

This module contains a group of C++ classes aimed to provide extra functions to XML gauges.

Some of these classes are in fact development tools for XML programmers while others were planned to enhance what is currently available for XML panel's development.

Conceptually, the intention in this case is to gather different C++ callback functions in a single library (.dll) that is loaded by FS at startup, so they are available to any panel without the need of adding individual C++ gauges and/or modules to do the task. Future versions of XMLTools might include sound, advanced graphics and any other specific functionality users may come up with and request.

A group of example and utility gauges are also provided with this release as a guide on how to properly make XML code interact with each one of the module classes.

XMLTools classes:

Included in v2.01:

XMLVARS v2.0: a class to handle custom variables, including strings, in single/array format.

LOCALVARS v1.0: a class to manage data from local (L:) variables actives in the current flight.

XMLEVENTS v1.1: a class that enhances the management of standard and custom events.

SIMVARS v1.1: a class that enables direct writing of specific aircraft (A:) variables.

XMLTABLES v1.0: a class used to extract values from nonlinear/multicolumn tables.

XMLKEYS v1.0: a class that enables custom key events.

LOGGER v2.0: Robbie McElrath's class for handling file I/O using XML code.

How to install in FSX

Run ***XMLToolsv201 Installer.exe*** included in the package.

Manual installation

Copy **XMLTools.dll** into FSX's main folder. Next, locate `dll.xml` file which should be in:

`C:\Users\username\AppData\Roaming\Microsoft\FSX\`

or

`C:\Documents and Settings\username\Application Data\Microsoft\FSX\`

Edit the file and add the following within `<Simbase.Document>`

```
<Launch.Addon>
  <Name>XMLTools</Name>
  <Disabled>False</Disabled>
  <ManualLoad>False</ManualLoad>
  <Path>XMLTools.dll</Path>
  <DllStartName>module_init</DllStartName>
  <DllStopName>module_deinit</DllStopName>
</Launch.Addon>
```

Important: if `XMLVars.dll` module is installed, just locate `XMLVars` entry and replace `XMLVars` and `XMLVars.dll` names with `XMLTools` and `XMLTools.dll`, as both modules must not be loaded at the same time. Also if `Logger.dll` is installed, remove its entry from the file.

XMLVARS v2.0

This class enables the use of custom variables -kind of LVars- for storing and retrieving data across the entire gauge set of an aircraft panel. Like LVars, these variables can have any name, can hold numeric values of any kind and can also handle string values. Besides, they can handle **standard* multidimensional array-style manipulation of data*, a powerful tool that is not possible to work with through the native LVarset.

This version supports *memory-limited* number of variables, with names up to 64 char, numeric values of type *FLOAT64* and string values up to 128 char. It can minimize the use of repetitive code in complex gauges like FMS, Navigation Display, TCAS and similar as one of its strongest points is the ability to maintain variable values across the different gauges, (like LVars do). New with this version is the ability to set the visibility of the variable values as *Local* to the current session, or *Global* –by default- which means values are retained upon reloading the aircraft.

XMLVars v2.0 is fully compatible with previous versions available in *XMLVars.dll* module. Current users of *XMLVars v1.x* must do proper replacements in *dll.xml* file as indicated in *XMLTools'* install instructions.

Tech note

Unless a reset command is used, names and values of custom variables are shared across the different gauges in where they are referenced, as *only one instance* of XMLVars class is created per aircraft panel. These variables can be defined in any XML or C++ gauge (*), but cannot be used inside a model (.MDL) file as this one doesn't support custom DLL callbacks.

Variable names and values are internally stored within an array structure of dynamic size. Starting with 1024 elements, it expands automatically in sections of 1024 elements each as soon as the current section is completely filled; resulting in a final size only limited by physical memory.

(*) using a C++ specific code style.

XMLVARS constant identifiers

A group of *constant identifiers* are used to communicate between an XML gauge and the class' internal array (or *database*):

(C:XMLVARS:StoreVarName,string) - Write only

This identifier is used to store the name of a custom variable in a record of the database.

For example, 'MyCustomVar' (>C:XMLVARS:StoreVarName,string) will search the database for a record ID equal to 'MyCustomVar'; if found then the index to that record is set in the module, otherwise the variable name is added to the database and the corresponding index value is set in the module. A variable name may contain any alphanumeric character (case insensitive) except "'" (single quote), "" (double quote) and "\" (escape); char ">" and spaces are ignored.

Then 'MyCustomVar' and 'MYCUSTOM VAR' are considered the same variable name.

(C:XMLVARS:SearchVarName,string) - Write only

This identifier is used to search for the name of a custom variable in the database.

For example, 'MyCustomVar' (>C:XMLVARS:SearchVarName,string) will search the database in a similar way as in the above example, but in this case if not found the index will point to EOF.

(C:XMLVARS:NumberValue,number) - Read/Write

(C:XMLVARS:StringValue,string) - Read/Write

These identifiers are used to store/retrieve the value of a custom var in/from a record of the database.

(C:XMLVARS:NumberValue,number)

holds numeric values and

(C:XMLVARS:StringValue,string)

strings of up to 128 alphanumeric chars (including spaces and excluding single and double quotes)

For example,

(A:FLAPS HANDLE POSITION,percent) (>C:XMLVARS:NumberValue,number)

will store the content of an FSX variable into a database's record in the current index position.

'This is a test' (>C:XMLVARS:StringValue,string)

will store a string of text using the same logic.

Likewise,

(C:XMLVARS:NumberValue,number) (>L:MyLocalVar,number)

will retrieve a value stored in the database, in the current index position, and assign it to

(L:MyLocalVar,number);

and

<String>%((C:XMLVARS:StringValue,string))%s!</String>

will display a text stored in the current index position.

(C:XMLVARS:NumberOfRecords,number) - Read only

This identifier is used to retrieve the total number of records that have a variable name assigned.

(C:XMLVARS:CurrentSize,number) - Read only (v2.0)

This identifier retrieves the current number of elements defined in the internal array of custom vars.

The internal array handles 1024 elements by default, and is dynamically expanded in chunks of 1024 elements when a new limit is reached.

(C:XMLVARS:Reset,number) - Write only

This identifier is used to clear the database and reset all the variable names to (empty) and values to default (0 for numbers and (empty) for strings).

(C:XMLVARS:XMLVarID,number) - Write only (v2.0)

This identifier is used to set the custom vars' array index to a desired position. Must be an integer, positive number between 0 and **(C:XMLVARS:NumberOfRecords,number) - 1** otherwise the index will point to EOF.

(C:XMLVARS:XMLVarName,string) - Read only (v2.0)

This identifier is used to retrieve the name of the custom variable defined in the current index position.

(C:XMLVARS:XMLVarsGlobalMode,bool) - Write only (v2.0)

This identifier sets the visibility of custom variables when reloading the aircraft. A value of 0 sets them to Local mode so their names and values are reset after an aircraft reload, the same of what happens with Local Vars (L:). A value of 1 (default) sets them to Global mode, meaning both names and values are retained after an aircraft reload.

Proper usage

The correct syntax for each kind of operation is:

-To store a numeric value:

```
'MyVarName' (>C:XMLVARS:StoreVarName,string)
nValue (>C:XMLVARS:NumberValue,number)
```

-To store a text string:

```
'MyVarName' (>C:XMLVARS:StoreVarName,string)
'This is a text' (>C:XMLVARS:StringValue,string)
'C:\\MyPath\\MyFile.ext' (>C:XMLVARS:StringValue,string) (*)
(*) to store a path, a double "\" (escape) char must be used.
```

-To retrieve a numeric value:

```
'MyVarName' (>C:XMLVARS:SearchVarName,string)
(C:XMLVARS:NumberValue,number)
```

If 'MyVarName' doesn't exist in the database, an empty (0) value is retrieved.

-To retrieve a text string:

```
'MyVarName' (>C:XMLVARS:SearchVarName,string)
(C:XMLVARS:StringValue,string)
```

If 'MyVarName' doesn't exist in the database, an empty (") text is retrieved.

-To retrieve the total number of records (variables) defined and store the value into a register:

```
(C:XMLVARS:NumberOfRecords,number) sp0
```

Then l0 might contain 0, 1, 23, 55, 1234, etc.

-To retrieve the current number of records (slots) available in the internal array of custom vars:

```
(C:XMLVARS:CurrentSize,number) sp0
```

Then l0 possible values are 1024, 2048, 3072, 4096, etc.

-To reset the database (for compatibility, not really necessary since this version):

```
(>C:XMLVARS:Reset,number)
```

-To retrieve the name of a variable giving its index position:

```
34 (>C:XMLVARS:XMLVarID,number)
(C:XMLVARS:XMLVarName,string) sp0
```

Then l0 might contain any valid name, ie. 'MyVar1'.

-To retrieve the value of a variable giving its index position:

```
34 (>C:XMLVARS:XMLVarID,number)
(C:XMLVARS:NumberValue,number) or (C:XMLVARS:StringValue,string)
```

-To set the visibility of custom variables after an aircraft reload:

```
0 (>C:XMLVARS:XMLVarsGlobalMode,number) - Reset names and values
1 (>C:XMLVARS:XMLVarsGlobalMode,number) - Retain names and values (*)
(*) names and values are lost if the user resets (CTRL+;) the current flight.
```

-Assign values to array-style variables:

```
'MyVarName' 0 scat (>C:XMLVARS:StoreVarName,string)
'This is a text in var' 0 scat (>C:XMLVARS:StringValue,string)

5 sp0
'MyVarName' 10 scat (>C:XMLVARS:StoreVarName,string)
'This is a text in var' 10 scat (>C:XMLVARS:StringValue,string)

23 (>L:OtherVar,number)
'MyVarName' (L:OtherVar,number) int scat (>C:XMLVARS:StoreVarName,string)
'Text in var' (L:OtherVar,number) int scat (>C:XMLVARS:StringValue,string)
```

Then

```
MyVarName0 contains 'This is a text in var0'
MyVarName5 contains 'This is a text in var5'
MyVarName23 contains 'Text in var23'
```

Examples

Here is a set of useful macros that should make it easier to work with:

-Single variable macro

```
<Macro Name="V">
  @1 d 0 symb '>' scmi 0 ==
  if{ (>C:XMLVARS:StoreVarName,string) (>C:XMLVARS:@2Value,@2) }
  els{ (>C:XMLVARS:SearchVarName,string) (C:XMLVARS:@2Value,@2) }
</Macro>
```

-Unidimensional array macro

```
<Macro Name="Val">
  @1 d 0 symb '>' scmi 0 ==
  if{ @3 scat (>C:XMLVARS:StoreVarName,string) (>C:XMLVARS:@2Value,@2) }
  els{ @3 scat (>C:XMLVARS:SearchVarName,string) (C:XMLVARS:@2Value,@2) }
</Macro>
```

-Bidimensional array macro

```
<Macro Name="Va2">
  @l d 0 symb '>' scmi 0 ==
  if{ @3 scat ':' scat @4 scat (>C:XMLVARS:StoreVarName,string)
    (>C:XMLVARS:@2Value,@2) }
  els{ @3 scat ':' scat @4 scat (>C:XMLVARS:SearchVarName,string)
    (C:XMLVARS:@2Value,@2) }
</Macro>
```

And the usage:

```
'Any Text' @V('>MYFIRSTstring',string)
or
nNumber/LVar/Avar @V('>MYFIRSTNumber',number)
for storing a single variable.
```

```
@V('MYFIRSTstring',string)
or
@V('MYFIRSTNumber',number)
for retrieving a single variable.
```

```
'Any Text' @Val('>MYFIRSTstring:',string,n)
for storing an unidimensional string array var, where n could be a static integer number, a register ID (I0,I1,etc) or a variable value (A:var,L:var) converted to integer.
```

```
nNumber/LVar/Avar @Va2('>MYFIRSTNumber:',number,n1,n2)
for storing a bidimensional number array var, where n1 and n2 can be static integer numbers, register IDs (I0,I1,etc) or variable values (A:var,L:var) converted to integers.
```

Code examples

XMLVarsExample.xml is available. Please check XMLTools_Examples.pdf

LOCALVARS v1.0

This class can be used to inspect the complete set of local variables (L:) defined in the flight's current session, including model (.MDL) exclusive (L:) variables and those that might be dynamically created at runtime.

Local variables are internally referenced by the class through a dynamic array of IDs, names, values and units. Saving them into an array brings special functionality: values can be restored after an aircraft's reload; sort and filter modes are also available, values and units are editable at run time, and much more.

Tech note

LOCALVARS identifiers are shared across the different gauges in where they are referenced, as *only one instance* of LOCALVARS class is created per aircraft panel. These identifiers can be defined in any XML or C++ gauge (*), but cannot be used inside a model (.MDL) file as this one doesn't support custom DLL callbacks.

(*) using a C++ specific code style

LOCALVARS constant identifiers

These are the *constant identifiers* defined to enable communications between any XML gauge and the class' internal array of LVars:

(C:LOCALVARS:LVarLinePos,enum) - Read/Write

This identifier has the following functions:

-*Read* mode: retrieves the LVar array's current index position.

-*Write* mode: sets the LVar array's index position.

(C:LOCALVARS:LVarID,enum) - Read/Write

This identifier works as follows:

-*Read* mode: retrieves an LVar's internal ID from the current index position.

-*Write* mode: sets the index position related to the ID assigned.

(C:LOCALVARS:LVarName,string) - Read only

This identifier is used to retrieve the name of the LVar in the current index position.

(C:LOCALVARS:LVarUnit,string) - Read/Write

This identifier can be used:

-in *Read* mode, to retrieve the unit's name of the LVar in the current index position. Units default to "number" in all cases.

-in *Write* mode, to set the unit's name of the LVar in the current index position.

(C:LOCALVARS:LVarValue,number) - Read/Write

This identifier can be used:

-in *Read* mode, to retrieve the value of the LVar in the current index position, expressed according to the units defined for that variable.

-in *Write* mode, to update the value of the LVar in the current index position, expressed according to the units defined for that variable.

(C:LOCALVARS:LVarSelected,bool) - Read/Write

This identifier:

-in *Read* mode returns the status of the LVar in the current index position, as Selected (1) or Deselected (0-default).

-in *Write* mode sets the status of the LVar in the current index position, as Selected (1) or Deselected (0).

(C:LOCALVARS:NumberOfLVars,number) - Read only

This identifier is used to retrieve the total number of LVars defined in the current flight session.

(C:LOCALVARS:NumberOfLVarsFiltered,number) - Read only

This identifier is used to retrieve the number of LVars returned by the current filter.

(C:LOCALVARS:LVarsOrderMode,number) - Read/Write

This identifier can be used:

-in *Read* mode, to retrieve the current sort order of the LVars array.

-in *Write* mode, to command a sort of the LVars array. Current modes are:

- 0 - No sort (default)
- 1 - Name, ascending order
- 2 - Name, descending order
- 3 - Value, ascending order (lower to greater)
- 4 - Value, descending order (greater to lower)

(C:LOCALVARS:LVarsFilterMode,number) - Read/Write

This identifier can be used:

-in *Read* mode, to retrieve the current filter mode of the LVars array.

-in *Write* mode, to apply a filter to the LVars array. Current modes are:

- 0 - No filter (default)
- 1 - Values greater than 0
- 2 - Values lower than 0
- 3 - Values equal to 0
- 4 - Values equal to 1
- 5 - Values that toggled from 0 to 1 in the current cycle
- 6 - Values that toggled from 1 to 0 in the current cycle
- 7 - Values that changed in the current cycle vs. the previous one.

(C:LOCALVARS:LVarsSaveToLog,bool) - Write only

This identifier saves the names and values of the LVars defined in the current flight session to a text file named `XMLTools_LVarsLog.txt`, and located in the same folder as XMLTools module (normally FSX's main folder).

(C:LOCALVARS:LVarsGlobalMode,bool) - Write only

This identifier sets the visibility of Local (L:) variables when reloading the aircraft. A value of 0 (default) sets them to *Local* mode so their values are reset after an aircraft reload, what indeed happens in standard FSX's operation. A value of 1 set them to *Global* mode, meaning values are retained after an aircraft's reload.

(C:LOCALVARS:LVarsGlobalModeException,string) - Write only

This identifier allows defining the name, or part of a name, to be used as a filter when restoring LVar values after an aircraft reloads in *Global* mode. Then, all those variables that contain this string as part of their names will reset to 0, like in standard FSX's operation.

Proper usage

-To retrieve the ID, name and value of a variable in the index position 5, with default order and filter:

```
5 (>C:LOCALVARS:LVarLinePos,enum)
(C:LOCALVARS:LVarID,enum) sp0
(C:LOCALVARS:LVarName,string) sp1
(C:LOCALVARS:LVarValue,number) sp2
(C:LOCALVARS:LVarUnit,string) sp3
```

If name is "MyLocal_Var" and value is 45.23 then:

```
10 contains 5
11 contains 'MyLocal_Var'
12 contains 45.23
13 contains 'number'
```

LVarLinePos and LVarID return the same value when there is no order or filter selected. Otherwise, LVarLinePos returns the current index position and LVarID the internal ID assigned to the variable by the simulator.

-To update a variable value that has an ID of 15 and unit "number":

```
15 (>C:LOCALVARS:LVarID,enum)
10 (>C:LOCALVARS:LVarValue,number)
```

-To update the unit of that variable and check its new value:

```
'feet' (>C:LOCALVARS:LVarUnit,string)
(C:LOCALVARS:LVarValue,number) sp0
```

Now 10 contains 32.8084, which is the original value with "number" unit converted to feet.

-To sort the LVars array alphabetically, ascending order:

```
<Click>
  1 (>C:LOCALVARS:LVarsOrderMode,number)
</Click>
```

-To filter the LVars with values equal to 1:

```
<Click>
  4 (>C:LOCALVARS:LVarsFilterMode,number)
</Click>
```

To avoid making a constant sort/filter on each gauge cycle, sort and filter commands should be placed within <Area> structures, or limited by flag variables as to make them run in a single action.

When a filter is active, setting the LVars array index with `(C:LOCALVARS:LVarLinePos,enum)` might return a different position than the assigned, depending on whether that position holds a variable included in the filter selection or not. If it is not included, the index will reposition to the next variable that meets the filter condition, or EOF otherwise.

-To save LVars names and values into a log file:

```
<Click>  
  1 (>C:LOCALVARS:LVarsSaveToLog,bool)  
</Click>
```

Data is saved to `XMLTools_LVarsLog.txt` . If the file exists, its content is removed and new data is appended from the beginning.

-To set the visibility of LVars as Global:

```
1 (>C:LOCALVARS:LVarsGlobalMode,bool)
```

Values are restored after the current aircraft is reloaded both by `RELOAD_USER_AIRCRAFT` event or by using the *Select Aircraft...* menu option. This last one makes it possible to restore the variables and its values into a different aircraft, which might be useful when working with different versions of the same model.

-To define an exclusion filter for Global mode:

```
'init' (>C:LOCALVARS:LVarsGlobalModeException,string)
```

Then values of all those variables containing *'init'* as part of their names won't be restored after a reload. Might be useful in cases where some special initialization is needed, for example related to certain aircraft variables (A:) values at startup, event captures, etc.

Code examples

`ReloadPanels.xml` and `LocalVarsLogger.xml` are available.
Please check `XMLTools_Examples.pdf`

XMLEVENTS v1.1

This class can be used to handle XML event traps in any view mode; trigger and trap custom events and also keep record of events being triggered in the current flight's session.

Code that would normally be included in <On Event> commands is actually exported and stored in a dynamic array of the class. Once an event is triggered, that code is processed by a single, internal event handler, giving the same result as it was captured by an XML equivalent command.

Tech note

XMLEVENTS identifiers are shared across the different gauges in where they are referenced, as *only one instance* of XMLEVENTS class is created per aircraft panel. These identifiers can be defined in any XML or C++ gauge (*), but cannot be used inside a model (.MDL) file as this one doesn't support custom DLL callbacks.

There are two different arrays defined by the class. One is used to store executable code that is going to be processed by the event handler's internal routine, and the other keeps record of events recently captured, up to a maximum of 100 triggered events per logged session.

(*) using a C++ specific code style

XMLEVENTS constant identifiers

These are the *constant identifiers* used to enable communications between any XML gauge and the class' internal arrays of selected/captured events:

(C:XMLEVENTS:EventName,string) - Write only

This identifier is used to store the name of an event in the internal array of selected events. A complete list of XML event names that can be assigned is available in `XMLEvents_reference.txt`, included along with this document. Custom events can be stored as well using hex values as names that must be within a certain range (see *Proper usage*).

(C:XMLEVENTS:EventString,string) - Write only

This identifier associates a script text to an event name in the internal array of selected events. When FS triggers the actual event, that script will be internally parsed and processed like a typical XML script from <Update>, <Value> and other similar sections.

(C:XMLEVENTS:ClearEvents,number) - Write only

This identifier resets the internal array of selected events. Reloading the aircraft or doing a flight's reset have the same effect.

(C:XMLEVENTS:EventsSize,number) - Read only

This identifier returns the number of selected events defined in the internal array.

(C:XMLEVENTS:CustomEventID,number) - Write only

This identifier assigns a custom event to be triggered by `CustomEventValue` property.

(C:XMLEVENTS:CustomEventValue,number) - Write only

This identifier triggers a custom event previously referenced by `CustomEventID` property, using the assigned value as its data source.

(C:XMLEVENTS:CaptureEvents,bool) - Read/Write

This identifier is used:

-in *Read* mode, to return the current status of events' inspection. True (1) means events are being captured and added to the inspection array; False (0) is the default option.

-in *Write* mode, to (1) enable/ (0) disable events' inspection routine.

(C:XMLEVENTS:CapturedEventData,string) - Read only

This identifier retrieves a string of text including the name and value of an event from the internal array of captured events. The format of that string is shown in the `XMLEventsLogger.xml` utility included with this package. A string containing a single `"#"` character is returned as a separator between event captures.

Proper usage

Code scripts used within `<On Event>` commands must be stored in the class' internal array before they can be recognized by its own event handler.

For example, this typical structure:

```
<On Event="FLAPS_DOWN">
    ScriptLine1
    ScriptLine2
    ScriptLine3
    etc...
</On>
```

must be replaced with:

```
<Update>
    'FLAPS_DOWN' (>C:XMLEVENTS:EventName,string)
    'ScriptLine1
    ScriptLine2
    ScriptLine3
    etc...' (>C:XMLEVENTS:EventString,string)
</Update>
```

Both the event name and the event's complete script must be enclosed in single quotes.

A code script may be written in one or multiple lines. Macros and `<!--comments -->` are also supported, but Gauge vars (G:) and properties from this and other custom classes (for example, XMLVars, SIMVARS, etc) are not.

Scripts are loaded only once into the class' internal array, so there is no harm putting them inside an `<Update>` or `<Value>` command. However, to avoid unnecessary parsing on every gauge's cycle, it is preferable to program a gauge that handles all the event traps and makes a single load on startup (see `EventHandler.xml` example).

Values assigned to an event can also be recognized by the class' event handler by using a specific convention, in this case the name of the event preceded by a `*` character.

For example, in a typical mouse click snippet from gauge*NN*:

```
<Click> ... 10200 (>K:THROTTLE_SET) ... </Click>
```

And in the Event gauge:

```
<Update>
  'THROTTLE_SET' (>C:XMLEVENTS:EventName,string)
  '*THROTTLE_SET' 15000 ==
    if{ do something1 }
  *THROTTLE_SET 10200 ==
    if{ do something2 }
    else{ do something else }' (>C:XMLEVENTS:EventString,string)
</Update>
```

The internal event handler will replace ***THROTTLE_SET** with the received value, 10200 in this case.

Custom events are also supported and can be both triggered and captured.

For example, from a modeldef.xml file:

```
<MouseRect>
  <Cursor>Hand</Cursor>
  <MouseFlags>LeftSingle</MouseFlags>
  <EventID>0x11023</EventID>
</MouseRect>
```

And in the Event gauge, a custom event's name is stored with a capital "C" before the 0x representation of the number:

```
<Update>
  'C0x00011023' (>C:XMLEVENTS:EventName,string)
  'ScriptLine1... etc...' (>C:XMLEVENTS:EventString,string)
</Update>
```

To trigger a custom event with a mouse click, use the hex representation *without* a capital C:

```
<Click>
  0x00011023 (>C:XMLEVENTS:CustomEventID,number)
  25 (>C:XMLEVENTS:CustomEventValue,number)
</Click>
```

And the passed value can also be identified by the event handler:

```
<Update>
  'C0x00011023' (>C:XMLEVENTS:EventName,string)
  '*C0x00011023' 25 ==
    if{ play sound 1 }
  *C0x00011023 30 ==
    if{ play sound 2 }' (>C:XMLEVENTS:EventString,string)
</Update>
```

Custom events must use the 0x00011000 to 0x0001FFFF range of hex numbers to be recognized by the internal event handler.

Examples

To simplify the use of this class, the following macros might help a lot:

-A mnemonic replacement of <On Event="FLAPS_UP">:

```
<Macro Name="FLAPS_UP">  
    Script...  
</Macro>
```

Now a macro for storing event scripts:

```
<Macro Name="OnEvent">  
    '@1' (>@e:EventName,string) '@2' (>@e:EventString,string)  
</Macro>
```

And the usage:

```
<Update>  
    @OnEvent(FLAPS_UP,@FLAPS_UP)  
</Update>
```

First parameter is the name of the event, followed by the same name preceded by "@" (see EventHandler.xml).

Code examples

EventHandler.xml and XMLEventsLogger.xml are available.
Please check XMLTools_Examples.pdf

History

V1.1: Bug correction for negative values improperly returned

SIMVARS v1.1

This class allows direct writing to aircraft (A:) variables using XML code. Not all the simulation variables described in FSX/ESP 1.0 SDKs accept this method; only those listed with “Y” in the [SETTABLE] column may receive custom user values.

Tech note

SIMVARS works by opening a single internal *Simconnect* client that is shared across the different gauges, as *only one instance* of the class is created per aircraft panel. Like happens with other XMLTools classes, SIMVARS identifiers cannot be used inside a model (.MDL) file.

Identifiers (“variable names”) that are referenced in single/multiple gauges are actually stored in a class’ internal array with the purpose of providing steady simulation data. This is necessary because, as *Simconnect* operates in asynchronous mode, values returned by direct reading of an updated (A:) variable tend to flicker a bit; so it becomes preferable to use its related SIMVARS identifier for displaying custom data (see *Proper usage*).

Because of Simconnect’s operation characteristics, some settable (A:) variables might not perform as expected when updated with custom values, principally those that depend on forward calculations from other variables.

SIMVARS constant identifiers

Check `SIMVars_reference.txt` for a list of supported variable names and units to be used as *constant identifiers*. Basic structure is:

(C:SIMVARS:Variable_Name,Variable_Unit) - Read/Write

This identifier is used:

- in *Write* mode, to update the equivalent (A:) variable with custom values.
- in *Read* mode, to return the latest value assigned to the equivalent (A:) variable.

The first variable described in `SIMVars_reference.txt` can be used to check the current view:

(C:SIMVARS:Current View Mode,Enum) - Read/Write

-in *Read* mode, returns the current view. Possible values are:

- 0-External View
- 1-2D Cockpit View
- 2-Virtual Cockpit View
- 4-Map View

-in *Write* mode, it does nothing (dummy mode).

The second variable can be used to check if the sim is active or in pause:

(C:SIMVARS:Pause State,Enum) - Read Only

- Possible values are:

- 0-Sim is active
- 1-Sim is in pause

The third variable can be used to check for sound's current status:

(C:SIMVARS:Sound State,Enum) - Read Only

- Possible values are:

- 0-Sound is active
- 1-Sound is off

Proper usage

-To update an aircraft variable, for example (A:Turb Eng N1:1,percent), write it as:

```
<Update/Value/etc>
    45 (>C:SIMVARS:Turb Eng N1:1,percent)
<Update/Value/etc>
```

To read the same custom updated variable:

```
<Element>
    <Position .../>
    <Text...>
        <String>%((C:SIMVARS:Turb Eng N1:1,percent))%!3.2f!</String>
    </Text>
</Element>
```

To replicate data from an aircraft variable into its corresponding SIMVARS' internal variable:

```
<Update/Value/etc>
    (A:Turb Eng N1:1,percent) (>C:SIMVARS:Turb Eng N1:1,percent)
<Update/Value/etc>
```

Code examples

TableAndSimExample.xml is available.

Please check XMLTools_Examples.pdf

History

V1.1: Added Pause and Sound status detection.

XMLTABLES v1.0

This class is used to extract interpolated values from nonlinear/multicolumn tables.

Tech note

XMLTABLES identifiers are shared across the different gauges in where they are referenced, as *only one instance* of XMLTABLES class is created per aircraft panel. As previously stated, these identifiers cannot be used inside a model (.MDL) file.

Custom tables are stored in dynamic arrays internally defined by the class following a specific design, and are created using plain XML code from panel gauges. There is no arbitrary limit on the number and/or size of tables that can be processed. Dynamic update of table rows within running gauge cycles is also supported.

Because a table creation is a single, unique procedure an error handler is provided to monitor whether the process was completed successfully.

XMLTABLES constant identifier

These are the *constant identifiers* used to operate with the tables from an XML gauge:

(C:XMLTABLES:TableBegin,bool) - Write only

This identifier is used to set the starting point of a new table. As it is a flag, no value needs to be assigned. If it is duplicated while the current table creation has not been finished, returns an error that can be checked using `TableError` identifier.

(C:XMLTABLES:TableAddName,string) - Write only

This identifier stores the name of a new table. Must come after `TableBegin` identifier. If another table exists with the same name, or if it is duplicated within the current table creation, returns an error that can be checked using `TableError` identifier.

(C:XMLTABLES:TableAddHeader,string) - Write only

This identifier adds a string containing header (X-axis) values for the table being created. Must come after `TableAddName` identifier. Only one header string must be added per table, otherwise an error returns and can be checked using `TableError` identifier.

(C:XMLTABLES:TableAddRow,string) - Write only

This identifier adds a string containing row values for the table being created. Must come after `TableAddHeader` identifier. Each row must contain a vertical reference (Y-axis) value on its first column, and resulting values on the rest of the columns. The number of columns of each row added must be the same and must coincide with the number of columns defined in the header, excepting the first column which is in fact the vertical reference value. There is no limit on the number of rows that can be added to a table. Using this identifier outside a table creation schema returns an error that can be checked using `TableError`.

(C:XMLTABLES:TableEnd,bool) - Write only

This identifier sets the end point of a new table. Must come after `TableAddRow` identifier. As it is a flag, no value needs to be assigned. A table is not completely saved until this identifier is used to indicate that. If issued outside a table creation schema returns an error that can be checked using `TableError`.

(C:XMLTABLES:TableSelectName,string) - Read/Write

This identifier is used:

-in *Read* mode to retrieve the name of the active table. If there is no active table, an error returns and can be checked using `TableError`.

-in *Write* mode to search for a table name and make it active. If the table is not found, an error returns and can be checked using `TableError`.

(C:XMLTABLES:TableUpdateRow,string) - Write only

This identifier is used to update a single row of the active table. The first column of the updating row must contain a vertical reference (Y-axis) value already present in the row to be updated; if that row is not found or there is no active table, an error returns and can be checked using `TableError` identifier.

(C:XMLTABLES:TableTopRef,number) - Write only

This identifier sets the value to be used as a header reference (X-value) for the active table.

(C:XMLTABLES:TableLeftRef,number) - Write only

This identifier sets the value to be used as a vertical reference (Y-value) for the active table.

(C:XMLTABLES:TableReturnValue,number) - Read only

This identifier retrieves the interpolated value that was obtained from `TableTopRef` and `TableLeftRef` references. If there is no active table, an error returns that can be checked using `TableError`.

(C:XMLTABLES:TableError,number) - Read only

This identifier retrieves the last error returned by the table creation system. Possible values are:

- 0 - No error
- 100 - No table defined
- 110 - Table not found
- 200 - Table not ready (`TableBegin`)
- 210 - Table not ready (`TableAddName`)
- 220 - Table not ready (`TableAddHeader`)
- 230 - Table not ready (`TableAddRow,TableUpdateRow`)
- 240 - Table not ready (`TableEnd`)
- 310 - Table not ready invalid name (`TableAddName,TableSelectName`)
- 320 - Table not ready invalid header (`TableAddHeader`)
- 330 - Table not ready invalid row (`TableAddRow,TableUpdateRow`)
- 400 - Table not ready invalid table (`TableSelectName`)

Proper usage

Due to the characteristics of table creations, it is preferable to put them all in a single, virtual cockpit gauge within an `<Update>` command, and include a `quit` operator to avoid processing the script once it is initialized.

Examples

This is a copy of default B737-800 .air file table 1502 – Turbine CN1 vs. CN2 and Mach N°- extracted using *Aircraft Airfile Manager V2.2*

	M0	M0.9
CN2	0	0.9
0	0	0
10	1.1	1
20	2.7	2.4
30	5.7	4.7
40	10.1	8.4
50	17.5	12.8
60	29.9	21.2
70	49.4	41.6
80	67.5	60.4
90	90.6	85.9
100	105.5	102.4
110	118	114.1

To configure this table, follow this method:

-In (gauge00) start a table creation process:

```
(>C:XMLTABLES:TableBegin,bool)
```

-Give a name to the table, for example CN1vsCN2:

```
'CN1vsCN2' (>C:XMLTABLES:TableAddName,string)
```

-Assign a header, in this case 0 for Mach 0 and 0.9 for Mach 0.9:

```
'0, 0.9' (>C:XMLTABLES:TableAddHeader,string)
```

-Now add the rows:

```
'0, 0, 0' (>C:XMLTABLES:TableAddRow,string)
'10, 1.1, 1' (>C:XMLTABLES:TableAddRow,string)
'20, 2.7, 2.4' (>C:XMLTABLES:TableAddRow,string)
'30, 5.7, 4.7' (>C:XMLTABLES:TableAddRow,string)
'40, 10.1, 8.4' (>C:XMLTABLES:TableAddRow,string)
'50, 17.5, 12.8' (>C:XMLTABLES:TableAddRow,string)
'60, 29.9, 21.2' (>C:XMLTABLES:TableAddRow,string)
'70, 49.4, 41.6' (>C:XMLTABLES:TableAddRow,string)
'80, 67.5, 60.4' (>C:XMLTABLES:TableAddRow,string)
'90, 90.6, 85.9' (>C:XMLTABLES:TableAddRow,string)
'100, 105.5, 102.4' (>C:XMLTABLES:TableAddRow,string)
'110, 118, 114.1' (>C:XMLTABLES:TableAddRow,string)
```

-And finish the process:

```
(>C:XMLTABLES:TableEnd,bool)
```

-Finally, to obtain a CN1 value in any gauge (gauge00 or gauge*NN*), first select the table's name:

```
'CN1vsCN2' (>C:XMLTABLES:TableSelectName,string)
```

-Then assign a header value:

```
(A:AIRSPPEED MACH,Mach) (>C:XMLTABLES:TableTopRef,number)
```

-And also CN2 value for an engine, be ENG1 in this case:

```
(A:TURB ENG CORRECTED N2:1,percent) (>C:XMLTABLES:TableLeftRef,number)
```

-Now to obtain CN1 data for the same engine:

```
(C:XMLTABLES:TableReturnValue,number) is ENG1 CN1 interpolated value, in percent
```

Code examples

TableAndSimExample.xml is available.

Please check XMLTools_Examples.pdf

XMLKEYS v1.0

This class can be used to handle keyboard and joystick traps with custom XML code triggered in any view mode.

Tech note

XMLKEYS identifiers are shared across the different gauges in where they are referenced, as *only one instance* of XMLKEYS class is created per aircraft panel. They can be used only in regular panel gauges and are not allowed inside a model (.MDL) file.

A single array is used to store executable code that is going to be processed by the key/controller handler's internal routine. Key capture takes precedence over XML <On Key> command.

Important: XMLKEYS uses *Simconnect* functions to interact with FSX's engine. Keys/controllers custom configured with FSUIPC have priority over XMLKEYS and will perform according to FSUIPC definition.

XMLKEYS constant identifiers

These are the *constant identifiers* used to enable communications between any XML gauge and the class' internal array of key/controller's executable scripts:

(C:XMLKEYS:KeyName,string) - Write only

This identifier is used to store the name of a key/combination of keys/joystick controller in the internal array of captured keys when referenced for the first time, or used to select a key/combination of keys/joystick controller and make it the currently operative for the rest of the properties.

Keyboard definitions can include a maximum of two modifiers (**Shift,Ctrl,Alt**) and two keys, case sensitive, joined by a plus (+) sign. Joysticks adopt the form **joystick:n:input[:i]**; where **n** is the joystick number (starting from 0), **input** is the joystick part/button's name (*Button,Slider,etc*) and **i** is an optional index number that might be required by the input name (for example *joystick:0:button:0*). Please check *XMLKeys_reference.txt* for further information.

(C:XMLKEYS:KeyString,string) - Write only

This identifier associates a script text to a key/controller name in the internal array of mapped captures. When a mapped key/controller operation is detected, the associated script will be executed instead of the default key/controller's assigned action, providing the capture is active.

(C:XMLKEYS:KeyValue,number) - Read only

This identifier contains the value returned by a captured key/controller's action. Useful only for joystick events other than button events (*sliders, POV, etc*). Please check *XMLKeys_reference.txt* for further information.

(C:XMLKEYS:KeyCaptureOn,bool) - Write only

This identifier enables –makes active- the capture of the key/controller's currently selected.

(C:XMLKEYS:KeyCaptureOff,bool) - Write only

This identifier cancels the capture of the key/controller's currently selected. Key/controller's default action and <On Key> command become actives.

Proper usage

Text scripts used in XMLKEYS must be stored in the class' internal array before they can be recognized by its own key handler.

For example, capture the “a” key and make it turn autopilot ON:

```
'a' (>@k:KeyName,string)
'(>K:AUTOPILOT_ON)' (>@k:KeyString,string)
(>@k:KeyCaptureOn,bool)
```

The capture has been configured and made active with the `KeyCaptureOn` identifier. To stop capturing the key –make it inactive, reverting to FS' default key behavior:

```
'a' (>@k:KeyName,string)
(>@k:KeyCaptureOff,bool)
```

Both key's name and key's complete script must be enclosed in single quotes:

```
'Shift+b' (>@k:KeyName,string)
'Ctrl+Shift+z' (>@k:KeyName,string)
'(L:Test1,bool) ++ (>L:Test1,bool)' (>@k:KeyString,string)
```

A key script may be written in one or multiple lines. Macros and `<!--comments -->` are also supported, but Gauge vars (G:) and properties from this and other custom classes (for example, XMLVars, SIMVARS, etc) are not.

Code examples

`KeysHandler.xml` is available.

Please check `XMLTools_Examples.pdf`

LOGGER v2.0

Author: Robbie McElrath

Original documentation: Robert “Bob” McElrath

This class provides file read and write capability for XML gauges, allowing hundreds of different number or string variables of any type (A:, E:, P:, G:, L:, or gps) to be written to or read from the hard disk. Also enables the user to write XML Flight Data Recorder gauges, save and load flight plans, save and load initial values for XML gauges that cannot otherwise be saved with Flight Sim's Save Flight, or record flights for subsequent playback as Google Earth Tracks and Tours.

Tech note

LOGGER identifiers handle private data exchange from each one of the different gauges in where they are defined, as a new *instance* of LOGGER class is created per referenced gauge. They can be used only in regular panel gauges and are not allowed inside a model (.MDL) file.

LOGGER reads and writes standard text files. The files can be any text file type such as .txt, .cfg, .ini, or .csv, but not .doc or .xls. The full directory path should be specified in the filename. If only the Filename.ext is specified, LOGGER will look for, or create the file in the root Flight Simulator directory.

LOGGER constant identifiers

These are the *constant identifiers* used to enable communications between any XML gauge and the class' internal file I/O commands:

(C:LOGGER:openAppend,string) - Read/Write

This identifier has the following functions:

-*Read* mode: Returns full path file name of file currently open for writing.

-*Write* mode: Opens the given file for writing. Appends to the end of it. Creates file if it doesn't exist. Writes using Line Format

(C:LOGGER:openCsvAppend,string) - Read/Write

This identifier has the following functions:

-*Read* mode: Returns full path file name of file currently open for writing.

-*Write* mode: Opens the given file for writing. Appends to the end of it. Creates file if it doesn't exist. Writes using CSV Format

(C:LOGGER:openWrite,string) - Read/Write

This identifier has the following functions:

-*Read* mode: Returns full path file name of file currently open for writing.

-*Write* mode: Creates the given file for writing. Erases file if it exists. Writes using Line Format

(C:LOGGER:openCsvWrite,string) - Read/Write

This identifier has the following functions:

-*Read* mode: Returns full path file name of file currently open for writing.

-*Write* mode: Creates the given file for writing. Erases file if it exists. Writes using CSV Format

(C:LOGGER:openRead,string) - Read/Write

This identifier has the following functions:

-*Read* mode: Returns full path file name of file currently open for reading.

-*Write* mode: Opens the given file for reading. Reading assumes File Format.

(C:LOGGER:openCsvRead,string) - Read/Write

This identifier has the following functions:

-*Read* mode: Returns full path file name of file currently open for reading.

-*Write* mode: Opens the given file for reading. Reading assumes CSV Format.

(C:LOGGER:countRead,number) - Read

This identifier returns the number of records read since opening the file. Does not count records bypassed using the skip command.

(C:LOGGER:number,number) - Read/Write

This identifier has the following functions:

-*Read* mode: Reads a number from the readable file.

-*Write* mode: Writes a number to the writable file.

(C:LOGGER:string,string) - Read/Write

This identifier has the following functions:

-*Read* mode: Reads a string from the readable file.

-*Write* mode: Writes a string to the writable file.

(C:LOGGER:newline,number) - Write

This identifier inserts a new line to the writable file.

(C:LOGGER:eof,number) - Read

This identifier returns 1 if End Of File has been reached, and returns 0 otherwise.

(C:LOGGER:skip,number) - Write

This identifier skips *n* number of records from the current record position.

(C:LOGGER:closeWrite,number) - Write

This identifier closes the file that's open for writing.

(C:LOGGER:closeRead,number) - Write

This identifier closes the file that's open for reading.

(C:LOGGER:delete,string) - Write

This identifier deletes the specified file by removing it to the Recycle Bin.

(C:LOGGER:version,number) - Read

This identifier returns the class' version number.

Proper usage

FILE OPEN and FILE CLOSE commands

The correct syntax for these commands is:

-To open an existing file that was written in standard line format:

```
'C:\Documents and Settings\TEMP\Desktop\LOGGER\File.txt'  
(>C:LOGGER:openRead,string)
```

-Or if a macro is used for the file path:

```
<Macro Name="Path">  
  'C:\Documents and Settings\TEMP\Desktop\LOGGER\  
</Macro>
```

```
@Path 'File.txt' scat (>C:LOGGER:openRead,string)
```

-To create and open a new file then write records to this file using standard line format:

```
@Path 'File.txt' scat (>C:LOGGER:openWrite,string)
```

-To create and open a new file then write records to this file using comma separated value (csv) format:

```
@Path 'File.txt' scat (>C:LOGGER:openCsvWrite,string)
```

-To open an existing file and write records to this file using standard line format, appending new data to the existing records starting after the last record:

```
@Path 'File.txt' scat (>C:LOGGER:openAppend,string)
```

-To open an existing file and write records to this file using comma separated value (csv), appending new data to the existing records starting after the last record:

```
@Path 'File.txt' scat (>C:LOGGER:openCsvAppend,string)
```

-To close the currently open Read file:

```
1 (>C:LOGGER:closeRead,number)
```

-To close the currently open Write file:

```
1 (>C:LOGGER:closeWrite,number)
```

Reading any of the open file variables:

```
(C:LOGGER:openAppend,string)
(C:LOGGER:openCsvAppend,string)
(C:LOGGER:openWrite,string)
(C:LOGGER:openCsvWrite,string)
(C:LOGGER:openRead,string)
(C:LOGGER:openCsvRead,string)
```

returns the full path file name of file currently open for reading or writing. This feature can be used to store the names of initialization or parameter files as they are written, then at the start of a flight, read those file names and pass to an `openRead` operation.

For example, `(C:LOGGER:openWrite,string)` returns the following:

```
C:\Documents and Settings\TEMP\Desktop\LOGGER\WriteFile.txt
```

and

```
(C:LOGGER:openWrite,string) (>C:LOGGER:string,string) writes that file name to file.
```

Reading `(C:LOGGER:openRead)` or `(C:LOGGER:openCsvRead)` is also used to verify that a file was successfully opened. As such, it's a 'best practice' to include in your code

```
@Path 'File.txt' scat (>C:LOGGER:openRead,string)
```

opens *File.txt*, but if that file does not exist, for example because of a typo error in the file name, then the file will not open and reading `(C:LOGGER:openRead)` will return an empty string. Consequently, code similar to the following can be used to verify a successful file open:

```
(C:LOGGER:openRead,string) slen 0 !=
if{ do your code }
els{ 1 (>L:ReadFailure, bool) }
```

or something equivalent.

Data READ and WRITE commands

LOGGER reads and writes number and string records using either standard line format or comma separated value format. Standard line format contains one record per line. Comma separated value format may contain multiple records per line, all separated by commas.

There are several combinations of open file and data read/write instructions.

Please check [LOGGER_File_Read_Write_Formats.pdf](#) for more detailed examples file read and write results.

READ commands - To read individual records from a file:

(C:LOGGER:number,number) – reads a number record from the file that was opened with openRead or openCsvRead commands.

Example: (C:LOGGER:number,number) (>L:OxygenBottle1Pressure, psi)

(C:LOGGER:string,string) – reads a string record from the file that was opened with openRead or openCsvRead commands.

Example: (C:LOGGER:string,string) (>C:fs9gps:FlightPlanNewWaypointICAO)

(C:LOGGER:eof,number) – End Of File. Used in reading files. Returns 0 if the end of file, or last record in the file has not been reached, and 1 when it has.

Example: (C:LOGGER:eof,number) 1 == if{ do something }

(C:LOGGER:skip,number) – Skips X number of records from the current record position.

Example:

```
9 (>C:LOGGER:skip,number)
(C:LOGGER:number,number) (>L:Num1, number)
4 (>C:LOGGER:skip,number)
(C:LOGGER:number,number) (>L:Num2, number)
```

L:Num1 will receive record number 10 and L:Num2 will receive record number 15.

(C:LOGGER:countRead,number) – Returns the number of records read since the file was opened. Does not count records bypassed using LOGGER:skip.

Example: The following counts the number of records in a file and stores that number into L:NumberOfRecords:

```
:2000
(C:LOGGER:string,string)
(C:LOGGER:eof,number) 1 == if{ g2001 }
g2000
:2001
(C:LOGGER:countRead,number) (>L:NumberOfRecords, enum)
```

The read command `LOGGER:string` is used in the example above, but nothing is done with the records as they are read - they are not stored into `L:Vars` or string variables or displayed. They are only counted. In this use, it does not matter if `LOGGER:string` or `LOGGER:number` is used or if an individual record type is string or number. Both types will be 'read' and `LOGGER:countRead` will count it.

WRITE commands - To write individual records to a file:

(>C:LOGGER:number,number) - writes a number record in the file that has been opened with openWrite, openCsvWrite, openAppend, or openCsvAppend commands.

Example: (A:AIRSPED INDICATED, knots) (>C:LOGGER:number,number)

(>C:LOGGER:string,string) - writes a string record in the file that has been opened with openWrite, openCsvWrite, openAppend, or openCsvAppend commands.

Example: (C:fs9gps:WaypointAirportName) (>C:LOGGER:string,string)

(>C:LOGGER:newline,number) - directs writing of the next record to a new line. Newline is used primarily when writing repetitive data to a file using the csv record format (openCsvWrite or openCsvAppend) such as in a *Flight Data Recorder* application.

Example: 1 (>C:LOGGER:newline,number)

READ and WRITE rules

LOGGER will allow only one *Write* file to be open at a time per gauge in where the class is referenced. Creating a second *Write* file within the same gauge will automatically close a previous *Write* file if it is open. As a consequence, there is never a need to execute more than one closeWrite command per gauge in order for all *Write* files to be closed. Example:

-Gauge (A)

'File1.txt' (>C:LOGGER:openWrite) - creates File1.txt
'File2.ini' (>C:LOGGER:openWrite) - closes File1.txt then creates File2.ini
1 (>C:LOGGER:closeWrite) - closes File2.ini. All *Write* files in Gauge (A) are now closed.

-Gauge (B)

'File3.txt' (>C:LOGGER:openWrite) - creates File3.txt
'File4.ini' (>C:LOGGER:openWrite) - closes File3.txt then creates File4.ini
1 (>C:LOGGER:closeWrite) - closes File4.ini. All *Write* files in Gauge (B) are now closed.

Additionally, openWrite or openCsvWrite will first erase the file if it already exists.

LOGGER will allow only one *Read* file to be open at a time per gauge in where the class is referenced. Opening a second *Read* file within the same gauge will automatically close a previous *Read* file if it is open. As a consequence, there is never a need to execute more than one closeRead command in order for all *Read* files to be closed. Example:

-Gauge (A)

'File1.csv' (>C:LOGGER:openRead) - opens File1.csv
'File2.txt' (>C:LOGGER:openRead) - closes File1.csv then opens File2.txt
1 (>C:LOGGER:closeRead) - closes File2.txt. All *Read* files in Gauge (A) are now closed.

-Gauge (B)

```
'File3.csv' (>C:LOGGER:openRead) - opens File3.csv  
'File4.txt' (>C:LOGGER:openRead) - closes File3.csv then opens File4.txt  
1 (>C:LOGGER:closeRead) - closes File4.txt. All Read files in Gauge (B) are now closed.
```

LOGGER will allow a single Read and a single Write file to be open at the same time in any gauge. Use of *Append* (openAppend or openCsvAppend) will close an open *Write* file first before opening the designated file in order to append records to it. Example:

```
'Write1.txt' (>C:LOGGER:openWrite) - creates Write1.txt  
'Read1.txt' (>C:LOGGER:openRead) - opens Read1.txt  
'Write2.ini' (>C:LOGGER:openAppend) - closes Write1.txt then opens Write2.ini. Read1.txt is  
not affected. If Write2.ini did not previously exist, it will be created and records will be appended to it.
```

Other rules

Triggering RELOAD_USER_AIRCRAFT event closes all LOGGER files currently open, the same that happens when loading a different aircraft; resetting the current flight or closing the simulator. Executing RELOAD_PANELS event has no effect in FSX.

Deleting a file with LOGGER can have unintended consequences if the file path and name are not designated perfectly. Assume that file AAA.txt has the following directory path:

C:\Documents and Settings\TEMP\Desktop\LOGGER\AAA.txt

Then:

```
'C:\Documents and Settings\TEMP\Desktop\LOGGER\AAA.txt'  
(>C:LOGGER:delete)
```

will delete that file to the *Recycle Bin* as intended. Now:

```
'C:\Documents and Settings\TEMP\Desktop\LOGGER\  
(>C:LOGGER:delete)
```

will have no effect. However:

```
'C:\Documents and Settings\TEMP\Desktop\LOGGER'  
(>C:LOGGER:delete)
```

will delete the LOGGER folder to the *Recycle Bin* (note the absence of the final \).

Managing output (WRITE) file size

A tremendous amount of data can be recorded to file using LOGGER. Hundreds of different variables of any type (A:, E:, P:, G:, L:, or fs9gps) can be written to file each gauge update cycle. Unless some cycles are skipped between recordings, the output file will get very large, very rapidly, especially if recording data every cycle at 18 update cycles per second.

Write file limitations and FS cycle time-out limit

Writing data to hard disk is one of the most time consuming operations a gauge can undertake. Unlike the Flight Simulator gps module (in which data base searches can also be very time consuming) that operates asynchronously with a gauge, LOGGER is synchronous with the host gauge, so it is possible to reach the update cycle time-out limit of around 55 milliseconds before all data have been written to hard disk.

The number of records that can be written to hard disk each update cycle is dependent upon the hard drive capability, system resource load, and especially gauge complexity. It is independent of update cycle frequency.

During tests using a fairly simple gauge, around 750 different variables were written to hard disk every update cycle (with Update running 18 cycles per second) before exceeding the cycle time-out limit and losing some records.

With a very complex gauge and a convoluted LOGGER write command structure, at least 250 variables were written to hard disk each update cycle before exceeding the time-out limit.

Effect on frame rate

Use of LOGGER has no, or negligible effect on frame rate, even when recording hundreds of variables each update cycle.

Code examples

XML gauge and script examples of several LOGGER applications are available.

Please check `XMLTools_Examples.pdf`

=====

Comments

XMLTools v2.01 can be installed and used on any commercial and non-commercial panel.

XMLTools v2.01 cannot be distributed or sold as a standalone product.

Commercial developers must ask for permission.

Use this software at your own risk.

Tom Aguilo
taguilo1@hotmail.com
February 2016