FSUIPC: Lua Library Reference

(for FSUIPC4, version 4.431 and later, and FSUIPC3 version 3.855 and later)

This document merely lists the facilities added to the standard Lua library complement via three new libraries "**ipc**", "**logic**" and "**event**".

The **ipc** library adds all of the facilities needed to interact with FS and FSUIPC (or ESPIPC), whilst the **logic** library justs adds bit-oriented logical operations which are otherwise missing from Lua but needed when dealing with arrays of bits for switches and options in FS. The **event** library provides ways of having dormant Lua plug-ins containing functions activated by events in FS. Events which can be so detected include joystick buttons, keyboard combinations being pressed/released, FS controls being used, and FSUIPC offsets changing values.

The IPC Library

Routine template	Description
`n = ipc.buttons(joynum)`	Get button settings: "joynum" is a joystick number, the same as shown in FSUIPC's Button assignments tab. Provided the joystick is one being scanned by FSUIPC (i.e. it has a button assignment), this function returns the 32-bit mask showing which buttons are currently "on" (1) and "off" (0). Use the **logic** functions to test or isolate bits. Button 0 is the lowest bit (2^0) and so on.
`n = ipc.ask("string")`	This prompts the user via a message window on the FS screen, displaying the "string" as a message. This can be single or multiple-lined (use '\n' for a new line). The user answers with a string value, which is the result of the call. It is then up to the Lua program as to how to interpret this. The window and the reply operate just like the Window used to prompt users for mouse macro names.

`ipc.control(n)`
`ipc.control(n, param)`
	Sends the FS or FSUIPC control 'n', with the optional parameter (assumed 0 if omitted).
	FS controls are listed in a List of ..." controls document provided separately. FSUIPC added control numbers are listed in the Advanced User's guide.

`ipc.display("string")`
`ipc.display("string", delay)`
	Displays the given string value in FS, in a sizeable and undockable window entitled "Lua display". The maximum string which will be displayed is 1023 characters, including new lines (\n) codes.
	If the delay parameter is provided (it is a number) it specifies how long the display should stay for, in seconds. To remove a display prematurely, send a null string ("").
	Note that there is only one such window for all Lua plug-ins. The last one wins!

`n = ipc.elapsedtime()`	This returns the number of milliseconds since FSUIPC was started. It is the same as the value shown in the Log files.
`ipc.exit()`	This terminates the current Lua plug-in thread. For plug-ins using the event library this is the only programmatic way of doing so, as the registration of the event processing functions effectively keeps the thread idling, waiting for those events, until the thread is forcibly killed by the Kill control or by re-loading the same plug-in.
`x = ipc.get("name")`	Retrieves a Lua value (any type) previously stored as a Global by "ipc.set". This mechanism provides a way for a Lua plug-in to pass values on to successive iterations of itself, or provide and retrieve values from other Lua plug-ins.

n = ipc.getLvarId("name") ***[FSUIPC4 only]***	This gets the ID of the current FS local panel variable identified by the name given. These variables are L: <name>. You can provide the L: part explicitly or leave it out. The value returned is numeric in the range 0 to 65535, or **nil** if the variable is not available.
n = ipc.getLvarName(id) ***[FSUIPC4 only]***	This gets the name of the current FS local panel variable identified by the id value, a numeric in the range 0 to 65535. These variables are L: <name> , but the result provided is only the ,name> part, without the L: The value returned is a string, or **nil** if the variable is not available. To get all current LVars you can iterate from 0 upwards until **nil** is returned.

ipc.keypress(keycode)
ipc.keypress(keycode, shifts)
	Sends the specified key press to FS. If the 'shifts parameter is omitted a normal unshifted keycode is sent and a press-and-release. The Advanced User's guide gives a list of keycodes and shifts.

ipc.log("string")	Logs the string provided. The log entry goes to the FSUIPC log gile unless either the Lua plug-in is being run in debug mode (Lua Debug control), or Lua logging is enabled in the FSUIPC options. In these two cases the log message goes to the Lua plug-in's log file instead.
n = ipc.readDBL(offset)	Reads the double floating point (64-bit) value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".
n = ipc.readFLT(offset)	Reads the single floating point (32-bit) value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".
n = ipc.readDD(offset)	Reads the 64-bit signed integer value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".
n = ipc.readLvar("name") ***[FSUIPC4 only]***	This reads the current value of the FS local panel variable called "name". These are L: <name> values. You can provide the L: part explicitly or leave it out. The value returned is numeric, or **nil** if the variable is not available.
n = ipc.readSB(offset)	Reads the 8-bit signed byte value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".
n = ipc.readSD(offset)	Reads the 32-bit signed integer value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".
n = ipc.readSTR(offset, length)	Reads the ASCII string at the given IPC offset, with the maximum length as specified. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".
x1, x2, x3 ... = ipc.readStruct(offset, valuelist, ...) ***for multiple groups:*** **x1, x2, x3 ... =**	Reads multiple values from one or more groups of successive IPC offsets, each starting with one given explicitly. The offsets can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC". The lists consist of one of more entries defining numbers and types

ipc.readStruct(offset1, valuelist1, offset2, valuelist2, ...)	of values, as ‘nTYPE’. Types supported are: UB unsigned 8-bit byte UW unsigned 16-bit word UD unsigned 32-bit dword SB signed 8-bit byte SW signed 16-bit word SD signed 32-bit dword DD signed 64-bit value DBL 64-bit double floating point FLT 32-bit single floating point STR string of ASCII characters (in this case the preceding number, n, gives the length *not* a repeat count) The values are assigned in order to the variables on the left-hand side. For example: A, B, C, S, V, W = ipc.readStruct(0x1234, 3SB, 12STR, 2DBL) Assigns 6 values (*not* 17), in order: A = the signed byte at 0x1234 B = the signed byte at 0x1235 C = the signed byte at 0x1236 S = the <= 12 character string at 0x1237 V = the double float value at offset 0x1243 W = the double float value at offset 0x124B
n = ipc.readSW(offset)	Reads the 16-bit signed word value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
n = ipc.readUB(offset)	Reads the 8 bit unsigned byte value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
n = ipc.readUD(offset)	Reads the 32-bit unsigned integer value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
n = ipc.readUW(offset)	Reads the 16-bit unsigned word value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
ipc.set(“name”, value)	Stores a Lua value (any type) as a Global with the given name. This can be retrieved by this or any other Lua plug-in by using “ipc.get”. This mechanism provides a way for a Lua plug-in to pass values on to successive iterations of itself, or provide and retrieve values from other Lua plug-ins.
ipc.sleep(msecs)	Suspends execution of the plug-in for the given number of milliseconds, allowing other threads to operate with less hindrance.
x = ipc.testbutton(joynum, btn)	Tests a scanned button. “joynum” is a joystick number, the same as shown in FSUIPC’s Button assignments tab. Provided the joystick is one being scanned by FSUIPC (i.e. it has a button assignment), this function returns the state of the specified button number (0–31) as TRUE or FALSE.
X = ipc.testflag(flagnum)	Tests one of the 32 flags (numbered 0–31) specifically available for this plug-in and controlled by the added FSUIPC controls (LuaFlag Set, Clear and Toggle). These are provided so that the user can communicate with the plug-ins via assigned buttons or keypresses.
ipc.writeDBL(offset, value)	Writes the value provided as a double floating point (64-bit) value at the given IPC offset.

	The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
`ipc.writeFLT(offset, value)`	Writes the value provided as a single floating point (32-bit) value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
`ipc.writeDD(offset, value)`	Writes the value provided as a 64-bit signed integer value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
`ipc.writeLvar(“name”, n)` ***[FSUIPC4 only]***	This writes to the FS local panel variable called “name”. These are L: <name> values. You can provide the L: part explicitly or leave it out. If the variable is not currently available, nothing happens.
`ipc.writeSB(offset, value)`	Writes the value provided as an 8-bit signed byte value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.

ipc.writeSD(offset, value)	Writes the value provided as a 32-bit signed integer value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.

ipc.writeSTR(offset, “string”)
ipc.writeSTR(offset, “string”, length)
	Writes the ASCII string at the given IPC offset, either with the same length or extended or truncated to the length optionally specified. The string will have a zero terminator added, so allow for this. If it is extended it is with zeroes.
	The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.

ipc.writeStruct(offset, valuelist, ...)
for multiple groups:
ipc.writeStruct(offset1, valuelist1, offset2, valuelist2, ...)
	Writes multiple values from one or more groups of successive IPC offsets, each starting with the one given explicitly.
	The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
	The list consists of one of more entries defining numbers and types of values, as ‘nTYPE’. Types supported are:
	UB unsigned 8-bit byte
	UW unsigned 16-bit word
	UD unsigned 32-bit dword
	SB signed 8-bit byte
	SW signed 16-bit word
	SD signed 32-bit dword
	DD signed 64-bit value
	DBL 64-bit double floating point
	FLT 32-bit single floating point
	STR string of ASCII characters (in this case the preceding number, n, gives the length *not* a repeat count)
	The values to be written must follow, in the parameter list, the Type specifier. For example:
	ipc.writeStruct(0x1234, 3SB, 55, 66, 77,
	12STR, “a string”, 2DBL, 1.234, 3.456)
	Writes 6 values (*not* 17), in order:
	55 to the signed byte at 0x1234
	66 to the signed byte at 0x1235
	77 to the signed byte at 0x1236
	“a string” with zero padding to the bytes at 0x1237
	1.234 to the double float value at offset 0x1243
	3.456 to the double float value at offset 0x124B

ipc.writeSW(offset, value)	Writes the value provided as a 16-bit signed word value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
ipc.writeUB(offset, value)	Writes the value provided as an 8 bit unsigned byte value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
ipc.writeUD(offset, value)	Writes the value provided as a 32-bit unsigned integer value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
ipc.writeUW(offset, value)	Writes the value provided as a 16-bit unsigned word value at the given IPC offset. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.

The Logic Library

Note that the names of all the functions provided in the logic library begin with a capitalised letter. This is important. It prevents Lua interpreter errors arising from the use of the reserved words “and”, “or” and “not”.

Note that all of these functions handle 32-bit unsigned values, no matter how the parameters are provided.

Routine template	Description
`X = logic.And(y, z)`	X = y & z For example, in binary, 0011 & 1010 = 0010
`X = logic.Nand(y, z)`	X = (~y) | (~z)., same as ~(y & z) For example, in binary, 0011 nand 1010 = 1101
`X = logic.Nor(y, z)`	X = (~y) & (~z)., same as ~(y | z) For example, in binary, 0011 nor 1010 = 0100
`X = logic.Not(y)`	X = ~y For example, in binary, ~ 0011 = 1100
`X = logic.Or(y, z)`	X = y | z For example, in binary, 0011 | 1010 = 1011
`X = logic.Shl(y, n)`	X = y << n For example, in binary, 0011 << 1 = 0110
`X = logic.Shr(Y, N)`	X = y >> n For example, in binary, 1100 >> 1 = 0110
`X = logic.Xor(Y, Z)`	X = y xor z. For example, in binary, 0011 xor 1010 = 1001

The Event Library

Routine template	Description

`event.button(joynum, button, “function-name”)`
`event.button(joynum, button, downup, “function-name”)`
Your processing function:
`function-name(joynum, button, downup)`
	Executes the named function (named as a string, “...”), which must be defined before this line, when a given joystick button changes.
	The button number provided can be 0–31 for normal buttons, 32–39 for 8-way POV, or 255 to indicate that the function should receive all 32 button states when any change.
	Except for the button “255” case, the optional “downup” parameter specifies the change to be detected:
	Omitted when pressed
	1 when pressed
	2 when released
	3 when pressed or released (see Note * below)
	The function is called with the joystick, button and downup details so that the same function can, if desired, be used for more than one such event.
	In the special case of the button being specified as 255, then *any* button change (buttons 0–31, not POV) on the specified joystick will result in the function being executed with the button state provided in the ‘button’ parameter as a 32-bit mask—bit 0 referring to button 0 and so on.

`event.control(controlnum, “function-name”)`
`event.control(controlnum, delta, “function-name”)`
Your processing function:
`function-name(controlnum, param)`
	Executes the named function (named as a string, “...”), which must be defined before this line, when the specified FS control occurs. FS controls are those numbered from 65536 upwards, and listed in my FS control lists.
	If the control is an axis-type control, with a parameter, you can limit the flood of calls you might otherwise get for a changing axis by specifying the “delta” parameter. This is a positive number which tells FSUIPC to only call the function when the parameter from FS changes by at least that amount.
	The control number and its parameter are supplied to the function so that the same function can, if desired, be used for more than one such event.

`event.key(keycode, shifts, “function-name”)`
`event.key(keycode, shifts, downup, “function-name”)`
Your processing function:
`function-name(keycode, shifts, downup)`
	Executes the named function (named as a string, “...”), which must be defined before this line, when a given keypress combination occurs.
	The key code provided is one of the standard list (see the FSUIPC Advanced User’s guide), and the “shifts” represent and combination of these (add them up). An 8 or zero value refers to the plain key:
	1 Shift
	2 Control
	4 Alt
	16 Tab
	32 Windows
	64 Apps
	The optional “downup” parameter specifies the change to be detected:
	Omitted when pressed
	1 when pressed
	2 when released
	3 when pressed or released (see Note * below)

	Note that repeated keys (auto-repeats resulting from holding the keys down) are not processed. The function is called with the key and downup details so that the same function can, if desired, be used for more than one such event.

`event.offset(offset, type, “function-name”)`
`event.offset(offset, “STR”, length, “function-name”)`
`Your processing function:`
`function-name(offset, value)`
	Executes the named function (named as a string, “...”), which must be defined before this line, when the specified FSUIPC offset changes.
	The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
	The type is one of these:
	UB unsigned 8-bit byte
	UW unsigned 16-bit word
	UD unsigned 32-bit dword
	SB signed 8-bit byte
	SW signed 16-bit word
	SD signed 32-bit dword
	DD signed 64-bit value
	DBL 64-bit double floating point
	FLT 32-bit single floating point
	STR string of ASCII characters
	The length parameter is omitted (or ignored) except for the “STR” type, where is can optionally define the string length (max 256). If the length is omitted for the STR type then the string will be zero terminated and will have a maximum length of 255 *not* including the final zero.
	The function is called with the offset, so that the same function can, if desired, be used for more than one such event, and also the current (new) value in that offset. This will be a Lua number for all types except STR where it will be a string.

`event.cancel(“function-name”)`	This simply removes all event tracking by the named function. This is typically used in a Lua program which uses one or two specific events to start a mode where many other events need to be monitored, but which are no longer needed. An example might be some processing for a landing aircraft. Perhaps the gear being lowered is the initiating event, at which more events are requested. After the aircraft has landed, the program can cancel these latter events and go back to waiting for the next time the gear is lowered.

* **Note**: If you really do need to detect both Key or Button presses *and* releases, and the action is possibly going to be quite fast (i.e. not latching, as with a toggle switch), then you should specify the event separately for “down” and “up” rather than use the combined facility. This is because there is no queuing of different event types within each event request—only a count of how many—so the order and nature of the press/release operations will be confused and some may be seen wrongly.

The separate event calls for the press and release can of course still both specify the same function-name, so the effect is still going to be similar. However, because of the asynchronous nature of the key/button scanning in relation to the plug-in threads, whilst you will not miss any presses or releases this way, you may process them in the wrong order.

You could, of course, deal with the problems either method may present by keeping a local flag showing the press or release state, rather than relying only on the “downup” parameter provided in the call to your function.