FSUIPC: Lua Library Reference

(for FSUIPC4, version 4.958 and later, and FSUIPC3 version 3.999z9 and later, and WideClient 6.999z2 and later)

This document lists the facilities added to the standard Lua library complement via ten libraries "**ipc**", "**logic**", "**com**", "**event**", "**sound**", "**gfd**", "**mouse**", "**ext**", "**wnd**" and "**display**" (the last two for WideClient only).

The **ipc** library adds all of the facilities needed to interact with FS and FSUIPC, 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 *(also WideClient except where indicated)*

Routine template	Description

`n = ipc.ask("string")`
[Not WideClient]
 	
`n = ipc.ask("string", COLOUR)`
[Not WideClient nor FSUIPC3]
	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 optional COLOUR parameter can be **RED** or **WHITE.**
	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.

`n = ipc.axis(joynum, "axis")` ***[Not WideClient]***	Returns the current assigned axis value as read from the device (i.e. before calibration). "joynum" is a joystick number, the same as shown in FSUIPC's Axis assignments tab. If you use joystick lettering, you can put the letter here instead but it must be "" quotes, as a string. The specified axis must be one of these: "X", "Y", "Z", "R", "U", "V" (as shown in the assignments tab)

`ipc.btnPress(btn-number)`
`ipc.btnRelease(btn-number)`
`ipc.btnToggle(btn-number)`
	These provide direct control over the virtual buttons supported by FSUIPC (those normally only controllable via offsets at 3340–3363).
	The button number is 0–287, and Press, Release, Toggle do as they suggest.
	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.

`n = ipc.buttons(joynum)`
`n = ipc.buttons("joyletter")`
 	
[Not WideClient]
	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.
	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.clearbitsUB(offset, mask)
n = ipc.clearbitsUW(offset, mask)
n = ipc.clearbitsUD(offset, mask)
	Clears those bits in the Byte (UB), Word (UW) or DoubleWord (UD) offset which correspond to those present in the Mask value.
	This is equivalent to the following where XX is UB, UW or UD:
	`n = ipc.readXX(offset)`
	`n = logic.And(n, mask)`
	`ipc.writeXX(offset, n)`

ipc.clearflag(flagnum) ***[Not WideClient]***	Clears the specified local Flag, 0-255. These flags are the same ones that can be changed by the FSUIPC assigned controls "LuaSet", "LuaClear" and "Lua Toggle". Test flags using the ipc.testflag function.
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)** **ipc.display("string", colour, delay)** ***[WideClient okay, but display is in FS, not on client PC]***	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 (""). With 3 parameters given, the second specifies whether the message should be in red (default, colour=0) or white (colour=1 or any non-zero value). *[This facility was added in FSUIPC 4.904, 3.999z3 and WideClient 6.996]* Note that with WideClient there is only one such window for all Lua plug-ins. The last one wins! This also applies to direct FSUIPC use, *unless* the **ipc.setowndisplay** function (see below) is used to name, position and size an individual window for this plug-in. *See also* **ipc.lineDisplay and ipc.setowndisplay**
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 forc-ibly killed by the Kill control or by re-loading the same plug-in.
x = ipc.get("name")	Retrieves a Lua value (any simple type -- i.e. numbers, strings, booleans) 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. With effect from FSUIPC version 4.958, in combination with WideClient version 6.999z2, the 'Globalness' of these values extends between Clients and Server in a WideFS network, so can be used to communicate values and strings over the network without resorting to user offsets. This only works if Server and Clients are in the same workgroup.

	Use of this should be sparing – the Windows Mailslot system is used and may not cope with excessive use very well. Also note that there is no backlog – the globals are only broadcast when being set (by ipc.set), so anything set before a client is actually running won't be seen by it. Also the Network protocol used is not checked – messages are not guaranteed to arrive. Retries, maybe by a system of Acknowledgement values, are up to the plug-in and would be advisable in any "mission-critical" application of this facility. Note that there are limits on the sizes for network Globalness: the variable names must not be greater than 32 characters, and string values should be no longer than 384 characters. Values outside these limits do not participate.
State, x, y, cx, cy = ipc.getdisplay() ***[Returns only zeroes in WideClient]***	This gets information about the current communal "Lua display" Window, as used by **ipc.display** by default. The values returned are: State = 0 for no display, 1 for docked, -1 for undocked x, y are the screen coordinates of the top left corner. cx, cy are the width and height, respectively.
n = ipc.getLvarId("name") ***[Not WideClient]***	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) ***[Not WideClient]***	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 (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)** ***[Not WideClient]***	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: 0 or omitted = press-and-release, as for **ipc.keypress** 1 = press key, not press-and-release 2 = release key, not press-and-release To which optionally one or both of these can be added: 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).
ipc.lineDisplay("string") **ipc.lineDisplay("string", line)**	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.

[WideClient okay, but display is in FS, not on client PC]	This function provides line selection and scrolling effects, controlled by the "line" parameter as follows: line = 0 (Or omitted): Clears the display and puts this text (if any) in the first line. Provide a null string ("") to simply initialise the text buffers. 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. 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. Note that with WideClient there is only one such window for all Lua plug-ins. The last one wins! This also applies to direct FSUIPC use, *unless* the **ipc.setowndisplay** function (see below) is used to name, position and size an individual window for this plug-in. *See also* **`ipc.display and ipc.setowndisplay`**
`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.
`ipc.macro("macroname")` **`ipc.macro("macroname", parameter)`**	Executes the named Macro, named in the same format as you see in the FSUIPC assignment drop-downs. For example: ipc.macro("PMDGquad: cutoff1") executes the macro named "cutoff1" in the Macro file "PMDGquad.mcro". The optional parameter should be an integer between -32768 and 32767 (or 0 and 65535 for unsigned values). Note that the facility can be used to execute other Lua plug-ins too, for example: ipc.macro("Lua display vals") or, indeed, any of the Lua controls. Note that when used in WideClient, the macro or Lua execution occurs on the FS PC, *not* on the local client PC.
`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")` ***`[Not WideClient]`***	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.readPOV(joynum)` ***`[Not WideClient]`***	Reads the POV value for a scanned joystick. "joynum" is the 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 POV ("Point of View" hat) as a direction-indicating pseudo button number (32–39), or 0 if it isn't pressed.
`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 string at the given IPC offset, with the length as specified. 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".
`x1, x2, x3 ... = ipc.readStruct(offset, valuelist, ...)` ***`for multiple groups:`*** **`x1, x2, x3 ... = ipc.readStruct(offset1, valuelist1, offset2, valuelist2, ...)`**	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 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.RestoreFriction() ***[Not WideClient]*** ***[FSUIPC4 only]***	Restores the SIM1.DLL friction tables to the state they were in when FS was loaded. Using this at the start of any Lua plug-in which sets the friction specifically to suit a particular aircraft profile will ensure that changes made previously are restored, stopping cumulative and unintended changes to those values not explicitly set. (See **ipc.SetFriction**, below.)
ipc.runlua("pathname") or **ipc.runlua("pathname", param)**	Runs the specified Lua program in its own thread. Unlike the ipc.macro method, when used on WideClient this will run the Lua plugin locally.
ipc.set("name", value)	Stores a Lua value (any simple type -- i.e. numbers, strings, booleans) 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. With effect from FSUIPC version 4.958, in combination with WideClient version 6.999z2, the 'Globalness' of these values extends between Clients and Server in a WideFS network, so can be used to communicate values and strings over the network without resorting to user offsets. This only works if Server and Clients are in the same workgroup. Use of this should be sparing – the Windows Mailslot system is used and may not cope with excessive use very well. Also note that there is no backlog – the globals are only broadcast when being set (by ipc.set), so anything set before a client is actually running won't be seen by it. Also the Network protocol used is not checked – messages are not guaranteed to arrive. Retries, maybe by a system of Acknowledgement values, are up to the plug-in and would be advisable in any "mission-critical" application of this facility. Note that there are limits on the sizes for network Globalness: the variable names must not be greater than 32 characters, and string values should be no longer than 384 characters. Values outside these limits do not participate.
n = ipc.setbitsUB(offset, mask) **n = ipc.setbitsUW(offset, mask)** **n = ipc.setbitsUD(offset, mask)**	Sets those bits in the Byte (UB), Word (UW) or DoubleWord (UD) offset which correspond to those present in the Mask value. This is equivalent to the following where XX is UB, UW or UD:

	`n = ipc.readXX(offset)`
	`n = logic.Or(n, mask)`
	`ipc.writeXX(offset, n)`

`ipc.setbtncol(btn, r, g, b)` ***[WideClient ONLY]***	This WideClient-only function is used with the ButtonScreen facility. It changes the button colour for the depicted button corresponding to the number given as 'btn' (0-287). Button number 0 corresponds to "joystick 64, button 0" with 287 being "joystick 72 button 31". This is not applicable to buttons denoted as "Toggle" buttons (T or TN) in the ButtonScreen definition. See **ipc.setbtnstate** for those. The colour to be set is specified by the values given for r (red), g (green) and b (blue), each of which can range from 0 (none) to 255 (full).
`ipc.setbtnstate(btn, state)` ***and*** **`ipc.setbtnstateonly(btn, state)`** ***[WideClient ONLY]***	These WideClient-only functions are used with the ButtonScreen facility. They change the button state (pressed or released) for the button corresponding to the number given as 'btn' (0-287). Button number 0 corresponds to "joystick 64, button 0" with 287 being "joystick 72 button 31". This is only applicable to buttons denoted as "Toggle" buttons (T or TN) in the ButtonScreen definition. A state value of 0 changes the state to "released" and also, in the case of **setbtnstate,** sends a "button up" indication to FSUIPC if the button was previously recorded as being pressed. Any non-zero value sets the state to 'pressed' and also, in the case of **setbtnstate,** sends a "button pressed" indication if it was recorded as released. This facility is designed to be used to allow Toggle buttons to correctly depict the current state of the option they control, even if that state is changed by other actions, such another button screen, keyboard, mouse or flight reloading. Note that you should really only use this for true Toggle actions -- i.e. ones for which the programmed actions in FSUIPC are different from Press and Release and have those actions occur twice in succession does no harm. Where this isn't the case try **setbtnstateonly** so that a single press does not result in a conflict condition.
`ipc.setdisplay(x, y, cx, cy)` ***[Dummy only in WideClient]***	This changes attributes of the current communal "Lua display" Window, if there is one displayed. This is the window used by **`ipc.display`** function by default. The values set are:: x, y give the screen coordinates of the top left corner. cx, cy give the width and height, respectively. These are in screen coordinates, but unless the Windows is undocked (by the user), in windowed mode the placement is limited by the size and position of the FS window itself. It is best to read the current values first, using **`ipc.getdisplay`**, modify them and write them back. Note that there is ever at most only one such "Lua display" window. This command operates on that even if it was instigated by another Lua plug-in. The **ipc.setowndisplay** function (see below) can instead be used to name, position and size an individual window for this specific plug-in.

	See also **`ipc.lineDisplay and ipc.display`**
`ipc.setflag(flagnum)` ***[Not WideClient]***	Sets the specified local Flag, 0-255. These flags are the same ones that can be changed by the FSUIPC assigned controls "LuaSet", "LuaClear" and "Lua Toggle". Test flags using the ipc.testflag function.
`x = ipc.SetFriction(class, surface, type, condition, value)` ***[Not WideClient]*** ***[FSUIPC4 only]***	This is a rather specialist function which specifically changes the rolling, sliding and braking coefficients within FSX's SIM1.DLL. It operates on-the-fly at run time so different plug-ins can be used to tailor these for different aircraft models. It operates with all versions of FSX plus Prepar3D version 1.4. **`Class`** = one of: BRAKE, WHEEL, SCRAPE, SKID, FLOAT, WRUDDER, SKI (where WRUDDER is short for Water Rudder). **`Surface`** = one of: CONCRETE, GRASS, WATER, GRASS_BUMPY, ASPHALT, SHORT_GRASS, LONG_GRASS, HARD_TURF, DIRT, CORAL, GRAVEL, OIL_TREATED, STEEL_MATS, SNOW, ICE, URBAN, FOREST, BITUMINUS, BRICK, MACADAM, PLANKS, SAND, SHALE, TARMAC, WRIGHT_FLYER_TRACK **`Type`** = one of: ROLLING, SLIDING (but only ROLLING for the BRAKE class) **`Condition`** = one of: DRY, RAIN, ICE, SNOW **`Value`** should be a number between 0 and 1 inclusive, but this is not checked. The Boolean result **`x`** is True if this was done, False if there's something wrong. See also **`ipc.RestoreFriction`**, above.
`ipc.setowndisplay("title", x, y, cx, cy)` ***[Not WideClient]***	This sets all further calls to **ipc.display** or **ipc.linedisplay** to operate on a private Window, owned by this Lua plug-in, with the title given. Note that the title is NOT optional. The x, y values give the top left corner position within the FS window in terms of a **percentage** of the window width and height respectively, so that 50, 50 is dead centre. Similarly the cx, cy values give the size in the same way, so that 25,25,50,50 would give a centred window with half the size (in both directions) of the FS window (or screen in full screen mode). The function can be used again to move, resize and/or re-title the window -- the previous one is automatically closed but its contents are retained. Note that the **ipc.getdisplay** and **ipc.setdisplay** functions do not operate on thisWindow -- those functions are purely for the communal Window "Lua Display". Additionally, whilst it can be moved, resized and undocked by the user, any such changes are not currently readable and no record is made of them in any configuration file. *See also* **`ipc.lineDisplay and ipc.display`**
`ipc.sleep(msecs)`	Suspends execution of the plug-in for the given number of

	milliseconds, allowing other threads to operate with less hindrance. Note that if the **ipcPARAM** value has been set by an external **LuaValue** control, the new value becomes available only after an **`ipc.sleep`** function call or an event is actioned.
`x = ipc.testbutton(joynum, btn)` ***[Not WideClient]***	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. You can test for the POV position too using button numbers 32-39, but you might want instead to read the POV state using ipc.readPOV.
`bool = ipc.testbuttonflag(joynum, btn)` ***[Not WideClient]***	Tests a button flag. Just like **testbutton** above except testing the state of the flag associated with the button instead. POVs do not have flags, so the **btn** number range is 0-31. The result is true or false.
`bool = ipc.testflag(flagnum)` ***[Not WideClient]***	Tests one of the 256 flags (numbered 0–255) 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. The result is true or false.

`n = ipc.togglebitsUB(offset, mask)`
`n = ipc.togglebitsUW(offset, mask)`
`n = ipc.togglebitsUD(offset, mask)`
	Inverts those bits in the Byte (UB), Word (UW) or DoubleWord (UD) offset which correspond to those present in the Mask value.
	
	This is equivalent to the following where XX is UB, UW or UD:
	
	`n = ipc.readXX(offset)`
	`n = logic.Xor(n, mask)`
	`ipc.writeXX(offset, n)`

`ipc.toggleflag(flagnum)` ***[Not WideClient]***	Toggles (i.e. inverts) the specified local Flag, 0-255. These flags are the same ones that can be changed by the FSUIPC assigned controls "LuaSet", "LuaClear" and "Lua Toggle". Test flags using the ipc.testflag function.
`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)`	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.

[Not WideClient]	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 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”.
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 One particularly useful application of this facility is to set FS's date and time in one operation so that it's reload of textures etc only occurs once. Here, this example sets the PC's current UTC date and time. It makes use of the built-in **os library** function **os.date**: t = os.date("!*t") ipc.writeStruct(0x023B, "1UB", t.hour, "3UW", t.min, t.yday, t.year)
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 *(also WideClient)*

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 Mouse Library *(not WideClient)*

This library provides some functions for manipulating the mouse, in order to access parts of add-on panels that no other methods appear to reach!

In FSUIPC4 (*only*) It can also be used to do all sorts of fancy things in conjunction with the extensive mouse events detectable using the **event** library. In particular, see the example plug-in supplied called "mrudder.lua".

The mouse position is measured in any one of three ways:

x, y Windows screen coordinates, or 'absolute' position, relative to the top left of the top left-most screen.

xr, yr FS's window coordinates, relative to the top left position of its main window (not including title bar, menu (if present) or borders (if present). If required, this pair can also be pointing to a position outside of the FS window

xp, yp A position inside the FS window denoted by the percentage of its width across (xp, left to right) and down (yp, top to bottom). This can only ever be in the FS window, values 0-100. The **getpos** function may return other values when the mouse pointer is outside the window, but these cannot be used within the positioning functions.

The percentage positions are useful in situations where you might resize the FS window, but the positions of elements you wish to control stay correctly positions proportionally. The other methods would generally be better if you use full screen all the time.

Routine template	Description
`x, y, xr, yr, xp, yp = mouse.getpos()`	Returns the current mouse pointer position in all three ways described above. This function is provided merely so that you can determine what values you need for your own positioning using the **mousemove** function.

`mouse.move(x, y)`
or
`mouse.move(x, y, method)`
	Moves the mouse pointer to the position indicated. If ***method*** is omitted or given as 0, the (x, y) value is assumed to be Windows screen coordinates.
	method = 1 indicates FS window relative coordinates.
	method = 2 indicates FS windows percentage position.

`mouse.click(button)`
or
`mouse.click(button, action)`
	Presses ("clicks") a mouse button.
	button = 0 left, 1 middle, 2 right
	If **action** is omitted or given as 0, a single click, ie. a press and release, is performed. Otherwise:
	action = 1 means hold the button down
	action = 2 means release the button
	If you need a double-click, use two subsequent **action** 0 calls, with a small delay (**ipc.sleep**) between.

`mouse.wheel(n)`	Turns the mouse wheel forward (+n) or back (–n) the specified number of 'clicks'.
`mouse.hwheel(n)`	If so equipped and supported by your version of Windows, pushs (or turns?) and holds the wheel in the horizontal mode, left (–n) or right (+n) the specified number of 'clicks'.

Routine template	Description
`ext.close(handle)` **`ext.close(handle, time)`**	Closes (terminates) a program started using the **ext.run** or **ext.runif** functions. If 'time' is provided (a number of milliseconds) then if the program has not terminated tidily in that time after being sent the polite 'CLOSE' message, it is ruthlessly terminated -- in the same manner as used in Task Manager.
`ext.focus(handle)` **`ext.focus(0)`** ***or*** **`ext.focus()`**	This does its level best to make the identified program the current foreground program, receiving keystrokes and mouse clicks. The program must be one started by **ext.run** or **ext.runif**. Using the function without a parameter, or 0, forces focus back to FS or WideClient.
`bool = ext.isrunning(handle)` **`bool = ext.isrunning("name")`**	Returns TRUE or FALSE depending on whether the identified program is running or not. You can identify the program by the handle returned by **ext.run** or **ext.runif**, or **ext.gethandle**, or by its name. The name can be a full pathname, but only the "name.exe" part is used. The result is true or false.
`ext.kill(handle)`	Forcibly terminates a program started using the **ext.run** or **ext.runif** functions.
`ext.move(handle, x, y, screen)` **`ext.move("name", x, y, screen)`** ***`where 'screen' can be omitted for default or only screen.`***	Moves the top level Window of the named program. The position can be on any attached screen (identified by 'screen', with 0=default, and 1, 2, 3 ... being the numbers shown by Windows monitor settings when using 'identify'). If the 'screen' parameter is omitted, the default screen is assumed. The x and y coordinates are for the top left corner and are in terms of a percentage of screen width and height respectively, and so are independent of actual screen size or resolution. For example 50, 50 would be dead centre. The program can be identified in one of three ways: * Handle: from an **ext.run** or **ext.runif** call. * Name of the process (i.e. "program.exe") * Title of the top-level Window. ("my window")
`ext.position(handle, x, y, cx, cy, screen)` **`ext.position("name", x, y, cx, cy, screen)`** ***`where 'screen' can be omitted for default or only screen.`***	Moves and sizes the top level Window of the named program. The position can be on any attached screen (identified by 'screen', with 0=default, and 1, 2, 3 ... being the numbers shown by Windows monitor settings when using 'identify'). If the 'screen' parameter is omitted, the default screen is assumed. The x and y coordinates are for the top left corner, and the cx, cy specify the width and height, and all four are in terms of a

	percentage of screen width and height respectively, and so are independent of actual screen size or resolution. For example 25,25,50,50 would position the window at half the screen's size in both dimensions and in the dead centre. The program can be identified in one of three ways: * Handle: from **ext.run, ext.runif** or **ext.gethandle.** * Name of the process (i.e. "program.exe") * Title of the top-level Window. ("my window")
`handle = ext.gethandle("name")`	This attempts to get an **ext** library type handle for the specified process or Window title. Check the handle returned. If it is 0 then the attempt failed. The process can be specified as a program name (complete with the .exe or whatever, but without any path detailed), or a Window title. There may of course be no such process running, or there may be more than one. In the latter case a handle to the first one listed by Windows will be attached. The handle can be used in any of the ext library functions which can use a handle, but theyt won't necessarily work. Since we don't actually "own" said process, there are limits to what can be done. You'd need to experiment a little.
`bool = ext.hasfocus()` **`bool = ext.hasfocus(handle)`** **`bool = ext.hasfocus("progname")`** **`Note: FSUIPC4 only`**	This tests whether a specific program currently has the focus (or, more accurately, is the owner of the current foreground window). If no parameter is provided, it is a test for the flight sim having focus. Otherwise you can provide a handle from the ext.run or ext.gethandle functions, or you can provide the executable name for the program being tested.
`handle, error = ext.run("pathname")` **`handle, error = ext.run("pathname", "command line parameters")`** ***`Both can have up to four extra parameters giving options -- see opposite.`***	Runs the program specified by a full pathname, including the EXE or COM filetype (or whatever). If it needs command line parameters provide these as a separate string, as the second parameter. Check the handle returned. If it is 0 then the attempt failed and 'error' will contain the error number returned by Windows, or, if negative, one of these: –2 = memory problem assigning control block –3 = bad string supplied, cannot use You can add up to four more parameters, selecting options from the following: EXT_HIDE to start the program hidden EXT_MIN to start the program minimised EXT_NRML to start it normally (defaulted anyway) EXT_MAX to start the program maximised EXT_LOW run at Low priority (using only idle time) EXT_HIGH run at High priority EXT_CLOSE close this automatically when FS (or WideClient) closes EXT_KILL terminate this forcibly when FS (or WideClient) closes

	EXT_FOCUS Transfer focus to the resulting top window, if possible. (If this is omitted, focus will be returned to the previous owner, probably FS, even though the program will grab it initially unless hidden). Note that not all programs are susceptible to all of these. In particular many define their own initialisation state (hidden, normal or whatever). Some of those may be susceptible to a subsequent **ext.state** change, however. Handles returned by **ext.run** and **ext.runif** are *not* local to the current Lua, but global for this FSUIPC or WideClient session, so can be saved in Lua global variables (see **ipc.get** and **ipc.set**) and that way passed among Lua plug-ins.
`handle, error = ext.shell("pathname")` **`handle, error = ext.shell("pathname", "command line parameters")`** ***`Both can have up to four extra parameters giving options -- see opposite.`***	Uses the Windows Shell to execute or otherwise process the program or file specified by a full pathname. If this needs command line parameters provide these as a separate string, as the second parameter. The normal action carried out will be 'open', bt others can be specified using the EXT_ keywords in the extra parameters, as described below. Of course not all options will be applicable to all file types. Check the handle returned. If it is 0 then the attempt failed and 'error' will contain the error number returned by Windows, or, if negative, one of these: –2 = memory problem assigning control block –3 = bad string supplied, cannot use The handle may or may not be associated with a Window, according to the filetype and action requested, so some of the functions defined for handles will not always work. You can add up to four more parameters, selecting options from the following: EXT_HIDE to start the program hidden EXT_MIN to start the program minimised EXT_NRML to start it normally (defaulted anyway) EXT_MAX to start the program maximised EXT_LOW run at Low priority (using only idle time) EXT_HIGH run at High priority EXT_CLOSE close this automatically when FS (or WideClient) closes EXT_KILL terminate this forcibly when FS (or WideClient) closes EXT_FOCUS Transfer focus to the resulting top window, if possible. (If this is omitted, focus will be returned to the previous owner, probably FS, even though the program will grab it initially unless hidden). The action to be carried out can be specified as one of these: EXT_OPEN To open the file (like double-clicking it). This is the default action. EXT_EDIT Attempt to open an appropriate editor with

	this file loaded. EXT_EXPLORE Open the folder in Explorer EXT_PRINT Attempt to print the file EXT_FIND Attempt to open Explorer to find a file EXT_PROPS Attempt to display the file's properties. Note that not all files are susceptible to all of these. Handles returned by **ext.shell** are *not* local to the current Lua, but global for this FSUIPC or WideClient session, so can be saved in Lua global variables (see **ipc.get** and **ipc.set**) and that way passed among Lua plug-ins.
handle, error = ext.runif(...) ***See ext.run for details***	This is identical to **ext.run**, above, except that the program is not run if it is already running. In other words it's equivalent to performing an **ext.isrunning** before an **ext.run** and bypassing the latter on a TRUE result. If the program is not run because it is already running, the handle is 0 and the error number is –1.
ext.size(handle, cx, cy, screen) **ext.size("name", cx, cy, screen)** ***where 'screen' can be omitted for default or only screen.***	Sizes the top level Window of the named program. The position can be on any attached screen (identified by 'screen', with 0=default, and 1, 2, 3 ... being the numbers shown by Windows monitor settings when using 'identify'). If the 'screen' parameter is omitted, the default screen is assumed. The cx, cy specify the width and height in terms of a percentage of screen width and height respectively, and so are independent of actual screen size or resolution. For example 50,50 would make the window half the screen's size in both dimensions. The program can be identified in one of three ways: * Handle: from **ext.run, ext.runif**, or **ext.gethandle**. * Name of the process (i.e. "program.exe") * Title of the top-level Window. ("my window")
ext.state(handle, state) **ext.state("name", state)**	Changes the current state of the top level window of the program identified by 'handle', started previously by an **ext.run** or **ext.runif** call. The 'state' can be one of: EXT_HIDE, EXT_MIN, EXT_NRML or EXT_MAX The program can be identified in one of three ways: * Handle: from **ext.run, ext.runif**, or **ext.gethandle**. * Name of the process (i.e. "program.exe") * Title of the top-level Window. ("my window") Note that not all programs are susceptible to these commands.
bool = ext.sendkeys(handle, ...) ***where for ... you can have any number of pairs of parameters denoting either:***	This sends keypresses to the window associated with the **ext** handle. To do this is has to transfer focus to that Window, so expect it to popup. Focus will be returned to the previous owner afterwards (e.g. FS). Note that not all handles will be associated with a Window --

virtual keycode, ***shift code*** ***or "string",*** ***shift code***	in particular those returned by **ext.gethandle** and **ext.shell** will often not be. In this case the result returned will be false. Otherwise it will be true, and the keystrokes wil be sent: but if the focus is not on a Window which can accept them, do not expect to see them! The pairs of parameters can be mixed between keycode+shifts and string+shifts as you need. The virtual keycodes are listed on the next page, as are the shift codes. The second of the pair for the last set can be omitted if no shifts are required, as 0 is then assumed. However, if there is more than one pair the intermediate shifts must be given, even if 0.
bool = ext.postkeys(handle, ...)	This *posts* keypresses to the window associated with the **ext** handle. Posting keyboard messages in this way does not require any change of focus, so may be more attractive on the FS PC. However, this method does not work with many programs. Best to try it first. Apart from posting instead of sending, this function is the same as **ext.sendkeys**, so please refer to the extra information for that.
bool = ext.postmessage(handle, **Message, wParam, lParam)**	This is for the programmers among you. It sends the specified Windows message (which you'll need to look up the number for) with the parameters given.

KeyCodes and Shifts

0	Null (+ Alt, Shift etc alone)	73	I	117	F6
8	Backspace	74	J	118	F7
9	Tab	75	K	119	F8
12	NumPad 5 (*NumLock Off*)	76	L	120	F9
13	Enter	77	M	121	F10
16	Shift	78	N	122	F11
17	Control	79	O	123	F12
18	Alt	80	P	124	F13
19	Pause	81	Q	125	F14
20	CapsLock	82	R	126	F15
27	Escape	83	S	127	F16
32	Space bar	84	T	128	F17
33	Page Up	85	U	129	F18
34	Page Down	86	V	130	F19
35	End	87	W	131	F20
36	Home	88	X	132	F21
37	Left arrow	89	Y	133	F22
38	Up arrow	90	Z	134	F23
39	Right arrow	91	Left Windows	135	NumPad Enter (or F24?)
40	Down arrow	92	Right Windows	144	NumLock
44	PrintScreen	93	Apps Menu	145	ScrollLock
45	Insert	96	NumPad 0 (*NumLock ON*)	186	; : Key*
46	Delete	97	NumPad 1 (*NumLock ON*)	187	= + Key*
48	0 on main keyboard	98	NumPad 2 (*NumLock ON*)	188	, < Key*
49	1 on main keyboard	99	NumPad 3 (*NumLock ON*)	189	- _ Key*
50	2 on main keyboard	100	NumPad 4 (*NumLock ON*)	190	. > Key*
51	3 on main keyboard	101	NumPad 5 (*NumLock ON*)	191	/ ? Key*
52	4 on main keyboard	102	NumPad 6 (*NumLock ON*)	192	' @ Key*
53	5 on main keyboard	103	NumPad 7 (*NumLock ON*)	219	[{ Key*
54	6 on main keyboard	104	NumPad 8 (*NumLock ON*)	220	\ | Key*
55	7 on main keyboard	105	NumPad 9 (*NumLock ON*)	221	] } Key*
56	8 on main keyboard	106	NumPad *	222	# ~ Key*
57	9 on main keyboard	107	NumPad +	223	` ¬ ¦ Key*
65	A	109	NumPad -		
66	B	110	NumPad .		
67	C	111	NumPad /		
68	D	112	F1		
69	E	113	F2		
70	F	114	F3		
71	G	115	F4		
72	H	116	F5		

* These keys will vary from keyboard to keyboard. The graphics indicated are those shown on my UK keyboard. It is possible that keys *in the same relative position* on the keyboard will respond similarly, so here is a positional description for those of you without UK keyboards. This list is in left-to-right, top down order, scanning the keyboard:

223	` ¬ ¦	is top left, just left of the main keyboard 1 key
189	- _	is also in the top row, just to the right of the 0 key
187	= +	is to the right of 189
219	[{	is in the 2nd row down, to the right of the alpha keys.
221	]}	is to the right of 219
186	; :	is in the 3rd row down, to the right of the alpha keys.
192	' @	is to the right of 186
222	# ~	is to the right of 192 (tucked in with the Enter key)
220	\ |	is in the 4th row down, to the left of all the alpha keys
188	, <	is also in the 4th row down, to the right of the alpha keys
190	. >	is to the right of 188
191	/ ?	is to the right of 190

The **shifts** value is a combination (add them) of the following values, as needed:

1	Shift
2	Control
4	Alt
8	Tab
16	Alt (*take care with this one—it invokes the Menu*)
32	Windows key (left or right)
64	Apps Menu key (the application key, to the right of the right Windows key)

The COM Library *(also WideClient)*

Note that this now handles both normal COM port serial transfers, whether via a USB serial adapter or direct via a COM port, but also HID (Human Interface Device) transfers, normally related exclusively to USB connections.

Routine template	Description
handle, rd, rdf, wr, initreport = com.openhid(VID, PID, unit, repno) ***Or*** **handle, rd, rdf, wr, initreport = com.openhid("vendor", "product", unit, repno)**	This opens the HID device identified either by the VendorID and ProductID, or by names or partial names identifying the same things. In the "vendor", "product" form the string will be used to match anywhere in the actual device details, so "Widget" will match "American Widgets Inc", as an example. Numerical VIDs and PIDs must match exactly, and are usually given in hexadecimal (0xXXXX). These can be found from the Device Manager details in Windows, or by using extra logging in FSUIPC or WideClient (see note at the end of this section). The unit parameter identifies one of several identical units, counting from 0. The unit numbers are assigned in the order in which Windows enumerates them, which probably depends on how they are plugged in. This parameter, along with the following one, can be omitted to default to unit 0 (okay for 1 unit), and Report #1 (the usual default). The repno value identifies the default input report number to be requested for the initial state. As you can see, there are several returned values. You do not have to use them all, of course. The handle returned will be zero if the device could not be opened. The 'rd', 'rdf' and 'wr' values provide the device-defined fixed sizes of input reports, SetFeature data, and output reports, respectively. You should use these values to define the size of data being read and written. The 'initreport' value is a string of bytes providing the first input report after opening. This gives you the initial state – usually switch positions and the like. For joystick type HID devices the additional data processing facilities embodied in the following functions should be used. The first three of these operate on Input Reports (length 'rd'): **com.gethidvalue** **com.gethidbuttons** **com.testhidbutton** **com.gethidcount**
handle = com.open("port", speed, handshake)	This opens the serial comms port named "port" (e.g. "COM1"), with settings: Speed = baudrate, e.g. 115200 for VRInsight devices, often 4800 or 9600 for GPSs. Handshake defines the protocol for controlling the flow: 0 = none 1 = RTS / DTR line levels 2 = XON / XOFF 3 = Both of the above The port is always opened in 8-bit no parity mode. 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.

com.close(handle)	This simply closes the port represented by the given Handle. It should always be used before the Lua program terminates.
n = com.test(handle)	Returns the number of bytes of data available to be read on the port represented by the given Handle.
str, n = com.read(handle,max) **str, n = com.read(handle,max,min)** **str, n = com.read(handle,max,min, term)**	Reads up to 'max' bytes from the port, returning them as a string in 'str' with the number actually read returned in 'n'. 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. If you specify -1 as the minimum then the terminating character ('term') must be seen before the function returns a non-zero result, unless of course the 'max' size is reached first. The 'term' parameter specifies an ASCII value (0 to 255) which is to be treated as a terminator for each block. This character is included in the returned count and string. **Note** that you can use the **event** library function, **event.com** to perform reads and call your Lua back when there is data to process. This can be more efficient and tidier than a program which uses continuous loops to scan the input.
str, n = com.readlast(handle, len) **str, n, discards = com.readlast(handle, len)**	This is the same as **com.read**, above, but with a fixed block size assumed (given by 'len'), and with all currently available blocks discarded except the last, which is supplied. The number of discarded blocks is returned as a third result should it be wanted. This function might be useful in polling situations where the rate at which data arrives might exceed the polling rate or capabilities of the Lua system. HID joysticks scanning is a prime example. Rather than process older records and get a larger unwanted lag that is necessary it enables an efficient way of only processing the most recently received state.
n = com.write(handle, "string") **n = com.write(handle, "string", len)**	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. The returned value gives the number of bytes actually sent (or at least, placed in the buffer).
n = com.writefeature(handle, "string", len)	Writes the string to the HID device via the "SetFeature" call. The length should equal the 'wrf' value returned by the com.openhid function. The returned value gives the number of bytes actually sent (or at least, placed in the buffer).
n = com.gethidvalue(handle, "axis", str) **n1, n2, ... = com.gethidvalue(handle, "axis", str)** ***(up to 16 results)***	If the open HID device is a joystick type, this can be used to read any analogue (axis) or POV value it might return. The 'str' parameter refers to the data, of length 'rd' (see **com.openhid**), returned by **com.read** or **com.readlast**, The "axis" parameter is one of the following axis names: "X", "Y", "Z", "R" (or "RZ"), "U" (or "RX"), "V" (or "RY"), "POV" (or "HAT"), "Rudder", "Slider", "Dial", "Wheel", or "Throttle". A HID device might support any number of each of these. The com library supports up to 16 axes of each of these types. Alternatively you can give a one-byte "usage" code for non-standard analogue values, ones not even supported by DirectInput. Use the **com.gethidcount** function to retrieve the numbers of any of the named axis types and the maximum value it can return.

N = com.gethidbuttoncount(handle) ***[NOT FSUIPC3]***	Returns the number of buttons on the optn HID device.
X = com.gethidbuttons(handle, str) **X1, X2, ... = com.gethidbuttons(handle, str)** ***(up to 8 results)***	If the open HID device is a joystick type, this can be used to read the state of all the buttons it might support, up to a maximum of 256 buttons. The 'str' parameter refers to the data, of length 'rd' (see **com.openhid**), returned by **com.read** or **com.readlast**, The values returned each contain up to 32 button states, buttons 0-31 in the first 32-63 in the second, and so on.
X = com.testhidbutton(handle, btn, str)	If the open HID device is a joystick type, this can be used to test the state of 256 buttons, numbered 0-255. The 'str' parameter refers to the data, of length 'rd' (see **com.openhid**), returned by **com.read** or **com.readlast**, The return is **true** or **false**.
n, max = com.gethidcount(handle, "axis")	This returns information about the analogue value (axis) named. 'n' gives the number of such axes supported (which may be 0) and 'max' gives the maximum value they can return. The axis names which can be used are "X", "Y", "Z", "R" (or "RZ"), "U" (or "RX"), "V" (or "RY"), "POV" (or "HAT"), "Rudder", "Slider", "Dial", "Wheel", or "Throttle". The com library supports up to 16 of each of these axis types.

NOTE: Logging HID devices in FSUIPC.

You can get a list of all HID devices connected to your PC, as well as their comings and goings, by adding these lines to the FSUIPC.INI or FSUIPC4.INI [General] section:

Debug=Please
LogExtras=512

To do the same in WideClient, change the Log= parameter in the [user] section to “Log=HID”.

The Event Library *(also WideClient, but not for all events, as noted)*

Events allow you to build plug-ins which rather than running continuously in a loop in order to interrogate things can be set to stay loaded but dormant waiting for those things to occur. Almost anything which can be done in a continuously running loop can be done more tidily and pleasingly using events instead.

Events rely on you specifying two things: what it is you want to monitor, and which pre-defined function, in your program (or called up by **Require**) you want to run when the monitored event occurs. There is no specific restriction on how many different events you can monitor in one program, nor how many times you can trap the same event for different functions. But note that, whilst FSUIPC does keep track of separate events, it does not queue multiple identical events. If a button is pressed 20 times before you process it, you only see it once. Therefore if you are monitoring things which can happen repetitively you will need to keep your processing short enough if you hope to catch them all.

The function name provided as a string in the Lua event function calls can now be functions in tables. This enables functions in Modules, brought in by the **require** function, to be used for event processing, because Modules so enabled provide tables of functions (and other values) for access in the current program. The format of the function reference string must be <table>.<function>, so if the Module is named (or equated to) "M", say, then function "fn" inside it would be referred to as "M.fn" in the event function. (The alternative form "M[fn]" is not allowed). The facility is actually extended to handle tables within tables, to no set limit other than the entire string name must be less than 64 characters (between the "").

A Lua plug-in with any events being monitored stays running (or rather dormant, awaiting those events) until it either explicitly terminates (via ipc.exit), fails through some error, or cancels its last outstanding event monitor.

Routine template	Description
`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)`** ***`[Not WideClient]`***	Executes the named function (named as a string, "..."), which must be defined before this line, when a given joystick button changes. "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. 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 *not* the Virtual Buttons. 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. 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) Special button number 40 can be specified to indicate that the event is required on any of the POV states 32-39, also according to the given 'downu' parmeter. 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.com(handle, max, min, term, "function-name") **event.com(handle, max, min, "function-name")** **event.com(handle, max, "function-name")** ***Your processing function:*** **function-name(handle, datastring, length)**	This event works with the **com** library—described earlier—providing a way of continuing to receive data using an event-driven program rather than a continuous loop doing **com.read** calls. Effectively the **event.com** call sets up FSUIPC do do the reads for you, passing received data to your function when available. The parameters **max**, **min**, and **term** are used as described in the section on the **com.read** function, earlier. The data read is passed back as the **datastring** parameter to the named function. Your program doesn't need to perform any reads itself, thought there's nothing stopping it—and it may be wise if there's more expected
event.control(controlnum, "function-name") **event.control(controlnum, delta, "function-name")** ***Your processing function:*** **function-name(controlnum, param)** ***[Not WideClient]***	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.flag("function-name") **event.flag(flag, "function-name")** ***Your processing function:*** **function-name(flag)** ***[Not WideClient]***	Executes the named function whenever one of this plug-ins Lua flags is changed (by one of the LuaSet, LuaClear or LuaToggle controls). If no flag number (0–256) is provided, any of the 256 changing will trigger the event. Otherwise only the selected flag will do so. The flag number provided to the named function is the one which changed to trigger the event.
event.gfd("function-name") **event.gfd(model, "function-name")** **event.gfd(model, unit, "function-name")** ***Your processing function:*** **function-name(model, unit)** ***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.*** ***[Not WideClient]***	This executes the named function whenever an input occurs on the identified GoFlight device(s). 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. 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. Unit numbers start from 0 and are assigned in ascending order by the Go-Flight interface, GFDev.DLL, which must be accessible. 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.
event.intercept(offset, "type", "function-name") **event.intercept(offset, "STR", length, "function-name")**	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

Your processing function: **function-name(offset, value)** ***[Not WideClient]***	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. 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. 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 it must define the string length (max 256). 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.key(keycode, shifts, "function-name") **event.key(keycode, shifts, downup, "function-name")** ***Your processing function:*** **function-name(keycode, shifts, downup)** ***[Not WideClient]***	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) To receive repeats the "downup" parameter must be specified, with '4' added to the documented values, so that: 4 pressed +repeats 5 pressed +repeats 6 same as 2, only release 7 pressed, + repeats, +release 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. The "downup" parameter in the called function will be 3 for a repeated press, 1 for an initial press and 0 for a release.

event.Lvar("lvarname", interval, "function-name")

Your processing function:

function-name(varname, value)

[Not WideClient]

This monitors the value of the given local gauge variable ("L:Var") at the given interval, and calls the function when it a change in the value is detected.

The L:var name can be in the form "L:name" or just "name", but when our declared function is called the name provided is without it's L: part. It is provided in the call so that the same function can be used for multiple variables if required.

The interval is in milliseconds and has a minimum value of 100. it is not optional.

The function will never be called if there is no matching L:Var, so you need not worry about checking the correct aircraft or gauge is loaded. No error will occur.

---

event.mousehoriz("function-name")

event.mousehoriz(method, "function-name")

Your processing function:

function-name(x, y, move, flags)

[FSUIPC4 only]

This causes the function to be called whenever the horizontal (left/right) movement of the mouse wheel is detected, on those mouse types which are so equipped.

The 'method' parameter, if given, specifies the type of x,y coordinates to be supplied in the call. It can be 0, 1 or 2, with 0 defaulted, for screen coordinates, FS window coordinates, or FS window proportion (0-100%, so 50,50 = centre). This is also the choice available in the **mouse** library.

The 'move' parameter supplied is +ve for right, -ve for left, with the vaue indicating how much (in 'clicks' or Windows 'delta' units, usually +1 or -1).

The flags parameter is a set of bits indicating these (columns are bit number, decimal value, meaning when set):

0	1	Left button is down
1	2	Right button is down
2	4	Shift is pressed
3	8	Ctrl is pressed
4	16	Middle button is pressed
5	32	eXtra button 1 is pressed
6	64	eXtra button 2 is pressed
15	32768	Mouse is outside FS window

---

event.mousehoriztrap("function-name")

event.mousehoriztrap(method, "function-name")

Your processing function:

function-name(x, y, move, flags)

[FSUIPC4 only]

This is identical to the **event.mousehoriz** function above except that the action of the mouse is trapped -- it is *not* passed on to FS or other programs further down the chain.

If you use this facility and you want to pass on the action in some circumstances you would need to use the **mouse.hwheel** function to pass it down.

---

event.mouseleft("function-name")

event.mouseleft(method, "function-name")

Your processing function:

function-name(x, y, move, flags)

[FSUIPC4 only]

This causes the function to be called whenever the left mouse button is pressed or released whilst within the FS window which has the mouse focus.

The 'method' parameter is used as described for **event.mousehoriz**, above. The 'move' parameter is always 0. The flags parameter is a set of bits indicating these (columns are bit number, decimal value, meaning when set):

0	1	Left button is down
1	2	Right button is down
2	4	Shift is pressed
3	8	Ctrl is pressed
4	16	Middle button is pressed
5	32	eXtra button 1 is pressed
6	64	eXtra button 2 is pressed

event.mouselefttrap("function-name") **event.mouselefttrap(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This is identical to the **event.mouseleft** function above except that the action of the mouse is trapped -- it is *not* passed on to FS or other programs further down the chain. If you use this facility and you want to pass on the action in some circumstances you would need to use the **mouse.click** function to pass it down.
event.mousemiddle("function-name") **event.mousemiddle(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This causes the function to be called whenever the middle mouse button is pressed or released whilst within the FS window which has the mouse focus. The parameters and results are all as described for **event.mouseleft**, above.
event.mousemiddletrap("function-name") **event.mousemiddletrap(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This is identical to the **event.mousemiddle**function above except that the action of the mouse is trapped -- it is *not* passed on to FS or other programs further down the chain. If you use this facility and you want to pass on the action in some circumstances you would need to use the **mouse.click** function to pass it down.
event.mousemove("function-name") **event.mousemove(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This causes the function to be called whenever the mouse is moved whilst within the FS window which has the mouse focus. The parameters and results are all as described for **event.mouseleft**, above.
event.mousemovetrap("function-name") **event.mousemovetrap(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This is identical to the **event.mousemove** function above except that the action of the mouse is trapped -- it is *not* passed on to FS or other programs further down the chain. If you use this facility and you want to pass on the action in some circumstances you would need to use the **mouse.move** function to pass it down.
event.mouseright("function-name") **event.mouseright(method, "function-name")**	This causes the function to be called whenever the right mouse button is pressed or released whilst within the FS window which has the mouse focus. The parameters and results are all as described for **event.mouseleft**, above.

Your processing function: **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	
event.mouserighttrap("function-name") **event.mouserighttrap(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This is identical to the **event.mouseright** function above except that the action of the mouse is trapped -- it is *not* passed on to FS or other programs further down the chain. If you use this facility and you want to pass on the action in some circumstances you would need to use the **mouse.click** function to pass it down.
event.mousewheel("function-name") **event.mousewheel(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This causes the function to be called whenever the forward/backward movement of the mouse wheel is detected, on those mouse types which are so equipped. The parameters and results are all as described for **event.mousehoriz**, above, except that if the wheel is moved very fast you can receive values for the 'move' parameter in excess or 1 and -1, for the number of 'clicks' of movement.
event.mousewheeltrap("function-name") **event.mousewheeltrap(method, "function-name")** **Your processing function:** **function-name(x, y, move, flags)** ***[FSUIPC4 only]***	This is identical to the **event.mousewheel** function above except that the action of the mouse is trapped -- it is *not* passed on to FS or other programs further down the chain. If you use this facility and you want to pass on the action in some circumstances you would need to use the **mouse.wheel** function to pass it down.
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 function is also executed initially, when the plugin is first run, in order to initialise things. This saves using an explicit call to do the same. 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 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 *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.offsetmask(offset, mask, "type", "function-name") **Your processing function:** **function-name(offset, value)** ***[Not FSUIPC3]***	Executes the named function (named as a string, "..."), which must be defined before this line, when the specified FSUIPC offset changes only in the bits set in the mask value -- i.e. the value given by logic.And(offsetvalue, mask) changes. The function is also executed initially, when the plugin is first run, in order to initialise things. This saves using an explicit call to do the same. 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 The difference between signed and unsigned values here has no effect . The values are treated as a collection of 8, 16 or 32 bits. 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) masked value in that offset. This will be a Lua number, with only the masked part present (i.e. the result after "Anding" the original with the mask).
event.param("function-name") ***Your processing function:*** **function-name(param)** ***[Not WideClient]***	The calls the declared function when the ipcPARAM variable for this plug-in is changed *externally* to the plug-n code. That is by use of the **LuaValue** <name-of-plugin> control. If this is assigned to an Axis in FSUIPC axis assignments it provides the value from that axis, otherwise it is the parameter given in the button or key assignment. The value 'param' provided as the parameter to the called function is also stored in the ipcPARAM variable. Note that if the LuaValue control for this plug-in is assigned to multiple sources there is no way to distinguish how the parameter value arose.
event.sim(event-type, "function-name") ***Your processing function:*** **function-name(event-type)**	This executes the named function when a specific type of event occurs in the Simulator. The types currently available are: CLOSE: the flight simulator is closing down (or, in the case of WideClient, WideClient is closing down or the simulator is closing down (you can't tell which). This gives the plug-in a chance to tidy things up before exiting tidily with an **ipc.exit** call. Note that if the tidy-up involves logging or sending stuff to a device, you may need an **ipc.sleep** call before the **exit** in order to allow those things to clear. FLIGHTLOAD: a flight has just been loaded. FLIGHTSAVE: a flight has just been saved AIRCRAFTCHANGE: the aircraft has been changed

	ANY: any of the above—use the parameter to determine which. Note that the ANY method is unlikely to catch all events. Only one event can be signalled at a time, but loading a flight often means an aircraft change too. ANY would only catch one of those. Similarly a flight save often occurs just before FS closes. Only one of those might be seen. Therefore, if catching everything is crucial, it is best to use the individual events.
event.textmenu(type, "function-name" ***Your processing function:*** **Function-name(type, colour, scroll, delay, id, n, msgs)** ***[WideClient with FSX SP2 or ACC, or P3D v2 and later, only]***	This event instructs FSUIPC4 (registered versions 4.924 and later) to intercept text and menu calls into SimConnect and send the details on to WideClient. Currently it is only implemented for FSX Acceleration and SP2, and in P3D since version 2. This does *not* apply to calls to SimConnect made through FSUIPC4 itself *nor* those from other internal add-ons (DLLs and Gauges), only those made directly by external applications. (I've not found the right place to hook for internal calls). This facility may need enabling in FSUIPC by setting this parameter in the INI file [General] section: InterceptTextMenu=Yes This is the default value. To remove the menu from the screen use InterceptTextMenu=Full At present it is not possible to suppress the display of menus on screen in FSX. Well, it can be done, but then SimConnect never sends the user selection back to the application. You can have the menu made smaller and moved top left, maybe out of the way a bit. In P3D the menu is fully removed from the P3D screen. The 'type' is 1 for a text message, and 2 for a menu. If you want to get both set the 'type' to 0 in the event function. The colour applies only to text messages and is one of these values: 0=black, 1=white, 2=red, 3=green, 4=blue, 5=yellow, 6=magenta, 7=cyan. The scroll parameter is true or false, but again only applies to text messages, not menus. The delay parameter is 0 for 'display till replaced, or answered', or otherwise is the number of seconds before the display should be cleared. The ID is a numeric value specified by the program which originated the request. It may or may not be unique, but it might be useful to filter specific messages. 'n' is the number of messages in this request. This is always 0 or 1 for a message type -- 0 would mean clear the message. For a menu it can be 0 to clear the menu, but otherwise it is the number of menu items PLUS TWO. The first two are the menu title and menu request for action. The remaining messages will be the choices. 'msgs' is a table containing n messages. Access them by msgs[1[through to msgs[n]. An example Lua is provided which combines the "Radar Contact" example in the **wnd** library section of this document with a handler for these texts and menu facilities, all on the

	same screen. (see "TextMenu.lua")
event.timer(msecs, "function-name") ***Your processing function:*** **function-name(time)**	This simply calls your function regularly, at the interval specified (in milliseconds). You can use this in event-driven plug-ins for polling, flashing lights, etc, rather than resorting to loops. Note that each Lua plug-in is restricted to one timer. If you specify another it replaces the previous one. If you need different intervals for different things, set the timer for the lowest common factor and use the time, in milliseconds, passed to your function as parameter to determine the intervals. The time provided is NOT the same as the one returned by the **ipc.elapsedtime** function.
event.vriread(handle, "function-name") ***Your processing function:*** **function-name(handle, "data")** ***[Not WideClient]***	This is an extension to the **com** library, described earlier, specifically for use with VRInsight devices. 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. 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**".
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. A Lua plug-in with any events being monitored stays running (or rather dormant, awaiting those events) until it either explicitly terminates (via ipc.exit), fails through some error, or cancels its last outstanding event monitor.
event.terminate("function-name")	This is called when the Lua thread is being forcibly terminated for any reason. the Lua program then has a further 5 milliseconds to do any essential tidying.

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 SOUND Library *(also WideClient)*

Routine template	Description
sound.adjust(ref) **sound.adjust(ref, vol)** **sound.adjust(ref, vol, posn)**	Adjusts the volume and/or position of a playing sound, defined by the 'ref' value returned from the play or playloop functions. The vol and posn values are as described below for the play function. If posn is omitted 0 is assumed (centre forward), and if they are both omitted full volume (100) centre forward position is assumed (i.e., like a "reset to norm").
str = sound.device(devnum)	This returns the device name string for the sound device known by the given number. Note that 0 means "default device" and is always the same as number 1. The device names are listed against their numbers in the [Sounds] section of the INI file.
sound.path("path-to-sounds")	The default path for wave files is the Sound subfolder in FS (or, for WideClient, a Sound sub-folder in the WideClient.exe folder). This can be changed by editing the Path parameter in the INI file, or, for the sounds called by this Lua program only, by setting a temporary path via this function. The path will be within the current default path if it does not contain a drive reference, such as "c:".
ref = sound.play("name-of-wave") **ref = sound.play("name-of-wave", devnum)** **ref = sound.play("name-of-wave", devnum, vol)** **ref = sound.play("name-of-wave", devnum, vol, posn)**	Plays the wave with the given name. This must be a "wav" file, but you can omit the ".wav" part. If the device number is omitted, the default is used (device 0). The vol (volume) is a % value between 0% and 100%. This isn't the complete range from silence ot max, but 0 is very quiet. This defaults to 100. The posn (position) is a value in degrees from 0 to 359 representing the circle around the PC, with 0 being forward centre. This is approximated as well as possible for the sound device being used. For a stereo setup, the circe collapses to a left-right spectrum. A position given as -1 will play on all speakers. Unlike FS sounds, this sound will play regardless of whether FS (or WideClient) has the current Focus. If you want the sound to be suppressed when FS/Wideclient does not have the focus, specify the volume parameter as a negative value 9e.g. -100 for max volume, but local to the program). The "ref" value returned is a number which can be used subsequently in the stop, adjust and query functions. Note that sounds are "global" in the sense that they don't stop when the Lua program ends or is killed..
ref = sound.playloop ***... with the same parameters and variations as for "sound.play" above***	This is identical to sound.play except that the wave file is looped—forever ,or until you stop it with a sound.stop call. In this case, it is *very* important that you note that sounds are "global" in the sense that they don't stop when the Lua program ends or is killed.
bool = sound.query(ref)	This returns true if the sound is playing, false if it is not or if there is no such sound.
sound.stop(ref)	This strops the sound indicated by the reference, if it is still playing. Otherwise it does nothing.

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
GFDIO	GF-DIO Digital Input/Output board (needs GFDev 1.20.0.1 or later)
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
GFMESM	GF-MESM Multi Engine Start Module
GFP8	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
GFWP6	GF-WP6 Annunciator Panel

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

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 configred 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 / extiguished.
5. The program then processes inputs forever (until Killed), displaying and logging the results.

Routine template	Description
gfd.BlankAll()	Blanks all digital displays (if possible) and switches all indicator lights off.
n = gfd.Buttons() ***or, for devices with more than 32 'buttons' (switch inputs really) as on the GFDIO:*** **n1, n2 = gfd.Buttons()**	Returns the state of all buttons supplied by the last call to **gfd.GetValues**, described below. The states of up to 32 or 64 buttons or switches are provided, and these are represented by one bit each in the returned values. Bit 0 (2^0, worth 1) is the first button, bit 1 (2^1, worth 2) is the second, and so on. When two resluts are read the second contains buttons 32 to 63.
gfd.ClearLight(model, unit, id) ***or, for multi-coloured units:*** **gfd.ClearLight(model, unit, id, 1)**	This simply turns off the indicator light identified by the id number, on the specified model and unit. For multi-coloured indicators other than the GF-WP6 you should use the second form. For example, the gear LEDs on the LGT would have 'ids' 1, 2 and 3 using the second form.
n = gfd.Dial(id)	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 nmubers are possible for faster turns -- except for the RP48, whose dials only ever seem to return -1, 0 or +1.
n = gfd.GetName(model)	This returns the string name of the device of type 'model'. For instance, the name of the model type GFMCPPRO is "GFMCPPRO".
n = gfd.GetNumDevices(model)	This returns the number of connected devices of type 'model'.
gfd.GetValues(model, id)	This obtains all of the current input values from the identified device. These values are subsequently accessible using these separate functions: **gfd.Buttons()** **gfd.Selector(id)** **gfd.Dial(id)** **gfd.TestButton(id)** **gfd.Lever(id)**
n = gfd.Lever(id)	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.
n = gfd.ReadLights(model, unit)	This reads the state of all the indicators on the specified module, if this is actually possible on this module. Indicators are numbered 0 to 31, with 0 being 2^0, worth 1 and so on. If there is an error the value returned will be negative, as follows: -1 = unknown model -2 = not a connected unit -3 = indicator reads not supported on this module Note that some versions of the LGT2 and RP48 modules (notably the "mouse edition" of the latter) have firmware deficiencies which cause the return here to be always zero.

n = gfd.Selector(id)	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.
gfd.SetBright(model, unit, n)	This sets the unit's display and indicator brightness, n=0 being off and n=15 being brightest.
gfd.SetDisplay(model, unit, id, "display text")	This attempts to write the given text (truncated if necessary) to the display identified by the id number, on the specified model and unit.
gfd.SetColour(model, unit, id, n) ***or*** **gfd.SetColour(model, unit, id, red, green, blue)** **(Colour can be spelled "Color" instead if you wish).**	This is currently only supported for the GF-WP6, and pre-sets the colour of indicator number 'id' either to the pre-determined colour 'n' (0-7, see below), ***or*** to the colour represented by separate red green and blue values, each one being a value from 0 - 100. Note that this function does NOT actually address the hardware at all, but merely *presets* the colour to be used by the SetLight or SetLights functions described below. Pre-determined colour values are: 0 black, 1 white, 2 red, 3 green, 4 blue, 5 magenta, 6 yellow, 7 cyan
gfd.SetLight(model, unit, id) ***or, for multi-coloured units:*** **gfd.SetLight(model, unit, id, val)**	This simply turns on the indicator light identified by the id number, on the specified model and unit. For multi-coloured indicators other than the GF-WP6 you should use the second form. For example, the gear LEDs on the LGT would have 'ids' 1, 2 and 3 using the second form, with "val" set as follows: 1 = red, 2 = green, 3 = amber For the GF-WP6 you can pre-set the colour using **gfd.SetColour**, above.
gfd.SetLights(model, unit, on, off)	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. Indicators 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**.
n = gfd.TestButton(id)	Returns ***true*** or ***false*** depending on the button/switch setting (0–63) in the input data supplied by the last call to **gfd.GetValues**, described above.

The Wnd Library *(WideClient only, added at version 6.953)*

Routine template	Description
w = wnd.open("title") ***or*** **w = wnd.open("title", maxlines)** ***or*** **w = wnd.open("title", x, y, w, h)** ***or*** **w = wnd.open("title", maxlines, x, y, w, h)** ***or*** **w = wnd.open("title", WND_FIXED, x, y, w, h)**	This creates the window and returns a window handle. The window is named by the title and this appears in the title bar if one is drawn. Except for the WND_FIXED case, the window has a title bar, a sizing border, and min/max/close buttons. The size and position can therefore be set by the user, and these values are saved, according to the "title" in the [ExtWindow] section of WideClient's INI file, and restored next time. The 'maxlines' parameter determines how many text lines are sent to the window before it scrolls. The scrolling is automatic -- no scroll bars are provided. The initial size and position can be specified in screen pixels by the x, y, w and h parameters. the x, y position refers to the top left corner. The WND_FIXED option is different. there are no sizing borders and no title bar. The user cannot change its size or position. In all cases you can use the **ext.position** function, referring to the "title", to change the position, size and even screen, using easier screen-relative measures. Multiple windows can be opened and controlled by one or more Lua plug-ins, and unless closed explicitly persist after the Lua program terminates. If you want other plug-ins to use the same Windows you need to pass the window handle on via the global variable facilities (**ipc.set** and **ipc.get**)
wnd.close(w)	Closes the window w. Windows created by WideClient are also closed when WideClient closes.
bool = wnd.clear(w)	Clears window w. Returns 'true' if okay, or 'false' if window w doesn't exist.
bool = wnd.backcol(w, 0xRGB)	Sets the backgrond colour of window w. Returns 'true' if okay, or 'false' if window w doesn't exist. The colour is given as 3 hexadecimal digits, 0x000 to 0xFFF, with the digits representing the amount of Red, Green and Blue in 16 steps. Thus 0x000 is black and 0xFFF is white. The background colour applies to the whole window, but can be changed at any time.
bool = wnd.textcol(w, 0xRGB)	Sets the text colour of window w. Returns 'true' if okay, or 'false' if window w doesn't exist. The colour is given as 3 hexadecimal digits, 0x000 to 0xFFF, with the digits representing the amount of Red, Green and Blue in 16 steps. Thus 0x000 is black and 0xFFF is white. The text colour applies text drawn from then on, until changed. You can mix text colours on screen.
bool = wnd.font(w, face, size)	Sets the text font details. Returns 'true' if okay, or

or **bool = wnd.font(w, face, size, options)**	'false' if window w doesn't exist. The 'face' is one of WND_ARIAL default WND_TIMES Times New Roman WND_COURIER Courier New, fixed space The size is usually given in points, ranging from 4.0 to 300.0, default 12.0, but you can have it computed automatically to provide n lines between the current line and the end of the current Window. For the latter use –n for the size, where n is the number of lines. The options can be any mix of these (add them together): WND_BOLD, WND_ITALIC, WND_UNDER, WND_STRIKE. The font details apply for text drawn from then until the font is changed again, so any mix can be used in one window.
bool = wnd.text(w, "text") ***or*** **bool = wnd.text(w, line, "text")**	Draws text in window w. Returns 'true' if okay, or 'false' if window w doesn't exist. The optional line number specifies which line, of those already drawn, should be replaced by this text. Unless this is the current line, you must have such a line already drawn if this is used. Line numbers start from 1. Control codes found in the Text are replaced by spaces except for new lines, returns and tabs. New lines and returns give rise to multilined text, whilst tabs are replaced by three spaces each. So, the text can be multi-lined, though if used this doesn't really work well with the 'line number' facility. Multiple lined text is best used on its own with a **wnd.clear** being used before display each time it changes.
wnd.bitmap(w, "pathfilename")	Draws a bitmap, loaded from the specified file, and stretched or compressed to exactly fit the window specified by 'w'. *Remember to use \\ for every \ in the full path -- Lua demands this, otherwise the \ just makes the next character a special control, or ignored.* The WXRadar.lua plug-in is an example of this.

EXAMPLE

This small Lua plug-in, saved as, say, "showtext.lua" in the same folder as WideClient, is used on my own system instead of ShowText.exe, to show the Radar Contact menu:

```
w = wnd.open("Radar Contact", WND_FIXED, 1280,0,1024,768)
wnd.backcol(w, 0x000)
wnd.textcol(w, 0x0f0)
wnd.font(w, WND_ARIAL, -10, WND_BOLD)

function update(off, val)
        wnd.clear(w)
        wnd.text(w, val)
end

event.offset(0x3380, "STR", "update")
```

The Display Library *(WideClient only, added at version 6.895)*

Routine template	Description
`handle = display.create("title", entries, xpos, ypos)`	This creates the dialogue display, consisting of a number of read-only edit boxes (the number given by '`entries`', which must be in the range 1 to 16). The width of the display is fixed, but the position on screen may be set to the `xpos` and `ypos` valuse, which determine the top left corner in screen pixels.
`display.clear(handle)`	This simply clears all the edit fields in the specified display.
`display.show(handle, entry, "text")` ***Or*** **`display.show(handle, entry, type)`**	Displays the given "`text`" string in the field numbered '`entry`' (counting from 1, the top-most, to 16 or the maximum in the created display. The alternative call, with '`type`' instead of a string is used for special pre-coded displays built into WideClient. The only ones currently available are: `RC1` displays a decode of Radar Contact 4's waypoint line from its menu, if it is currently available. Otherwise blank. `RC2` displays a decode of Radar Contact 4's runway line from its menu, if it is currently available. Otherwise blank.
`display.close(handle)`	Closes the display. The hande is not valid after this call.

Here's an example of such a display, this one created using the supplied example 'MyDisplay.lua":