

Nspire Scripting Application Programming Interface

Table of Contents

1	Copyright	4
2	Standard Libraries	5
2.1	Math	5
2.2	String	5
2.3	Table	5
2.4	Coroutine	5
2.5	Built-in Function	5
2.6	Unimplemented	5
3	Image Library	6
3.1	new	6
3.2	width	6
3.3	height	6
3.4	copy	6
4	Graphics Library	8
4.1	drawArc	8
4.2	drawImage	8
4.3	drawLine	8
4.4	drawPolyLine	8
4.5	drawRect	8
4.6	drawString	9
4.7	fillArc	9
4.8	fillPolygon	9
4.9	fillRect	9
4.10	getStringHeight	9
4.11	getStringWidth	9
4.12	setColorRGB	9
4.13	setFont	9
4.14	setPen	10
5	String Library Extension	11
5.1	split	11
5.2	uchar	11
5.3	usub	11
6	Math Library Extension	12
6.1	eval	12
7	Clipboard Library	13
7.1	addText	13
7.2	getText	13
8	Document Library	14
8.1	markChanged	14
9	Symbol Table Library	15
9.1	list	15

9.2	recall	15
9.3	recallstr	15
9.4	store	15
9.5	monitor	15
9.6	unmonitor	15
10	Locale Library	16
10.1	name	16
11	Timer Library	17
11.1	getMilliSecCounter	17
11.2	start	17
11.3	stop	17
12	2D Editor	18
12.1	getText	18
12.2	move	18
12.3	newRichText	18
12.4	resize	18
12.5	setFormattedText	18
12.6	setReadOnly	18
12.7	setText	18
13	Platform Library	19
13.1	gc	19
13.2	isColorDisplay	19
13.3	isDeviceModeRendering	19
13.4	window	19
14	Class Library	21
14.1	class	21
15	Event Handling	22
15.1	activate	22
15.2	arrowDown	22
15.3	arrowKey	22
15.4	arrowLeft	22
15.5	arrowRight	22
15.6	arrowUp	22
15.7	charIn	23
15.8	backspaceKey	23
15.9	backtabKey	23
15.10	clearKey	23
15.11	contextMenu	23
15.12	copy	23
15.13	create	23
15.14	cut	23
15.15	deactivate	24
15.16	deleteKey	24
15.17	destroy	24
15.18	enterKey	24
15.19	escapeKey	24

15.20	help	24
15.21	mouseDown	24
15.22	mouseMove	24
15.23	mouseUp.....	24
15.24	paint	25
15.25	paste	25
15.26	resize	25
15.27	restore.....	25
15.28	returnKey	25
15.29	rightMouseDown	25
15.30	rightMouseUp.....	25
15.31	save	25
15.32	tabKey.....	26
15.33	timer	26
15.34	varChange	26
16	Application Life Cycle	27

1 Copyright

This material is copyrighted by Texas Instruments © 2011.

2 Standard Libraries

The Nspire implementation of Lua implements most standard libraries that come with the Lua distribution. See the Lua Reference Manual (<http://www.lua.org/manual/5.1/manual.html#5>) for definitions of the standard functions.

2.1 *Math*

abs, acos, asin, atan, atan2, ceil, cos, cosh, deg, exp, floor, fmod, frexp, huge, ldexp, log, log10, max, min, modf, pi, pow, rad, random, randomseed, sin, sinh, sqrt, tan, tanh.

2.2 *String*

byte, char, dump, find, format, gmatch, gsub, len, lower, match, rep, reverse, sub, upper.

String routines *lower* and *upper* are not tailored to the current locale. The conversion of strings to upper and lower case letters operate only on the 26 letters of the Latin alphabet. This restriction also applies to the alphabetic matching patterns (%a, %l, %u, and %w) employed by the *find*, *gmatch*, and *match* functions.

2.3 *Table*

concat, insert, maxn, remove, sort.

2.4 *Coroutine*

create, resume, running, status, wrap, yield.

2.5 *Built-in Function*

assert, collectgarbage, error, gcinfo, getfenv, getmetatable, ipairs, load, loadstring, next, pairs, pcall, rawequal, rawget, rawset, select, setfenv, setmetatable, tonumber, tostring, type, unpack, xpcall.

2.6 *Unimplemented*

Some standard Lua libraries are not implemented in Nspire: file, io, os, and package.

Some standard functions are not implemented in Nspire: dofile, loadfile

3 Image Library

An “image” object is a container for graphical images, typically small GUI objects such as buttons, arrow heads, and other such graphical adornments.

3.1 *new*

`image.new(str)`

This function returns a new image object from a string input. The string consists of the image header followed by the binary representation of the image pixels.

The header consists of 20 bytes of data arranged as presented in the following table. All fields are little endian integers.

Table 1, Image header

Offset	Width (bytes)	Contents
0	4	Pixel width of image
4	4	Pixel height of image
8	1	Image alignment (0)
9	1	Flags (0)
10	2	Pad (0)
12	4	The number of bytes between successive raster lines
16	2	The number of bits per pixel (16)
18	2	Planes per bit (1)

The image pixel data immediately follows the header. Pixels are arranged in rows. Each pixel is a little endian 16-bit integer with five bits for each color red, green, and blue. The top bit determines if the pixel is drawn. If it is zero (0), the pixel is not drawn. If it is one (1), the pixel is drawn in the RGB color of the remaining 15 bits.

0x8000 is black, 0x801F is blue, 0x83E0 is green, 0xFC00 is red, and 0xFFFF is white.

3.2 *width*

`image:width()`

Returns the pixel width of the image.

3.3 *height*

`image:height()`

Returns the pixel height of the image.

3.4 *copy*

`image:copy(width, height)`

Returns a copy of the input image scaled to fit the specified pixel width and height.

The width and height default to the size of the input image.

The routine raises an error if the new width or height scaling factors are less than or equal to 0 or greater than 25000. This routine raises an error if there is insufficient memory to create the new image.

4 Graphics Library

A graphics context is a module which has a handle to the script app's graphics output window and a library of graphics routines which are used to draw on the window. A graphics context is supplied to the script "on.paint" event handler each time the window needs to be redrawn.

The graphics context employs a pixel-based coordinate system with the origin in the upper left corner of the drawing window.

4.1 *drawArc*

```
gc:drawArc(x, y, width, height, startAngle, endAngle)
```

Draws an arc in the rectangle with upper left corner (x,y) and pixel *width* and *height*. The arc is drawn beginning at *startAngle* degrees and ending at *endAngle*. Zero degrees points to the right and 90 degrees points up (standard mathematical practice but worth mentioning since the y axis is inverted).

To draw a circle, the *width* and *height* must be equal length and the start and end angles must be 0 and 360. If *width* and *height* are different lengths, this routine will draw an oval.

4.2 *drawImage*

```
gc:drawImage(image, x, y)
```

Draws an image at (x, y). The image must have been created by a previous call to `image.new(...)`.

4.3 *drawLine*

```
gc:drawLine(x1, y1, x2, y2)
```

Draws a line from (x1,y1) to (x2,y2).

4.4 *drawPolyLine*

```
gc:drawPolyLine({x1, y1, x2, y2, ..., xn, yn})
```

Draws a series of lines connecting the (x,y) points. The polygon is not automatically closed; the first x-y coordinate pair must be repeated at the end of the array of points to draw a closed polygon.

4.5 *drawRect*

```
gc:drawRect(x, y, width, height)
```

Draws a rectangle at (x,y) with the given pixel *width* and *height*.

4.6 *drawString*

```
gc:drawString("text", x, y [,verticalalignment])
```

Draws text on the window beginning at pixel location (x,y). Vertical alignment may be "baseline", "bottom", "middle", or "top". This aligns the text in the height of the characters' bounding rectangle.

Returns the x pixel position after the text.

4.7 *fillArc*

```
gc:fillArc(x, y, width, height, startAngle, endAngle)
```

Fills an arc with the preset color. See setColorRGB to set the fill color.

4.8 *fillPolygon*

```
gc:fillPolygon({x1, y1, x2, y2, ... xn, yn})
```

Fills a polygon with the preset color. The array of points bounds the polygon.

4.9 *fillRect*

```
gc:fillRect(x, y, width, height)
```

Fills a rectangle with the preset color.

4.10 *getStringHeight*

```
gc:getStringHeight("text")
```

Returns the pixel height of the text. The pixel height is determined by the font setting previously set by a call to setFont.

4.11 *getStringWidth*

```
gc:getStringWidth("text")
```

Returns the pixel width of text. The pixel width is calculated using the font setting previously set by a call to setFont.

4.12 *setColorRGB*

```
gc:setColorRGB(red, green, blue)
```

Sets the color for subsequent draw and fill routines. The red, green, and blue components of the color are values in the range 0 to 255. Black is 0,0,0 and white is 255,255,255.

4.13 *setFont*

```
gc:setFont(family, style, size)
```

Sets the font for drawing text and measuring text size. Family may be "sansserif" or "serif". Style may be "r" for regular, "b" for bold, "i" for italic, or "bi" for bold italic. Returns the font family, style, and size previously in effect.

4.14 *setPen*

```
gc:setPen(thickness, style)
```

Sets the pen for drawing lines and borders. Thickness may be “thin”, “medium”, or “thick”. Style may be “smooth”, “dotted”, or “dashed”.

5 String Library Extension

In addition to the standard Lua string functions there are a few routines which aid handling Unicode strings.

5.1 *split*

```
string.split(str [,delim])
```

Divides *str* into substrings based on a delimiter, returning a list of the substrings. The default pattern for the delimiter is white space (“%s+”).

5.2 *uchar*

```
string.uchar(chnum, chnum, ...)
```

Unicode characters may be included in strings by encoding them in UTF-8. This routine converts one or more Unicode character numbers into a UTF-8 string.

5.3 *usub*

```
string.usub(str, startpos, endpos)
```

This routine returns a substring of *str*. It is the Unicode version of `string.sub`. It accounts for multi-byte characters encoded in UTF-8.

6 Math Library Extension

In addition to the functions that come with the standard Lua math library there is an interface to the Nspire math server. The interface allows access to the advanced mathematical features of the Nspire product.

6.1 *eval*

```
math.eval(math_expression)
```

This function sends an expression or command to the Nspire math server for evaluation. The input expression must be a string that the Nspire math server can interpret and evaluate.

If the math server successfully evaluates the expression, it returns the numerical results. The *eval* function returns no result if the math server does not return a calculated result or if the calculated result cannot be represented as a fundamental Lua data type.

If the math server cannot evaluate the expression because of a syntax, simplification, or semantic error, *eval* returns two results: nil and an error number.

7 Clipboard Library

7.1 *addText*

```
clipboard.addText(string)
```

This routine adds the contents of *string* to the clipboard as plain text, MIME type “text/plain”.

7.2 *getText*

```
clipboard.getText()
```

This routine returns the contents of the clipboard as a string of plain text. If the clipboard does not contain any text (MIME type “text/plain”) then this routine returns nil.

8 Document Library

8.1 *markChanged*

```
document.markChanged()
```

This routine marks the current document as changed. The user will be prompted to save the Nspire document before closing.

9 Symbol Table Library

The symbol table is used by the Nspire math engine to calculate and store values. This library gives scripts access to the symbol table.

Not all values in the symbol table have compatible types in Lua. But many important variable types are supported: real and integer numbers, strings, and lists of numbers and strings.

9.1 *list*

```
var.list()
```

This routine returns a list of names of variables currently defined in the symbol table.

9.2 *recall*

```
var.recall(name)
```

Returns the value of a math variable with the given *name*. If the type of the named variable has no compatible Lua type, then nil and an error message are returned.

9.3 *recallstr*

```
var.recallstr(name)
```

Returns the value of a math variable with the given *name* as a string. Some math types have no compatible Lua type but all math types can be represented as a string. If the value cannot be recalled even as a string, this function returns nil and an error message.

9.4 *store*

```
var.store(name, value)
```

Stores *value* as a math variable with the given *name*. If the value cannot be stored, then an error message is returned. Otherwise, nil is returned.

9.5 *monitor*

```
var.monitor(name)
```

Turns on monitoring of the math variable with given *name*. Whenever another application changes the math variable, this script application's on.varChange handler is called. See the description of on.varChange below.

The math variable must already exist before monitoring can be requested. You can assure that the variable exists by calling var.store to give the math variable a value before calling var.monitor.

9.6 *unmonitor*

```
var.unmonitor(name)
```

Turns off monitoring of the named math variable.

10 Locale Library

10.1 *name*

`locale.name()`

Returns the name of the current locale. The locale name is a two-letter language code. The language code may be followed by an underscore and two-letter country code.

11 Timer Library

Each script application has one timer at its disposal. The resolution of the timer depends on the platform. It is about 0.01 seconds on the hand-held device.

The script application should implement the “on.timer()” function to respond to timer ticks.

The timer continues to send ticks to the script application even when its window is not visible on the screen.

The timer is automatically stopped when the document containing the script application is closed or if the script application is deleted.

11.1 *getMilliSecCounter*

```
timer.getMilliSecCounter()
```

Returns the value of the internal millisecond counter. The counter rolls over to zero when it passes 2^{32} milliseconds.

11.2 *start*

```
timer.start(period)
```

Starts the timer with the given period in seconds. If the timer is already running when this routine is called, the timer is reset to the new period.

11.3 *stop*

```
timer.stop()
```

Stops the timer.

12 2D Editor

12.1 *getText*

```
D2Editor.getText()
```

Returns the contents of the text editor as a UTF-8 encoded string.

12.2 *move*

```
D2Editor.move(x, y)
```

Sets the parent-relative location of the upper left corner of the text editor.

12.3 *newRichText*

```
D2Editor.newRichText()
```

Creates and returns a new rich text editor. The program must set the location (see `move()` above) and size (see `resize()` below) before the text editor widget is painted the first time.

12.4 *resize*

```
D2Editor.resize(width, height)
```

Changes the width and height of the text editor.

12.5 *setFormattedText*

```
D2Editor.setFormattedText(text)
```

Sets the formatted (?) text of the text editor.

12.6 *setReadOnly*

```
D2Editor.setReadOnly(true or false)
```

Makes the content of the text editor modifiable (false) or unmodifiable (true) by the user.

12.7 *setText*

```
D2Editor.setText(text, cursor, selection)
```

Sets the text content of the text editor. The cursor position is set to 0 (beginning of text), -1 (end of text) or a value from 0 to the length of the text. Text can be selected by specifying a selection index indicating the end of the selection. No text is selected if *selection* = -1.

13 Platform Library

Platform specific information is available through the platform library.

13.1 *gc*

`platform.gc()`

Returns a dummy graphics context. It is typically used to measure pixel lengths and heights of strings when a normal graphics context is not available. This may be the case when creating new text elements when the script app is initialized. A graphics context is available only during paint events, and that may be too late to create and size the containers for text fields.

This graphics context cannot be used to draw graphics since it is not tied to a window.

Here is an example of using the dummy graphics context to get the pixel length and height of a string.

```
local gc = platform.gc()           -- Get the dummy graphics context
gc:begin()                         -- Make it the current graphics context
local width = gc:getStringWidth(a_string) -- Get the pixel length of a_string
local height = gc:getStringHeight(a_string) -- Get the pixel height of a_string
gc:finish()                        -- Restore previous graphics context
```

It is important to use `gc:begin()` to set up the graphics context before using it in the `getString` routines and to call `gc:finish()` to relinquish it when finished with it.

13.2 *isColorDisplay*

`platform.isColorDisplay()`

Returns true if the display of the host platform is color. Returns false if the display is gray scale.

13.3 *isDeviceModeRendering*

`platform.isDeviceModeRendering()`

Returns true if the script is running on the hand-held device or in the emulator of the desktop software. Returns false if the script is running in the normal view of the desktop software.

13.4 *window*

`platform.window()`

Returns the window object currently owned by the script application. The window object has several methods of particular interest.

13.4.1 height and width

```
platform.window:height()  
platform.window:width()
```

Routines `height()` and `width()` return the pixel height and width respectively of the display window.

13.4.2 invalidate

```
platform.window:invalidate(x, y, width, height)
```

This routine invalidates a region of the window and forces it to repaint. `x` and `y` default to (0, 0) and `width` and `height` default to the pixel width and height of the window. The entire window can be forced to repaint with a call to `platform.window:invalidate()` allowing all parameters to take their default values.

14 Class Library

The class library implements basic object-oriented class definitions.

14.1 *class*

```
class([parent_class])
```

Returns a new class. If a parent class is specified, the new class inherits the methods of the parent class.

```
Widget = class()  
function Widget:init() ... end
```

```
Button = class(Widget)  
function Button:init() ... end
```

With these definitions, when the script calls `Button()`, a new `Button` is created, the `Button:init()` function is called to initialize the button, and the newly minted `Button` object is returned as the function result of the call.

Class `Button` in this example inherits all the methods and class variables defined in class `Widget`. Class `Button` can override any methods of its parent class.

15 Event Handling

Script applications respond to external stimuli by implementing event handlers. All the event handlers are grouped in the “on” module.

For example, the application script implements `on.paint` to be notified when it is time to redraw its window. It is passed a graphics context which it can use to call drawing routines on its window.

```
function on.paint(gc)
    gc.drawLine(...)
    :
end
```

15.1 *activate*

`on.activate()`

This routine is called when the script application is activated. The dimensions of the drawing window may not be initialized at this point so it is not a good place to create and position graphical elements if they depend on the window size.

15.2 *arrowDown*

`on.arrowDown()`

This routine is called when the user presses the down arrow key.

15.3 *arrowKey*

`on.arrowKey(key)`

This routine is called when the user presses an arrow key. The *key* parameter may be “up”, “down”, “left”, or “right”. This routine will not be called if the script implements a specific arrow key handler (`on.arrowDown` for instance) for the particular arrow key type.

15.4 *arrowLeft*

`on.arrowLeft()`

This routine is called when the user presses the left arrow key.

15.5 *arrowRight*

`on.arrowRight()`

This routine is called when the user presses the right arrow key.

15.6 *arrowUp*

`on.arrowUp()`

This routine is called when the user presses the up arrow key.

15.7 *charIn*

on.charIn(char)

This routine is called when the user types a letter, digit, or other characters. Parameter *char* is normally a one-byte string but since it can contain a UTF-8 encoded character, it may be two or more bytes long. It might also contain the letters of a function name from one of the short-cut keys; “sin” from the trig menu, for instance.

15.8 *backspaceKey*

on.backspaceKey()

This routine is called when the user presses the [backspace] key on the desktop keyboard or the [del] key on the hand-held device keypad.

15.9 *backtabKey*

on.backtabKey()

This routine is called when the user presses [shift] [tab].

15.10 *clearKey*

on.clearKey()

This routine is called when the user presses the [clear] key on the hand-held keypad.

15.11 *contextMenu*

on.contextMenu()

This routine is called when the user presses the context menu key.

15.12 *copy*

on.copy()

This routine is called when the user selects the copy command either from a menu or by pressing ctrl+C.

15.13 *create*

on.create()

This routine is called when the script application is created. The window size and graphics context are valid at this point.

Use this routine to initialize graphical objects based on the window size. Don't actually paint the screen in this routine since the on.paint event handler will be called soon after this routine finishes.

15.14 *cut*

on.cut()

This routine is called when the user selects the cut command either from a menu or by pressing ctrl+X.

15.15 deactivate

on.deactivate()

This routine is called when the script is deactivated. This happens when the user moves the focus to another page or to another application on the same page.

15.16 deleteKey

on.deleteKey()

This routine is called when the user presses the [delete] key on the desktop keyboard. This is not the [del] key on the hand-held keypad.

15.17 destroy

on.destroy()

This routine is called just before the script application is deleted. A script app is deleted when it is cut to the clipboard and when the document that contains it is closed.

15.18 enterKey

on.enterKey()

This routine is called when the user presses the [enter] key.

15.19 escapeKey

on.escapeKey()

This routine is called when the user presses the [esc] key.

15.20 help

on.help()

This routine is called when the user presses the help key. On the desktop the help key is [ctrl] [?]. On the hand-held device it is [ctrl] (?), the control key over the [trig] button.

15.21 mouseDown

on.mouseDown(x, y)

This routine is called when the user clicks the mouse. x and y are in window-relative pixel coordinates.

15.22 mouseMove

on.mouseMove(x, y)

This routine is called when the user moves the mouse pointer. The mouse button does not have to be pressed to receive these events.

15.23 mouseUp

on.mouseUp(x, y)

This routine is called when the user releases the mouse button.

15.24 *paint*

`on.paint(gc)`

This routine is called when the script application's window needs to be painted. The *gc* graphics context is used in the script code to draw on the window.

15.25 *paste*

`on.paste()`

This routine is called when the user selects the paste command either from a menu or by pressing ctrl+V.

15.26 *resize*

`on.resize(width, height)`

This routine is called when the script application's window changes size. This is a good place to initialize (or reinitialize) graphical objects based on the window size.

15.27 *restore*

`on.restore(state)`

This routine is called when the script application is restored from its saved state in a document or when the app is pasted into a document. It is only called if state was saved with the application when it was previously copied to the clipboard or saved in a document. See the `on.save` handler.

Parameter *state* is the table that the `on.save` event handler returned.

15.28 *returnKey*

`on.returnKey()`

This routine is called when the user presses the carriage return key on the hand-held keypad.

15.29 *rightMouseDown*

`on.rightMouseDown(x, y)`

This routine is called when the user clicked the right mouse button. *x* and *y* are in window-relative pixel coordinates.

15.30 *rightMouseUp*

`on.rightMouseUp(x, y)`

This routine is called when the user releases the right mouse button.

15.31 *save*

`on.save()`

This routine is called when the script app is saved to the document or copied to the clipboard. The script should return a table of whatever data it needs to properly restore when the `on.restore` event handler is called.

15.32 *tabKey*

on.tabKey()

This routine is called when the user presses the [tab] key.

15.33 *timer*

on.timer()

If the script application implements on.timer, then the system will call this routine each time the timer ticks.

15.34 *varChange*

on.varChange(varlist)

This routine is called when a monitored variable is changed by another application. The *varlist* is a list of variable names whose values were changed. This handler must return a value to indicate if it accepts the new value(s) or vetoes the change.

Valid return values are:

- 0 Success. The script application accepts the change.
- 1 Veto range. The new value is unsatisfactory because it is outside the acceptable range, i.e. too low or too high.
- 2 Veto type. The new value is unsatisfactory because its type cannot be used by the script application.
- 3 Veto existence. Another application deleted the variable and this application needs it.

16 Application Life Cycle

Life Cycle of a ScriptApp

