Introduction:

FS2002 introduced a new format for gauges. The ”old” .gau gauges still work, but a new type of script based .xml gauges are introduced. Since no official SDK is available for these. The author decided to write down what can be found out by analyzing the samples provided by the default xml gauges. Together with the information the author found by himself a lot of information collected on the flightsim.com‘s and avsim.com’s panel design forums is here included.
This document is not meant to be a tutorial, but only a frequently updated collection of what is known about the XML gauge programming so far.
The author will give of course no warranty at all on the information in this document and possible resulting damages.

Even if the XML syntax is all new I sugest to read also the FS2000 Panel SDK from Microsoft and the sd2gau12.zip tutorial of Dai Griffiths. A lot of the infos are still usable in XML gauges.

Folder Structure or Cab Stucture

FS2002 comes with three *.cab files which contain XML gauges and are placed in the gauges folder:

- simicons.cab
- boeing747-400.cab
- beech_baron.cab
- minipanel.cab

The Cabinet (cab) format is a Microsoft format to store compressed files and folders. And in some Windows versions (XP) they are treated as folders. WinZip can extract these cab files, but can’t repack them. You need either a CabPacker (e.g. CabPack 1.4 available at http://home.t-online.de/home/lars.hederer/english.htm) or you extract the cab files with WinZip or a comparable product and leave the extracted folders in your gauges folder, then the xml files are read directly from these folders, and it’s easyer to edit them.

Referencing

The referencing of xml gauges in a panel.cfg is straightforward:

gauge00=Beech_Baron!Annunciator Gear Up, 130, 22

means the xml file ”Annunciator Gear Up.xml” located in ”Beech_Baron.cab” is used.
I stated before that it is also possible to extract ”Beech_Baron.cab” to a folder ”Beech_Baron” and treat all files in it as if they were in a cab file, so what happens if cab and folder with the same name exist, or even an ”old style” .gau file?
Asuume the above panel line, what does Fs2002? It will first check if a ”Beech_Baron” folder exists with an ”Annunciator Gear Up.xml” in it, if yes it will use it. If not Fs2002 searches a cab file ”Beech_Baron.cab” for an ”Annunciator Gear Up.xml”, if even that fails it will search for a conventional Fs2000 style multigauge ”Beech_Baron.gau” which contains a subgauge ” Annunciator Gear Up”.

Reading the Files

For those familiar with html and xml syntax the xml files might look familiar, there are however some strange symbols like ">","&lt”,”&” what does that mean? Since the ”<”,”>” and also ”&” are already used by the xml syntax itself you have to use ”special

cahracters" sometimes. ">" means ">", "&lt" means "<", "amp;" means "&" etc. (NOTE: if you intend to publish your xml code on one of the forums it might be convenient to introduce special characters before you paste it in the forum post. e.g. <Gauge Name="DME" Version="1.0"> might not show at all in the forum whereas <Gauge Name="DME" Version="1.0"> will give the correct result, the other way to prevent ">" showing up as ">" use ">" instead).
The xml syntax for FS2002 is mostly relative straightforward so everyone is invited to make first some trial and error exercises on the easier gauges to see "how things work". Some points have to be kept in mind: you can't store a xml file that is in use by FS2002 so you have to select a different aircraft, save the xml file and reload the first aircraft to see changes. The syntax is in general very sensitive to errors, if only on small space is is in the wrong place the whole gauge might dissapear, so it's always advisable to start from simple arrangements and work slowly up to more complex forms.

File Placement

A look in the default .cabs show a certain subfolder structure. Most of them have a "640" and "1024" subfolder containing the used bitmaps in two resolutions. The bitmap sizes correspond with the "ImageSizes=..." where the first two numbers are the width and height of the low resolution bitmap and the last two numbers width and height of the high resulution bitmap. If you don't want to use two sets of bitmaps for each gauge set the last two numbers to zero or omit them and place the bitmaps in "640", or omit the bitmap subfolders completely and place them directly in the cab or folder together with the xml files (example: minipanels.cab).

Variable Types

A look in the xml files show several different types of variables and events.

A: and P: variables are read-only variables that carry information of the state of FS2002 (see also a_vars.txt and p_vars.txt).

K: events are write-only informations that set FS2002 in a specific state. K: events often need special conversion calculations since they use always integer arguments, the already found conversions used in standard gauges are still valid here (see also k_events.txt). For standard gauge developers: `"arg (>K:EVENT)"` is equivalent to `"trigger_key_event(KEY_EVENT,arg);"`

G: and L: are user definable red-write variables that are only available to the gauges of the panel itself, there is no further link to FS2002. G: variables are only valid inside the gauge they are defined in ("G" for gauge?), they don't need conversions, they are limited to (G:Var1) to (G:Var9). L: variables are valid in all gauges of one panel ("L" for linked?)

In difference to standard gauges a typical read operation of A:, P:, L: variables does include already a conversion like `(A:Eng1 manifold pressure,inHg)` or `(A:Eng1 manifold pressure,millibar)`. A list of possible conversions is given in units.txt, but you have to consider that not all conversions make sense. The standard conversion that always seems to work is "number", which seems to give the intrinsic value of a variable.

"Callback" Syntax

The most complex part of XML-gauges are possibly the calulations, something you would name "callbacks" in conventional gauges. These calculations are usually part of <Value>...</Value>, <String>...</String>,<Visible>...</Visible> or <Click>...</Click> lines,

seldom they are part of <Tooltip>...</Tooltip> lines or in the ”Color=” entries of <Text>...</Text>.
Sometimes a <Value>...</Value> line looks quite decent like:

```
<Value>(A:Eng1 manifold pressure,inHg)</Value>
```
(beech_baron\Left Manifold Pressure.xml line 9)

the purpose is clear; display the manifold pressure of engine one in inches Hg. But there are also others:

```
<Value>(A:Indicated Altitude,feet) 10000 /</Value>
```
(beech_baron\Altimeter.xml line 24)

what does this mean? Or a real nightmare:

```
<Value>(A:Autopilot mach hold, bool) if{ 1 (&gt;L:Mach-IAS select, bool) }
(A:Autopilot airspeed hold, bool) if{ 0 (&gt;L:Mach-IAS select, bool) }
(L:Mach-IAS select, bool) if{ 9 } els{ (A:Autopilot airspeed hold var,
knots)  s1 100 / 10 % flr l1 100 % 99 &gt; if{ l1 10 % 9 - + } }</Value>
```
(boeing747-400\Autopilot IAS-Mach Window.xml line 14)

What the heck is this? Backward, upside-down or what? The solution is not hard at all. The calculations are made in a ”stack” syntax. Values are ”pushed” down a stack and operators modify and ”pop” them up again.

Take `<Value>(A:Indicated Altitude,feet) 10000 /</Value>` as an example. First `(A:Indicated Altitude,feet)` is placed on the stack, then `10000` is placed on top of the stack and the previous entry is pushed one down, as a last entry / is placed on top.As a binary operator it removes the both previous entries from stack and pushes the result of the operation on the stack.

$$(\text{A : Indicated Altitude, feet}) \rightarrow \underset{(\text{A : Indicated Altitude, feet})}{10000} \rightarrow \underset{(\text{A : Indicated Altitude, feet})}{\overset{/}{10000}} \rightarrow (\text{A : Indicated Altitude, feet})/10000$$

The result is a division of `(A:Indicated Altitude,feet)` by `10000` . As an exercise the reader can try to resolve the IAS-Mach window example <g>. Why is this strange way chosen? My assumption is that those lines are easily interpretable by machines, no need for checking precedence of multiplication before addition, no bracketing etc. The other side is that the designer has to resolve all this stuff before the writing of the syntax, and it is pretty bad readable for humans. Those of the readers who still know the early HP pocket calculators might recognize the similarity to the old HP syntax. It is also known under ”Reverse Polish Notation”. Thanks to Bob Samuelson an interesting site is http://www.hpmuseum.org/rpn.htm (see also rpn.zip).

In general the syntax is pretty C like especially in the writing of boolean operators. In the following a summary of all used operators, functions and modifiers is added:

- \+ (binary operator) : adds the last two stack entries
- \- (binary operator) : subtracts the last two stack entries
- * (binary operator) : multiplies the last two stack entries
- / (binary operator) : divides the last two stack entries
- % (binary operator) : remainder divides the last two stack entries

- /-/ (unary operator) : reverses sign of last stack entry
- -- (unary operator) : decrements last stack entry
- ++ (unary operator) : increments last stack entry
- sqrt (unary operator) : returns square root of last entry

- > (binary operator) : ”>” gives 1 if last stack entry is greater than forelast stack entry
- < (binary operator) : ”<” gives 1 if last stack entry is smaller than forelast stack entry

- &gt=; (binary operator) : ”>=” gives 1 if last stack entry is greater than or equal to forelast stack entry
- &lt=; (binary operator) : ”<=” gives 1 if last stack entry is smaller than or equal to forelast stack entry
- == (binary operator) : gives 1 if both last last stack entries are equal
- && (binary operator) : ”&&” logical AND, if both last stack entries are 1 gives 1 otherwise 0
- || (binary operator) : logical OR, if one of the last stack entries is 1 result is
- ! (unary operator) : logical NOT, toggles last stack entry from 1 to 0 or 0 to 1
- ? (ternary operator) : ”short if-statement”, if the last entry is 1, the forelast entry is used, else the fore-forelast (or the other way round. Try it, see it)

- & (binary operator) : ”&” bitwise AND
- | (binary operator) : bitwise OR
- ~ (unary operator) : bitwise NOT, toggles all bits
- << (binary operator) : ”<<” shift bits of forelast stack entry by last stack steps to the left
- >> (binary operator) : ”>>” shift bits of forelast stack entry by last stack steps to the right

- d : duplicates last stack entry
- r : swaps last two stack entries
- s0,s1,s2,.. : stores last stack entry in storage for later use
- sp0,sp1,sp2,... : (presumably) the same as above
- l0,l1,l2,... : loads value from storage and places on top of stack

- abs (unary operator) : returns absolute value (removes ”–”)
- int (unary operator) : cuts away digits after decimal
- near (unary operator) : rounds to nearest integer
- flr (unary operator) : gives next smallest integer
- dnor (unary operator) : normalizes degrees (all values are ”wrapped round the circle” to 0°-360°)
- rnor (unary operator) : normalizes radians (all values are ”wrapped round the circle” to 0-2π) (NOTE: doesn’t work too reliable)
- dgrd (unary operator) : converts degrees to radians (also rddg available?)
- sqrt (unary operator) : returns the square root of last stack entry

- pi : places π on top of stack
- atg2 (binary operator) : gives atan2 in radians (other trigonometric functions? sin,cos,tg? Other functions? sqrt, ln?)
- max (binary operator) : gives the greater of last two stack entries
- min (binary operator) : gives the smaller of last two stack entries

others:

if{ ... }	if last stack entry is 1, the code inside the brackets is executed (note that there is no SPACE between ”if” and ”{” but one after it and at least one SPACE before ”}”)
if{ ... } els{ ... }	if last stack entry is 1, the code inside the brackets is executed, else the code in the second set of brackets (take also care to where SPACEs are allowed and where not)
quit	leaves the execution immediately, last stack entry is used for further purposes
case	hard to describe, therefore an example:

```
30 25 20 10 5 1 0 7 (A:Flaps handle index, number) case
```

the numbers 30 25 20 10 5 1 0 are pushed down the stack, 7 says how much entries, based on the result of (A:Flaps handle index, number) ”case” extracts one of the seven numbers. If (A:Flaps handle index, number) is 0 -> 0, 1->1, 2->5,... 6->30.

NOTE: <String>s use also other forms of ”if” and ”case” which aren’t discussed here.

In the following I’ll try to illustrate the ”stack syntax” with some examples from the default gauges, where the stack syntax is resolved step by step to something which is pretty near to a standard callback written in C. This chosen mainly because I’m used to C and secondly the XML syntax, once resolved, is very C-like.

First example: The division operator ”/” from beech_baron\Altimeter.xml line 24, the goal is a division of the indicated altitude by 10000 for the 10000ft needle on the altimeter:

```
<Value>(A:Indicated Altitude,feet) 10000 /</Value>

step 1:     (A:Indicated Altitude,feet) 10000 /

stack[0]:   (A:Indicated Altitude,feet)

----

step 2:     (A:Indicated Altitude,feet) 10000 /

stack[0]:   10000
stack[1]:   (A:Indicated Altitude,feet)

----

step 3:     (A:Indicated Altitude,feet) 10000 /

stack[0]:   ((A:Indicated Altitude,feet)/10000)
```

Second example: The functions abs, min and the ternary operator ”?” from beech_baron\Altitude Selector.xml line 30. The goal is to display the vertical speed or the altitude dialed on the autopilot depending on a user defined variable Var1, the sign is displayed in a different image element, so the absolute value is used up to 9000ft/min or 90000ft.

```
<String>%((A:Autopilot vertical hold var, feet/minute) abs 9000 min
(A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?
)%!5d!</String>

step 1:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   (A:Autopilot vertical hold var, feet/minute)

----

step 2:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   abs((A:Autopilot vertical hold var, feet/minute))

----

step 3:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   9000
stack[1]:   abs((A:Autopilot vertical hold var, feet/minute))

----

step 4:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   min(9000,abs((A:Autopilot vertical hold var, feet/minute)))

----

step 5:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   (A:Autopilot altitude lock var, feet)
stack[1]:   min(9000,abs((A:Autopilot vertical hold var, feet/minute)))

----

step 6:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   abs((A:Autopilot altitude lock var, feet))
stack[1]:   min(9000,abs((A:Autopilot vertical hold var, feet/minute)))

----

step 7:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   90000
stack[1]:   abs((A:Autopilot altitude lock var, feet))
stack[2]:   min(9000,abs((A:Autopilot vertical hold var, feet/minute)))

----

step 8:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   min(90000,abs((A:Autopilot altitude lock var, feet)))
stack[1]:   min(9000,abs((A:Autopilot vertical hold var, feet/minute)))

----

step 9:     (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   (G:Var1)
```

```
stack[1]:   min(90000,abs((A:Autopilot altitude lock var, feet)))
stack[2]:   min(9000,abs((A:Autopilot vertical hold var, feet/minute)))

----


step 10:    (A:Autopilot vertical hold var, feet/minute) abs 9000 min
            (A:Autopilot altitude lock var, feet) abs 90000 min (G:Var1) ?

stack[0]:   (G:Var1) ? min(9000,abs((A:Autopilot vertical hold var,
            feet/minute))) : min(90000,abs((A:Autopilot altitude lock var,
            feet)))


(value1 ? value2 : value3 means: if value1 then value2 else value3)
```

Another example: Relational operators (<,>) and logical operators(AND,OR) from beech_baron\Annunciator Gear Up.xml line 4. The goal is to get a 1, if both engines are on low power or the flaps are extended and the gear isn't fully extended, for a "gear up" warning light:

```
<Value>(A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
(A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full flaps*) (*OR*)
(A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*) (A:Eng2 manifold
pressure,inHg) 13 &lt; &amp;&amp; || &amp;&amp;</Value>

step 1:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   (A:Gear left position,percent)

----

step 2:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   99
stack[1]:   (A:Gear left position,percent)

----

step 3:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   ((A:Gear left position,percent)<99)

----

step 4:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   (A:Trailing edge flaps0 left percent,percent)
stack[1]:   ((A:Gear left position,percent)<99)

----
```

```
step 5:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   90
stack[1]:   (A:Trailing edge flaps0 left percent,percent)
stack[2]:   ((A:Gear left position,percent)<99)

----

step 6:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[1]:   ((A:Gear left position,percent)<99)

----

step 7:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   (A:Eng1 manifold pressure,inHg)
stack[1]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[2]:   ((A:Gear left position,percent)<99)

----

step 8:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   13
stack[1]:   (A:Eng1 manifold pressure,inHg)
stack[2]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[3]:   ((A:Gear left position,percent)<99)

----

step 9:     (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   ((A:Eng1 manifold pressure,inHg)<13)
stack[1]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[2]:   ((A:Gear left position,percent)<99)

----

step 10:    (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   (A:Eng2 manifold pressure,inHg)
stack[1]:   ((A:Eng1 manifold pressure,inHg)<13)
stack[2]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[3]:   ((A:Gear left position,percent)<99)
```

```
----

step 11:    (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   13
stack[1]:   (A:Eng2 manifold pressure,inHg)
stack[2]:   ((A:Eng1 manifold pressure,inHg)<13)
stack[3]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[4]:   ((A:Gear left position,percent)<99)

----

step 12:    (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   ((A:Eng2 manifold pressure,inHg)<13)
stack[1]:   ((A:Eng1 manifold pressure,inHg)<13)
stack[2]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[3]:   ((A:Gear left position,percent)<99)

----

step 13:    (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   (((A:Eng2 manifold pressure,inHg)<13)&&((A:Eng1 manifold
            pressure,inHg)<13))
stack[1]:   ((A:Trailing edge flaps0 left percent,percent)>90)
stack[2]:   ((A:Gear left position,percent)<99)

----

step 14:    (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   ((((A:Eng2 manifold pressure,inHg)<13)&&((A:Eng1 manifold
            pressure,inHg)<13))||((A:Trailing edge flaps0 left
            percent,percent)>90))
stack[1]:   ((A:Gear left position,percent)<99)

----

step 15:    (A:Gear left position,percent) 99 &lt; (*gear is up*) (*AND*)
            (A:Trailing edge flaps0 left percent,percent) 90 &gt; (*full
            flaps*) (*OR*) (A:Eng1 manifold pressure,inHg) 13 &lt; (*AND*)
            (A:Eng2 manifold pressure,inHg) 13 &lt; &amp;&amp; ||
            &amp;&amp;

stack[0]:   (((((A:Eng2 manifold pressure,inHg)<13)&&((A:Eng1 manifold
            pressure,inHg)<13))||((A:Trailing edge flaps0 left
            percent,percent)>90))&&((A:Gear left position,percent)<99))
```

I know these examples aren’t easy, but so are a lot of the xml gauges. It often helps to scribble down step by step how the stack of a specific gauge works.

Apendices:

- a_vars.txt: a list of possible (A:...,...) type variables extracted from sim1.dll
- p_vars.txt: a list of possible (P:...,...) type variables extracted from main.dll
- k_events.txt: a list of possible <ClickEvent=...>s and (>K:...) events extracted from controls.dll
- units.txt: a list of possible conversion units (NOTE not all conversion fit to all value, ”number” seems to work always) extracted from util.dll
- colorst.txt: a list of possible colors extracted from acontain.dll
- xmloptions.txt: a list of possible ”tags” extracted from panels.dll
- rpn.zip two files from the HP webside dealing with ”Reverse Polish Notation” (RPN)
- Format Specification Fields.zip a copy from the VC++ help file dealing with C/C++ style formatting, it might have certain value for ”String”s, not all formats are allowed in XML gauges though.

There is no limit for using the information included in this document, it is however nice to mention the author if used for your own projects.

The document itself and all attached documents aren’t allowed to be included on any commercial CD/DVD or other commercial media without the written permission of the author. The publishing on freeware internet sites, is allowed without restrictions.

Final Note: No warranty at all for these lists, they are all made by the motto: ”parts of it fit so maybe also the rest” so don‘t wonder if they don’t work.

Copyright 2002 Arne Bartels.
arne.bartels@nwn.de