IOCards hardware

An interesting site with lots of information and pictures and schema's of Opencockpits IOCards hardware, such as pin numbering, can be found here. I will only describe a few particular pieces of Opencockpits IOCards hardware.

It is recommended to use the USB expansion card and to connect a Master Card to this USB expansion card and NOT to a LPT port. The USB solution works flawlessly and USB is common practice these days, besides lots of new PC's do not even have a LPT port.

Don't forget to connect +5V and Gnd to the Master card. The easiest way to do this is to take one of the spare Molex connectors of your PC power supply, take a red (+5V) and a black (GND) wire from that 4 pin Molex.

A cockpit has lots of rotary switches (1 x n) and rotary encoders. I strongly recommend to use gray type encoders.

Also find her some information about the pins at the In- and Output connection boxes and some info about the USB LCD card and the USB Outputs card.

Gray Type Rotary Encoder

You can connect a gray type rotary encoder (to be precise: 2-BIT GRAY TYPE ROTARY ENCODER WITH 1/4 CYCLE PER DETENT and 16 DETENTS/REVOLUTION) directly to two inputs of the Master Card. One does not need the Opencockpits Encoder card.

The encoder has three pins. Two go to inputs of the MC and the gnd to the common gnd of the same group where you connected the two inputs from the encoder. The inputs should be consecutive.

Gray-code means that only one bit of the 2 bits output changes at a time. The sequence is: 00 (zero), 01 (one), 11 (three), 10 (two) and back to 00 (zero).

The sequences "zero-one-three-two" is called a cycle. 1/4 cycle per detent means for gray-code encoders, that for each detent (step) just one of the two bits changes and that there are no bit changes in between detents. A condition for use with IOCards Master card is that the gray type encoder changes only a 1/4 cycle per detent!

Watch out:

There are also gray type encoders that change a full cycle per detent or 1/2 cycle per detent, these types will not work correctly when connected directly to the Master Card.

You should take encoders with 16 detents per revolution (360 degrees), no more.

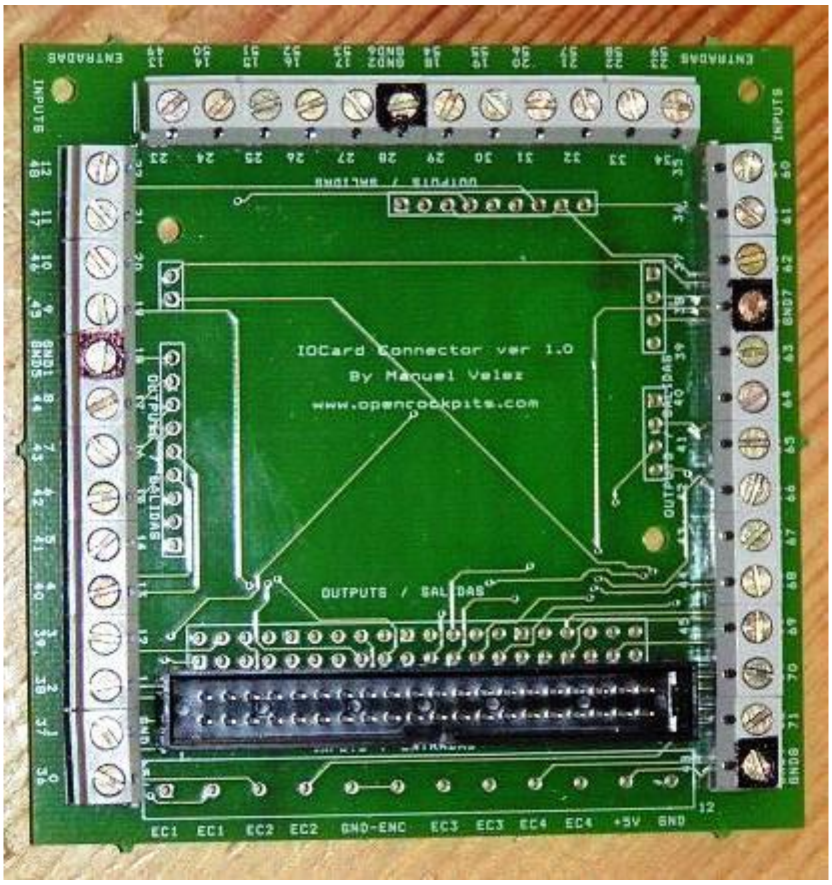
An encoder that is gray type and will work is CTS corporation type 288. A CTS type 288 encoder is available in the Opencockpits shop. Examples of how to program a gray type encoder in SIOCcan be found at the howto page.

In- and Output Connection Boxes

Opencockpits also sells Input- and Output connection boxes. You can connect your in- and outputs to the Master card without having to solder.

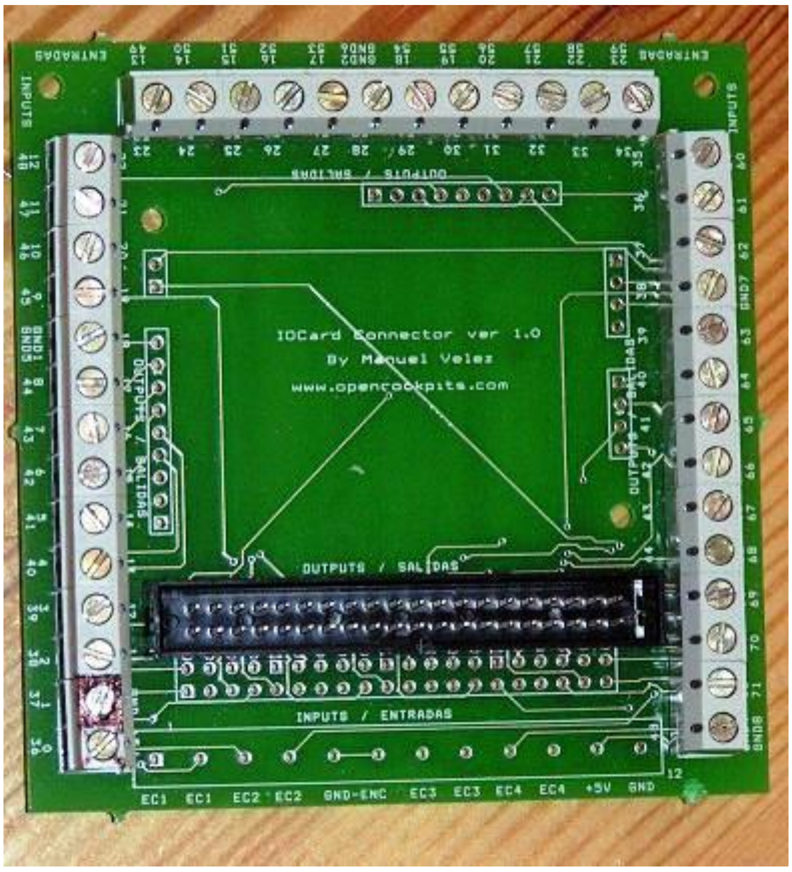
The connections card is bi-sexual... Depending on the placement of the 40 pins connector it's used as 4 * (9 + ground) input card or as 38 outputs with a ground.

Here is the Input version:



The grounds of each group are indicated in this picture with a black spot. The pins are well indicated. Note that 9 input pins share 1 ground, the pin ordering indicated is the logical ordering. If you connect a gray type rotary you should take pins n and n+1 within one and the same group.

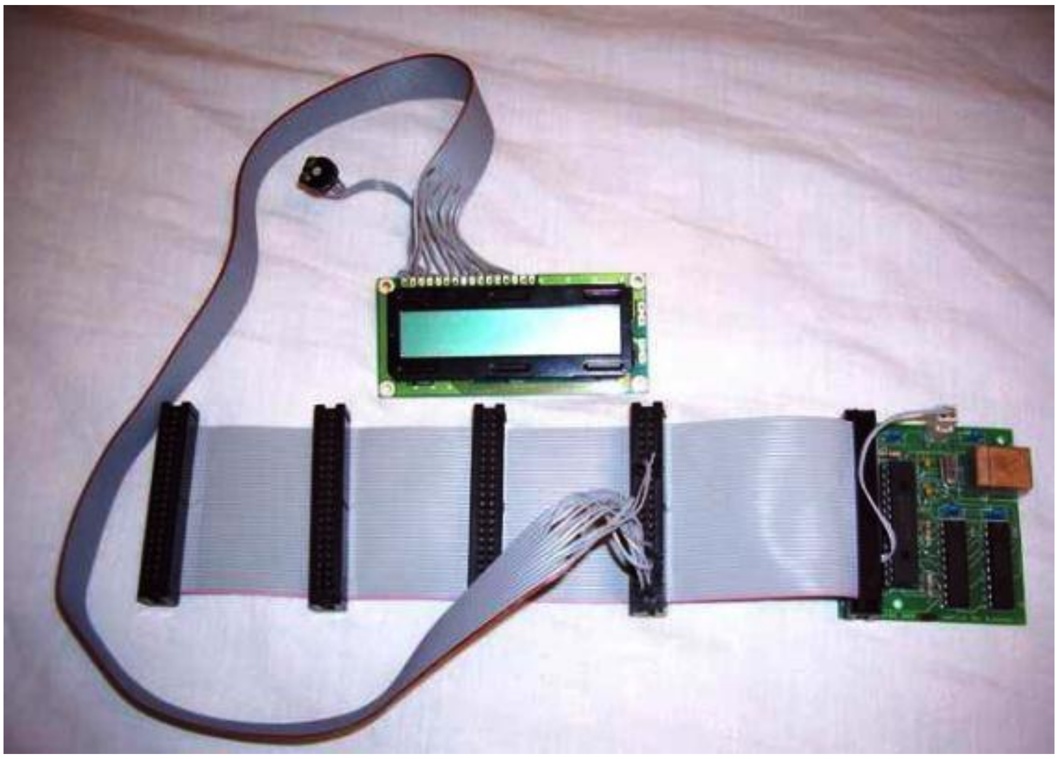
And here is the somewhat more difficult to interpret Outputs version:



Note that pin 1/37 is the GROUND (second from the bottom left, see the black spot). Pin 0/36 is +5V (not really needed) and the other 38 pins are the Outputs.

USB LCD Card

With the Opencockpits USB LCD card you can control up to 4 LCD's.

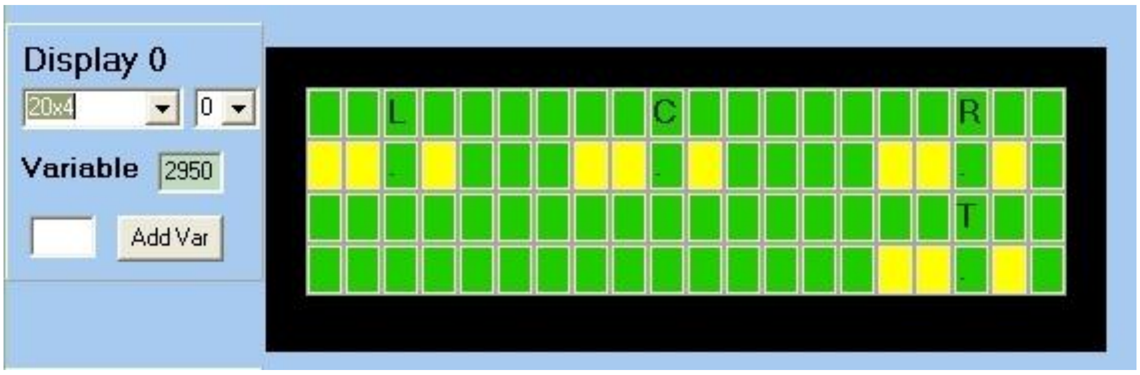


Contrary to other Opencockpits USB cards/modules (like the Outputs card, or the MCP, or a Master card), this card is controlled by a separate software package.This program, called IOClcd.exe, acts as a IOCP client to SIOC. In our SIOC script we can use SIOC variables that will be displayed via IOClcd.exe.

To make this more clear I'll show you an example: the display of the Left, Center, Right and Total fuel tank values of the Level-D Boeing 767 (FS9 version).

First we use the IOClcd_config.exe program to define a so-called virtual screen at a 20x4 character display. You may define up to 4 virtual screens per display. With a SIOC variable you can switch between virtual screens, so you can switch for instance between a Fuel screen and a Engine parameters screen.

In this example I'll only define the Fuel screen:



Display 0

20x4 0

Variable 2950

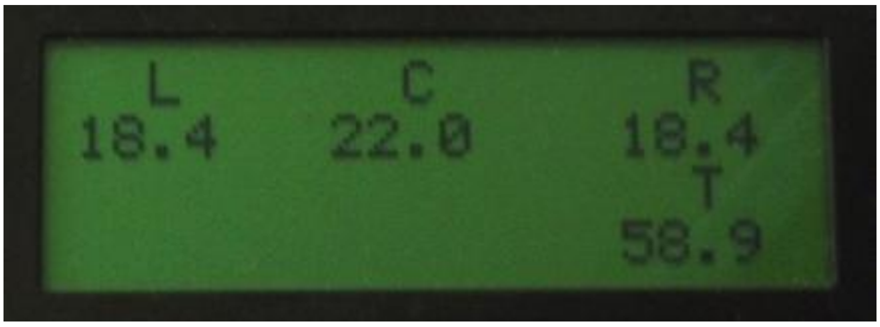
Var 2950 is used as virtual screen selector, if we assign it he value 0, then this fuel screen will show up. Certain positions are given fixed text values, like the L, the C and the R, and the decimal points.

The yellow blocks are the fields where the actual fuel tank values will be shown. I have defined them two positions wide for the integer values and 1 position wide for the decimal values. Each field has a SIOC Variable attached, it starts with 2951 for the integer value of the Left fuel tank and it ends with 2958 for the decimal value of the Total Fuel value. All these definitions are entered with the IOClcd_config.exe program and at the end we save this configuration in a file called fuel767LCD_fs9.lcd.

The SIOC script used to compute and display the Fuel Tank Values of the Level-D Boeing 767 can be found in this zip file, together with all the other related configuration files. The interesting part of the SIOC script is the definition of the variables that are implicitly "linked" to the 20x4 display via the IOClcd.exe program:

```
 Var 2950 value 0 // virtual display selector

 Var 2951 name FuelLHLCD // FuelL High part at LCD (2 digits)
 Var 2952 name FuelLLLCD // FuelL Low (decimal) part at LCD (1
digit)
 Var 2953 name FuelCHLCD // FuelC High part at LCD (2 digits)
 Var 2954 name FuelCLLCD // FuelC Low (decimal) part at LCD (1
digit)
 Var 2955 name FuelRHLCD // FuelR High part at LCD (2 digits)
 Var 2956 name FuelRLLCD // FuelR Low (decimal) part at LCD (1
digit)
 Var 2957 name FuelTHLCD // FuelT High part at LCD (2 digits)
 Var 2958 name FuelTLLCD // FuelT Low (decimal) part at LCD (1
digit)
```

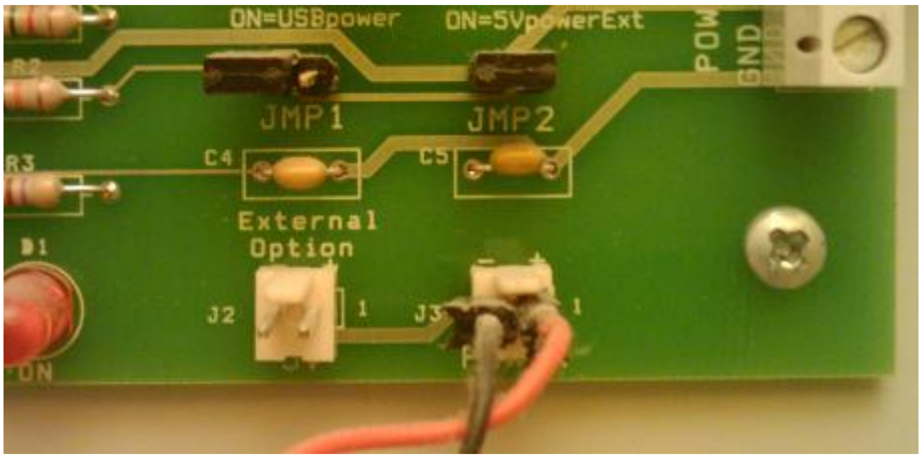
If we run this SIOC script (and FS9, and IOClcd.exe) the LCD display will show a screen like this:



USB Outputs Card

Many people have asked me how to configure and power the USB Outputs card. Note that I use the 64 outputs of this card to drive led's.

Here a picture of my jumper settings:



JMP1 is open, JMP2 is closed. At connector J3 I have connected a GND wire (black) and a +5V wire (red). The +5V power comes from my flightsim PC's power supply.

IMPORTANT:

The bad news: The 64 outputs at this card share a common +5V. This is different from the 38 outputs at a Master card that share a common GND. Take good notice of this difference when you connect your led's!

The good news: in SIOC you do not have to take these differences into account, writing a '1' to an output in SIOC will always make the led lit.

SIOC Software

SIOC is a programming language/system that is used to program a flight simulator based on Opencockpits IOCards. Basically this means connecting a switch in your home build cockpit to the aircraft add-on in MS Flight simulator and, the other way around, using information in that add-on to control led's and digits in your cockpit.

The documentation of Opencockpits about SIOC is huge and overwhelming, and not always correctly translated into English... There is often confusion about how to use SIOC. Therefore I will try to explain the basics.

To start with one should forget about running at multiple PC's, forget about the IOCP protocol/variables and forget about 'servers'. One only needs:

a text editor such as Notepad, to write your script(-s);

the config-sioc.exe program to compile your script(-s) into an executable file (.ssi);

the sioc.exe program to run your program.

Furthermore, the siocmonitor.exe program (since SIOC version 4.0!) is very handy to check whether your hardware is correctly working or not. SIOC, config-sioc and siocmonitor are all installed by the sioc40.zip package (free).

A presentation (from Opencockpits) can be downloaded here.

SIOC: The Golden Rule

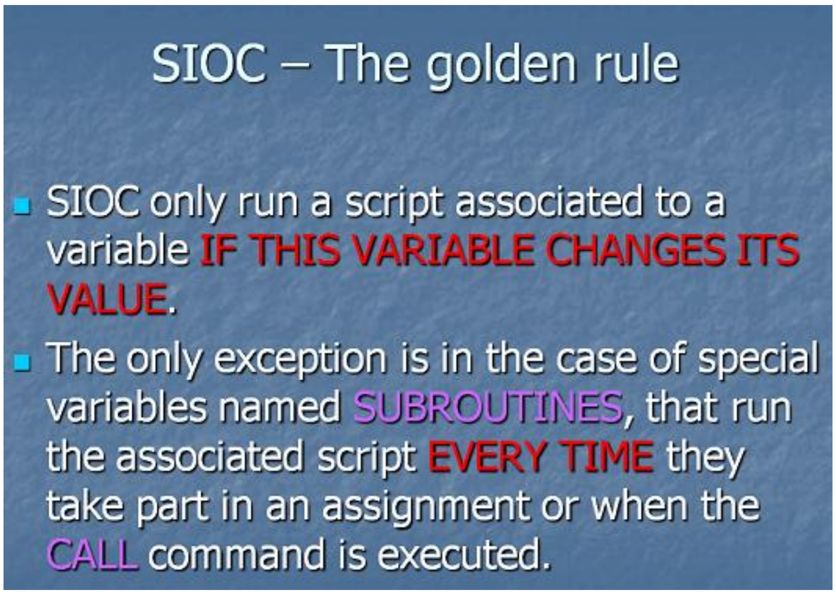
Very important to understand is that SIOC is not a 'normal' procedural programming language with a clear start and a clear end (like Pascal, C, C++, Java, ...), but an event driven scripting language ...

Ok... (uh, yes...), but what does that mean???

Well, SIOC works with Variable declarations. Variables represent connections to the outside world, such as FSUIPC offsets (your link with your aircraft in FS9/FSX), push buttons, switches, rotary switches, rotary encoders, led's, 7-segment display, servo's, and so on, or they represent internal program states or subroutines.

SIOC monitors the Variables connected to the outside world say 50 times per second in order to detect any changes in values. If a Variable gets another value, then (and only then!) the code in the body of the Variable, enclosed by curly brackets { }, is executed. This can cause another Variable to change value and so you may get a chain of events. Ideal for simulating the logics of an aircraft!

It is very important to understand this behavior of SIOC. One might call it the SIOC Golden Rule:



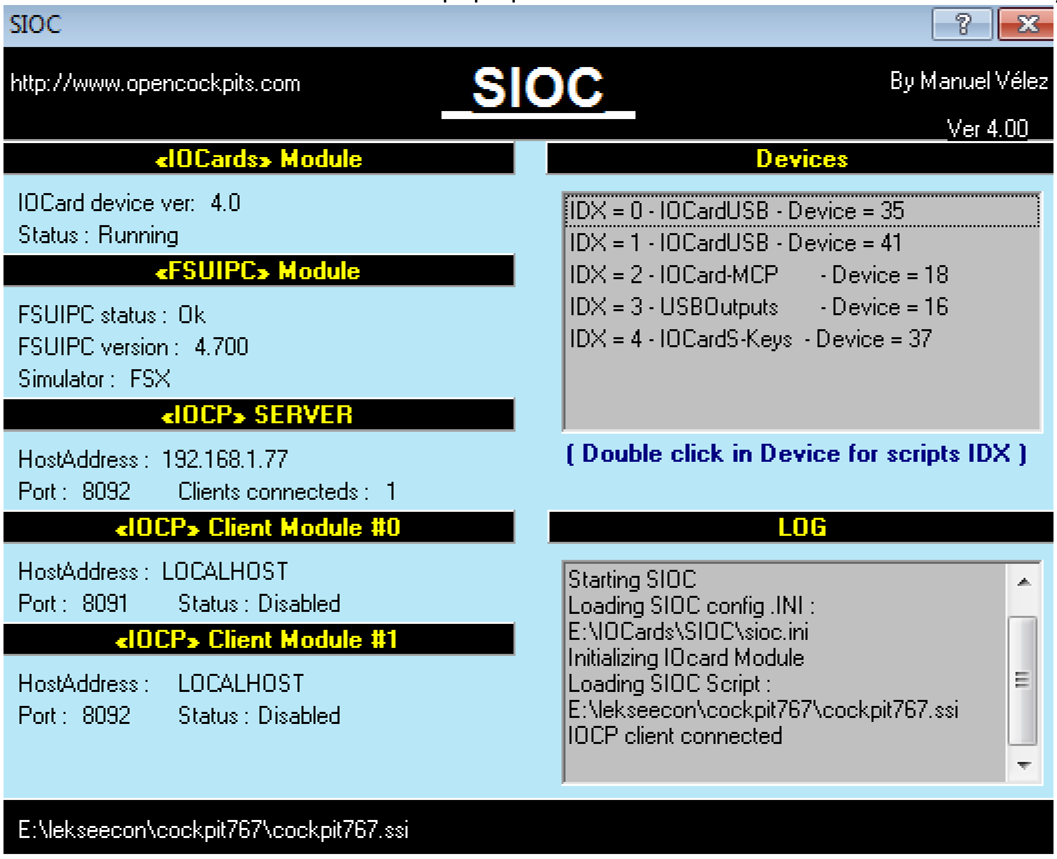
SIOC – The golden rule

- SIOC only run a script associated to a variable IF THIS VARIABLE CHANGES ITS VALUE.
- The only exception is in the case of special variables named SUBROUTINES, that run the associated script EVERY TIME they take part in an assignment or when the CALL command is executed.

[sheet from a power point presentation by Opencockpits]

SIOC window

If you start sioc.exe the SIOC window will pop up. Below is a screen shot of that window at my PC.



```
SIOC
http://www.opencockpits.com        SIOC        By Manuel Vélez
                                                    Ver 4.00

<IOCards> Module                       Devices
IOCard device ver:  4.0                IDX = 0 - IOCardUSB - Device = 35
Status : Running                       IDX = 1 - IOCardUSB - Device = 41
<FSUIPC> Module                        IDX = 2 - IOCard-MCP    - Device = 18
FSUIPC status :  Ok                    IDX = 3 - USBOutputs    - Device = 16
FSUIPC version :  4.700                IDX = 4 - IOCardS-Keys  - Device = 37
Simulator :  FSX
<IOCP> SERVER
HostAddress :  192.168.1.77            [ Double click in Device for scripts IDX ]
Port :  8092       Clients connecteds :  1
<IOCP> Client Module #0                LOG
HostAddress :  LOCALHOST               Starting SIOC
Port :  8091       Status : Disabled   Loading SIOC config .INI :
<IOCP> Client Module #1                E:\IOCards\SIOC\sioc.ini
HostAddress :   LOCALHOST              Initializing IOcard Module
Port :  8092       Status : Disabled   Loading SIOC Script :
                                       E:\lekseecon\cockpit767\cockpit767.ssi
                                       IOCP client connected

E:\lekseecon\cockpit767\cockpit767.ssi
```

The <IOCards> Module sub window should indicate Running.

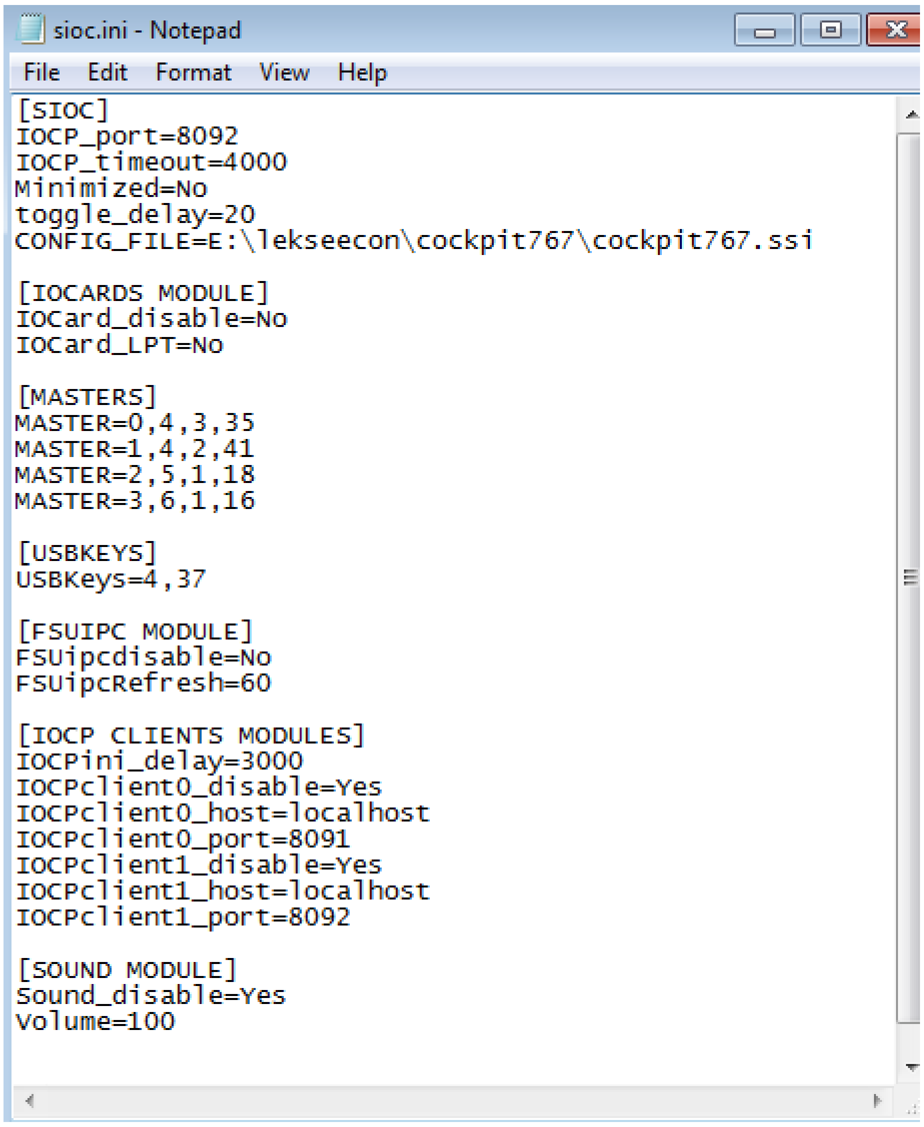
The <FSUIPC> Module sub window gives information about use and status of FSUIPC. This depends on your setting in sioc.ini (see below); whether you have enabled FSUIPC or not. Note if you are not using FSUIPC offsets in your SIOC script you better disable FSUIPC.

The <IOCP> Client Modules #0 and #1. For almost all users these should be Disabled, don't pay much attention to it, just take the settings in your sioc.ini as I have (see below).

The LOG sub window shows that SIOC has loaded and is running the script 'cockpit767.ssi'.

The Devices sub window tells you which Opencockpits USB cards SIOC has detected. At my PC it has detected my two USB expansion cards, my USB MCP module, my IOCards-Keys card and my USB Outputs card.

The Line "`IDX = 0 - IOCardUSB - Device = 35`" gives important information: it means that in my computer the (physical) Device number of this USB expansion card has got number 35. In the sioc.ini file you can specify the logical device number for the use of this device in SIOC. This logical number is used as parameter for the DEVICE attribute for SIOC variables. The link between the logical device number used in SIOC and the physical device number at my PC is defined in the sioc.ini file in the SIOC folder. Below a screen shot of my sioc.ini:



```
[SIOC]
IOCP_port=8092
IOCP_timeout=4000
Minimized=No
toggle_delay=20
CONFIG_FILE=E:\lekseecon\cockpit767\cockpit767.ssi

[IOCARDS MODULE]
IOCard_disable=No
IOCard_LPT=No

[MASTERS]
MASTER=0,4,3,35
MASTER=1,4,2,41
MASTER=2,5,1,18
MASTER=3,6,1,16

[USBKEYS]
USBKeys=4,37

[FSUIPC MODULE]
FSUipcdisable=No
FSUipcRefresh=60

[IOCP CLIENTS MODULES]
IOCPini_delay=3000
IOCPclient0_disable=Yes
IOCPclient0_host=localhost
IOCPclient0_port=8091
IOCPclient1_disable=Yes
IOCPclient1_host=localhost
IOCPclient1_port=8092

[SOUND MODULE]
Sound_disable=Yes
Volume=100
```

Let me explain the first line in the [MASTERS] block:

```
MASTER=0,4,3,35
```

The first number is the logical device number (0).

The second number defines the type of the USB device (4=USB expansion card)

The third number defines the number of Master Cards connected to this USB device (3)

The fourth number defines the physical IDX number of this USB device at my PC (35).

You see? By doing this we have linked `DEVICE 0` to physical `IDX` device 35 (as you can see in the SIOC main window). Be aware that if you unplug your USB card and plug it into another USB port you will get another number then 35 ... and you will have to update your sioc.ini. But the advantage of this concept is that you do not have to change your SIOC scripts.

More info about the `DEVICE` attribute in SIOC is given in this how-to example.

To get your settings right just start SIOC and write down the `IDX` number(-s) detected. Then close SIOC, update your sioc.ini and start again.

Note that there is also a Config_SiocIni.exe tool available at Opencockpits (in downloads, SIOC), that may help you to get your settings right.

The <IOCP> SERVER sub window gives information about the Host address (192.168.1.77) and the Port (8092) over which an IOCP server that is build in SIOC, is available. My lekseecon program connects to this IOCP server to exchange information about the Variables in the SIOC program. In the LOG sub window you will find the message "IOCP Client Connected" as soon as lekseecon connects to SIOC. The actual number of Clients connected is indicated in the IOCP SERVER sub window.

config-sioc.exe

My preferred way of developing SIOC programs is to use a simple text editor, such as Microsoft's NotePad, to write SIOC scripts in text format. This approach gives us a lot of flexibility in editing, copying and pasting text. Much better then the Graphical User Interface (GUI) in config-sioc.exe.

The config-sioc.exe program is then used to compile the SIOC script in text format into a format that can be executed (run) by sioc.exe.

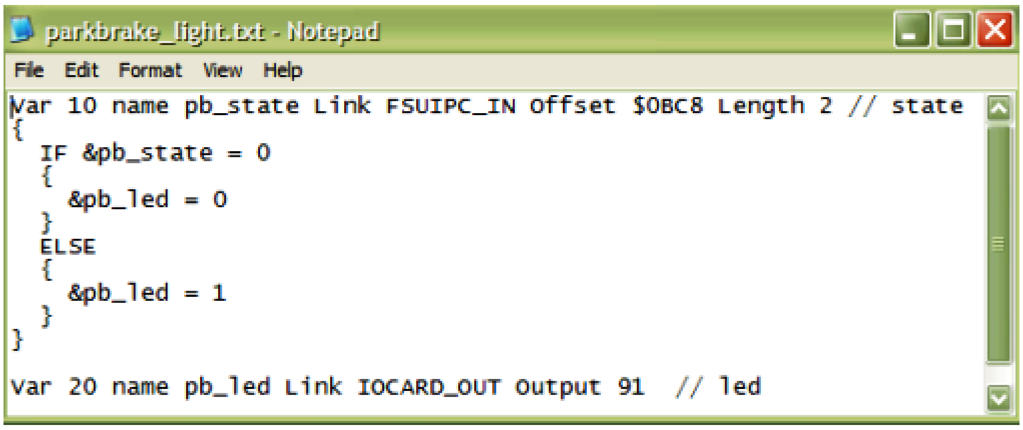
A .txt file may contain several SIOC declarations and statements. Config_sioc.exe compiles one text file, or several text files at once, into a single .ssi file. I will describe both compilation methods here.

Note that sioc.exe can only run one .ssi file, indicated by the CONFIG_FILE parameter in sioc.ini.

Compile a single Text File

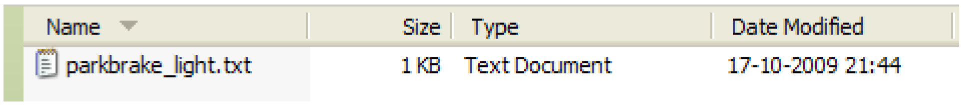
In this example I will show the steps needed to write and compile howto example read from FSUIPC offset as a single text file.

First we use the Notepad editor to write the code. It is a good practice to assign symbolic names ('pb_state' and 'pb_led' in this example) for variable numbers. Your code is easier to read and it is less prone to errors when you change variable numbers. Also note that using comma's and leading zeroes in the variable declaration, like is used by the 'Export TXT' feature of config-sioc.exe (see further down), is not necessary.



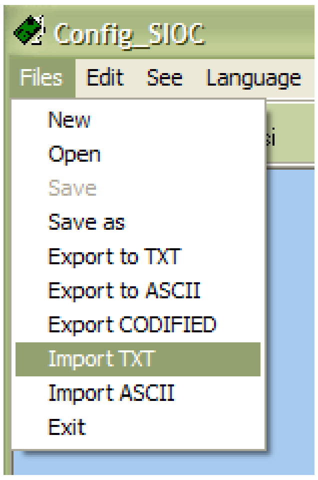
```
Var 10 name pb_state Link FSUIPC_IN Offset $0BC8 Length 2 // state
{
  IF &pb_state = 0
  {
    &pb_led = 0
  }
  ELSE
  {
    &pb_led = 1
  }
}

Var 20 name pb_led Link IOCARD_OUT Output 91   // led
```

This file is saved as parkbrake_light.txt

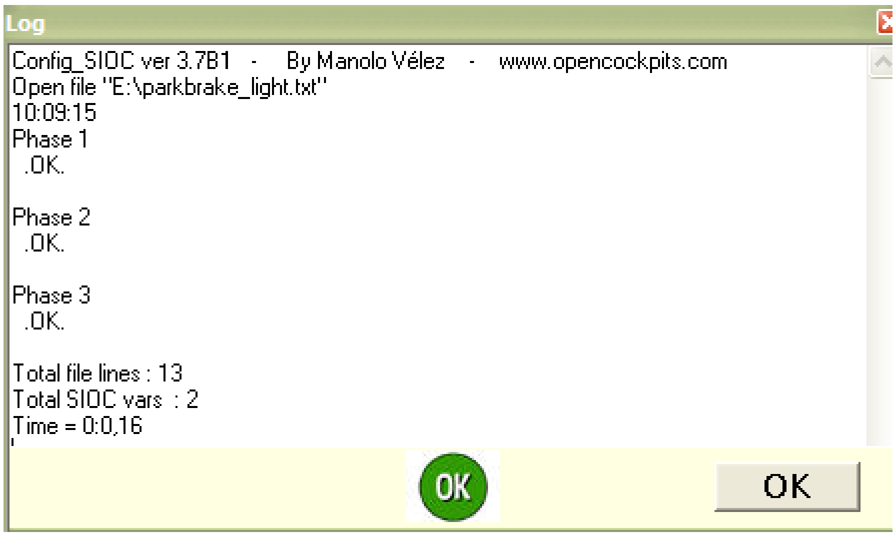


Name	Size	Type	Date Modified
parkbrake_light.txt	1 KB	Text Document	17-10-2009 21:44

Then we run config-sioc.exe and select 'Files, Import TXT'

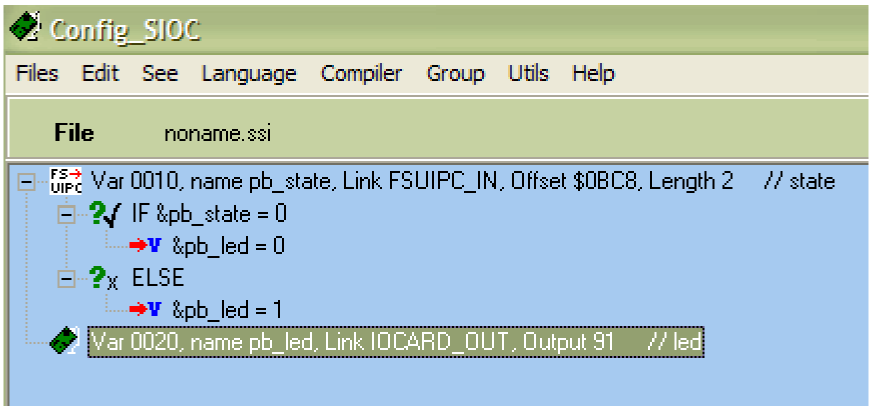


We select the file 'parkbrake_light.txt' and compilation will start immediately



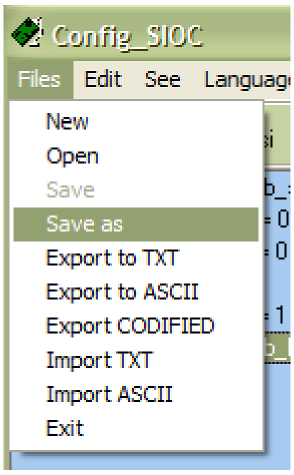
```
Config_SIOC ver 3.7B1   -   By Manolo Vélez   -   www.opencockpits.com
Open file "E:\parkbrake_light.txt"
10:09:15
Phase 1
 .OK.

Phase 2
 .OK.

Phase 3
 .OK.

Total file lines : 13
Total SIOC vars  : 2
Time = 0:0,16
```

There are no errors, so we click at the gray OK button, bottom right, to close the log window, and we inspect the config-sioc.exe main window showing the compiled code

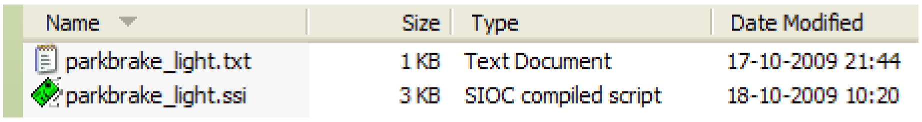


```
Var 0010, name pb_state, Link FSUIPC_IN, Offset $0BC8, Length 2     // state
    IF &pb_state = 0
        &pb_led = 0
    ELSE
        &pb_led = 1
Var 0020, name pb_led, Link IOCARD_OUT, Output 91     // led
```

This view is only of importance if you would enter code via the GUI (Graphical User Interface) of config-sioc (not using a text editor), so we directly go to the Files menu item and select 'Save as':



We fill in parkbrake_light.ssi, and we are ready. The folder also shows the compiled file (type SIOC compiled script):



Name	Size	Type	Date Modified
parkbrake_light.txt	1 KB	Text Document	17-10-2009 21:44
parkbrake_light.ssi	3 KB	SIOC compiled script	18-10-2009 10:20

It is good to notice that the compiled .ssi file is a bit different from compiled code of programming languages such as C/C++ or Java. The file not only contains executable code but also contains the complete source, comments included. Therefore config-sioc.exe also has the capability to export a .txt file (or to recreate one could say) from a .ssi file (Export to TXT).

I recommend to only use this feature if you have got a .ssi file without the original .txt file, this is for instance the case with .ssi files created via the config-sioc GUI. Otherwise there is no need to Export to TXT, always stick to your text file created with the Notepad Editor.

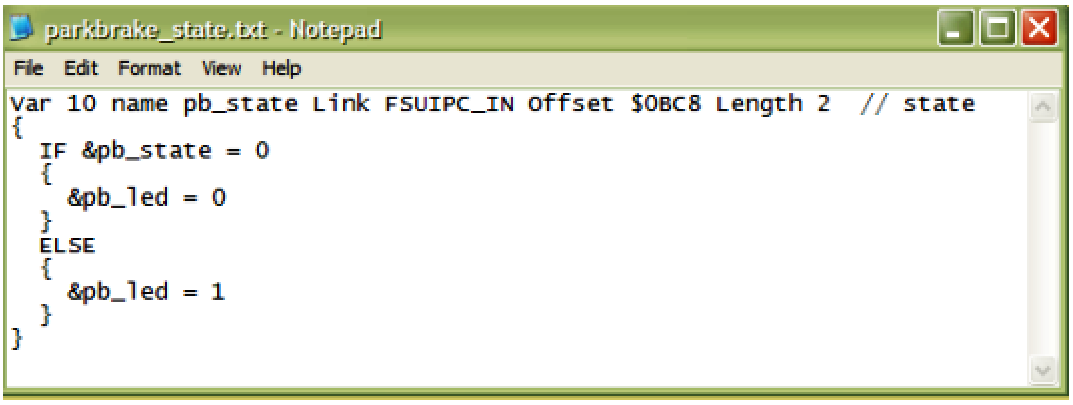
Compile multiple Text Files

If the number of scripts for your cockpit grows it becomes more difficult to put them into one file without variable number clashes. You also have to manually group your Var 0 statements and getting the overall view over your program becomes difficult.

In those situations it is good software engineering practice to split up your application into several self-contained logical units. Config_sioc supports this practice.

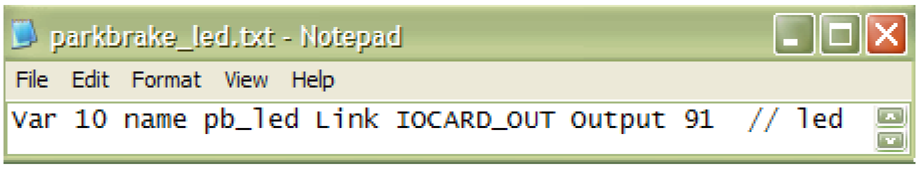
I will show the necessary steps based on the same example as above in the single file approach. Of course one would not split up such a small file, but it is just the mechanism I'd like to show you.

We start with the Notepad editor and create first a file that handles the state of the parking brake



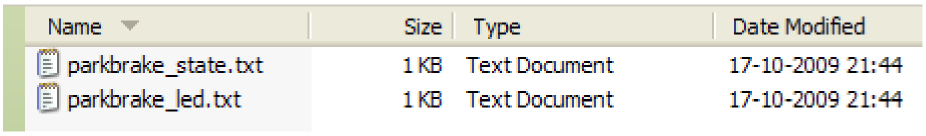
```
var 10 name pb_state Link FSUIPC_IN Offset $0BC8 Length 2  // state
{
  IF &pb_state = 0
  {
    &pb_led = 0
  }
  ELSE
  {
    &pb_led = 1
  }
}
```

and then a text file that handles the led of the parking brake (note that both files have a variable numbered 10).



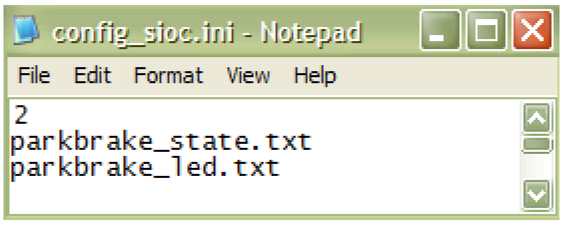
```
Var 10 name pb_led Link IOCARD_OUT Output 91  // led
```

Both files are saved



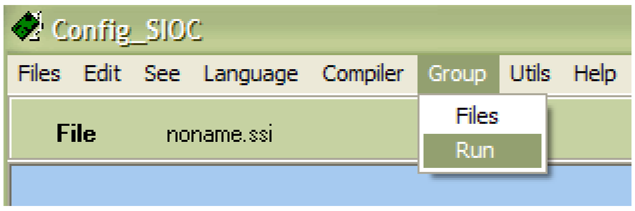
Name	Size	Type	Date Modified
parkbrake_state.txt	1 KB	Text Document	17-10-2009 21:44
parkbrake_led.txt	1 KB	Text Document	17-10-2009 21:44

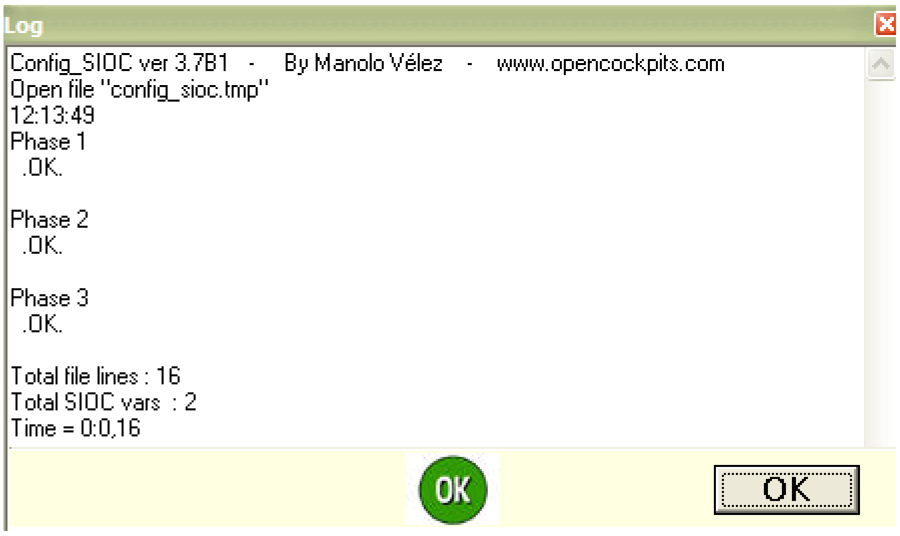
In order to be able work with multiple files and config-sioc.exe we create a config-sioc.ini file that defines the text files that make up our application. We do that by writing such a file with the Notepad editor. The first line is fixed and defines the languages setting of config-sioc.exe (1 = Spanish, 2 = English), but the second and third line specify where config-sioc.exe should look for our script files. This file has to be put in the same folder as config-sioc.exe (normally the SIOC folder).



```
2
parkbrake_state.txt
parkbrake_led.txt
```

Instead of Files, Import TXT, like we do in the single file approach, we now select 'Group, Run', and compilation will start immediately

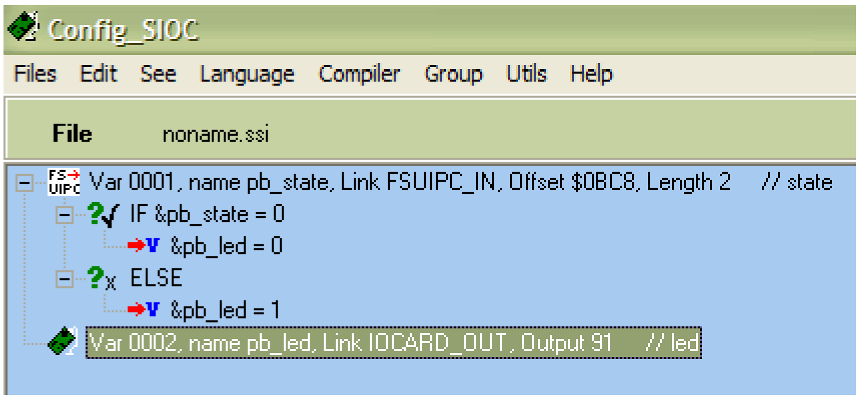




```
Config_SIOC ver 3.7B1   -   By Manolo Vélez   -   www.opencockpits.com
Open file "config_sioc.tmp"
12:13:49
Phase 1
 .OK.

Phase 2
 .OK.

Phase 3
 .OK.

Total file lines : 16
Total SIOC vars  : 2
Time = 0:0,16
```

There are no errors. Note that config-sioc compiles in three phases, which is very convenient for us programmers, because we do not have to import missing variable names or -numbers into our file, we can just make use of a variable that is defined in another file.

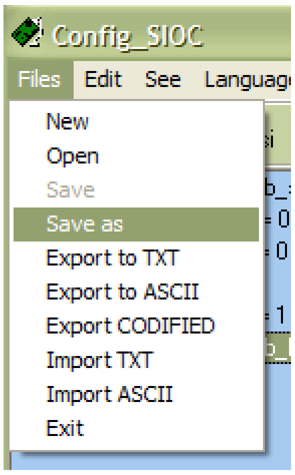
We now click at the gray OK button, bottom right, to close the log window, and inspect the config-sioc.exe main window showing the compiled code.



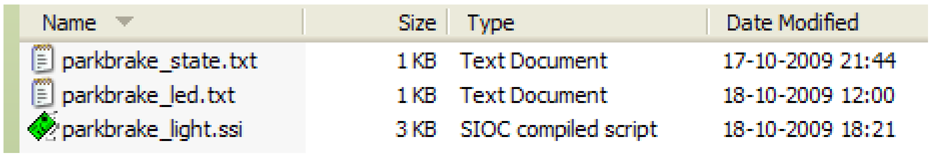
```
Var 0001, name pb_state, Link FSUIPC_IN, Offset $0BC8, Length 2     // state
    IF &pb_state = 0
        &pb_led = 0
    ELSE
        &pb_led = 1
Var 0002, name pb_led, Link IOCARD_OUT, Output 91     // led
```

There is one big difference with the single file approach: config-sioc has automatically renumbered the variable numbers in our files, instead of twice var number 10, we now see vars 1 and 2. It has resolved the number clash, something we as programmers no longer have to worry about.

Just like in the single file approach we select 'Files, Save as' and save the compiled code as parkbrake_light.ssi



We are ready, the folder shows the compiled file (type SIOC compiled script), next to the two text files



Name	Size	Type	Date Modified
parkbrake_state.txt	1 KB	Text Document	17-10-2009 21:44
parkbrake_led.txt	1 KB	Text Document	18-10-2009 12:00
parkbrake_light.ssi	3 KB	SIOC compiled script	18-10-2009 18:21

Note: the end result, the file parkbrake_light.ssi, is semantically the same as in the single file approach, only the way to get there is different. Sioc.exe will not see any difference at all.

How to program in SIOC

This page gives some basic examples of how to program in SIOC (by writing scripts). All these examples are not specific for a certain type of aircraft (add-on) in Microsoft Flight Simulator.

Specific examples for the Level-D 767, based on the pre-defined *lekseecon* variables in SIOC, are described at a separate page.

Push Button

The push button is a switch that makes an electrical circuit when it is pushed. So it closes the contact when pushed and it opens the contact when released (not pushed). (note: for the purists, there are also push buttons that 'brake' a contact when pushed).

It has two terminals, one has to be connected to the ground of a group of inputs of the master card and the other to one of the 9 inputs of that same group at the Master Card. What terminal is connected to what does not matter. If the button is not pushed the input is high (1) and if pushed the input is low (0).

In SIOC one programs a push-button like this, define a Variable and **link** it to a button:

```
 Var 1 Link IOCARD_SW Input 23 Type P
```

Type **`P`** will make SIOC latch the button changes, so each time the button is pushed, the state of `Var 1` changes (from 0->1, from 1->0, and so on). If the button is pushed the input changes state and the code attached to the Var is executed.

Another possible way of using a push-button in SIOC is (note Type **`I`** instead of Type **`P`**):

```
 Var 2 Link IOCARD_SW Input 23 Type I
```

`Var 2` will be 1 if the push-button is pushed and 0 if the push-button is not pushed. So there is no latching of states like with type **`P`**.

And yet another way of using a push-button in SIOC is:

```
 Var 3 Link IOCARD_SW Input 23 Type P
 {
   IF v4 = 0
   {
     v4 = 1
   }
   ELSE
   {
     v4 = 0
   }
 }

 Var 4
```

The value of `Var 3` does not matter now, each time the button is pushed `Var 4` will change value.

On/off switch with 2 terminals

This switch (Single Pole Single Throw or SPST) has two mechanically different positions. In one position the contact is closed and in the other position the contact is open. Most popular are the toggle type ones with a small lever and the button type ones. The terminals have to be connected to the Master Card in the same way as a push-button.

In SIOC one codes an on/off switch with two terminals like this:

```
 Var 1 Link IOCARD_SW Input 23 Type I // I means "on/off"
switch.
```

`Var 1` is 0 or 1 depending on the physical position of the switch (toggle or push type does not matter).

Rotary Switch

This switch has **n** mechanically different positions. One terminal is the common ground and the other n terminals represent each position of the rotary switch.

The common ground has to be connected to the ground of a group of 9 inputs of a Master card and the n other terminals have to be connected to n inputs of the same group.

In SIOC one codes a rotary switch with *n* (say 3) positions with *n* vars, one for each input of the Master card:

```
 Var 1 Link IOCARD_SW Input 23 Type I // I means "on/off"
switch.
 Var 2 Link IOCARD_SW Input 24 Type I
 Var 3 Link IOCARD_SW Input 25 Type I
```

Due to the mechanical design of the rotary switch only one contact can be closed at a time, so only one of the three Vars will be 1, and the others will be zero, when the switch is in a position.

On/off switch with 3 terminals

The Single Pole Double Throw or SPDT switch has two mechanically different positions. It's a so called Change Over switch, there is always one closed and one open position. Most popular are the toggle type ones with a Lever and the button type ones. One terminal is the common ground and the other two terminals are for each position. The common ground has to be connected to the ground of a group of 9 inputs of a Master card and the two other terminals have to be connected to two inputs in the same group.

In SIOC one codes an on/off switch with three terminals with two Vars, one for each input of the Master card:

```
 Var 40 Link IOCARD_SW Input 23 Type I // I means "on/off"
switch.
 Var 41 Link IOCARD_SW Input 24 Type I
```

`Var 40` is 0 or 1 depending on the physical position of the switch (lever or push type does not matter). `Var 41` is 0 or 1 depending on the physical position of the switch and `Var 40` and `Var 41` always have different values!
Note that the behavior of a SPDT switch is exactly the same as a rotary switch with 2 positions.

Rotary Encoder

The rotary encoder has three terminals, one common ground and one for each direction. We can connect a Gray Type rotary encoder directly to the Master Card in the same way as a three terminal On/off Switch (see above), with the restriction that you have to use two (logically) consecutive inputs of the Master Card.

In SIOC one codes a gray type rotary encoder like this (note the Type 2 attribute):

```
Var 1 Link IOCARD_ENCODER Input 40 Aceleration 2 Type 2
```

`Input 40` is the first logical input, the second input is assumed to be input 41. With aceleration you can specify if you want to increase with higher delta's if you turn faster.

Here an example from a Heading Rotary:

```
Var 1 Link IOCARD_ENCODER Input 40 Aceleration 2 Type 2
{
   L0 = v1 // * -1 turning clockwise should be plus
   v2 = ROTATE 0 359 L0
}

Var 2 // heading (0 .. 359)
```

`Var 1` contains the encoder value (increment/decrement) caused by turning the rotary encoder. With the `ROTATE` function this delta is added/subtracted to Variable 2.

The `ROTATE` function does so by taking the actual value of `Var 2` first, then it adds the increment and checks the resulting value against the range (defined by the first two attributes supplied to the `ROTATE` function) and if necessary wraps around and then assigns the resulting heading to `Var 2`. `Var 2` contains the value of the heading, ranging from 0 to 359. By default var 2 starts with 0, if necessary give it another initial value with the Value attribute.

Make sure the encoder is generating positive values when turning clockwise. Note that the encoder value is dependent on the way you have connected the two pins of the encoder to the two inputs of the Master Card. If turning clockwise does not increment the value (but decrement) you can correct that that by interchanging the two wires. You can also do that quite easily in SIOC (without hardware changes) by multiplying the value coming from the encoder by -1, so the example above would then read:

```
Var 1 Link IOCARD_ENCODER Input 40 Aceleration 2 Type 2
{
   L0 = v1 * -1 // turning clockwise should be plus
   v2 = ROTATE 0 359 L0
}
```

Dual Rotary Encoder

A dual Rotary Encoder is composed of two rotary encoders. It has 2 * 3 terminals. We can connect a Gray Type dual rotary encoder directly to the Master Card in the same way as two single rotary encoders.

Here a SIOC script for a dual Rotary Encoder controlling a VOR frequency (in the range from 108.00 to 135.00):

```
Var 1 Value 0 // decimal digits
Var 2 Value 8 // freq digits

Var 4 Link IOCARD_ENCODER Input 22 Aceleration 1 Type 2
{
  L0 = v4 // * -1 make sure turning right increments
  L0 = L0 * 5 // steps of 5
  v1 = ROTATE 0 99 L0
}

Var 5 Link IOCARD_ENCODER Input 24 Aceleration 1 Type 2
{
  L0 = v5 // * -1 make sure turning right increments
  v2 = ROTATE 8 35 L0
}
```

The decimal digits are controlled by `Var 4`. First the increment is multiplied by 5, and then the ROTATE function will keep the value in the range 0 .. 95. (so in steps from 00, via 05 to 95)

The frequency digits are controlled by `Var 5`. Due to the `ROTATE` function the value in v2 is incremented or decremented between 8 and 35.

So together with a fixed fifth digit (always being '1') we get VOR frequencies between 108.00 and 135.00

Rotary Encoder with Push Button

If you want to control a frequency you preferably use a Dual Rotary Encoder, so you can set as well the integer as the decimal part of the frequency. However, these dual rotary encoders are expensive and hard to get so either you build a dual rotary encoder your self or you use a Rotary Encoder with a Push Button in the shaft.

In the following example we control the integer and decimal part of a VOR frequency with a single gray type rotary encoder. Switching between integer/decimal is done with the push button in the shaft.

```
Var 1 Value 0 // decimal digits
Var 2 Value 8 // freq digits

Var 3 Link IOCARD_SW Input 20 Type P // in the shaft

Var 4 Link IOCARD_ENCODER Input 22 Aceleration 1 Type 2
{
  L0 = v4 // * -1 make sure turning right increments
  IF v3 = 0
  {
    L0 = L0 * 5 // steps of 5
    v1 = ROTATE 0 99 L0
  }
  ELSE
  {
    v2 = ROTATE 8 35 L0
  }
}
```

Depending on the value of `Var 3` the rotary increment is added to the integer frequency part or to the decimal part.

Read from a FSUIPC offset

If you want to read from a FSUIPC offset you should define a Variable linked to FSUIPC, by using the attribute **FSUIPC_IN**. The Offset attribute defines the hexadecimal address of the offset, and the Length attribute is the number of bytes.

In the following example `Var 1` is Linked to offset $0BC8 which represents the state of the Parking Brake (see FSUIPC for Programmers.doc in the FSUIPC SDK).

```
Var 1 Link FSUIPC_IN Offset $0BC8 Length 2 // parking brake
state
{
  IF v1 = 0
  {
    v2 = 0
  }
  ELSE
  {
    v2 = 1
  }
}

Var 2 Link IOCARD_OUT Output 91 // led indicating parking
brake set
```

Each time the offset in FSUIPC changes value, SIOC will write that new value into Var 1 and the statement in the body of `Var 1` will be executed. So dependent on the value of `Var 1`, `Var 2` will be assigned a 0 or a 1. `Var 2` is linked to a Master Card output and controls a led. So with this simple piece of program we have got a led that indicates the state of the parking brake.

Note that without any more specification then this, the value of `Var 1` will be treated as unsigned value. If you want to treat a FSUIPC offset as signed value you have to add the attribute `Type 1` to the Link definition. Take for example the elevator trim offset in FSUIPC:

```
 Var 1 Link FSUIPC_IN Offset $0BC2 Length 2 Type 1 // elev.
trim
 {
    // var 1 has range -16383 .. 16383
 }
```

Sometimes a FSUIPC offset contains a value that needs further calculations. That is for instance the case with the TRUE heading offset $0580. In the FSUIPC for Programmers Manual we can read that you have to take the offset *360/(65356*65356) in order to get a TRUE heading. In this specific example the heading ranges from -180 to 180 degrees, whereby 0 degrees is North. In order to get a heading in the range 0 - 359 we have to multiply the offset by 0,000000083819 and for negative values add 360. The SIOC code for that is:

```
 Var 1 name HDG Link FSUIPC_IN Offset $0580 Length 4 // true
heading
 {
   L0 = &HDG * 8.3819E-008
   IF L0 < 0
   {
     L0 = L0 + 360
   }
   &D_HDG = ROUND L0
 }
```

Note that at we round the resulting value to the nearest integer. This value is always in the range 0 - 359 and ready to be written to a display.

 The previous example showed TRUE heading, but if you want a Magnetic heading you have to take the variation (dependent on where you are) into account. Luckily that variation is also available as FSUIPC offset. Here is the modified code:

```
 Var 1 name HDG Link FSUIPC_IN Offset $0580 Length 4 // true
heading
 {
   L0 = &HDG * 8.3819E-008
   IF L0 < 0
   {
     L0 = L0 + 360
   }
   L1 = &VARIATION * 0.0054932
   L0 = L0 - L1
   &D_HDG = TRUNC L0
 }

 Var 2 name VARIATION Link FSUIPC_IN Offset $02A0 Length 2
```

Write to a FSUIPC offset

If you want to write to a FSUIPC offset you should define a Variable linked to FSUIPC by using the attribute **FSUIPC_OUT**. The Offset attribute defines the hexadecimal address of the offset, and the Length attribute is the number of bytes.

```
 Var 1 Link FSUIPC_OUT Offset $0C4E Length 2 // Set NAV1 CRS

 Var 2
 {
   v1 = 123
 }
```

`Var 1` is a link to the FSUIPC offset that is used to set the CRS of Navigational Radio 1.

`Var 2` is a variable that, if it changes value and its body (between curly brackets) is executed, will assign `Var 1` with a value of 123. This value is passed to FSUIPC offset 0x0C4E by SIOC.

But when will this happen? At the moment never, because there is no event that assigns `Var 2` a value ...

In order to make this work we can for instance connect a push button to `Var 2` like this:

```
 Var 2 Link IOCARD_SW Input 23 Type P
 {
   v1 = 123
 }
```

If the button is pushed, `Var 2` will become 1 or 0. In this example we do nothing with that particular value, but due to the fact that `Var 2` has got a new value the statements between the curly brackets of `Var 2` are executed and `Var 1` will become 123 and thus the FSUIPC offset 0x0C4E becomes 123 and the CRS of the Navigational Radio 1 becomes 123.

Read/Write to a FSUIPC offset

A lot of FSUIPC offsets are as well used to give you information about a value in the flight sim panel as to set that value in the panel. If you want to read/write to a FSUIPC offset you may define a separate Var for reading and a separate Var for writing. However, it is more efficient to define a Var for read and write operations. This is done by linking with **FSUIPC_INOUT**.

Examples of this type of use are rotaries for controlling heading, course, frequency, and so on. If you would like to be synchronized with the flight sim panel, you should at least take the initial values in the panel (while loading a flight) of the course, heading, and so on into account. This to display these value at your hardware displays and to let your rotaries work with the actual values.

Also during operation, if you would use the mouse in the panel to change the course, you also would like your rotaries and your displays to take that into account.

The following example is a fully synchronized NAV1 Crs script. It does all the things mentioned above.

```
 Var 1 Link FSUIPC_INOUT Offset $0C4E Length 2 // NAV1 CRS in
FSUIPC
 {
   v2 = v1 // new value from FSUIPC ? (see note below)
 }

 Var 2 // internal NAV1 CRS
 {
   v4 = v2 // write to my 3-digit display
 }

 Var 3 Link IOCARD_ENCODER Input 22 Aceleration 4 Type 2
 {
   L0 = v3 // * -1 make sure turning right increments
   v2 = ROTATE 0 359 L0 // update NAV1 CRS
   v1 = v2 // write to FSUIPC
 }

 Var 4 Link IOCARD_DISPLAY Digit 25 Numbers 3 // Display
```

Note: the statement `v2 = v1` in the body of `Var 1` will have no effect if the new value comes from the rotary (from the body of `Var 3`) because `v2` then already has the value of `v1` (SIOC Golden rule).

Only values coming from FSUIPC that are different from the value in `v2` (at start up or if you have used the mouse in the panel) will have a synchronization effect at `Var 2`. This is exactly what we want to achieve!

It is very important to understand this SIOC script with just four variables, because it demonstrates the power (letting a rotary, a 3 digit display and a FSUIPC offset working together) of SIOC very elegantly. Please study this carefully and try to explain its behavior, the flow of control, in your own words.

Led

A single led can to be connected to an output at the Opencockpits Master Card or to an output at the Opencockpits USB Outputs card, always with a resistor of 280 or 330 ohm placed in series.

Important: Note the fundamental difference in approach between these two cards, the outputs of the Master Card share a common ground, while the outputs at the USB outputs card share a common Voltage (+5 normally). So take that into account while designing your electrical circuits.

In SIOC you can forget about these electrical differences and connect the output to a variable like this, see Var 1.

```
Var 1 Link IOCARD_OUT Device 1 Output 91 // led
```

In sioc.ini the logical device number 1 is linked to a physical USB device number; there you can make the difference between a Master Card and a USB Expansion card.

Note that you should only assign values 0 or 1 to `Var 1`. A 1 will imply that the led will lit.

Digit

The Opencockpit Display card can control up to 16 seven-segment digits.

You can connect four Display cards to a Master Card, so in total you can control up to 64 digits per Master Card. For the numbering consult this document.

Digits can be dimmed.

If you want to use just one digit you can use the following SIOC code:

```
Var 1 Link IOCARD_DISPLAY Digit 0 Numbers 1
```

Attribute Digit indicates which of the 64 digits and Attribute Numbers indicates how many digits, starting from 0 (in this example).

If you want to write a '5' to this digit, just assign `Var 1` the value 5, like this:

```
v1 = 5
```

Note: If you write a '6' to this digit you will get a (normal) 6 without the upper segment of the 7-segment display. However, if you write the code -999997 to the digit you will get a 6 with the upper segment on, as in use in some cockpits, for instance in the Level-D 767.

Write to a 3 digit display

The previous example showed how to write to a single digit. However, in a cockpit you often want to write a HDG or CRS to a 3 digit display (build out of three 7-segment display units).

Suppose the CRS is in Variable 1 (ranging between 0 and 359). We then simply write that value to a Variable 3 that is Linked to a DisplayII card of Opencockpits. The Digit attribute indicates the (physical) number at your DisplayII card of the least significant digit of the 3 digit value (the '9' in case of a value of 359). The Numbers attribute defines the number of digits, in this example it is 3. This means that we are using Digits 0, 1 and 2 for the CRS value. Important: you have to physically assemble them, from left to right, as 2, 1, 0.

```
Var 1 // CRS
{
  v3 = v1
}

Var 3 Link IOCARD_DISPLAY Digit 0 Numbers 3
```

For more advanced writing to displays, see the advanced example.

Suppress (blank) leading zeroes in a display

Suppose you have a 2 digit display showing numbers between 1 and 99 and you would like to see leading zeroes blanked, like " 7" instead of "07". How do you code that in SIOC?

The approach I recommend is to treat each digit separately, so define two displays of one digit each instead of one display of two digits. Start computing digits from right to left, so the least significant digit first, like this:

```
Var 1 name Value // 1 .. 99
{
  L0 = &Value
  L1 = MOD L0 10 // value Modulo 10 gives the right digit
  &DigitR = L1
  L1 = DIV L0 10 // Value / 10 gives us the left digit
  IF L1 = 0 // if 0 do not write but send blank display code
  {
    &DigitL = -999999 // blank
  }
  ELSE
  {
    &DigitL = L1
  }
}
```

```
Var 2 name DigitR Link IOCARD_DISPLAY Digit 0 Numbers 1
Var 3 name DigitL Link IOCARD_DISPLAY Digit 1 Numbers 1
```

Writing negative numbers to a display

If you want to write a negative number to a display you should always take one display 'digit' extra into account for showing a minus sign or a blank. Note that it is not possible to display a + sign, so a blank means plus.

Suppose you want to display numbers ranging from -9 to +9; then you need a display of one digit for the number and a display of one digit for the minus sign, like this:

```
Var 1 name Value // -9 to +9
{
  &Digit = ABS &Value // take the absolute value
  IF &Value < 0 // negative
  {
    &Sign = -999998 // minus sign '-'
  }
  ELSE
  {
    &Sign = -999999 // blank
  }
}

Var 2 name Digit Link IOCARD_DISPLAY Digit 0 Numbers 1
Var 3 name Sign Link IOCARD_DISPLAY Digit 1 Numbers 1
```

Write to a display (advanced)

As your cockpit grows and grows you are no longer happy with the fact that your displays are always on, even if your cockpit is cold and dark... And for maintenance you would like to be able to test all segments of your displays.

This can be done in SIOC if you adapt to the following approach. Never write to a Display directly but use a Subroutine for each display. In the subroutine we can test for ColdAndDark and/or Lights test and act accordingly, if life is normal we output the value to the Display. Don't forget to call the OutCrs subroutine as well as the CRS changes value (in Var 1) as if the cockpit changes state (to/from cold and dark) or if we push or release the Lights Test button:

```
Var 1 name CRS
{
  CALL &OutCRS
}
```

```
 Var 2 name OutCRS Link SUBRUTINE
 {
   IF &ColdAndDark = 1
   {
     v3 = -999999 // blank the display
   }
   ELSE
   {
     IF &TestLights = 1
     {
       v3 = 888 // put all segments on by writing 8's
     }
     ELSE
    {
       v3 = &CRS // normal case, write the value to the
display
     }
   }
 }

 Var 3 Link IOCARD_DISPLAY Digit 0 Numbers 3
```

I have not coded ColdAndDark and TestLights in this example. At the howto-767 page I give an example of how to code that for the Level-D 767. Note that in light aircraft (general aviation) you may use the avionics switch as a cold and dark indicator.

Dim a Display

The Opencockpits DisplayII unit can be dimmed, which is a nice feature.

Undimmed the DisplayII card is able to control 16 7-segment displays, numbered from 0 to 15. When dimming is used the DisplayII card is able to control 15 7-segment displays, numbered from 0 to 14, display 15 is then used to control the brightness of display units 0..14.

So we loose one 7-segment display, a pity but not a big deal because we can connect up to 4 DisplayII cards to one Master Card and they are not very expensive. But it is something to take into account when building a MCP for instance: always plan for future extensions (there will come the day that you will want this feature...), so do not use display segment 15 for a digit, even if you are not using Dimming.

The next example shows how you dim a display using a gray type rotary encoder:.

```
 Var 1 name Dimmer

 Var 2 name RO_Dimmer Link IOCARD_ENCODER Input 40 Aceleration
1 Type 2
 {
   L0 = &RO_Dimmer
   &Dimmer = LIMIT 1 15 L0
   &D_Dimmer = -999994
   &D_Dimmer = &Dimmer
 }

 Var 3 name D_Dimmer Link IOCARD_DISPLAY Digit 15 Numbers 1
```

There are 15 possible brightness control values, indicated with values 1 to 15.

The rotary encoder generates values between 1 and 15 (due to the LIMIT function), which are kept in `Var 1` during the time you are not turning the rotary encoder.

The brightness control value is send to "digit" 15 after we have sent -999994, a sort of escape (attention) value to tell the Display2 unit that we want to send a brightness control value.

That's all...

Note that this code does not interfere with the normal SIOC code used to write values to DisplayII cards, that code remains unchanged, also a nice feature.

Internal Variables

Internal Variables can be used to to store intermediate results. You can use these variables inside the body of a Variable.

There are six Internal Variables, three of type **Boolean**: `C0, C1 and C2`, and three of type **Integer/Float**: `L0, L1 and L2`.

Example of **`Boolean`** usage:
If we have to test `IF v1 > 2 AND V1 < 10` then we put the result of each comparison in a boolean internal variable first:

```
 Var 1
 {
   C0 = v1 > 2
   C1 = v1 < 10
   IF C0 AND C1
   {
    // ...
   }
 }
```

Example of **`Integer`** usage:
Suppose we want to assign `Var 3` the value of the total `of Var 1 * 10 + Var 2`.
We cannot write `v3 = v1 * 10 + v2` because only one operator is allowed in a statement, so we write it as follows:

```
 Var 10
 {
   L0 = v1 * 10
   v3 = L0 + v2
 }
```

Initialization constructs

There are two SIOC language constructs that can be used to give a SIOC variable an initial value at the time the program starts to run. One is to give a Variable definition a Value Attribute like this:

```
Var 1 Value 345
```

`Var 1` will get a value of 345 when the program starts, and only then.

The other way is Variable 0, note that all variables are treated equal with the exception of `Var 0` with Attribute Value.

Variable 0 is the Variable, if present because it is optional, and if it has an attribute Value attached, that will be executed first when the program starts. In this variable you can assign other variables a value. Variable 0 execution also precedes the Value attribute of another Variable (described above)..

```
 Var 0 Value 0
 {
 // initial assigments of values to variables in here
 }
```

An example of an initialization procedure can be found in the example about Flow of Control.

Flow of Control at start up

[example based on text from Peter Depoortere, thank you Peter!]

SIOC is a program and the SIOC script is just telling the program which are the Variables it has to work with and what to do when one of those Variables changes value.

Initially the SIOC program sets all Variables to a value of *'unknown'*. Then it starts its function. It checks first Variable 0 (Var 0) and if you set this value to an initial value of 0 (or something else) the value of Var 0 changes from unknown to 0 (or something else). Since Var 0 changes value it will execute the body between the `{ }` attached to this Var. This body can of course set values to other Variables which have, until this moment, the value of unknown. Thus generating a chain of events. Like this it will continue until all Vars are initialized.

When all these initialization events are over, the SIOC program continues its course by examining all possible inputs it can have, like hardware inputs closing/opening, FSUIPC_IN changes, FSUIPC_INOUT changes, etc. It will read these fsuipc offsets that you defined as IN or INOUT which a certain pace, defined in sioc.ini, default at a refresh rate of 50 msec (20 times /sec).

When one of these values change, the body attached to the Variable will be executed and this can generate a chain of Variable changes and other bodies to be executed. [Something special are the timers since these will be able to run in another thread and change a Variable in a certain timely manner.]

I give a small example:

```
Var 0 name INIT value 123
{
  &Var1 = 10
  &Var2 = 20
}

Var 1 name Var1

Var 2 name Var2
{
  IF &Var1 = 10
  {
    &led = 1
  }
}

Var 3 name led Link IOCARD_OUT
Var 4 name gear Link FSUIPC_IN offset XXXX length y
```

Initially the SIOC program will build a database registering 5 Variables with value 'unknown' and it will register for each of these Variables a body to execute. `Var 1` and `Var 3` and `Var 4` will have an empty body.

Now SIOC will treat first `Var 0` and set the value to 123 so its value changes and the body needs to be executed. So it will set Var 1 to 10, so its changes value from unknown to 10 but there is no body to execute so SIOC goes to the next line where it will set `Var 2` to 20. Now `Var 2` changes from unknown to 20 and there is a body defined so it will execute that body. This `Var 2` body checks to see if `Var 1` is 10 and this is true so it sets `Var 3` to 1. `Var 3` changes from unknown to 1 so it needs to do the body. But there is no body. However it is linked to an output so it will send this to the hardware where the output will become logically 1 and the led will lit.

Now SIOC finished treating `Var 0` so it goes to `Var 1`, this one does not have an initialization value so nothing needs to happen. SIOC goes to `Var 2`, again no individualization value specified so nothing to do. The same for `Var 3`.

Now SIOC ends up at `Var 4` which is an FSUIPC input at offset XXXX, so it will copy the value from offset XXXX to `Var 4`. If `Var 4` would have a body it would execute this one too...

Finally the complete initialization is done and SIOC will read continuously FSUIPC offset XXXX, since this is the only input defined. If it changes value it will copy the new value into `Var 4`.

Subroutine Var

A special type of Var is a `Subroutine`. It is defined by adding `Link` `SUBRUTINE` to the Var declaration.

A subroutine Var is special because it is the only exception to the SIOC Golden Rule that says that a script attached to a Var is only executed if the var changes value. The script attached to a subroutine Var is always executed, whether the Var is assigned a value or called.

One can define subroutine Vars that do not use the Value of the Var in the attached script:

```
Var 1 name OutHDG Link SUBRUTINE
{
 &O_HDG = &HDG // assign to a value outside this script
}
```

Or one can define subroutine Vars that use the actual value of the Var in the attached script:

```
Var 2 name OutIAS Link SUBRUTINE
{
 &O_IAS = &OutIAS // assign to the value of Var 2
}
```

An example of assigning a value to a subroutine Var:

```
&OutIAS = 3
```

An example of calling a subroutine Var:

```
CALL &OutHDG
```

If you `CALL` a subroutine, you may (optionally) add a parameter. The Subroutine Var will get that value:

```
CALL &OutIAS 3
CALL &OutIAS L0
CALL &OutIAS v908
```

Hold up execution of program flow

Most procedural programming languages have `Wait` functions that hold up execution. SIOC has not.

If you want to hold up execution you have to spread your code in two variables, start execution in the first variable and jump to the second variable with a `DELAY` statement like this:

```
Var 1
{
  // some code here
  v2 = DELAY 1 100 // after 1 second delay jump to v2
}

Var 2
{
  // delayed code here
}
```

Putting the delayed code right after the `DELAY` statement would not work, because SIOC "executes the `DELAY` statement in the sense that it sets the assignment to v2 aside for execution at a later time" and it continues immediately with any statement after the `DELAY` statement...

Note: If you want this flow of control to be reentrant you must assign a different value to var 2 each time. The first parameter of the `DELAY` statement cannot be a fixed value then.

An example of this technique can be found in the HowTo example Making a led blink a fixed number of times.

Names instead of Numbers

Compare these two small programs, that do exactly the same thing:

```
Var 1 Link FSUIPC_IN Offset $0BC8 Length 2 // parking brake
state
{
  IF v1 = 0
  {
    v2 = 0
  }
  ELSE
  {
    v2 = 1
  }
}
```

```
 Var 2 Link IOCARD_OUT Output 91 // led indicating parking
brake set
```

and

```
 Var 1 name ParkingBrake Link FSUIPC_IN Offset $0BC8 Length 2
 {
   IF &ParkingBrake = 0
   {
     &PBrakeLed = 0
   }
   ELSE
   {
     &PBrakeLed = 1
   }
 }

 Var 2 name PBrakeLed Link IOCARD_OUT Output 91
```

The second program is to be preferred, because:

- If you renumber your Vars, you only have to do it at one place, you don't have to search for other occurences of the same Variable number.
- There is less need to comment your program because you can put semantics in the Variable names.
- Names can be used in other SIOC scripts if you compile multiple files in one go.
- And finally your program is easier to comprehend by other people.

Toggle a value

Suppose we have a Flag Variable with values 0 or 1.

```
 Var 1 name Flag Value 0
```

There are two ways in the SIOC software to change this Flag value from 0 to 1 or from 1 to 0 (called toggling)

```
 Var 2
 {
   &Flag = CHANGEBITN 0 &Flag
 }
```

or

```
 Var 3
 {
   IF &Flag = 0
```

```
    {
      &Flag = 1
    }
    ELSE
    {
      &Flag = 0
    }
  }
```

The `CHANGEBIT` method in `Var 2` is to be preferred.

Landing Lights toggle

This example is based on the generic FSUIPC offset for the lights, it will work with almost any aircraft. This is how the SIOC code reads:

```
Var 1 Link IOCARD_SW Input 50 Type I
{
   v2 = CHANGEBIT 2 v1
}

Var 2 Link FSUIPC_OUT Offset $0D0C Length 2
```

`Var 1` is a link to your on/off toggle switch, connected to input 50 of the IOCards Master card. *(Note that 50 is just an example)*. Type `I` means on/off switch. If switch is on, `Var 1` is 1 and if switch is off, `Var 1` is 0.

`Var 2` is a link to the generic FS2004 FSUIPC offset with the lights controls (2 bytes long). Bit 2 in that offsets represents the landing lights *(See FSUIPC for Programmers.doc).*

Each time you toggle your hardware switch, the body (the part between curly brackets) of Var 1 is executed. There is only one statement in that body; it means that bit 2 in `Var 2` will be set according to the value of var 1. So if the toggle switch is ON, bit 2 will be 1, and if switch is OFF bit 2 will be 0. Note that with the Changebit function no other bits in offset 0x0D0C will be effected.

Light a led after n seconds

If you want to light a led with a **delay** of n seconds, you can use the `DELAY` function in SIOC.

In this example the led will be lit after 4 seconds. The second parameter of the `DELAY` function is times 1/100 second and the first parameter is the value that will be assigned after the delay.

```
Var 1
{
  &Led = DELAY 1 400
}

Var 2 name Led Link IOCARD_OUT Output 92
```

In the next example the led will be lit after a random value between 1 and 4 seconds. The parameters for the `RANDOM` function are the lower and upper bounds.

```
Var 1
{
  L0 = RANDOM 100 400
  &Led = DELAY 1 L0
}

Var 2 name Led Link IOCARD_OUT Output 92
```

Light a led for n seconds

If you want to light led for n seconds based on another event, you can use the `DELAY` function in SIOC.

In this example the event is a 'button push'. If that happens we light a led for 4 seconds.

```
Var 0
{
  &Led = 0
}

Var 1 name Button Link IOCARD_SW Input 27 Type I
{
  IF &Button = 1
  {
    IF &Led = 0
    {
      &Led = 1
      &Led = DELAY 0 400
    }
  }
}

Var 2 name Led Link IOCARD_OUT Output 92
```

Note that although you are using a push button the Type Attribute of `Var 1` should be I.

Make a led blink for n seconds

In order to make a led blink for a number of seconds you can use the `TIMER` function in SIOC.

We take the same event as the previous example, a 'button push'. If that happens we let a led blink for 4 seconds.

```
Var 0
{
  &Led = 0
}

Var 1 name Button Link IOCARD_SW Input 27 Type I
{
  IF &Button = 1
  {
    IF &Led = 0 // only start if led does not blink
    {
      &BlinkLed = 10 // begin value of timer
      &BlinkLed = TIMER 0 -1 40
    }
  }
}

Var 2 name BlinkLed
{
  L0 = MOD &BlinkLed 2
  IF L0 = 0
  {
    &Led = 0
  }
  ELSE
  {
    &Led = 1
 }
}

Var 10 name Led Link IOCARD_OUT Output 92
```

The `TIMER` parameters mean:

- 0 = end value,
- -1 = increment value (actually a decrement in this negative value case),
- 40 = interval (* 10 msec)

So we are counting down from 10 to 0 with steps of -1, in intervals of 0.4 seconds. Note that counting up is of course also possible.

The `BlinkLed Var` will regularly be assigned the Timer value (10, 9, 8, 7, 6, ...0). Due to the Modulo 2 function in its body (10 MOD 2 = 0, 9 MOD 2 =1, 8 MOD 2 = 0), it will set the value of the Led to 1 or 0 and we get the blinking effect.

Stop a blinking led

How do you stop a 'continuous' blinking led? This might be needed for blinking annunciators that only stop blinking after pushing a button.

In this example a led starts blinking at the start of the SIOC program, but you can of course also take other events to start the blinking of a led. The blinking is done with a `TIMER`, just as in the previous example. The difference is that we now program the Timer for a very long period, it counts down from 500.000 to 0 in steps of .4 seconds, long enough for a long haul flight ;-).

As soon as you push the button, the code attached to the button will **force** the Timer value to 1 step before the end value. In the next Timer operation the Timer will end and the blinking will stop. *Make sure that the end value of the timer is an even value, so the last led value will be 0.*

```
Var 0
{
  &Led = 0
  &BlinkLed = 500000 // begin value of timer
  &BlinkLed = TIMER 0 -1 40
}

Var 1 name Button Link IOCARD_SW Input 28 Type P
{
  &BlinkLed = 1 // equal to end value of timer (0) + 1
}

Var 2 name BlinkLed
{
  L0 = MOD &BlinkLed 2
  IF L0 = 0
  {
    &Led = 0
  }
  ELSE
  {
    &Led = 1
 }
}

Var 10 name Led Link IOCARD_OUT Output 92
```

Make a led blink a fixed number of times

If you want to blink a led for a fixed number of times you can make use of the Blinks subroutine approach defined below.

The parameter for this subroutine specifies the number of blinks. You can call Blinks with a constant

```
  CALL &Blinks 4
```

or a variable

```
  CALL &Blinks L0
```

Here the subroutine Blinks. Note it is called recursively until all blinks have been generated.

```
 Var 1 name Blinks link SUBRUTINE // parameter is number of
blinks
 {
   &Led = 1 // led on
   &LedOff = DELAY &Blinks 50 // half a sec ON
 }

 Var 2 name LedOff
 {
   L0 = &LedOff
   IF L0 > 0
   {
     &Led = 0 // led off
     &Complete = DELAY L0 50 // half a sec OFF
   }
 }

 Var 3 name Complete
 {
   L0 = &Complete
   IF L0 > 0
   {
     IF L0 > 1
     {
       L0 = L0 - 1
       CALL &Blinks L0 // recursive subroutine call
     }
     ELSE
     {
       &LedOff = 0 // make responsive for another series of
Blinks.
       &Complete = 0 // make responsive for another series of
Blinks.
     }
```

```
    }
  }

Var 4, name Led Link IOCARD_OUT Output 92
```

The width of the ON pulse and the length of a cycle can be specified with the parameters for the `DELAY` functions.

Note that in the Blinks subroutine (Var 1 and Var 2) I use a technique to hold up execution of program flow.

An Endless Timer

If you want a function to be performed say every 50 msec, you should make use of an Endless Timer, this is a Timer with increment **zero**, so it will never reach the end value...

It goes like this:

```
Var 0 Value 0
{
  &control = 0
  &control = TIMER 1 0 5
}

Var 1 name control Link SUBRUTINE
{
  // regular control work in here (is called every 50 msec)
}
```

The Timer calls a subroutine control after each interval. Put SIOC code for your regular control work, such as motor control, in that subroutine. Note that the interval is x10 msec, in this example 5x10 msec = 50 msec or 20 times per second.

Add auto repeat

It is good to have an auto repeat function for a Switch. For example the FSUIPC offset 0x4602 is used to increment the Fuel quantity in your aircraft's tank. But you do not want to toggle a switch over and over again, you just want to set a switch to on, and after a while you will want to set it to off.

We use the `TIMER` function for that. Start a timer (endlessly) counting up, during counting we generate the right commands to FSUIPC, and we force the timer to stop when the switch is set to off again.

```
 Var 1 Link IOCARD_SW Input 23 Type I
 {
   IF v1 = 1
   {
     &Fuel = 0 // startvalue of timer
     &Fuel = TIMER 1000 1 100 // 1 second interval, from 0 to
1000
   }
   ELSE
   {
     &Fuel = 999 // force timer to stop very soon ...
   }
 }

 Var 2 name Fuel
 {
   L0 = MOD &Fuel 2
   IF L0 = 0
   {
     &FSUIPC_Fuel = 177 // Set to value 177 to increment fuel
   }
   ELSE
   {
     &FSUIPC_Fuel = 0 // clear, so it will detect a new value
   }
 }

 Var 3 name FSUIPC_Fuel Link FSUIPC_OUT Offset $4602 Length 1
```

Set Altimeter(QNH)

In this example I'll show you how to set the Altimeter (QNH) with a gray type rotary encoder. I use FSUIPC offset 0x3110 for that (explanation), this approach will work for almost any aircraft.

```
 Var 1 Link IOCARD_ENCODER Input 23 Aceleration 1 Type 2
 {
   L0 = v1 // * -1 turning clockwise should be plus
   IF L0 >0
   {
     v2 = 65883 // FS Control Kohlsman increment
     v2 = DELAY 0 10
   }
   ELSE
   {
     IF L0 < 0
     {
       v2 = 65884 // FS Control Kohlsman decrement
       v2 = DELAY 0 10
     }
   }
 }

 Var 2 Link FSUIPC_OUT Offset $3110 Length 4
```

If you turn clockwise, the rotary encoder values in Var 1 will be positive, and the FS Control Kohlsman increment code (see FSUIPC documentation for that) is send to FSUIPC via Var 2. After a delay of 100 msec Var 2 is reset to 0, so it will be ready to recognize a new value.

If you turn anti-clockwise, the rotary encoder values in Var 1 will be negative, and the decrement code is generated. See the section on Rotary Encoder for information about how to force the encoder to generate positive values when turing clockwise.

Play sound

In a cockpit you would like to play sounds. SIOC supports that in several ways. Let's start with a basic example (that hopefully will not occur ...), a flight attendant telling you that there is smoke coming out of the right engine (smoke.wav) ...

Add the smoke.wav file to the sioc folder.

Enable the `SOUND` Module in the sioc.ini file and define the smoke.wav file to be number 1:

```
 [SOUND MODULE]
 Sound_disable=No
 Volume=100

 [ #1 ]
 Sound=smoke.wav
```

Make a short SIOC script. We define `Var 1` as the SmokeStart control variable (it is linked to the `SOUND` Module of SIOC). In `Var 2` we write the sound file number to `Var 1` if an on/off switch at input 75 is set to 1, and the sound will start to play ...

```
 Var 1 name SmokeStart Link SOUND

 Var 2 Link IOCARD_SW Input 75 Type I
 {
   IF v2 = 1
   {
     &SmokeStart = 1 // number of the sound file defined in
sioc.ini
     &SmokeStart = 0
   }
 }
```

Note that we have to set `&SmokeStart` back to 0 immediately, otherwise it will not recognise a new value (other then 0).

If you want to play a sound **repeatedly** just put an asterisk before the name of the sound file in sioc.ini:

```
 [ #2 ]
 Sound=*wiper.wav
```

If you want to **stop** a playing sound you need a second sound control variable that does the stopping:

```
 Var 1 name WiperStart Link SOUND
 Var 3 name WiperStop Link SOUND Type S // add S for Stop
variable

 Var 2 Link IOCARD_SW Input 75 Type I
 {
   IF v2 = 1
   {
     &WiperStart = 2 // number of the sound file defined in
sioc.ini
     &WiperStart = 0
   }
   ELSE
   {
     &WiperStop = 2 // number of the sound file defined in
sioc.ini
     &WiperStop = 0
   }
 }
```

GMT HH:MM clock

As an example of a complete application in SIOC, here the code of a simple HH:MM clock showing GMT time (or Zulu time). The time is obtained from two offsets in FSUIPC (see also the read from FSUIPC example) and the hours and minutes are shown at a 4 digit display (see also the example about how to write to a display).

```
 Var 1 name Hours Link FSUIPC_IN Offset $023B Length 1 // (0-
23)
 {
   &D_ClockH = &Hours
 }

 Var 2 name Minutes Link FSUIPC_IN Offset $023C Length 1 //
(0-59)
 {
   &D_ClockM = &Minutes
 }

 Var 3 name D_ClockM Link IOCARD_DISPLAY Digit 3 Numbers 2 //
2 digits
 Var 4 name D_ClockH Link IOCARD_DISPLAY Digit 5 Numbers 2 //
2 digits
```

This is just a basic clock. One can easily add functionality, such as an option to show local time, a push button to switch between local and GMT, and support for lights test and 'cold and dark' cockpit situations.

Key generation in SIOC

Important Note: Although personally I hate to use keys as interface method, and I'm glad it is not necessary for the Level-D 767, for other add-ons this is often the only option.

There are two possible ways to send keys from within a SIOC script.

Method 1: **SIOC as a keyboard emulator.**

You send a value to a variable with Link KEYS:

```
 Var 1 Link IOCARD_SW Input 116 Type I
 {
   IF v1 = 1
   {
     &Key = 37
   }
   ELSE
   {
     &Key = 0
   }
 }

 Var 2 name Key Link KEYS
```

Note that you always have to reset the `Key` variable to a value that has no key attached (in this example 0) otherwise SIOC will not detect a second assignment of value 37 to `&Key` (because the value does not change, nothing will happen).

In the sioc.ini you define what key will be sent if this Keys variable receives a certain value:

```
 [************** KEYBOARD EMULATOR MODULE ***************]
 window = "Microsoft Flight Simulator X"
 #37=B
```

This method works ok but it is rather basic in the ways that keys can be defined.

An advantage is that you can specify the window the keys will be send to, so you can use this to send keys to any program window (although only one, and it must have focus). A drawback is that you can only use this method with Flight Simulator and SIOC running at the same PC.

Method 2: SIOC toggling FSUIPC virtual buttons

There is an easier, more user friendly and and more powerful way of generating keys from within SIOC, it is based on the facilities that FSUIPC provides.

FSUIPC contains the concept of a virtual joystick with 32 buttons. Nine of of these joysticks, each DWORD long (4 bytes), are available starting from FSUIPC offset 0x3340 (see FSUIPC for Programmers.doc in the FSUIPC SDK):

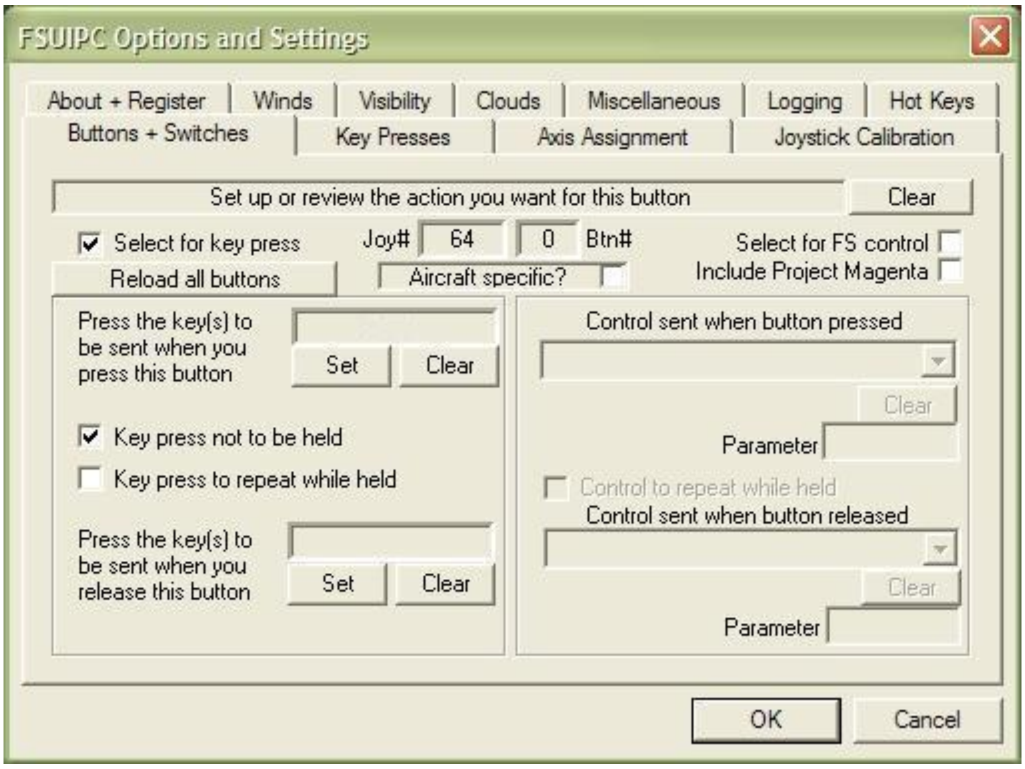
> *"Each DWORD represents one joystick with 32 buttons. If an external program sets or clears a bit in any of these 9 DWORDS the "Buttons" page in FSUIPC will register the change of a button operation in one of Joystick numbers 64 to 73 (corresponding to the 9 DWORDS). So, FSUIPC can be used to program whatever action the user wants."*

In the next small part of SIOC code we let SIOC toggle bit 0 of FSUIPC Joystick 64 upon a press of a momentary push button connected to input 116:

```
 Var 1 Link IOCARD_SW Input 116 Type I
 {
   &FO_JoyStick64 = CHANGEBIT 0 v1 // toggle bit 0 of joystick
64
 }

 Var 2 name FO_JoyStick64 Link FSUIPC_OUT Offset $3340 Length
4
```

Simple is not it? We could do exactly the same for the other 31 buttons of Joystick 64 or for each button in the other 8 Joysticks.

Assuming SIOC is running this SIOC code, we start Flight Simulator, open the FSUIPC module and we select the "Buttons+Switches" tab. If we now push our button the following information will appear:


FSUIPC Options and Settings

Set up or review the action you want for this button

☑ Select for key press Joy# 64 0 Btn# Select for FS control ☐ Include Project Magenta ☐ Aircraft specific? ☐

☑ Key press not to be held

☐ Key press to repeat while held

☐ Control to repeat while held

We see the Joystick number 64 and button 0 is recognised!

Now we 'tick' Select for key press, click Set, and type in the key(-s) (at our keyboard) that we want to have sent by FSUIPC if this virtual button is pressed (again).

Contrary to Method 1, FSUIPC always sends the keys to the **current active window** at the Flightsim PC. Note that this also works over a network, via wideFS.

Note that FSUIPC is very powerful, we can do much more like specifying sending the key repeatedly when the button press is held, or to generate another key if the button is released. We can also do other things than sending keys, like selecting a FS control or one of the PM controls.

Note also that it does not matter where your SIOC program runs, the virtual buttons will always be recognized by FSUIPC at your Flightsim PC.

Send FSUIPC Control in SIOC

From the "FSUIPC for Programmers.pdf":

> *FSUIPC offset $3110 (length 8) operates a facility to send any 'controls' to Flight simulator. Write all 8 bytes for controls which use a value, but just 4 will do for 'button' types. (The list of controls can be found in "List of FS2004 Controls.pdf")*

In SIOC we can send such a FSUIPC control (with optional parameter) like this:

```
Var 1 name FS_CONTROL Link FSUIPC_OUT Offset $3110 Length 4
Var 2 name FS_PAR Link FSUIPC_OUT Offset $3114 Length 4

Var 3 Link IOCARD_SW Input 12 Type P
{
 &FS_PAR = 1
 &FS_CONTROL = 66712
 &FS_CONTROL = DELAY 0 10
}
```

In this example each time the button is pressed we send a value 1 first and then the control (code) 66712. Note that after a delay of 100 msec we reset Var FS_CONTROL, otherwise it will not detect that another FS_CONTROL 66712 needs to be send.

A fine example of how you can use this is setting the QNH.

Use of the DEVICE attribute

A USB expansion card can support up to 4 Master Cards. The Input- Output- and Display numbers provided by these 4 cards are given in this Numbering document. If you need more in- outputs or displays, no problem, just add another USB expansion card. The numbering of inputs, outputs and displays at this second USB card is exactly the same as the ones at the first.

In order to avoid conflict, SIOC has introduced the Attribute **DEVICE**, e.g. Input 21 at logical DEVICE 0 is defined like this:

```
Var 1 Link IOCARD_SW DEVICE 0 Input 21
```

A device number is the logical number that you assign to the corresponding USB expansion card (DEVICE 0 is default). Input 21 at logical DEVICE 1 is defined like this:

```
Var 2 Link IOCARD_SW DEVICE 1 Input 21
```

The connection between a logical USB device number in SIOC and a physical device number in reality (as shown in the SIOC main window), is defined in a MASTER statement in sioc.ini.

For instance we could have defined

```
MASTER=0,4,4,13
```

meaning: Logical DEVICE 0 is linked to a USB expansion card at real Device 13, and it contains 4 master Cards.

```
MASTER=1,4,2,19
```

meaning: Logical DEVICE 1 is linked to a USB expansion card at real Device 19, and it contains 2 master Cards.

Other Opencockpits USB based cards behave exactly the same. Take the USB outputs card for instance, after we have added this line to the MASTERS section of sioc.ini

```
MASTER=2,6,1,22
```

meaning: Logical DEVICE 2 is linked to a USB outputs card at real Device 22.

we can define in our SIOC script again a output 21 at DEVICE 2 (being the USB outputs card).

```
Var 3 Link IOCARD_OUT DEVICE 2 Output 21
```

Cold and dark and lights test

Suppose we have this piece of code in which a led Q is controlled by the value of an FSUIPC offset. If the value is >28 the led will lit.

```
 Var 1 name LedValue Link FSUIPC_IN Offset $abcd Length 2
 {
  IF &LedValue >28
  {
   &O_Q = 1
  }
  ELSE
  {
   &O_Q = 0
  }
 }

 Var 2 name O_Q Link IOCARD_OUT Output 95 // led Q
```

This works perfectly fine.

QUESTION:

The question now is, how do we add Cold and Dark Cockpit support and Lights Test support to this (output) led? Suppose we have a Switch for lights test connected to an input:

```
 Var 101 name LightsTest Link IOCARD_SW Input 116 Type I
```

We cannot just write:

```
 Var 101 name LightsTest Link IOCARD_SW Input 116 Type I
 {
  IF &LightsTest = 1
  {
   &O_Q = 1
  }
 }
```

Because: If we toggle the Switch back to normal (0) nothing will happen, the led remains lit ... How do we give the led the value it had before the lights test??? The value of Var 1 has not changed during the lights test so the code to set the led will not run again ...

A similar problem arises if we have Cold and Dark Cockpit info available in our panel *(in this example it is out of scope how we obtain or derive such info).*

```
 Var 100 name ColdAndDark Link FSUIPC_IN Offset $abcd Length 2
```

Again, we cannot just write:

```
 Var 100 name ColdAndDark Link FSUIPC_IN Offset $abcd Length 2
 {
  IF &ColdAndDark = 1
  {
   &O_Q = 0
  }
 }
```

Because: If your aircraft comes to live, ColdAndDark becomes 0, and nothing will happen, the led remains off ... How do we set the led at the right value? The value of Var 1 may have changed during the cold and dark period, but the code to set the led will not run again at the cold and dark state change ...

ANSWER

A **generic solution** for this **synchronization** problem is to apply an approach in which we do not write to an output directly, but via a separate subroutine (for each output). In that subroutine we first test for the special situations Cold and Dark and Lights Test and act accordingly, and if they do not apply, we write the actual value of the led.

This subroutine is to be called from **three (!)** places in our code: 1) if the value of the led changes. 2) if there is a change of state in cold and dark (both ways, from cold to normal and from normal to code) and 3) if there is a change in LightsTest state (both ways too).

Below I give a complete script. I only describe one output, but for other outputs (and for digits) it will go exactly the same.

```
 Var 1 name LedValue Link FSUIPC_IN Offset $034C Length 2
 {
  IF &LedValue >28
  {
   &Q = 1
  }
  ELSE
  {
   &Q = 0
  }
 }

 Var 2 name O_Q Link IOCARD_OUT Output 95

 Var 3 name Q // always keep the actual value for the led
 {
  CALL &OutQ // call the routine to write the value to the led
 }

 Var 4 name OutQ Link SUBRUTINE
 {
  IF &ColdAndDark = 1
  {
   &O_Q = 0 // put the led OFF
  }
  ELSE
  {
   IF &LightsTest = 1
   {
    &O_Q = 1 // put the led ON
   }
   ELSE
   {
    &O_Q = &Q // write the actual led value
   }
  }
 }

 //[ Here you can put code for more outputs ]
```

The common part to detect changes in cold and dark and lights test and to call the output routines for led's (and displays):

```
 Var 100 name ColdAndDark Link FSUIPC_IN Offset $abcd Length 2
 {
  CALL &Refresh
 }

 Var 101 name LightsTest Link IOCARD_SW Input 116 Type I
 {
  CALL &Refresh
 }

 Var 103 name Refresh Link SUBRUTINE
 {
  CALL &OutQ
// [ add here all other CALL's to Out-routines for leds and
displays ]
 }
```

Generic SIOC code

This section contains SIOC code that can be used for almost all aircraft (without using lekseecon or FSCONV).

Several SIOC sources for building an **ADF** receiver based on generic FSUIPC offsets. Can be used for any aircraft that supports these FSUIPC offsets (most of them do).

Generic SIOC sources for building a **NAV1** receiver based on generic FSUIPC offsets. Can be used for any aircraft that supports these FSUIPC offsets (most of them do).
