

FSUIPC: Lua Library Reference

(for FSUIPC4, version 4.60 and later, and FSUIPC3 version 3.98 and later)

This document merely lists the facilities added to the standard Lua library complement via four libraries “**ipc**”, “**logic**”, “**event**” and “**gfd**”.

The **ipc** library adds all of the facilities needed to interact with FS and FSUIPC (or ESUIPC), whilst the **logic** library just 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
<code>n = ipc.ask("string")</code>	<p>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).</p> <p>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.</p> <p>The window and the reply operate just like the Window used to prompt users for mouse macro names.</p>
<code>ipc.btnPress(btn-number)</code> <code>ipc.btnRelease(btn-number)</code> <code>ipc.btnToggle(btn-number)</code>	<p>These provide direct control over the virtual buttons supported by FSUIPC (those normally only controllable via offsets at 3340–3363).</p> <p>The button number is 0–287, and Press, Release, Toggle do as they suggest.</p> <p>Note that because Lua plug-ins are running in a separate thread (one per plug-in), any running Lua plug-in which is operating the virtual buttons can be detected doing so in FSUIPC4’s “Buttons” tab, and therefore such buttons can be programmed therein—provided the plug-in IS actually looping and toggling a fixed button, of course.</p>
<code>n = ipc.buttons(joynum)</code> <code>n = ipc.buttons("joyletter")</code>	<p>Get button settings: “joynum” is a joystick number, the same as shown in FSUIPC’s Button assignments tab. If you use joystick lettering, you can put the letter here instead but it must be "" quotes, as a string.</p> <p>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.</p>
<code>ipc.control(n)</code> <code>ipc.control(n, param)</code>	<p>Sends the FS or FSUIPC control ‘n’, with the optional parameter (assumed 0 if omitted).</p> <p>FS controls are listed in a List of ...” controls document provided separately. FSUIPC added control numbers are listed in the Advanced User’s guide.</p>
<code>ipc.display("string")</code> <code>ipc.display("string", delay)</code>	<p>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.</p> <p>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 (“”).</p> <p>Note that there is only one such window for all Lua plug-ins. The</p>

	<p>last one wins!</p> <p><i>See also ipc.lineDisplay</i></p>
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.
State, x, y, cx, cy = ipc.getdisplay()	<p>This gets information about the current "Lua display" Window, as used by the ipc.display function. The values returned are:</p> <p>State = 0 for no display, 1 for docked, -1 for undocked</p> <p>x, y are the screen coordinates of the top left corner.</p> <p>cx, cy are the width and height, respectively.</p>
n = ipc.getLvarId("name")	<p>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.</p> <p>The value returned is numeric in the range 0 to 65535, or nil if the variable is not available.</p>
n = ipc.getLvarName(id)	<p>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:</p> <p>The value returned is a string, or nil if the variable is not available.</p> <p>To get all current LVars you can iterate from 0 upwards until nil is returned.</p>
ipc.keypress(keycode) ipc.keypress(keycode, shifts)	Sends the specified key press to FS (provided it has keyboard focus). 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.keypressplus(keycode) ipc.keypressplus(keycode, shifts) ipc.keypressplus(keycode, shifts, options)	<p>Same as the ipc.keypress function, above, except that the keypresses are still sent whilst FS is inside a menu dialogue, and the following additional options are provided, according to the value of the "options" parameter:</p> <p>0 or omitted = press-and-release, as for ipc.keypress 1 = press key, not press-and-release 2 = release key, not press-and-release</p> <p>To which optionally one or both of these can be added:</p> <p>4 = change focus to FS before keystroke 8 = return focus to originally active window after keystroke (this needs a previous or concurrent '4' option to get the active window remembered).</p>
ipc.lineDisplay("string") ipc.lineDisplay("string", line)	<p>A variation on the ipc.display function, this also displays the given string value in FS, in a sizeable and undockable window entitled "Lua display", but in this case the maximum string is 255 characters, and any new line codes will be stripped out.</p> <p>This function provides line selection and scrolling effects, controlled by the "line" parameter as follows:</p> <p>line = 0 (Or omitted): Clears the display and puts this text</p>

	<p>(if any) in the first line. Provide a null string ("") to simply initialise the text buffers.</p> <p>line > 0 Specifies the line number for this text, from 1 to 32 (max). Line 1 is the top line. Lines above this, between it and the last line written, are cleared.</p> <p>line < 0 Adds this text as another line in the list, following the last one sent. The line parameter gives the negative of the maximum line number to be used (counting from 1, max 32), and if this line would be placed there, the display is scrolled up one line before it is added.</p>
<code>ipc.log("string")</code>	<p>Logs the string provided. The log entry goes to the FSUIPC log file 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.</p>
<code>ipc.macro("macroname")</code> <code>ipc.macro("macroname", parameter)</code>	<p>Executes the named Macro, named in the same format as you see in the FSUIPC assignment drop-downs. For example:</p> <p style="padding-left: 40px;"><code>ipc.macro("PMDGquad: cutoff1")</code></p> <p>executes the macro named "cutoff1" in the Macro file "PMDGquad.mcro".</p> <p>The optional parameter should be an integer between -32768 and 32767 (or 0 and 65535 for unsigned values).</p> <p>Note that the facility can be used to execute other Lua plug-ins too, for example:</p> <p style="padding-left: 40px;"><code>ipc.macro("Lua display vals")</code></p> <p>or, indeed, any of the Lua controls.</p>
<code>n = ipc.readDBL(offset)</code>	<p>Reads the double floating point (64-bit) value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>n = ipc.readFLT(offset)</code>	<p>Reads the single floating point (32-bit) value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>n = ipc.readDD(offset)</code>	<p>Reads the 64-bit signed integer value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>n = ipc.readLvar("name")</code>	<p>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.</p> <p>The value returned is numeric, or nil if the variable is not available.</p>
<code>n = ipc.readSB(offset)</code>	<p>Reads the 8-bit signed byte value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>n = ipc.readSD(offset)</code>	<p>Reads the 32-bit signed integer value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>n = ipc.readSTR(offset, length)</code>	<p>Reads the string at the given IPC offset, with the length as specified.</p> <p>The string can contain any byte values, including zeroes. It is not restricted to being ASCII. In this respect it can be considered as a</p>

	<p>block of offsets, or a structure without named elements.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>																				
<pre>x1, x2, x3 ... = ipc.readStruct(offset, valuelist, ...) for multiple groups: x1, x2, x3 ... = ipc.readStruct(offset1, valuelist1, offset2, valuelist2, ...)</pre>	<p>Reads multiple values from one or more groups of successive IPC offsets, each starting with one given explicitly.</p> <p>The offsets can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p> <p>The lists consist of one or more entries defining numbers and types of values, as 'nTYPE'. Types supported are:</p> <table> <tr><td>UB</td><td>unsigned 8-bit byte</td></tr> <tr><td>UW</td><td>unsigned 16-bit word</td></tr> <tr><td>UD</td><td>unsigned 32-bit dword</td></tr> <tr><td>SB</td><td>signed 8-bit byte</td></tr> <tr><td>SW</td><td>signed 16-bit word</td></tr> <tr><td>SD</td><td>signed 32-bit dword</td></tr> <tr><td>DD</td><td>signed 64-bit value</td></tr> <tr><td>DBL</td><td>64-bit double floating point</td></tr> <tr><td>FLT</td><td>32-bit single floating point</td></tr> <tr><td>STR</td><td>string of ASCII characters (in this case the preceding number, n, gives the length <i>not</i> a repeat count)</td></tr> </table> <p>The values are assigned in order to the variables on the left-hand side. For example:</p> <p>A, B, C, S, V, W = ipc.readStruct(0x1234, "3SB", "12STR", "2DBL")</p> <p>Assigns 6 values (<i>not</i> 17), in order:</p> <p>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</p>	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 <i>not</i> a repeat count)
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 <i>not</i> a repeat count)																				
n = ipc.readSW(offset)	<p>Reads the 16-bit signed word value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>																				
n = ipc.readUB(offset)	<p>Reads the 8 bit unsigned byte value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>																				
n = ipc.readUD(offset)	<p>Reads the 32-bit unsigned integer value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>																				
n = ipc.readUW(offset)	<p>Reads the 16-bit unsigned word value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>																				
ipc.set("name", value)	<p>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.</p>																				
ipc.setdisplay(x, y, cx, cy)	<p>This changes attributes of the current "Lua display" Window, if there is one displayed. This is the window used by the ipc.display function. The values set are::</p> <p>x, y give the screen coordinates of the top left corner.</p> <p>cx, cy give the width and height, respectively.</p> <p>It is best to read the current values first, using ipc.getdisplay, modify them and write them back.</p> <p>Note that there is ever at most only one Lua display window. This</p>																				

	command operates on that even if it was instigated by another Lua plug-in.
<code>ipc.sleep(msecs)</code>	Suspends execution of the plug-in for the given number of milliseconds, allowing other threads to operate with less hindrance.
<code>x = ipc.testbutton(joynum, btn)</code>	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.
<code>x = ipc.testflag(flagnum)</code>	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.
<code>ipc.writeDBL(offset, value)</code>	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”.
<code>ipc.writeFLT(offset, value)</code>	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”.
<code>ipc.writeDD(offset, value)</code>	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”.
<code>ipc.writeLvar("name", n)</code>	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.
<code>ipc.writeSB(offset, value)</code>	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”.
<code>ipc.writeSD(offset, value)</code>	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”.
<code>ipc.writeSTR(offset, "string")</code> <code>ipc.writeSTR(offset, "string", length)</code>	Writes the 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 you don't specify a length. If it is extended it is with zeroes. The string can contain any byte values, including zeroes. It is not restricted to being ASCII. In this respect it can be considered as a block of offsets, or a structure without named elements. The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. “0AEC”.
<code>ipc.writeStruct(offset, valuelist, ...)</code> <i>for multiple groups:</i> <code>ipc.writeStruct(offset1, valuelist1, offset2, valuelist2, ...)</code>	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 or more entries defining numbers and types of values, as ‘nTYPE’. Types supported are:

	<p> 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 <i>not</i> a repeat count) </p> <p>The values to be written must follow, in the parameter list, the Type specifier. For example:</p> <pre>ipc.writeStruct(0x1234, "3SB", 55, 66, 77, "12STR", "a string", "2DBL", 1.234, 3.456)</pre> <p>Writes 6 values (<i>not</i> 17), in order:</p> <p> 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 </p>
<code>ipc.writeSW(offset, value)</code>	<p>Writes the value provided as a 16-bit signed word value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>ipc.writeUB(offset, value)</code>	<p>Writes the value provided as an 8 bit unsigned byte value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>ipc.writeUD(offset, value)</code>	<p>Writes the value provided as a 32-bit unsigned integer value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>
<code>ipc.writeUW(offset, value)</code>	<p>Writes the value provided as a 16-bit unsigned word value at the given IPC offset.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p>

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
<code>X = logic.And(y, z)</code>	$X = y \& z$ For example, in binary, $0011 \& 1010 = 0010$
<code>X = logic.Nand(y, z)</code>	$X = (\sim y) (\sim z)$., same as $\sim(y \& z)$ For example, in binary, $0011 \text{ nand } 1010 = 1101$
<code>X = logic.Nor(y, z)</code>	$X = (\sim y) \& (\sim z)$., same as $\sim(y z)$ For example, in binary, $0011 \text{ nor } 1010 = 0100$
<code>X = logic.Not(y)</code>	$X = \sim y$ For example, in binary, $\sim 0011 = 1100$
<code>X = logic.Or(y, z)</code>	$X = y z$ For example, in binary, $0011 1010 = 1011$
<code>X = logic.Shl(y, n)</code>	$X = y \ll n$ For example, in binary, $0011 \ll 1 = 0110$
<code>X = logic.Shr(Y, N)</code>	$X = y \gg n$ For example, in binary, $1100 \gg 1 = 0110$
<code>X = logic.Xor(Y, Z)</code>	$X = y \text{ xor } z$. For example, in binary, $0011 \text{ xor } 1010 = 1001$

The COM Library

Routine template	Description
<code>Handle = com.open("port", speed, handshake)</code>	<p>This opens the serial comms port named "port" (e.g. "COM1"), with settings:</p> <p>Speed = baudrate, e.g. 115200 for VRInsight devices, often 4800 or 9600 for GPSs.</p> <p>Handshake defines the protocol for controlling the flow:</p> <ul style="list-style-type: none">0 = none1 = RTS / DTR line levels2 = XON / XOFF3 = Both of the above <p>The port is always opened in 8-bit no parity mode.</p> <p>The handle returned will be zero if the port could not be opened. If the port is already opened by FSUIPC for use in its handling of VRInsight devices, the com.open call will succeed and be granted access to the same port.</p>
<code>com.close(Handle)</code>	<p>This simply closes the port represented by the given Handle. It should always be used before the Lua program terminates.</p>
<code>n = com.test(Handle)</code>	<p>Returns the number of bytes of data available to be read on the port represented by the given Handle.</p>
<code>str, n = com.read(handle,max)</code> <code>str, n = com.read(handle,max,min)</code>	<p>Reads up to 'max' bytes from the port, returning them as a string in 'str' with the number actually read returned in 'n'.</p> <p>If the 'min' parameter is also given, this returns a null string and n=0 until at least that minimum number of bytes are available. It does not block waiting for them.</p>
<code>n = com.write(Handle, "string")</code> <code>n = com.write(Handle, "string", len)</code>	<p>Writes the string to the port. If the length parameter is provided, the string is either extended by zero bytes to that length, or truncated, whichever is the more appropriate.</p> <p>The returned value gives the number of bytes actually sent (or at least, placed in the buffer).</p>

The Event Library

Routine template	Description								
<pre> event.button(joynum, button, "function-name") event.button(joynum, button, downup, "function-name") event.button("joyletter", button, "function-name") event.button("joyletter", button, downup, "function-name") Your processing function: function-name(joynum, button, downup) </pre>	<p>Executes the named function (named as a string, "..."), which must be defined before this line, when a given joystick button changes.</p> <p>"joynum" is a joystick number, the same as shown in FSUIPC's Button assignments tab. If you use joystick lettering, you can put the letter here instead but it must be "" quotes, as a string. Note, however, that the function is called with the translated number as its first parameter.</p> <p>The joystick device concerned can be any supported device on the FS PC or any WideFS client. This includes Windows joysticks, GoFlight modules, and EPIC devices, but <i>not</i> the Virtual Buttons.</p> <p>The button number provided can be 0–31 for normal buttons, 32–39 for 8-way POV (local Windows devices only), or 255 to indicate that the function should receive all 32 button states when any change.</p> <p>Except for the button "255" case, the optional "downup" parameter specifies the change to be detected:</p> <table data-bbox="719 909 1358 1032"> <tr> <td>Omitted</td><td>when pressed</td></tr> <tr> <td>1</td><td>when pressed</td></tr> <tr> <td>2</td><td>when released</td></tr> <tr> <td>3</td><td>when pressed or released (see Note * below)</td></tr> </table> <p>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.</p> <p>In the special case of the button being specified as 255, then <i>any</i> 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.</p>	Omitted	when pressed	1	when pressed	2	when released	3	when pressed or released (see Note * below)
Omitted	when pressed								
1	when pressed								
2	when released								
3	when pressed or released (see Note * below)								
<pre> event.control(controlnum, "function-name") event.control(controlnum, delta, "function-name") Your processing function: function-name(controlnum, param) </pre>	<p>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.</p> <p>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.</p> <p>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.</p>								
<pre> event.flag("function-name") event.flag(flag, "function-name") Your processing function: function-name(flag) </pre>	<p>Executes the named function whenever one of this plug-ins Lua flags is changed (by one of the LuaSet, LuaClear or LuaToggle controls).</p> <p>If no flag number (0–31) is provided, any of the 32 changing will trigger the event. Otherwise only the selected flag will do so.</p> <p>The flag number provided to the named function is the one which changed to trigger the event.</p>								

<pre>event.gfd("function-name") event.gfd(model, "function-name") event.gfd(model, unit, "function-name")</pre> <p><i>Your processing function:</i></p> <pre>function-name(model, unit)</pre> <p><i>This should normally start with a call to gfd.GetValues(model, unit) which makes all the the inputs accessible within the event processing function. See details in the gfd section below.</i></p>	<p>This executes the named function whenever an input occurs on the identified GoFlight device(s).</p> <p>If both "model" and "unit" parameters are omitted, inputs from all connected devices will attempt to trigger this function, whilst if only the "model" is given, only all units of that model type will.</p> <p>The "model" parameter should normally be one of the fixed model names listed in the gfd library section below. these are pre-defined and equated to internal model numbers, in the range 1 to the maximum number of model types.</p> <p>Unit numbers start from 0 and are assigned in ascending order by the Go-Flight interface, GFDev.DLL, which must be accessible.</p> <p>Note: If it is likely that you will get simultaneous events from different devices, whether of the same model or not, then you should consider having separate Lua plug-ins, as otherwise you may lose some events—only one event is handled at a time, and they are not queued.</p>																				
<pre>event.intercept(offset, "type", "function-name") event.intercept(offset, "STR", length, "function-name")</pre> <p><i>Your processing function:</i></p> <pre>function-name(offset, value)</pre>	<p>Executes the named function (named as a string, "..."), which must be defined before this line, when the specified FSUIPC offset is written to by any FSUIPC or WideFS client application or internal module or gauge. The write is intercepted—i.e. prevented from actually affecting the specified offset. It is then up to the intercepting Lua function to decide whether to write the (possibly modified) value to the same offset or not. If it does it must actively do it using the appropriate ipc.writeXXX() function as described earlier.</p> <p>Note that the offset write is only intercepted if it is explicitly addressed in the request from the FSUIPC client. If the client writes to the offset as part of a larger area, with an earlier starting point, the intercept will not occur. However, for all practical applications this should not present any problems.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p> <p>The type is one of these:</p> <table data-bbox="810 1400 1173 1680"> <tbody> <tr><td>UB</td><td>unsigned 8-bit byte</td></tr> <tr><td>UW</td><td>unsigned 16-bit word</td></tr> <tr><td>UD</td><td>unsigned 32-bit dword</td></tr> <tr><td>SB</td><td>signed 8-bit byte</td></tr> <tr><td>SW</td><td>signed 16-bit word</td></tr> <tr><td>SD</td><td>signed 32-bit dword</td></tr> <tr><td>DD</td><td>signed 64-bit value</td></tr> <tr><td>DBL</td><td>64-bit double floating point</td></tr> <tr><td>FLT</td><td>32-bit single floating point</td></tr> <tr><td>STR</td><td>string of ASCII characters</td></tr> </tbody> </table> <p>The length parameter is omitted (or ignored) except for the "STR" type, where it must define the string length (max 256).</p> <p>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.</p>	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
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																				

<pre>event.key(keycode, shifts, "function-name") event.key(keycode, shifts, downup, "function-name") Your processing function: function-name(keycode, shifts, downup)</pre>	<p>Executes the named function (named as a string, "..."), which must be defined before this line, when a given keypress combination occurs.</p> <p>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:</p> <table data-bbox="762 383 951 551"> <tr><td>1</td><td>Shift</td></tr> <tr><td>2</td><td>Control</td></tr> <tr><td>4</td><td>Alt</td></tr> <tr><td>16</td><td>Tab</td></tr> <tr><td>32</td><td>Windows</td></tr> <tr><td>64</td><td>Apps</td></tr> </table> <p>The optional "downup" parameter specifies the change to be detected:</p> <table data-bbox="715 636 1337 759"> <tr><td>Omitted</td><td>when pressed</td></tr> <tr><td>1</td><td>when pressed</td></tr> <tr><td>2</td><td>when released</td></tr> <tr><td>3</td><td>when pressed or released (see Note * below)</td></tr> </table> <p>Note that repeated keys (auto-repeats resulting from holding the keys down) are not processed.</p> <p>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.</p>	1	Shift	2	Control	4	Alt	16	Tab	32	Windows	64	Apps	Omitted	when pressed	1	when pressed	2	when released	3	when pressed or released (see Note * below)
1	Shift																				
2	Control																				
4	Alt																				
16	Tab																				
32	Windows																				
64	Apps																				
Omitted	when pressed																				
1	when pressed																				
2	when released																				
3	when pressed or released (see Note * below)																				
<pre>event.offset(offset, "type", "function-name") event.offset(offset, "STR", length, "function-name") Your processing function: function-name(offset, value)</pre>	<p>Executes the named function (named as a string, "..."), which must be defined before this line, when the specified FSUIPC offset changes.</p> <p>The offset can be specified in Lua format hexadecimal, e.g. 0x0AEC, or in decimal, or as a string e.g. "0AEC".</p> <p>The type is one of these:</p> <table data-bbox="810 1205 1168 1480"> <tr><td>UB</td><td>unsigned 8-bit byte</td></tr> <tr><td>UW</td><td>unsigned 16-bit word</td></tr> <tr><td>UD</td><td>unsigned 32-bit dword</td></tr> <tr><td>SB</td><td>signed 8-bit byte</td></tr> <tr><td>SW</td><td>signed 16-bit word</td></tr> <tr><td>SD</td><td>signed 32-bit dword</td></tr> <tr><td>DD</td><td>signed 64-bit value</td></tr> <tr><td>DBL</td><td>64-bit double floating point</td></tr> <tr><td>FLT</td><td>32-bit single floating point</td></tr> <tr><td>STR</td><td>string of ASCII characters</td></tr> </table> <p>The length parameter is omitted (or ignored) except for the "STR" type, where it 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 <i>not</i> including the final zero.</p> <p>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.</p>	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
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																				
<pre>event.vriread(handle, "function- name") Your processing function: function-name(handle, "data")</pre>	<p>This is an extension to the com library, described earlier, specifically for use with VRInsight devices.</p> <p>It executes the named function whenever an apparently valid VRInsight input arrives on the identified VRInsight device. The latter is identified by the "handle" to the device returned by a com.open call made previously.</p> <p>The "data" provided to the function will be the 1 to 8-character string supplied by the device. Please see the document "Lua plugins for VRInsight devices".</p>																				

<code>event.cancel("function-name")</code>	<p>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.</p> <p>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.</p>
--	---

NOTES

* 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.

The Go-Flight Device (gfd) Library

This library provides full facilities for reading inputs from Go-Flight devices and writing to their displays. It is currently programmed to cover the following devices ("models". note the model code, which is used when addressing the model type in all functions, including the **event.gfd** function already described.

GF166	GF-166 Versatile Radio Panel
GF45	GF-45 Avionics Simulation Unit or GF-45PM Display Panel Module
GF46	GF-46 Multi-Mode Display Module
GFATC	GF-ATC Headset Comms Panel
GFEFIS	GF-EFIS Control Panel Module
GFFMC	GF-FMC Flight Management Computer Module
GFLGT	GF-LGT Landing Gear/Trim control module
GFLGT2	GF-LGT II Landing Gear/Trim control module
GFMCP	GF-MCP Advanced Autopilot Module
GFMCPPRO	GF-MCP Pro Mode Control Panel Module
GMESM	GF-MESM Multi Engine Start Module
GF-P8	GF-P8 Pushbutton/LED Module
GFRP48	GF-RP48 Rotary/Pushbutton/LED Module
GFSECM	GF-SECM Single Engine Aircraft Control Module
GFT8	GF-T8 Toggle Switch/LED Module
GFTPM	GF-TPM Throttle/Prop/Mixture Control Module*
GFTQ6	GF-TQ6 Throttle System**

* The TPM is currently not recognised because of missing support in GFDev.DLL, and no information available on its data formats.

** The TQ6 support is dubious at present. It may be withdrawn—feedback is welcome.

As with FSUIPC's GoFlight button support, you need **GFDev.dll** installed on the FS PC to use this library. Normally it is installed for you by the GoFlight installer—in this case it should be in the same folder as your GFconfig program, probably in Program Files\GoFlight. When installed correctly, FSUIPC should be able to find it automatically, via the GFconfig installed registry entry. If not, you will have to place GFDev.dll in an accessible place. For FSX this can be the FSX Modules folder. For FS9 and before do NOT, repeat NOT, put it into the Modules folder or you will crash FS. Try the main Windows folder.

The list above is based on the latest version of GFDev.dll available at the time of publication: **1.92.0.8**, dated 30th November 2009. The latest version I have is always available from my Support Forum.

Another important point to know when trying to operate GoFlight displays and indicators, whether using GFdisplay.exe or these new Lua facilities, is that (at present at least), the GoFlight drivers do not co-operate well with other using programs. By all means you can share access to knobs and switches, but the GF drivers seem to want to write to all displays and indicators on a module even if only configured to use some of them. For each GoFlight unit you may have to make the choice: GF driver or Lua/FSUIPC plug-in.

The full reference to the functions available in the Library is tabulated on the next page.

The GoFlight coverage may be revised from time to time. The test program (gfdDisplay.lua) supplied with this package will show you what is covered. If you run this test program (i.e. assign a keypress or button to the drop-down entry "Lua gfddisplay.lua" and use it), this is what you should expect, with a descriptive display in a "Lua display" window on screen:

1. All LEDs lit and all Displays showing 8888....
2. The brightness is modified, from 0 to 15 (full) in steps
3. LEDs are alternated 1 0 1 0 1 0 and 0 1 0 1 0 1 four times whilst the displays are alternated 123456 and 654321 (or as many of those digits as can be accommodated).
4. All displays and LEDs should then be blanked / extinguished.
5. The program then processes inputs forever (until Killed), displaying and logging the results.

Routine template	Description
<code>gfd.BlankAll()</code>	Blanks all digital displays (if possible) and switches all indicator lights off.
<code>n = gfd.Buttons()</code>	Returns the state of all buttons supplied by the last call to gfd.GetValues , described below. The states of up to 32 buttons or switches are provided, and these are represented by one bit each in the returned value. Bit 0 (2^0 , worth 1) is the first button, bit 1 (2^1 , worth 2) is the second, and so on.
<code>gfd.ClearLight(model, unit, id)</code>	This simply turns off the indicator light identified by the id number, on the specified model and unit.
<code>n = gfd.Dial(id)</code>	Returns $-n$ (counter-clockwise turn), 0 (no turn) or $+n$ (clockwise turn) for the rotary dial identified by 'id' (0–7) in the input data supplied by the last call to gfd.GetValues , described below. The values of 'n' will be 1 for a slow turn, but larger numbers are possible for faster turns -- except for the RP48, whose dials only ever seem to return -1, 0 or +1.
<code>n = gfd.GetName(model)</code>	This returns the string name of the device of type 'model'. For instance, the name of the model type GFMCPRO is "GFMCPRO".
<code>n = gfd.GetNumDevices(model)</code>	This returns the number of connected devices of type 'model'.
<code>gfd.GetValues(model, id)</code>	This obtains all of the current input values from the identified device. These values are subsequently accessible using these separate functions: <div style="display: flex; justify-content: space-around; margin-top: 5px;"> <div> gfd.Buttons() gfd.Dial(id) gfd.Lever(id) </div> <div> gfd.Selector(id) gfd.TestButton(id) </div> </div>
<code>n = gfd.Lever(id)</code>	Returns the input value from the lever axis identified by 'id' (0–7) in the input data supplied by the last call to gfd.GetValues , described above.
<code>n = gfd.Selector(id)</code>	Returns the numeric position of the selector switch (multi-position switch) identified by 'id' (0–7) in the input data supplied by the last call to gfd.GetValues , described above.
<code>gfd.SetBright(model, unit, n)</code>	This sets the unit's display and indicator brightness, n=0 being off and n=15 being brightest.
<code>gfd.SetDisplay(model, unit, id, "display text")</code>	This attempts to write the given text (truncated if necessary) to the display identified by the id number, on the specified model and unit.
<code>gfd.SetLight(model, unit, id)</code>	This simply turns on the indicator light identified by the id number, on the specified model and unit.
<code>gfd.SetLights(model, unit, on, off)</code>	This sets selected indicator lights on or off, on the specified model and unit. The 'on' and 'off' parameters are masks to determine those indicators to be turned on (bits set in 'on') and those to be turned off (bits set in 'off') Indicators not referenced by bits in either mask are unchanged. Bits are numbered 0 to 31, with 0 being 2^0 , worth 1 and so on. These numbers correspond to the indicator ids used in SetLight and ClearLight .
<code>n = gfd.TestButton(id)</code>	Returns <i>true</i> or <i>false</i> depending on the button/switch setting (0–31) in the input data supplied by the last call to gfd.GetValues , described above.