

Lua Plug-ins for VRInsight devices

Using VRInsight's SerialFP2 and FSUIPC together with re-programming using plug-ins.

Introduction

This document follows on from the Appendix in your FSUIPC Advanced Users Guide entitled "**Handling VRInsight devices in FSUIPC**". If you have not read and acted upon that first, please do so before moving on to the facilities detailed here.

This is not going to teach you how to completely program VRInsight devices in Lua, using the "com" library facilities. You most certainly could do that if you so wished, but it is beyond the scope of this package.

What this will do is show how some parts of what the devices do can be changed a little to more suit your needs. To do this we shall concentrate on two examples (both of which are useful now, but both of which have apparently been promised as future SerialFP2 improvements by VRInsight):

VRI_SetMach: a plug-in to provide Mach speed control setting and readout from the MCP Combi device. This one is well-tested on my own MCP Combi.

VRI_SetBaro: a plug-in to provide the option to show and adjust the altimeter BARO setting on the M-Panel device in millibars (hectoPascals) as an alternative to inches of mercury. At the time of publication this has not been tested at all on the device (so look out for updates).

However, before the examples, some reference information about parameter setups in FSUIPC's INI file.

FSUIPC INI file settings

If you've already set up your system for using FSUIPC with VRInsight devices, as you should have, you will have a [VRInsight] section in your INI file already. This will be something like this:

```
[VRInsight]
1=COM5,COM2
```

where COM5 is your VRInsight device connection, and COM2 is one of a pair of Virtual Ports. You COM numbers will be different, of course, and if you have multiple VRI devices there will no doubt be more entries in this section, one parameter line for each device.

To extend this facility to use Lua plug-ins easily and automatically you add a new section to the INI file for each device you want so handled. This is the general format for this section:

```
[VRInsight.<devicetype>]
Lua=<name of plugin>
DriverReads=<option>
DriverWrites=<option>
```

where the last two are optional, and the parts in < > are:

<devicetype> is one of these (the current known VRInsight device names as shown in SerialFP2):

FMER	MCP Combi
CDU	CDU panel
CDU2	CDU II panel
COM	Instrument Radio Stack
FltMn	<i>Unknown future device</i>
GPS5	GPS panel
JetPt	Jet Pit
MFD	<i>Unknown future device</i>
MPanl	M Panel
ProPt	Prop Pit
uJPit	Micro Jet Pit
uPPit	Micro Prop Pit

<name of plugin> gives the name of the Lua plug-in to be automatically loaded if this device is detected (and for each such detection). Omit the '.lua' part.

<option> is one of these:

- NO** to block data to ("reads") or from ("writes") the SerialFP2 driver from / to the device. Note that the Lua plug-in can read from and write to the device in any case. This option merely cuts off the SerialFP2 driver from all data arriving ("reads") or, more likely ("writes") stops it writing anything to the device.
- YES** to allow the driver reads or writes through to the device as normal. The plug-in can still read and write both This is the default setting if the parameter line is omitted.
- FILTER** will be found to be the most useful. It allows you to make a list of those commands and responses which should not be allowed to go between the driver and the device.

It is the FILTER facility which allows your plug-in to take control of some aspects of the device actions without interfering with it all. If you set this option then you add a list of filters afterwards. These are in this form:

WrFilter.n=<filter string>

for Write filters (that is, writes TO the device FROM SerialFP2, the driver, and

RdFilter.n=<filter string>

for Read filters (that is, reads FROM the device TO SerialFP2, the driver.

As already mentioned, but worth emphasising, the filters act between driver and device. The device reads are all accessible to the Lua plug-in, as are the driver writes.

The number, 'n', starts at 1 for each of the groups (Rd and Wr) and increases normally, just being a listing reference. You can have a maximum of 32 filters of each of the two types.

The strings listed in the filters represent those being sent back and forth between SerialFP2 and the device. The best way for you to know what they are and understand what they do is to use the FSUIPC Log facility (**Log -> Custom** and enter value **x4**). The log will show the exact command and response strings being used.

However, there are some special facilities in the filter string specifications:

The character * at the end means "any characters at all, or none".

The character ? at the end means "any number of decimal digits, only, until the end"

The characters ?+ at the end means "any number of decimal digits, only, until the end *or* a + or -

Other special filtering options may be added if the need arises. Those are the few found to be needed for the examples we shall look at. in detail below.

The Lua plug-in

The Lua program which is automatically loaded via the parameters in the INI file VRI device sections has to interact with at least the device and quite possibly also the SerialFP2 driver. It does this using the **com** library functions provided by FSUIPC.

Since you may have more than one VRI device of the same type, the actual COM ports being used should not be built into the Lua program. Instead, it is supplied with three pieces of information, as pre-defined Lua string variables:

VRImodel the short model device name as in the list above (FMER etc)

VRIdevice the string name of the COM port to which the device is attached

VRIdriver the string name of the virtual port which the SerialFP2 driver is ultimately connected to.

These will always be set if the plugin is started automatically via the FSUIPC INI settings we discussed, but will be "nil" if the user starts the plug-in in any other way. You can use this fact whilst testing before having it automatically loaded and run—the difficulty of allowing the latter before your Lua programming is fully working is that it then needs an FS restart to get it running again after you encounter a problem. By making it work in a free-standing way too you can do all the testing and debugging in one FS session.

This precautionary way of programming is demonstrated in the first example which will now be fully explained.

VR_SetMach: An example for the MCP Combi Device

This describes in detail the **VR_SetMach.lua** plug-in included with the Lua package and acts as a useful example of the mixing of SerialFP2 and FSUIPC handling.

If you have an MCP Combi device you can try this example now. The extra section you need in your FSUIPC INI file looks like this:

```
[VRInsight.FMER]
Lua=VR_SetMach
DriverWrites=Filter
WrFilter.1=SPD?
WrFilter.2=SPDOF
WrFilter.3=SPDON
DriverReads=Filter
RdFilter.1=SPD?+
```

The driver commands we are stopping are all those concerned with displaying the Speed (SPDnnn) and turning the "*" on and off showing the speed mode enabled or not (SPDON and SPDOF). These are filtered off because we handle them in the plug-in.

The device sending SPDnnn+ or SPNnnn-, or the fast versions SPDnnn++ and SPDnnn-- are stopped from going to the driver, because they always set the IAS and we want to be able to set the Mach instead.

You will see these commands and responses going back and forth in the FSUIPC log, if you've enabled the special VRI logging. This is how you find out what to intercept and how to do it yourself. However, I am also attempting to put together a complete listing of SerialFP2 output commands for each device. That will follow in due course.

The VR_SetMach.lua code can now be examined, step by step:

```
if VRImodel == nil then
    -- Set known ports if testing with Lua not loaded automatically
    VRIdriver = "COM2"
    VRIdevice = "COM5"
end
```

All this part is doing is allowing me to run the plug-in separately, without the automatic loading in FSUIPC INI. I assigned one keypress to "Lua VR_SetMach" and another to "LuaKill VR_SetMach" so I could run tests and keep changing it till it worked properly (yes, things don't always work properly first time!).

When the plug-in is loaded automatically the values of VRImodel, VRIdriver and VRIdevice will already be set correctly for the program.

```
speed = 115200 -- is this the same for all VRI devices? (YES!)
handshake = 0  -- No handshake
minsize = 8
maxsize = 8    -- VRI seems to use fixed length blocks of 8 bytes
```

These are standard settings for VRI devices. You can't really change the first two in any case when we are opening ports already opened by FSUIPC, but it is best to have the correct values in the calls we make

```
dev = com.open(VRIdevice, speed, handshake)
```

In this case we only need to talk to the device. We have no need to hear what the driver says or to send anything to it, so we don't open the VRIdriver port.

Skip now to the end:

```
setmchmode(0x7e4, ipc.readUD(0x7e4))
setspdmode(0x7dc, ipc.readUD(0x7dc))
```

```

event.offset(0x7e2, "UW", "setspeed")
event.offset(0x7e8, "UD", "setmach")
event.offset(0x7e4, "UD", "setmchmode")
event.offset(0x7dc, "UD", "setspdmode")
event.VRIread(dev, "setmachtofs")

```

Here's where you will need the list of FSUIPC offsets, the references to internal FS values will usually need those. You'll need to download the FSUIPC SDK. For FS2004 and before the best reference for offsets is the list in the "FSUIPC for Programmers" document—bypass the chatty stuff at the start and go straight to the tabulated list. You can use search to find things in there. For FSX and ESP you should use the FSUIPC7 Offsets Status document, also in the SDK.

Back to the code. The first two lines in the extract above are merely testing the current IAS/Mach state and setting things up accordingly. The functions they call are those which are also called when those offsets change -- 07E4 is the A/P Mach mode switch and 07DC is the A/P speed mode switch.

The sequence of event.offset function calls ensure that one of our routines is called on a change of A/P mode and when either the speed value (in offset 07E2) or the Mach value (in offset 07E8) change. Those routines in turn take care of updating the display, as here, from the "setmach" function:

```

val = (val / 655.36) + 0.5 -- rounded
str = string.format("SPD%03d", val)
com.write(dev, str, 8)

```

Note the conversion from FS units (65536 times the Mach value) to 100ths, then formatting the result into the command "SPDnnn" which displays the value. This makes Mach 0.82 show as 082. (There's no way to get a decimal point).

The really new entry in this part of the program, though, is that "event.VRIread". This calls the function "setmachtofs" every time anything arrives in FSUIPC from the device, whether it is going to be filtered or not. So that function must check whether it is one we should handle:

```

-- need to check only for SPDnnn+/- and send speed or mach to FS
speed = tonumber(string.match(str, "SPD(%d%d%d)"))
if speed ~= nil then

```

This is using one of those really clever little Lua string library functions to not only match "SPDnnn ..." responses, ignoring the + or - characters on the end, but also extract the value of the 3 digit numerical part. Then it can update the speed or mach value offset as appropriate for the current mode.

Okay. I think that's all that really needs explaining here. Have a look though it, try it. Note that there's extra debugging lines still left in, those "ipc.log" function calls to log whatever is happening. For streamlining those should be removed now, but I left them in for your use.

VRI_SetBaro: An example for the M Panel Device

This describes in outline only the **VRI_SetBaro.lua** plug-in included with the Lua package. If you have an M Panel device you can try this example now. The extra section you need in your FSUIPC INI file looks like this:

```

[VRInsight.MPanel]
Lua=VRI_SetBaro
DriverWrites=Filter
WrFilter.1=BAR?
DriverReads=Filter
RdFilter.1=BAR?+

```

If you got through the last example this one should be easy for you. As it says in the comment in the code, it uses the local Lua flag 0 as the "use milibars" selector. You will need to program a button or keypress to "LuaToggle VRI_SetBaro" with parameter 0, and you can then toggle between the two modes.

The important parts of the program are all here, in three little lumps. First the part called when the FS BARO value changes—it's in offset 0330 and is in 1/16ths of a millibar:

```
function setbarodisplay(off,val)
  -- val is BARO setting in 16ths of millibar
  if ipc.testflag(0) then
    -- need whole number of millibars
    val = (val + 8) / 16 -- Note rounding by addition of 8
  else
    -- need to convert to 100ths of inch
    val = ((val * 2992) / (1013.2 * 16)) + 0.5 -- note rounding
  end
  str = string.format("BAR%04d", val)
  com.write(dev, str, 8)
end
```

Then there's the part called when the dial on the M-Panel is turned. It provides "BARnnnnn+" or "BARnnnnn-":

```
function setbarovalue(handle, str)
  -- need to check only for BARnnnnn+/-
  baro = tonumber(string.match(str, "BAR(%d%d%d%d)"))
  if baro ~= nil then
    if ipc.testflag(0) then
      -- if flag set, using millibars
      baro = baro * 16 -- FS units are 16ths
    else
      -- using inches
      baro = ((baro * 1013.2 * 16)/2992)+ 0.5 -- note rounding
    end
    ipc.writeUW(0x330, baro)
  end
end
```

And the small function called when the flag (0) is changed, so that the display can be toggled between inches and mb (hPa):

```
function flagchanged(n) -- n is the flag number, but we know that is 0
  setbarodisplay(0x330, ipc.readUW(0x330))
end
```

Finally, right at the end there's all-important part initialising the display and setting the event "traps" for the flag changes, FS offset changes, and the dial being turned on the device:

```
flagchanged(0) -- ensure initial value set
event.flag(0, "flagchanged")
event.offset(0x330, "UW", "setbarodisplay")
event.VRIread(dev, "setbarovalue")
```

That's it! Have fun!