [SOURCES] [GROUP]
     EXAMPLE    : GROUPIFY (...) GROUP_1

     -> ONLY A SINGLE GROUP NAME IS ALLOWED ...
     -> ALTHOUGH MULTIPLE SOURCES CAN BE USED


UNGROUP    : REMOVE NAMED SOURCES FROM GROUPS, OR EMPTY NAMED GROUPS

     SYNTAX     : UNGROUP [NODE]
     EXAMPLE    : UNGROUP GROUP_1


UNGROUPIFY : UNGROUP SPECIFIED GROUPINGS

     SYNTAX     : UNGROUP [NODES]
     EXAMPLE    : UNGROUP S2 S3 S4 GROUP_1

     -> AS WITH GROUPIFY, MULTIPLE SOURCES CAN BE USED
     -> BUT ONLY A SINGLE GROUP


SET_ROTATION    : EXPLICIT SETTING OF GROUP ROTATION

     SYNTAX     : SET_ROTATION [GROUP] [S:SPEED] (OPTION)
     (OPTION)   : ON | OFF | TRUE | FALSE | YES | NO
     EXAMPLE    : SET_ROTATION GROUP_1 S:0.1
                : SET_ROTATION GROUP_1 S:0.1 ON
                : SET_MOVEMENT GROUP_2 FALSE


SET_MOVEMENT    : EXPLICIT SETTING OF GROUP MOVEMENT

     SYNTAX     : SET_ROTATION [GROUP] [X:SPEED Y:SPEED] (OPTION)
     (OPTION)   : ON | OFF | TRUE | FALSE | YES | NO
     EXAMPLE    : SET_MOVEMENT GROUP_1 X:1.1 Y:-1.5 ON
                : SET_MOVEMENT GROUP_1 OFF
```

```
TOGGLE     : TOGGLE VARIOUS ASPECTS OF THE TOOLBOX AND INTERFACE

      SYNTAX     : TOGGLE [OPTION]
      [OPTION]   : ROOM_VISIBILITY | TARGET_VISIBILITY | GROUP_VISIBILITY |
                   SOURCE_VISIBILITY | CABLE_VISIBILITY | X_HAIR_VISIBILITY
                   GRID_VISIBILITY | ANNOTATION_VISIBILITY
                   ROOM_LABELS | TARGET_LABELS | GROUP_LABELS
                   SOURCE_LABELS | STAGE_LOCKS | ROOM_LOCKS
                   TARGET_LOCKS | GROUP_LOCKS | SOURCE_LOCKS
                   ANNOTATION_LOCKS | GUI_SNAP | VISUALISATIONS
                   ROTATION | MOVEMENT | TOOLBOX | MASTER | DEBUG
                   HEATMAP | FULLSCREEN | STEALTH
                   AUTO_SAVE | AUTO_BACKUP
      EXAMPLE    : TOGGLE ANNOTATION_VISIBILITY

      -> ROTATION & MOVEMENT APPLIES TO GLOBAL-LEVEL SWITCHES
      -> AND NOT INDIVIDUAL GROUPS - SEE SET_ROTATION AND SET_MOVEMENT


DISMISS    : DISMISS SWITCHES (MATCHING THOSE IN THE TOOLBOX)

      SYNTAX     : DISMISS [OPTION]
      [OPTION]   : SOURCE_MUTES | SOURCE_SOLOS
                   TARGET_MUTES | TARGET_SOLOS
                   GROUP_MUTES | GROUP_SOLOS
                   ROOM_MUTES | ROOM_SOLOS
                   CABLE_MUTES | CABLE_SOLOS
                   SOURCE_CLIPPING | TARGET_CLIPPING
                   FULLSCREEN
      EXAMPLE    : DISMISS SOURCE_CLIPPING


ENTER      : USE TEXT ENTRY TO EDIT SETTINGS

      SYNTAX     : ENTER [OPTION]
      [OPTION]   : SOURCE_COUNT | TARGET_COUNT | GROUP_COUNT | ROOM_COUNT
      EXAMPLE    : ENTER SOURCE_COUNT

      -> CRASH ALERT -> THIS SHOULD NOT BE USED IN A PATCH
      -> BUT INSTEAD CALLED FROM A MACRO OR A SCRIPT ...
      -> OR SCRIPTED BUTTON ...
      -> OR A TYPED CLI COMMAND


RESET_TOOLBOX   : RESET THE TOOLBOX TO ITS ‘FACTORY’ STATE
```

The following commands deal with node selection and syndication. They determine what happens when the parsing engine auto-dereferences thr keywords “SELECTION” and “SYNDICATION”.

```
ADDSELECT  : ADD LISTED NODES TO SELECTION

     SYNTAX     : ADDSELECT [NODES]
     EXAMPLE    : ADDSELECT (...)


UNSELECT   : UNSELECT NAMED NODES AND MODIFY MARQUEE TO SELECTED NODES

     SYNTAX     : UNSELECT [NODES]
     EXAMPLE    : UNSELECT SOURCE_1


TOGGLE_SELECT : TOGGLE LISTED NODES IN-AND-OUT OF SELECTION

     SYNTAX     : TOGGLE_SELECT [NODES]
     EXAMPLE    : TOGGLE_SELECT {...}


SYN_MODE   : SET SYNDICATION MODE

     SYNTAX     : SYN_MODE [MODE]
     [MODE]     :
          NONE       : NO SYNDICATION IS APPLIED TO MOUSE ACTIONS
          MARQ       : ONLY NODES UNDER MARQUEE ARE SYNDICATED
          SELECT     : ONLY NODES IN CURRENT SELECTION ARE SYNDICATED
          BOTH       : BOTH MARQUEE AND SELECTION ARE IN SYNDICATION


STORING SELECTIONS - SETTING-UP SYNDICATION GROUPINGS

     STORE_SELECT [SLOT]        : SAVE SELECTION TO SLOT
     RETRIEVE_SELECT [SLOT]     : RETRIEVE SELECTION FROM SLOT
     SWITCH_SELECT [SLOT]       : STORE CURRENT, GET SLOT

     EXAMPLE         : STORE_SELECT 4


RESET_SELECT    : RESET ALL SELECTION SLOTS

     -> THIS CLEARS/RESETS ALL SELECTIONS IN ALL SLOTS


PRINT_SELECTED  : PRINTS THE CURRENT SELECTION
```

```
WRAP            : ENCAPSULATE CURRENT SELECTION IN MARQUEE


TOUCH      : SIMULATE CLICKING ON A NODE TO SHOW IT’S PANEL SETUPS

     SYNTAX    : TOUCH [NODE]
     EXAMPLE   : TOUCH SPEAKER_11

     -> X-HAIRS WILL BE DIRECTED TO THIS NODE
     -> ONLY THE FIRST NODE IN ANY [LIST] IS TOUCHED
```

The following commands are used to control the position, size, and content of the function panels. Whilst these are generally used at patch load/save time, they also may be useful in writing scripts aimed at extending the functionality of SoundSquares :

```
SHOW_PANEL      : SHOW NAMED PANEL
HIDE_PANEL      : HIDE NAMED PANEL
TOGGLE_PANEL    : SHOW/HIDE PANEL

     SYNTAX     :    TOGGLE_PANEL [PANEL]
     [PANEL]    :    VIRTUAL | DELAY | FILTERS | DYNAMICS
                     MUTES | ENVELOPES | SCOPE | FFT
                     MASTER | TOOLBOX | CLI


PANEL_SETUP     : CONFIGURE A PANEL

     SYNTAX     : PANEL_SETUP [PANEL] X:PX Y:PX W:PX H:PX S:SCALE(1 or 2)

     -> PX VALUES ARE ALL PIXEL-COORDINATES
     -> SCALE DETERMINES HOW LARGE THE PANEL CONTENTS ARE DISPLAYED



POSITION_PANEL_AT    : MOVE A PANEL

     SYNTAX     : POSITION_PANEL_AT [PANEL] X:PX Y:PX
     EXAMPLE    : POSITION_PANEL_AT DYNAMICS X:-200 Y:-200

     -> X:0 Y:0 IS AT THE CENTRE OF THE VIEW-WINDOW
     -> THIS POSITIONS THE PANEL SO ITS CENTRE
     -> IS PUT AT THE COORDINATES IN THE COMMAND


VIEW_ENVELOPES  : SHOW ENVELOPES FOR NAMED NODES IN ENVELOPES PANEL

     SYNTAX     : VIEW_ENVELOPES [NODES]
     EXAMPLE    : VIEW_ENVELOPES LEFT RIGHT

     -> SHOWS ENVELOPES FOR NAMED NODES
     -> IF A GROUP IS NAMED, ONLY MEMBERS OF GROUPS ARE SHOWN

VERBOSE         : TOGGLES DENSITY OF CLI RESPONSE TEXT

     SYNTAX     : VERBOSE [0 or 1]
     EXAMPLE    : VERBOSE 0                        -> CONCISE CLI (DEFAULT)
                : VERBOSE 1                        -> LONG-WINDED CLI
```

The folllowing commands are used to control stage navigation and to switch pages within the interface :

```
ZOOM_IN    : ZOOM IN A STEP (*1.1 MAGNIFICATION)
ZOOM_OUT   : ZOOM OUT A STEP (/1.1 MAGNIFICATION)

ZOOM_LEVEL : ZOOM TO A SPECIFIED LEVEL

     SYNTAX    : ZOOM_LEVEL [0.5 - 4.0]
     EXAMPLE   : ZOOM_LEVEL 2.0


FLAT_ZOOM_IN   : STRAIGHT ZOOM, TAKING NO ACCOUNT OF STAGE DISPLACEMENT
FLAT_ZOOM_OUT  : STRAIGHT ZOOM, TAKING NO ACCOUNT OF STAGE DISPLACEMENT


DISPLACE_STAGE  : MOVE THE VIEWING WINDOW RELATIVE TO STAGE CENTRE

     SYNTAX    : DISPLACE_STAGE X:PX Y:PY
     EXAMPLE   : DISPLACE_STAGE X:100 Y:100

     -> POSITIONS THE CENTRE OF THE DISPLACE
     -> OVER STAGE COORDINATE PX PY


FIND       : FIND NAMED NODE

     SYNTAX    : FIND [NAME]
     EXAMPLE   : FIND GROUP_1

     -> MOVES INTERFACE TO PUT NAMED NODE AT CENTRE OF SCREEN
```

The following commands are used to control which page of the interface is currently being viewed :

```
PAGE       : TAKE THE GUI TO A NAMED PAGE

     SYNTAX     : PAGE [PAGE]
     [PAGE]     : STAGE | OUTPUTS | INPUTS | MATRIX | FEEDBACK
     EXAMPLE    : PAGE INPUTS



NEXT_PAGE       : GO TO NEXT PAGE
PREVIOUS_PAGE   : GO TO PREVIOUS PAGE

     -> PAGE ORDER IS AS FOLLOWS :

          STAGE
          INPUTS
          OUTPUTS
          MATRIX
          FEEDBACK
```

The folllowing commands are used to control stage navigation and to switch pages within the interface.

```
ZOOM_IN    : ZOOM IN A STEP (*1.1 MAGNIFICATION)
ZOOM_OUT   : ZOOM OUT A STEP (/1.1 MAGNIFICATION)

ZOOM_LEVEL: ZOOM TO A SPECIFIED LEVEL

     SYNTAX    : ZOOM_LEVEL [0.5 - 4.0]
     EXAMPLE   : ZOOM_LEVEL 2.0


FLAT_ZOOM_IN    : STRAIGHT ZOOM, TAKING NO ACCOUNT OF STAGE DISPLACEMENT
FLAT_ZOOM_OUT   : STRAIGHT ZOOM, TAKING NO ACCOUNT OF STAGE DISPLACEMENT


DISPLACE_STAGE  : MOVE THE VIEWING WINDOW RELATIVE TO STAGE CENTRE

     SYNTAX    : DISPLACE_STAGE X:PX Y:PY
     EXAMPLE   : DISPLACE_STAGE X:100 Y:100

     -> POSITIONS THE CENTRE OF THE DISPLACE
     -> OVER STAGE COORDINATE PX PY


FIND       : FIND NAMED NODE

     SYNTAX    : FIND [NAME]
     EXAMPLE   : FIND GROUP_1

     -> MOVES INTERFACE TO PUT NAMED NODE AT CENTRE OF SCREEN
```

The following commands are used to control which PRE and POST mixers. For patches requiring a one-to-one relationship between SOURCE (as input channel) and TARGET (as speaker) then these mixers are likely not being used. However, when it comes to the equivalent of controlling multiple output 'headphone mixes', or creating common side-chains, the PRE and POST mixers offer an additional layer of processing.

```
INPUT_MIXER     : SETUP THE PRE-MIXERS
OUTPUT_MIXER    : SETUP THE POST-MIXERS
```

I/O mixers use both CHANNEL and PIN to index the member of the mix - PIN indexing refers to the PLUGIN I/O pins, and CHANNEL indexing refers to the individual channels within the I/O mixer itself.

```
     SYNTAX     : INPUT_MIXER [NODES] [ACTION] [OPT] [OPT]

     [ACIONS]   : FOR BOTH INPUT AND OUTPUT MIXERS

          RESET           : SET MIXER TO STRAIGHT-THROUGH ROUTING FOR
                            SINGLE INPUT PIN CORRESPONDING TO ITS INDEX

          NORMALISE       : SET LEVEL OF ALL CHANNELS TO 1.0/CHANNEL-COUNT
          LEVEL_MASTER    : LEVEL MIX MASTER LVL    [dB-LEVEL]

          LEVEL_CHANNEL   : LEVEL A MIXER CHANNEL   [dB-LEVEL]
          MUTE_CHANNEL    : MUTE A MIXER CHANNEL    [INDEX]
          UNMUTE_CHANNEL  : MUTE A MIXER CHANNEL    [INDEX]
          PHASE_CHANNEL   : PHASE A MIXER CHANNEL   [+1 / -1]
          REMOVE_CHANNEL  : REMOVE A MIXER CHANNEL  [INDEX]

          ADD_PIN         : ADD A I/O MIXER PIN     [INDEX]
          REMOVE_PIN      : REMOVE I/O MIXER PIN    [INDEX]
          REPLACE_PIN     : REPLACE PIN INPUT A <- B     [INDEX] [INDEX]
          LEVEL_PIN       : LEVEL A MIXER PIN       [dB-LEVEL]
          MUTE_PIN        : MUTE A MIXER PIN        [INDEX]
          UNMUTE_PIN      : UNMUTE A MIXER PIN      [INDEX]
          PHASE_PIN       : SWITCH PHASE OF PIN     [INDEX]

     [ACIONS]   : FOR ONLY INPUT MIXERS

          MID             : SETUP MID CHANNEL FROM NAMED STEREO PAIR
          SIDE            : SETUP SIDE CHANNEL FROM NAMED STEREO PAIR

          -> BOTH MID AND SIDE TAKE 2 INPUT INDEXES IN ORDER TO WORK
```

I/O mixer example commands, using S1 as the source channel, and therefore refering to its INPUT_MIXER. Wtih the exception of MID and SIDE actions, if using T1 as target channel, all of the following also apply to OUTPUT_MIXER, except that terminology of Input and Output is reversed.

```
-> RESET S1 MIXER
     INPUT_MIXER S1 RESET

-> SET S1 MIXER'S MASTER LEVEL TO -6dB
     INPUT_MIXER S1 LEVEL_MASTER -6.0

-> SET LEVEL OF PIN 4 ON S1's MIXER TO -12dB
     INPUT_MIXER S1 LEVEL_PIN 4 -12

-> ADD PIN 7 TO THE MIXER
     INPUT_MIXER S1 ADD_PIN 7

-> REMOVE PIN 7 FROM THE MIXER
     INPUT_MIXER S1 REMOVE_PIN 7

-> REMOVE CHANNEL 5 FROM THE MIXER
     INPUT_MIXER S1 REMOVE_CHANNEL 5

-> INVERT PHASE OF PIN 7
     INPUT_MIXER S1 PHASE_PIN 7 -1

-> INVERT PHASE OF CHANNEL 5
     INPUT_MIXER S1 PHASE_CHANNEL 5 -1

-> CONFIGURE MIXER TO COMPOSE A MID CHANNEL FROM PINS 1 & 2
     INPUT_MIXER S1 MID 1 2

-> CONFIGURE MIXER TO COMPOSE A SIDE CHANNEL FROM PINS 3 & 4
     INPUT_MIXER S1 SIDE 3 4
```

The following commands are used to control the delay processors associated with each source and target node.

```
DELAY      : SET THE DELAY STATUS FOR A NODE

     SYNTAX     : DELAY [NODES] [STATUS] TIME:(ms)
     [STATUS]   : ON | OFF
     EXAMPLE    : DELAY S1 ON TIME:450
                : DELAY S1 ON
                : DELAY S1 OFF
                : DELAY S1 TIME:-100


     -> TIME: RANGE IS (-1000.00 to 1000.00) IN MILLISECONDS
     -> EITHER, OR BOTH STATUS AND TIME CAN BE USE IN A COMMAND
```

The following commands are used to control the filter processors associated with each source and target node.

```
FILTER     : SET THE FILTERING STATUS FOR A NODE

-> CONFIGURE A FILTER SETUP

     SYNTAX     : FILTER [NODES] [ENABLE] [BAND] [SETTINGS]
     [ENABLE]   : ENABLE indicates band is operational
     [BAND]     : 1 - 5
     [SETTINGS] : TYPE: F: G: BW:
          TYPE :
               HP : HIGH PASS
               LP : LOW PASS
               BP : BANDPASS
               HS : HIGH SHELF
               LS : LOW SHELF
               PEAK : PEAKING/BELL
          F    : FREQUENCY (20-20000 HZ)
          G    : GAIN (dB)
          BW   : BANDWIDTH (0.1-5.0 OCTAVES)
     EXAMPLE    : FILTER S1 BAND:1 ENABLE TYPE:HP F:839.11 G:12.00 BW:1.70

-> SWITCH ON OR OFF FILTERS FOR THIS NODE

     SYNTAX     : FILTER [NODES] [ON | OFF]

EQ_PRESET : LOAD AN EQ PRESET INTO SELECTED NODE

     SYNTAX     : EQ_PRESET [OPTION]
     [OPTION]   : NAMED ‘CREATIVE’ PRESET (SEE EQ_PRESETS.TXT FOR DETAILS)
                  OR CROSSOVER PRESET USING THE FORM :
                     [MODEL]_[TYPE]_[POLES]_[FREQ]
     MODELS     :
          BS    : BESSEL
          BW    : BUTTERWORTH
          LR    : LINKWITZ-RILEY
     TYPE :
          LP    : LOWPASS (FOR SUBS)
          HP    : HIGHPASS (FOR MAINS)
     POLES:
               BS : 2/4/6/8          BW : 2/4/6/8/10       LR : 2/4/8
     FREQ :    50 - 300HZ IN 25HZ INCREMENTS

-> SETUP A BESSEL FILTER, LOW-PASS, 4-POLES @ 125HZ ON NODE SUB_1 :

     EXAMPLE    :    EQ_PRESET SUB_1 BS_LP_4_125
```

```
COPY_FILTER     : COPY THE CURRENT FILTER TO THE CLIPBOARD

          SYNTAX     : COPY_FILTER [NODE]

PASTE_FILTER    : PASTE CLIPBOARD FILTER TO THE SELECTED NODE’S FILTERS

          SYNTAX     : PASTE_FILTER

RESETFILTERS    : FLATTEN AND SWITCH-OFF THE FILTERS

          SYNTAX     : RESET_FILTERS [NODES]
          EXAMPLE    : RESET_FILTERS SOURCES
```

The following commands are used to control the dynamics processors associated with each source and target node.

```
DYNAMICS  : SETUP THE DYNAMICS PROCESSORS

     SYNTAX    : DYNAMICS [NODES] [ENABLED] [MODE] [PARAMS]
     [ENABLED] : ON | OFF
     [MODE]    : COMPRESSOR | GATE | LIMITER
                 COMPANDER | FREEFORM | FOLLOWER
                 SHAPER | EQ
     [PARAMS]  :
               ATTACK : MILLISECONDS
               RELEASE : MILLISECONDS

               COMPRESSOR     : THRESHOLD:   [dB]
                              : RATIO:       [1.0 to 100]
                              : KNEE:        [dB]
               GATE           : THRESHOLD:   [dB]
                              : KNEE:        [dB]
                              : LIMIT:       [dB]
               LIMITER        : LIMIT:       [dB]
                              : KNEE:        [dB]
               COMPANDER      : NODE POSITIONS X1:-X5, Y1:-Y5: in [dB]
               FREEFORM       : NODE POSITIONS X1:-X5, Y1:-Y5: in [dB]
               FOLLOWER       : NO PARAMETERS
               SHAPER         : NODE POSITIONS X1:-X5, Y1:-Y5: in [dB]
               EQ             : NO PARAMETERS
     EXAMPLE :

DYNAMICS S1 ON COMPRESSOR ATTACK:10 RELEASE:130 THRESHOLD:-30 RATIO:1.000 KNEE:18.000
DYNAMICS S1 ON LIMITER ATTACK:250 RELEASE:250 KNEE:18.000 LIMIT:-20.625

     SIMPLE OPERATION : SWITCH ON / OFF DYNAMICS NEEDS JUST :
          DYNAMICS [NODES] [ENABLED]

DYNAMICS_MODE  : CHANGE NODE'S DYNAMICS MODE

     SYNTAX    : DYNAMICS_MODE [NODES] [MODE]
     [MODE]    : COMPRESSOR | GATE | LIMITER
                 COMPANDER | FREEFORM | FOLLOWER
                 SHAPER | EQ
     EXAMPLE   : DYNAMICS_MODE SUB_WOOFER LIMITER
```

```
SIDECHAIN       : CONFIGURE THE SIDECHAIN FOR A DYNAMICS PROCESSOR

     SYNTAX     : SIDECHAIN [NODE] INTERNAL (LOC)
                : SIDECHAIN (FROM) (TO) (LOC)

     (FROM)     : NODE SUPPLYING THE SIDECHAIN
     (TO)       : NODE CONSUMING THE SIDECHAIN
     (LOC)      : PRE_DELAY | PRE_FILTER | POST_FILTER | POST_DYNAMICS

     EXAMPLE    : SIDECHAIN SOURCE_1 SOURCE_2 POST_FILTER
                  SIDECHAIN SOURCE_1 TARGET_6 POST DYNAMICS

     -> POST_FILTER IS THE DEFAULT SIDECHAIN LOCATION
     -> POST_DYNAMICS SI ONLY APPLICABLE WHEN
     -> A TARGET USES A SOURCE AS A SIDECHAIN


COPY_DYNAMICS   : COPY DYNAMICS TO THE CLIPBOARD

     SYNTAX     : COPY_DYNAMICS [NODE]

PASTE_DYNAMICS  : PASTE DYNAMICS TO THE SELECTED DYNAMICS

     SYNTAX     : PASTE_DYNAMICS [NODE]

RESETDYNAMICS   : RESET AND SWITCH-OFF DYNAMICS PROCESSORS

     SYNTAX     : RESET_DYNAMICS [NAMES]
     EXAMPLE    : RESET_DYNAMICS SYNDICATION
```

The following commands are used to control feedback settings. They use a similar mechanism to I/O mixers.

```
FEEDBACK        : SETUP FEEDBACK PATH

     SYNTAX     : FEEDBACK [SOURCES] [TARGETS] [LEVEL]
     [LEVEL]    : dB

     EXAMPLE    : FEEDBACK MAIN_OUTPUT SOURCE_1 -6



REMOVE_FEEDBACK: REMOVE A FEEDBACK PATH BETWEEN SOURCES

     SYNTAX     : REMOVE_FEEDBACK [SOURCES] [TARGETS]
     EXAMPLE    : REMOVE_FEEDBACK SOURCE_1 MAIN_OUTPUT



NO_FEEDBACK     : REMOVE ALL FEEDBACK PATHS TO/FROM

     SYNTAX     : NO_FEEDBACK [NODE]
     EXAMPLE    : NO_FEEDBACK SOURCE_1
                : NO_FEEDBACK MAIN_OUTPUT


TOGGLE_FEEDBACK_MUTE     : TOGGLE MUTE STATUS OF A FEEDBACK PATH

     SYNTAX     : TOGGLE_FEEDBACK_MUTE [TARGET] [SOURCE] [OPTION]
     [OPTION]   : SEE REGULAR TOGGLE_MUTE -> SAME OPTIONS APPLY



TOGGLE_FEEDBACK_PHASE    : TOGGLE PHASE STATUS OF A FEEDBACK PATHSAME AS

     SYNTAX     : TOGGLE_FEEDBACK_PHASE [TARGET] [SOURCE] [OPTION]
     [OPTION]   : SEE REGULAR TOGGLE_PHASE -> SAME OPTIONS APPLY



FEEDBACK_LEVEL : SET THE LEVEL OF A FEEDBACK PATH

     SYNTAX     : FEEDBACK_LEVEL [TARGET] [SOURCE] [LEVEL]
     [LEVEL]    : dB
```

The following commands are used to control SoundSquares mix-slicing mechanism. A slice contains information for multiple sources - their locaion on the stage, and volumes.

```
     SLICE FORMAT :

          SLICE_n [INDEX] [COUNT] {DATA}

          SLICE PARAMETERS :

               S    : SLICE INDEX
               X    : NUMBER OF MEMBER NODES

          SLICE DATA :

               A TAB-SEPARATED LIST OF DATA POINTS

     SLICE STORAGE COMMANDS

     SLICE_M   -> SLICE MASTER CONFIGURATION - NAMES OF SLICED NODES
     SLICE_X   -> X-POSIION OF SLICED NODES
     SLICE_Y   -> Y-POSTION OF SLICED NODES
     SLICE_S   -> SIZE OF SLICED NODES
     SLICE_L   -> LEVEL OF SLICED NODES

     -> ALL SLICE STORAGE COMMANDS INCLUDE [INDEX] AND [COUNT]

          SLICE_M S:1 X:3 ......

     -> INDICATES SLICE [ONE] MASTER CONTAINS NAMES FOR [3] NODES

          SLICE_X S:2 X:5     -> X-POSIION OF SLICED NODES

     -> INDICATES SLICE [TWO] MASTER CONTAINS X-POSITION FOR [5] NODES

STORE_SLICE    : STORE SOURCE NODES UNDER MARQUEE AS A SLICE

     SYNTAX    : STORE_SLICE [INDEX]
     EXAMPLE   : STORE_SLICE 7


JUMP_TO_SLICE  : IMMEDIATE RECALL OF SLICE DATA

     EXAMPLE   : JUMP_TO_SLICE 4

DRIFT_TO_SLICE S:SPEED [INDEX]

     SYNTAX    : DRIFT_TO_SLICE [S:SPEED] [INDEX]
     EXAMPLE   : DRIFT_TO_SLICE S:0.1 5
```

```
LOAD_MACRO_FILE     : LOAD MACRO USING FILE OPEN DIALOG

LOAD_MACRO          : LOAD AND RUN A MACRO FILE
                      FROM A PATH WITHIN PLUGIN'S FOLDER

          EXAMPLE   : LOAD_MACRO PATCHES/5.1.txt

SAVE_NOW            : OVERWRITE THE CURRENT OPEN PATCH
                      USING CURRENT SETUP'S DATA

SAVE_PATCH          : (SAVE AS)
                    : SAVES CURRENT SETUP TO PATCH VIA FILE SAVE DIALOG

AUTO_SAVE           : TURN ON/OFF AUTO-SAVE FUNCTION
AUTO_BACKUP         : TURN ON/OFF AUTO-BACKUP FUNCTION

          EXAMPLE : AUTO_SAVE ON
          EXAMPLE : AUTO_BACKUP OFF

     -> AUTO_SAVE AND AUTO_BACKUP ARE ALSO HANDLED
        VIA THE GENERIC TOGGLE COMMAND
        (SEE TOOLBOX AND FEATURES SECTION)

     -> AUTO_SAVE AND AUTO_BACKUP INTERVAL IS 5 MINUTES
```

The following commands work with a drawing engine that targets the macro panel. This can be used to create custom mini-interfaces of buttons, faders, and text-labels.

```
DRAW_IN_MACRO  : Direct the drawing engine to place following commands
                 into the MACRO PANEL
```

The additional command DRAW_WITH_STAGE is used to instruct the drawing engine to place elements on top of the STAGE.

In order to be able to create reusable sections of drawing code, the engine uses a rolling coordinates system which applies an offset to the drawn nodes. The same is true for the colours of these node elements.

```
_AT       : move the drawing location to optional X: and Y:
_AT+      : positive shift the drawing location by optional X: and Y:
_AT-      : negative shift the drawing location by optional X: and Y:

     EXAMPLE   : _AT X:100 Y:100
               -> position drawing relative 100,100

               : _AT+ Y:200
               -> add 200 to the Y drawing offset

               : _AT- X:100
               -> subtract 100 from the X drawing offset


INK       : set the ink colour using RGB

     EXAMPLE   : INK RED:1.0 GREEN:0.0 BLUE:0.0 L:0.7
     -> Use red ink with 0.7 alpha
     -> Ink applies to text labels

PAINT     : set the paint colour using RGB

     EXAMPLE   : PAINT RED:0.0 GREEN:0.0 BLUE:1.0 L:0.8
     -> Use blue paint with 0.8 alpha
     -> paint applies to rectangles, buttons, and faders

     -> for both INK and PAINT commands, the RED: GREEN: BLUE and L:
     -> parameters are optional, as their impact is cumulative

               : INK RED:1.0 GREEN:0.5 BLUE:0.0 L:1.0
               -> creates orange ink with alpha 1.0
               : INK GREEN:1.0 L:0.5
               -> the ink is now yellow with alpha 0.5
```

```
RECT      : DRAW A RECTANGLE

     SYNTAX    : RECT X: Y: W: H:
     EXAMPLE   : RECT X:-100 Y:-50 W:200 H:100

     -> Draws a rectangle 200x100 positioned at -100,-50
     -> RELATIVE to the drawing offset as defined by _AT commands



TEXT      : DRAW TEXT

     SYNTAX    : TEXT (ALIGN) [X: Y:] “TEXT”
     (ALIGN)   : _LEFT_ | _CENTRE_ | _RIGHT_
     EXAMPLE   : TEXT _CENTRE_ X:100 Y:50 “HELLO WORLD”

     -> Draws the text “Hello World” positioned at 100,50
     -> RELATIVE to the drawing offset as defined by _AT commands
     -> Text is centre-aligned
     -> TEXT command is different to NOTE command -
     -> NOTE is for user-annotations
     -> TEXT is for shims, overlays, and additional interface layout


BUTTON    : DRAW A BUTTON

     SYNTAX    : BUTTON [X: Y: W: H] “COMMAND STRING”
     EXAMPLE   : BUTTON X:0 Y:0 W:50 H:12 “MUTE (0)”

     -> button uses alpha*0.5 For regular state
     -> and alpha*1.0 For mouse_over
     -> when pressed, the button executes “MUTE (0)”
     -> in the root scripting context
```

In the following listing,

```
     [A] and [B] are previously declared registers/variables
     [i] is an immediate value written into the body of the script
     [NAME] is an immediate string
     [LABEL] is a jump label notated as either >>LABEL or ->LABEL
     [

# VARIABLE/REGISTER DECLARATION

     INT              [NAME]     [INT]             int i = 123;
     FLOAT            [NAME]     [FLOAT]           float f = 123.456;
     STRING           [NAME]     [STRING]          string s = “hello world”;
     DEL              [A]                          delete [] A;

# FLOW CONTROL

     CALL             [LABEL]                      do_this();
     GOTO             [LABEL]                      goto label;
     JUMP             [LABEL]                      pause(); goto label;
     RTN                                           return;
     END                                           exit();
     PAUSE            [i]                          pause(milliseconds);
     NOP                                           do_nothing();

# INDIRECTION

     ALIAS            [NAME]     [POINTER_NAME]

          float *NAME = &POINTER_NAME;
          int   *NAME = &POINTER_NAME;

# STACK OPERATIONS

     PUSH             [A]                          stack[n] = A;
     PUSHI            [TYPE]     [i]               stack[n] = (TYPE)immediate;

     POP              [A]                          A = stack[n];
     POPI             [A]                          new(A) = stack[n];

     MOV              [A]  [B]                     A = B;
     MOVI             [A]  [i]                     A = i;

     SWAP             [A]  [B]                     C = A; A = B; B = C;

# THREADING

     DISPATCH         [LABEL] [NAME]
```

```
          run_new_thread_from(&ENTRY_POINT, THREAD_NAME);

     READY              [ENTRY_POINT]    [THREAD_NAME]

          setup_new_thread_from(&ENTRY_POINT, THREAD_NAME);

     TELL               [THREAD NAME]    [COMMAND]        thread[n].do(COMMAND);

     TERMINATE          [THREAD NAME]                     thread[n].die();
                        -> if no [THREAD_NAME] specified, all threads terminate

     PROTECT_THREAD                                       thread.isSpecial = true;
     UNPROTECT_THREAD                                     thread.isSpecial = false;

     THREAD_INFO                                          print(thread_info);
                        -> PROXY : TI

# NUMERICAL OPERATIONS

     INC                [A]                               A++;
     DEC                [A]                               A--;
     ADD                [A] [B]                           A += B;
     ADDI               [A] [i]                           A += i;
     SUB                [A] [B]                           A -= B;
     SUBI               [A] [i]                           A -= i;
     MUL                [A] [B]                           A *= B;
     MULI               [A] [i]                           A *= i;
     DIV                [A] [B]                           A /= B;
     DIVI               [A] [i]                           A /= i;
     NEG                [A]                               A *= -1;
     TOG                [A]                               A = A == 0 ? 1 : 0;
     SQRT               [A]                               A = sqrt(A);
     RND                [A]                               A = round(A);
     MIN                [A] [B]                           A = min(A, B);
     MINI               [A] [i]                           A = min(A, i);
     MAX                [A] [B]                           A = max(A, B);
     MAXI               [A] [i]                           A = max(A, i);
     FLOOR              [A]                               A = floor(A);
     CEIL               [A]                               A = ceil(A);
     ROUND              [A] [i]                           A = round(A, i);
          -> # DECIMAL POINTS [i] IS OPTIONAL
     SIN                [A]                               A = sin(A);
     SIND               [A]                               A = sin(A*(PI/180));
     COS                [A]                               A = cos(A);
     COSD               [A]                               A = cos(A*(PI/180));
     TAN                [A]                               A = tan(A);
     TAND               [A]                               A = tan(A*(PI/180));
```

```
# COMPARISON OPERATIONS

     CMP_EQ          [A]  [B]  [COMMAND]                if(A == B) COMMAND();
     CMP_EQI         [A]  [i]  [COMMAND]                if(A == i) COMMAND();
     CMP_NE          [A]  [B]  [COMMAND]                if(A != B) COMMAND();
     CMP_NEI         [A]  [i]  [COMMAND]                if(A != i) COMMAND();
     CMP_LT          [A]  [B]  [COMMAND]                if(A < B) COMMAND();

     CMP_LTI         [A]  [i]  [COMMAND]                if(A < i) COMMAND()
     CMP_LTE         [A]  [B]  [COMMAND]                if(A <= B) COMMAND()
     CMP_LTEI        [A]  [i]  [COMMAND]                if(A <= i) COMMAND()
     CMP_GT          [A]  [B]  [COMMAND]                if(A > B) COMMAND();
     CMP_GTI         [A]  [i]  [COMMAND]                if(A > i) COMMAND();
     CMP_GTE         [A]  [B]  [COMMAND]                if(A >= B) COMMAND();
     CMP_GTEI        [A]  [i]  [COMMAND]                if(A >= i) COMMAND();

# LOGICAL

     NOT             [A]                                A = !A;
     AND             [A]  [B]                           A = A && B;
     NAND            [A]  [B]                           A = !(A && B);
     OR              [A]  [B]                           A = A || B;
     NOR             [A]  [B]                           A = !(A || B);
     XOR             [A]  [B]                           A = A ^ B

# BITWISE

     SHL             [A]  [B]                           A = A << B;
     SHLI            [A]  [i]                           A = A << i;
     SHR             [A]  [B]                           A = A >> B;
     SHRI            [A]  [i]                           A = A >> i;

# STRING

     STREVAL         [A]                                A = eval(A);
```

PLUS A WHOLE LOAD OF UNDOCUMENTED COMMANDS AROUND COLOUR, SERIAL, AND VARIOUS OTHER THINGS ...

POKE-ABOUT AND LISTEN TO THE CLI TRACER TO SEE WHAT ELSE YOU MIGHT FIND ;)