1. GMFileSystem

GMFileSystem is an extensions for GameMaker studio. It allows users to ignore the(arbitrary) sandbox restrictions set up by yoyogames. GMFileSystem tries to “look like” gamemaker's built in functions, As such one of the core components is to make the functions as transparent as possible; the functions. This means no helpfile should be necessary, the help comes from the GM manual.

There are some differences, some improvements as well as extra restrictions. These differences will be discussed during this small manual. Also some performance notes will be explained as well as an indication of how GMFileSystem works behind the schemes. The final parts of the manual contain a reference list, where each function is quickly explained (and the corresponding function inside the manual is shown).

Updates to the GMFileSystem extension can always be found at the GMC topic (http://gmc.yoyogames.com/index.php?showtopic=567528). GMFileSystem is licensed under the new BSD license; the source is available on google (http://gm-filesystem.googlecode.com/).

GMFileSystem consist of several parts, those parts can be separated at will, as to prevent unnecessary file size increase. Current the parts are as following

	GMFile	**GMResource**	**GMIni**	**GMXML**
Size (Kb)	200	2500	240	190
Uses	File system functions Text file functions binary file functions	Sprite management Background management Sound management Image saving	Ini handling	Xml handling
Files	GMFile.dll	GMResource.dll GMResource.gml	DMIni.dll	GMXML.dll

Table of Contents

2. Manual

During this chapter the different parts of GMFileSystem will be discussed. Each part has its own files & can be completely separated from the other parts & the extension without making those other parts no longer work. To remove a part simply delete all files that belong to the part from the extension in the resource tree (generally all files belonging to the same part have the same name, just different extensions).

Currently the parts implemented are as following:

	GMFile	**GMResource**	**GMIni**	**GMXML**
Size (Kb)	200	2500	240	190
Uses	File system functions Text file functions binary file functions	Sprite management Background management Sound management Image saving	Ini handling	Xml handling
Files	GMFile.dll	GMResource.dll GMResource.gml	DMIni.dll	GMXML.dll

Each function starts with the prefix “FS_”, and most functions have names equal to gamemaker's equivalent. The biggest difference is that these functions work not like the sandboxed versions, they work more straightforward (as the GM8.x & earlier versions). The functions often require a string as filename, in this string you can use the general environment variables – like in windows. (%APPDATA%, %PROGRAMDATA%, %TMP% etc) – or things like “%USERPROFILE%\documents” get the “my documents” folder in windows vista.

3. GMFile

Size: 200Kb

Files: GMFile.dll

GMFile handles all basic file management functions. Text file functions (reading writing), binary file functions (with some extra strength), and generic filesystem functions (copying files, deleting/creating directories & files).

First the text file functions will be handled, followed by the binary functions; after that some information will be given on error handling (as this is the same for both text & binary files). Finally the functions of filesystem will be shortly discussed

3.1. Text files

Function	Description
`File = FS_file_text_open_read(fname)`	Opens file for reading
`FS_file_text_read_string(file)`	Reads contents of current line as string
`FS_file_text_read_char(file, number)`	Reads a number of characters
`FS_file_text_read_real(file)`	Reads a number from file
`FS_file_text_unread(file)`	Unreads a character.
`FS_file_text_readln(file)`	Goes to the next line
`File = FS_file_text_open_append(fname)`	Opens the file for writing at the end
`File = FS_file_text_open_write(fname)`	Opens the file for writing
`FS_file_text_write_string(file, string)`	Writes a string to the file
`FS_file_text_write_read(file, number)`	Writes a number to a file
`FS_file_text_writeln(file)`	Writes a newline
`FS_write_flush(file)`	Flushes the buffer to the hard drive
`FS_set_locality(localstring)`	Sets locality

Locality for text files is handled correctly. (possix file systems will have a LN byte at the ends of a line, windows based have CR+LN). Further locality regarding numeric values will be handled in the next paragraph.

The text files act almost exactly as GameMaker's built in functions, in the part *reference > File Handling > Files*. When reading a numeric value from a file the parser will skip leading spaces, and ends the moment a number is no longer a number. This allows you to have multiple numbers separated by spaces on a line.

The function `FS_file_write_flush()` is added: to allow you to force writing of the buffer to the hard drive. Another extra function is `FS_file_text_read_char()`. This function reads a given number of characters – or until an end of line character is found. Finally the function `FS_text_unread()` is added.

This function allows you to “unread” characters: so to move back in the file.

3.2. Locality

Function	Description
`FS_set_locality(localstring)`	Sets locality

When the program starts the locality is **set to the current system locality (changed from 1.3.0)**. This is different from the “standard” c-locality, though more in line with GM & other office programs. The function given above allows you to change the locality manually during program execution.
There some value for localstring that work

1) providing an empty string sets locality to the system locality
2) providing “C” sets the locality to original c-style
3) On windows (when the DLL is compiled with VS) there are a lot of locality strings. These have the form Language_Country.CodePage. (IE: German_Germany.1252, or English_United States.1252). All values can be found at http://www.mydigitallife.info/ansi-code-page-for-windows-system-locale-with-identifier-constants-and-strings/

The function returns whether the locale set was successful, in effect whether the locale is installed on the user's PC. (“C” and “” should always work).

3.3. Binary files

Function	Description
`File=FS_file_bin_open(fname, mode)`	Opens a file for writing or reading
`FS_file_bin_read_byte(file)`	Reads a byte from the file
`FS_file_bin_read_word(file)`	Reads a short (2 bytes) from the file
`FS_file_bin_read_dword(file)`	Reads a long (4 bytes) from the file
`FS_file_bin_write_byte(file, value)`	Writes a byte to the file
`FS_file_bin_write_word(file, value)`	Write a short (2 bytes) to the file
`FS_file_bin_write_dword(file, value)`	Writes a long (4 bytes) to the file
`FS_file_bin_size(file)`	Returns the size of the file
`FS_file_bin_position(file)`	Returns the current position
`FS_bin_seek(file, pos)`	Sets the position
`FS_bin_seek_relative(file, offset, rel)`	Sets the position relative to the target

The binary files work exactly the same as the build in binary functions *reference > File Handling > Binary Files.* There are some extra features added to the binary functions to improve usage of binary files.

First of are the improvement in binary file reading & writing.

`FS_file_bin_read_word, FS_file_bin_write_word` These 2 functions allow one to read/write a 2-

byte-value at once. (So instead of a value between 0-255, the value can be between 0-65,535). The format in which the bytes are stored is depending on the system used to read/write the bytes. (Little endian on intel pcs, big endian on other systems). As such you should take a lot of care when writing a file in -say- mac osx and reading it in windows.
`FS_file_bin_read_dword, FS_file_bin_write_dword` Similar to above, but instead thse write a 4-byte-value at once. (So instead of a value between 0-255, the value can be between 0-4,294,967,295).

The other improvement is the function `FS_bin_seek_relative` This function allows more control over setting the position. The 3rd argument describes from what position the offset is calculated, it can have 3 values:

0 - relative to the beginning of the file (default operation, like in gamemaker)
1 - relative to the end of the file
2 - relative to the current position

3.4. Error Handling

File functions can easily give errors. The error handling is the same for both text as well as binary files. Where gamemaker only allows one to check if reading a file has reached the end-of-file, GMFileSystem allows for a lot more control. They do use the `_text_` part instead of `_bin_` because GM's also does.

Function	Description
`FS_file_text_eof(file)`	Whether reading has reach the end of file
`FS_file_text_fail(file)`	Failbit has been set
`FS_file_text_bad(file)`	Badbit has been set
`FS_file_text_good(file)`	Whether any of previous 3 bits has been set
`FS_file_text_set_fail(file, fail)`	Sets the failbit to “fail”
`FS_fail_text_set_bad(file, bad)`	Sets the failbit to “bad”

`FS_file_text_eof()` This is an actual GM function, though it is also part of the group to handle exceptions. Returns true if filereading has reached end of file.
`FS_file_text_fail()` Means an exception happened. Has three prime causes: when opening a file this exception happens if the file can't be opened (isn't there, something locked it out etc). When reading data it can happen if trying to read past end of file. And finally it can happen if you try to write bad formatted data to a file.
`FS_file_text_bad()` Means the file has crashed. Main cause of this is when the file can't be written to anymore while it was opened. (IE storage medium got removed).
`FS_file_text_good()` Means non of the above states is set.
Only when `good()` returns true the functions actually work, otherwise they are simply ignored and a default value is returned. In case you wish to continue after the file-reading crashed (ie; you know the storage medium got reconnected) you can set the fail and bad bits to “true”

3.5. File System

File system part handles all file management tasks. The filesystem is almost a direct copy of the functions inside *reference > File Handling > File System* though some extra function from the main part (*reference > File Handling*), mainly the "directory" functions.

Function	Description
`FS_directory_exists(dir)`	Tests if dir exists
`FS_directory_create(dir)`	Creates directory (and parent directories)
`FS_directory_delete(dir)`	Deletes directory and all files/subdirectories
`FS_file_exists(fname)`	Tests if fname exists
`FS_file_delete(fname)`	Deletes file fname
`FS_file_rename(fname, newname)`	Renames fname to newname
`FS_file_copy(fname, newname)`	Copies fname to newname
`FS_file_attributes(fname)`	Returns attributes of fname
`FS_file_find_first(mask, attributes)`	Finds first file that satisfies mask &attributes
`FS_file_find_next()`	Finds next file that satisfies mask & attributes
`FS_file_find_close()`	Frees file find memory
`FS_max_open_file()`	Returns maximum open files

These functions mostly work exactly the alike to the built in functions. With three notable exceptions; `file_rename()` can be used to move files to another directory – simply providing a different directory will do this. (Notice that for newname you will have to provide a full path).

`FS_file_attributes` will povide many more attributes than those specified by gamemaker. (Though testing if a certain exists using `FS_file_attributes("file") & fa_readonly = fa_readonly` works. The full attribute list can be found at MSDN ([1]). Also note that (like in gamemaker) for POSSIX filesystems only the `fa_directory` works.

FS_max_open_file returns the maximum number of simultaneously opened files. Windows uses a 9-bit number for this so the maximum open files is "512". On possix systems this is limited to the internal format (32 bit or 64 bit), or the available memory.

1 File attribute list: http://msdn.microsoft.com/en-us/library/windows/desktop/gg258117%28v=vs.85%29.aspx

4. GMResource

Size: 2500 Kb

Files: GMResource.dll
GMResource.gml

GMResource handles input & output of gamemaker, it basically allows users to load (and save) external resources from anywhere on the disk. (And in the future from “memory” might become possible too). During this chapter first a short explanation of the internal works will be made, after which the actual loading will be described.

4.1. Internal behaviour

The basic structure uses almost always the same very simple steps:

(File is converted) > copied to working_directory > using GM's functions to import

You will notice that this part uses 2 files, the copying and conversion are done by the dll, while the importing is done by a GM-script.

Now I said “copying”, but this isn't always the correct term, as disk writing is often a factor of low importance. To understand this more we separate two cases:

When the file has correct format and can be imported
When the file already has a correct format we don't have to do any internal work, the file will be provided to GM “as-is”. To speed this up, the file will not be copied, instead a hard-link will be created (hard link is similar to a shortcut with a few differences, like the file will only be deleted from a hard drive when the last hard link to it is removed). Providing a hard link is very fast, and the whole operation takes (on my core 2 duo, 54000 RPM drive) only around 5 micro seconds to complete. This speed is independent of the size of the file. Only in obscure cases the file has to be copied (like when there exist already 1023 hard links to the file).

When file has to be converted to correct format
When file conversion has to be done, this is the main limiting factor. Conversion is done by the openCV library. The complete process of converting & saving a file this way depends a lot on the size of an image, but for a large image (1280 * 800 pixels) it takes around 100 micro seconds to complete.

Generic functions

As the copying is very fast, the decision was made to not delete the temporary file each time, rather the files will be deleted when gamemaker normally closes. - Or if you wish you could force a delete of temporary files at any moment. Also you have some control over where the temporary files are stored

Function	Description
`FS_set_working_directory(dir)`	Sets the temporary directory
`FS_clean_temporary()`	Cleans temporary files
`FS_set_gm_save_area(dir)`	Sets the GM save area.
`FS_get_unique_filename(dir, ext)`	Gets a randomly-based unique filename

The working directory is normally somewhere in the temporary directories. You can set it to any directory from **where gm can read**. Beware of the bug currently haunting background_add() though[2]. The gm-save-area is a special directory that you have to manually identify, it is always "%localappdata %\<YOURGAMENAME>. It is explained more thoroughly below.

4.2. Loading resources

The resource loading is mostly transparent, and copies gamemaker. Notice however that the sound_add function only works with legacy sound systems. With the new audio there is no way to load external files. There are basically three sets of 2 functions for this:

Function	Description
`FS_sprite_add(fname, imgnumb, removeback, smooth, xorig, yorig)`	Adds an image from a file
`FS_background_add(fname, removeback, smooth)`	Adds a background from a file
`FS_sound_add(fname, kind, preload)`	Adds a sound from a file
`FS_sprite_replace(ind, fname, imgnumb, removeback, smooth, xorig, yorig)`	Replaces sprite ind with image from a file
`FS_background_replace(ind, fname, removeback, smooth)`	Replaces background ind with image from a file
`FS_sound_add(ind, fname, kind, preload)`	Replaces sound ind with image from a file

The file formats supported are:
images: *portable network graphics, bitmaps, jpeg files, jpeg 2000, portable image format, sun rasters & tiff files*

sounds: *mp3*

4.3. Exporting resources

Exporting resources is a bit a strange child, it is impossible to make it completely transparent. This is because there is no way to programmatic get the location where the items are stored by gamemaker (see this[3] bug report). To get around this limitation in gamemaker you (as programmer) have to manually give this folder to GMFileSystem. Luckily this folder is always at a similar location,

2 Can't load background from %localappdata%/<gamename> http://bugs.yoyogames.com/view.php?id=9013
3 Can't get gamename programmatically: http://bugs.yoyogames.com/view.php?id=9059

%LOCALAPPDATA%\<GAME>. So in the case of a game called “test_game” you would call `FS_set_gm_save_area("%localappdata%\test_game")` before any exporting operations.

Functions	Description
`FS_background_save(ind, fname)`	Saves background
`FS_background_save_adv(ind, fname, param)`	Saves background with parameter
`FS_screen_save(fname)`	Saves a screenshot
`FS_screen_save_adv(fname, param)`	Saves a screenshot with parameter
`FS_screen_save_part(fname, x, y, w, h)`	Saves a part of the screen
`FS_screen_save_part_adv(fname, x, y, w, h, param)`	Saves a part of the screen with parameter
`FS_sprite_save(ind, subimg fname)`	Saves a sprite
`FS_sprite_save_adv(ind, subimg, fname, param)`	Saves a sprite with parameter
`FS_sprite_save_strip(ind, fname)`	Saves a sprite as strip
`FS_sprite_save_strip_adv(ind, fname, param)`	Saves a sprite as strip with parameter
`FS_surface_save(id, fname)`	Saves a surface
`FS_surface_save_adv(id, fname, param)`	Saves a surface with parameter
`FS_surface_save_part(id, fname, x, y, w, h)`	Saves a part of a surface
`FS_surface_save_part_adv(id, fname, x, y, w, h, param)`	Saves a part of a surface with parameter

First of all, notice that each function comes in two flavours, one with and one without the extra `_adv` extension. The advanced saving options have an extra parameter that is used to define the storage manner for the compressed images. For a PNG this value describes simply the compression level, the value can be between 0-9 where “9” is slower but also better compressed. (Default would be 3). For jpeg this describes the quality level, this value can be between 0-100, where 100 would be best quality – but highest filesize (default is 95).

Another thing to notice is the huge bugs currently in GameMakerStudio (1.1.785). ([4]) This prevents background saving to work with large images. (unless those resources are created from a screen/surface). In the future added sprite/backgrounds will work, however the build in won't – to use them you will have to make a duplicate (sprite_duplicate()).

4 Won't save large background images http://bugs.yoyogames.com/view.php?id=9175

5. GMIni

Ini files are a method to store options. The ini specification is very simple, and hence easy to use for small amounts of data. Gamemaker (even previous version) has only allowed users to open ini files in specific locations (program directory in GM 8.x and earlier, aplication data in gm studio); and on top of that only a single ini file can be opened at once. GMIni tries to lift those problems, while keeping compatibility should this be wanted. All ini functions hence have two version, a normal version that behaves exactly as GM's ini functions. And a function with the affix _ext – these also take an ini-file-index. And behave as text files (with ini functions).

5.1. GM-like interface

Functions	Description
`FS_ini_open(fname)`	Opens the ini file
`FS_ini_close()`	Closes the opened ini file.
`FS_ini_read_string(section, key, def)`	Read a string from the section, key
`FS_ini_read_real(section, key, def)`	Read a numeric from the section, key
`FS_ini_write_string(section, key, val)`	Sets the section, key to the given value
`FS_ini_write_real(section, key, val)`	Sets the section, key to the given value
`FS_ini_key_exists(section, key)`	Whether the given key exists
`FS_ini_section_exists(section)`	Whether the given section exists
`FS_ini_key_delete(section, key)`	Deletes the given key
`FS_ini_section_delete(section)`	Deletes the given section (and all keys)

The most simplistic interface is gained when using the normal version of each function. When using these, the functions copy GM's behaviour exactly (apart from being able to read/write outside gm's default directories). They do however limit you to 1 open ini file at once.

5.2. Multiple ini file interface

Functions	Description
`FS_ini_open_ext(fname)`	Opens the ini file
`FS_ini_close_ext(ini)`	Closes the given ini file.
`FS_ini_read_string_ext(ini, section, key, def)`	Read a string from the section, key
`FS_ini_read_real_ext(ini, section, key, def)`	Read a numeric from the section, key
`FS_ini_write_string_ext(ini, section, key, val)`	Sets the section, key to the given value
`FS_ini_write_real_ext(ini, section, key, val)`	Sets the section, key to the given value
`FS_ini_key_exists_ext(ini, section, key)`	Whether the given key exists
`FS_ini_section_exists_ext(ini, section)`	Whether the given section exists

`FS_ini_key_delete_ext(ini, section, key)`	Deletes the given key
`FS_ini_section_delete_ext(ini, section)`	Deletes the given section (and all keys)

The extended interface is also relativelly easy to use. All functions do correspond gm's functionality, however this ini-interface also allows you as user to open multiple ini files at once. To do this the open-function (`FS_ini_open_ext`) returns a file-handle. This file handle is passed to the other functions.

6. GMXML

From version 1.4 and newer a complete new feature has been added to GMFilesystem: reading and writing XML files. XML files are ideal for markup and storage of data. The xml-part is named “GMXML.dll”. The core library used for GMXML is tinyxml2, a lightweight heavily optimized XML parser. The interface and constants correspond to the interface of that library.

The data in an xml file is written in a (DOM) tree-structure. Just as with ini files the whole file is read t once. (and changes are only saved when you close the file). As tree structures are alien to gamemaker a new syntax/handling had to be written. Most functions return a “pointer” (actually a pointer binary copied into a double floating point), where get & set functions have to be used to get the exact values.

6.1. Three datatypes

Functions	Description
`FS_xml_get_node_type(xml, node)`	Gets the type id of a node
`FS_xml_node_make_element(xml, node)`	Changes element “node” to element
`FS_xml_same_node(node_left, node_right)`	Whether two handles point to the same node

GMFilesystem's XML functions basically work over 3 “data types”. These are “attributes”, “nodes” & “elements”. Below I will quickly describe what is what and how you can know what function takes what.

First consider the “element”, an element is similar to a “branch” or “leaf” in XML, an element can contain attributes, other elements or values. Elements always have a name. The shortname used for elements is “elem”.

Secondly we have “nodes”. Nodes are a more generic concept of elements: any element is considered a node. However things like the actual text under an element is also a node – as are comments. Node access should be considered closer to the metal than using element-based functions. However in some cases (example below) this is necessary. The type of node can be requested, and a node can -if the type is correct- be converted to an element. The follow types are possible:

Value	Description	Value is
`GMXML_UKNOWN`	-	-
`GMXML_TEXT`	Raw text data	The actual text
`GMXML_ELEMENT`	Contains sub nodes	Name of the node
`GMXML_DOCUMENT`	Root of the XML file (not root node)	Name of the file
`GMXML_DECLARATION`	Meta data	
`GMXML_COMMENT`	Comment	Comment text

Finally there are attributes. Attributes are rather simple: as they are exactly the same as attributes standard in xml files. In GMXML they are often abbreviated to “attr”. Attributes have a name and a value, and for each element the name of an attribute is always unique.

To summarize, consider the following snippet from an xml file

```
<u>normal text here
<b>this is bold text</b>
 This text is hidden when using element-based functions
</u>
```

The main element “root” has as data “text” - the data is only read read upto the first child element. It contains 1 child element. Using node access it has 3 child nodes:

1) GMLXML_TEXT with value “normal text here ”
2) GMLXML_ELEMENT with value “b”
3) GMLXML_TEXT with value “ This text is hidden when using element-based functions”

So if an element in the xml tree can contain data + sub elements it is probably better to use the node-functions to access these.

In GMXML functions and arguments are always named to show what they operate over, by “node” “elem” or “attr”. Function are also clearly marked if returning an element or a node. Notice however that the actual data returned by such a function is rather abstract: it is a handle to the element/node. Other functions have to be used to read this handle into actual text.

Finally there is a function to compare handles. (GM's rounding with floating point values prevent a native comparison) You can compare two nodes, elements or even nodes & elements to see if they point to the same “location”.

6.2. Loading an xml file

Functions	Description
`FS_xml_open(filename)`	Opens xml file.
`FS_xml_open_ext(filename, whitespace)`	Opens xml file and gives a parameter for whitespace handling
`FS_xml_close(xml)`	Closes and saves xml file.

Opening a file is similar to opening text files – you give the filename and it returns a handle for future operations. However do notice that the complete file is read & the DOM is constructed the moment you open a file. The extended open function has a parameter called “whitespace” - this determines the manner whitespace is handled, the options are:
`FS_XML_COLLAPSE_WHITESPACE` – This collapses multiple spaces into a single space, and removes all linebreaks. - This is the behaviour you see in HTML.
`FS_XML_PRESERVE_WHITESPACE` – This preserves internal whitespace and linebreaks. However do notice that whitespace between elements is removed.

```
<root>hello    world
       <sub>
...
```

The value of above function would be “hello   world” (4 spaces).

6.3. Iterating over elements

Functions	Description
`FS_xml_root_element(xml)`	Gets the root element
`FS_xml_num_elem(xml, parent_node)`	Gets number of child elements under parent
`FS_xml_elem_first(xml, parent_node)`	Gets the first element under parent_node
`FS_xml_elem_last(xml, parent_node)`	Gets the last element under parent_node
`FS_xml_elem_next(xml, elem)`	Gets the next element
`FS_xml_elem_prev(xml, elem)`	Gets the previous element
`FS_xml_named_elem_first(xml, parent_node, name)`	Gets the first element with name "name" under parent_node
`FS_xml_named_elem_last(xml, parent_node, name)`	Gets the last element with name "name" under parent_node
`FS_xml_named_elem_next(xml, elem)`	Gets the next element with name "name"
`FS_xml_named_elem_prev(xml, elem)`	Gets the previous element
`FS_xml_num_node(xml, parent_node)`	Gets number of child nodes under parent
`FS_xml_node_first(xml, parent_node)`	Gets the first node under parent_node
`FS_xml_node_last(xml, parent_node)`	Gets the last node under parent_node
`FS_xml_node_next(xml, node)`	Gets the next node
`FS_xml_node_prev(xml, node)`	Gets the previous node
`FS_xml_find_elem(xml, pathstring)`	Finds an element based on a string-path
`FS_xml_find_elem_under(xml, elem, pathstring`	Finds an element under the given element with the specified path
`FS_xml_parent_elem(xml, node)`	Gets the parent element of the given node

Iterating over elements works similar to maps. With the main difference that the function do not return a "workable" value (key), but instead return a handle/id for the element. (Which can be compared for equality, or used to get the name/value). The iterating functions also need to know the parent function.
If you know the name of an child element you can also iterate over only the elements with this name (and if the name is unique, getting the first element with this name obviously also gets THE element with the name). The _node_ functions work over the "nodes" where the _elem_ functions only iterate the elements under a certain node. An example code to read and display below xml file:

```
<student>
    <name>Paul</name>
    <id>100254</id>
    <study>AE</study>
</student>
```

```
var root_element = FS_xml_root_element(xml)
var cur_elem = FS_xml_child_elem_first(xml, root_elem);
```

```
var num = FS_xml_num_child_elem(xml, root_elem)

for (var num = FS_xml_num_child_elem(xml, root_elem); num > 0; --n) {
        var name = FS_xml_get_elem_name(xml, cur_elem);
        var value = FS_xml_get_elem_data(xml, cur_elem);
        var str = "Node name: " + name;
        if (value != "") {
            str += "#Node value: " + value;
        }
        show_message(str);

        cur_elem = FS_xml_child_elem_next(xml, cur_elem)
}
```

A complete different manner to get an element is using the `FS_xml_child_find_elem(xml, pathstring)` functions. These provide a shortway to get unique elements if you know the path. The function takes a pathstring, which are the childs separated by dots. to get the name from above example one would do `FS_xml_child_find_elem(xml, "student.name")`. The second version allows a user to provide the root element where the path starts.

6.4. Element access

Functions	Description
`FS_xml_get_node_type(xml, node)`	Gets the type of a node
`FS_xml_get_elem_name(xml, elem)`	Gets the name of an element
`FS_xml_get_elem_data(xml, elem)`	Gets the string data of an element
`FS_xml_get_node_raw_data(xml, node)`	Gets data of a node, actual data depends on node type.
`FS_xml_set_elem_name(xml, elem, name)`	Sets the name of an element
`FS_xml_set_elem_data(xml, elem, val)`	Sets the string data of an element
`FS_xml_set_node_raw_data(xml, node, val)`	Sets value of a node, What value actually represents depends on node type.
`FS_xml_get_attribute(xml, elem, name)`	Gets the attribute of element with the given name
`FS_xml_set_attribute(xml, elem, name, value)`	Sets/Adds the attribute under element
`FS_xml_delete_attribute(xml, elem, name)`	Deletes the attribute with the given name

Once you have an element (or node) handle you can get data from this element using above functions. The first function is explained in chapter 6.1. The second function gets the name of the xml-tag, and the third function the value it contains. When using nodes, the 4th function can be used to quickly get the data stored at the node. What the data represents depends on the node type as following

Value	Value is
GMXML_UKNOWN	-
GMXML_TEXT	The actual text stored at the node
GMXML_ELEMENT	Name of the node
GMXML_DOCUMENT	Name of the xml-file
GMXML_DECLARATION	
GMXML_COMMENT	Comment text

The _set_ functions can be used to change this data. Finally the attribute functions can be used to have a map-like interface to attributes. Setting a non-existing attribute adds it, while an exesiting attribute will get changed.

6.5. Changing xml tree-structure

Functions	Description
`FS_xml_insert_begin_elem(xml, parent_elem, name, value)`	Inserts a new element at the start under parent_element
`FS_xml_insert_end_elem(xml, parent_elem, name, value)`	Inserts a new element at the end under parent_element
`FS_xml_insert_elem(xml, parent_elem, after_node, name, value)`	Inserts a new element after the given node under parent_element
`FS_xml_insert_begin_node(xml, parent_elem, type, value)`	Inserts a new node at the start under parent_element
`FS_xml_insert_end_node(xml, parent_elem, type, value)`	Inserts a new node at the end under parent_element
`FS_xml_insert_node(xml, parent_elem, after_node, type, value)`	Inserts a new node after the given node under parent_element
`FS_xml_delete_node(xml, parent, node)`	Deletes the given node under parent
`FS_xml_clear(xml, parent_node)`	Clears all subnodes (and value etc) from parent_node

Above functions are useful for updating the xml-tree structure. You can add child-elements easily, by giving the name of the element and the value, there are convenience functions to add elements at the start, end or after a specific other element.
Also you can add raw nodes, where the interpretation of the “value” (and what actually gets added to the tree) depends on the given type – see chapter 6.1.
Deleting is even more easy, and you can use the same function to both delete elements or nodes. All subnodes will get deleted when you delete the parent. There is also a convenience function to delete all childs at once – notice that this deletes all child NODES, so the value (text node) also gets cleared.

6.6. Attribute access

Functions	Description
`FS_xml_get_attribute(xml, elem, name)`	Gets the attribute of element with the given name
`FS_xml_set_attribute(xml, elem, name, value)`	Sets/Adds the attribute under element
`FS_xml_delete_attribute(xml, elem, name)`	Deletes the attribute with the given name
`FS_xml_num_attributes(xml, elem,)`	Gets number of attributes for element
`FS_xml_attribute_first(xml, elem)`	Gets handle to first attribute
`FS_xml_attribute_last(xml, elem)`	Gets handle to last attribute
`FS_xml_attribute_next(xml, attr)`	Gets handle to next attribute
`FS_xml_attribute_prev(xml, attr)`	Gets handle to previous attribute
`FS_xml_attribute_get_name(xml, attr)`	Gets name of the given attribute handle
`FS_xml_attribute_get_value(xml, attr)`	Gets value of the given attribute handle

The first three functions give a convenience named interface to the attributes. You simply give the name and the attribute handle is found in the function itself. The latter function require (or return) an attribute handle used in other functions.

7. Reference

FS_file_text_open_read(fname)

Opens file for reading

		Type	Description
Return		**real**	**File index**
Arguments			
	fname	**string**	**filename**

FS_file_text_open_write(fname)

Opens file for writing

		Type	Description
Return		**real**	**File index**
Arguments			
	fname	**string**	**filename**

FS_file_text_open_append(fname)

Opens file for appending data at end

		Type	Description
Return		**real**	**File index**
Arguments			
	fname	**string**	**filename**

FS_file_text_read_string(file)

Reads a string

		Type	Description
Return		**string**	**line**
Arguments			
	file	**real**	**file index**

FS_file_text_read_char(file, number)

Reads a number of characters on the current line

		Type	Description
Return		**string**	**line**
Arguments			
	file	**real**	**file index**
	number	**real**	**number of character to read**

FS_file_text_read_real(file)

Reads a number

		Type	Description
Return		**real**	**number**
Arguments			
	file	real	file index

FS_file_text_readln(file)

Reads to next line

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**

FS_file_text_unread(file)

Unreads a character

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**

FS_file_text_write_string(file, string)

Writes a string

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	string	**string**	**line to write**

FS_file_text_write_real(file, number)

Writes a numeric value

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	number	number	number to write

FS_file_text_writeln(file)

Writes a newline character

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**

FS_file_text_eof(file)

Whether file reached end-of-file

		Type	Description
Return		**bool**	**eof bit**
Arguments			
	file	**real**	**file index**

FS_file_text_fail(file)

Whether file has failed

		Type	Description
Return		**bool**	**fail bit**
Arguments			
	file	**real**	**file index**

FS_file_text_bad(file)

Whether file has crashed

		Type	Description
Return		**bool**	**bad bit**
Arguments			
	file	**real**	**file index**

FS_file_text_good(file)

Whether file can be read

		Type	Description
Return		**bool**	**!(eof | bad | fail)**
Arguments			
	file	**real**	**file index**

FS_file_text_set_fail(file, fail)

Sets the failbit

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	fail	**bool**	**failbit**

FS_file_text_set_fail(file, bad)

Sets the badbit

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	fail	**bool**	**badbit**

FS_file_bin_open(fname, mode)

Opens binary file

		Type	Description
Return		**real**	**File index**
Arguments			
	fname	**string**	**filename**
	mode	**real**	**modus (read: 0, write: 1, both: 2)**

FS_file_bin_read_byte(file)

Reads a byte

		Type	Description
Return		**real**	**byte-value**
Arguments			
	file	**real**	**file index**

FS_file_bin_read_word(file)

Reads a word

		Type	Description
Return		**real**	**word-value**
Arguments			
	file	**real**	**file index**

FS_file_bin_read_dword(file)

Reads a dword

		Type	Description
Return		**real**	**dword-value**
Arguments			
	file	**real**	**file index**

FS_file_bin_write_byte(file, value)

Writes a byte

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	value	**real**	**byte-value**

FS_file_bin_write_word(file, value)

Writes a word

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	value	**real**	**word-value**

FS_file_bin_write_dword(file, value)

Writes a dword

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	value	**real**	**dword-value**

FS_file_bin_write_byte(file, value)

Writes a byte

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	value	**real**	**byte-value**

FS_file_bin_size(file)

File size

		Type	Description
Return		**real**	**file size**
Arguments			
	file	**real**	**file index**

FS_file_bin_position(file)

Get current position

		Type	Description
Return		**real**	**position**
Arguments			
	file	**real**	**file index**

FS_file_bin_seek(file, pos)

Sets the position

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	pos	**real**	**position**

FS_file_bin_seek(file, offset, rel)

Sets the position relative to target

		Type	Description
Return		**void**	
Arguments			
	file	**real**	**file index**
	offset	**real**	**offset from target**
	rel	**real**	**target (beginning: 0, end: 1, current:2)**

FS_directory_exists(dir)

Tests if dir exists

		Type	Description
Return		**bool**	**if dir exists**
Arguments			
	dir	**string**	**directory**

FS_directory_create(dir)

Creates directory dir

		Type	Description
Return		**void**	
Arguments			
	dir	**string**	**directory**

FS_directory_delete(dir)

deletes directory dir

		Type	Description
Return		**void**	
Arguments			
	dir	**string**	**directory**

FS_file_exists(filename)

Tests if filename exists

		Type	Description
Return		**bool**	**if** filename **exists**
Arguments			
	filename	**string**	**file path**

FS_file_delete(filename)

deletes file filename

		Type	Description
Return		**void**	
Arguments			
	filename	**string**	**file path**

FS_file_rename(filename, newname)

renames file filename to newname

		Type	Description
Return		**void**	
Arguments			
	filename	**string**	**file path**
	newname	**string**	**new file path**

FS_file_copy(filename, newname)

copies file filename to newname

		Type	Description
Return		**void**	
Arguments			
	filename	**string**	**file path**
	newname	**string**	**new file path**

FS_file_attributes(filename)

Returns attributes associated with filename

		Type	Description
Return		**real**	**attributes**
Arguments			
	filename	**string**	**file path**

FS_file_find_first(mask, attributes)

Returns first file associated with mask & attributes

		Type	Description
Return		**string**	**filename**
Arguments			
	mask	**string**	**file mask**
	attributes	**real**	**attribute mask**

FS_file_find_next()

Returns next file after fs_file_find_first()

		Type	Description
Return		**string**	**filename**
Arguments			

FS_file_find_close()

Frees memory from file_find

		Type	Description
Return		**void**	
Arguments			

FS_max_open_file()

Returns maximum opened files by dll

		Type	Description
Return		**real**	**number of files**
Arguments			

FS_set_working_directory(dir)

Sets directory where to store temporary files

		Type	Description
Return		**void**	
Arguments			
	dir	**string**	**directory**

FS_set_gm_save_area(dir)

Give GameMaker's save area

		Type	Description
Return		**void**	
Arguments			
	dir	string	directory

FS_clean_temporary()

Cleans temporary files

		Type	Description
Return		**void**	
Arguments			

FS_get_unique_filename(dir, ext)

Gets random unique filename in dir with extension ext

		Type	Description
Return		**string**	**filename**
Arguments			
	dir	string	directory
	ext	string	extension of file

FS_sprite_add(fname, numb, removeback, smooth, xorig, yorig)

add sprite from image file

		Type	Description
Return		**real**	**spit index**
Arguments			
	fname	string	filename
	numb	real	number of sub images
	removeback	bool	whether to remove the background
	smooth	bool	whether to smooth the edges
	xorig	real	xposition of the origin
	yorig	real	yposition of the origin

FS_sprite_replace(ind, fname, numb, removeback, smooth, xorig, yorig)

Replaces sprite ind with image from fname

		Type	Description
Return		**void**	
Arguments			
	ind	real	sprite index
	fname	string	filename
	numb	real	number of sub images
	removeback	bool	whether to remove the background
	smooth	bool	whether to smooth the edges
	xorig	real	xposition of the origin
	yorig	real	yposition of the origin

`FS_background_replace(ind, fname, removeback, smooth)`

Replaces background ind with image from fname

		Type	Description
Return		**void**	
Arguments			
	ind	real	background index
	fname	string	filename
	removeback	bool	whether to remove the background
	smooth	bool	whether to smooth the edges

`FS_background_add(fname, removeback, smooth)`

Adds background from file fname

		Type	Description
Return		**real**	**background index**
Arguments			
	fname	string	filename
	removeback	bool	whether to remove the background
	smooth	bool	whether to smooth the edges

`FS_sound_replace(ind, fname, kind, preload)`

Replaces sound ind with sound from fname

		Type	Description
Return		**void**	
Arguments			
	ind	real	sound index
	fname	string	filename
	kind	real	type of sound
	preload	bool	whether to load sound directly in memory

`FS_sound_add(fname, kind, preload)`

Add sound with sound from fname

		Type	Description
Return		**real**	**sound index**
Arguments			
	fname	string	filename
	kind	real	type of sound
	preload	bool	whether to load sound directly in memory

`FS_background_save(ind, fname)`

Saves background

		Type	Description
Return		**void**	
Arguments			
	ind	real	background index
	fname	string	filename

`FS_background_save_adv(ind, fname, param)`

Saves background

		Type	Description
Return		**void**	
Arguments			
	ind	real	background index
	fname	string	filename
	param	real	compression parameters

`FS_sprite_save(ind, subimg, fname)`

Saves sprite

		Type	Description
Return		**void**	
Arguments			
	ind	real	sprite index
	subimg	real	image index
	fname	string	filename

`FS_sprite_save_adv(ind, subimg, fname, param)`

Saves sprite

		Type	Description
Return		**void**	
Arguments			
	ind	real	sprite index
	subimg	real	image index
	fname	string	filename
	param	real	compression parameters

`FS_screen_save(fname)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	fname	string	filename

`FS_screen_save_adv(fname, param)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	fname	string	filename
	param	real	compression parameters

`FS_screen_save_part(fname, x, y, w, h)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	fname	string	filename
	x	real	x position
	y	real	y position
	w	real	width
	h	real	height

`FS_screen_save_part_adv(fname, x, y, w, h, param)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	fname	string	filename
	x	real	x position
	y	real	y position
	w	real	width
	h	real	height
	param	real	compression parameters

`FS_surface_save(id, fname)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	id	real	surface index
	fname	string	filename

`FS_surface_save_adv(id, fname, param)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	id	real	surface index
	fname	string	filename
	param	real	compression parameters

`FS_surface_save_part(id, fname, x, y, w, h)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	id	real	surface index
	fname	string	filename
	x	real	x position
	y	real	y position
	w	real	width
	h	real	height

`FS_surface_save_part_adv(id, fname, x, y, w, h, param)`

Saves part of surface

		Type	Description
Return		**void**	
Arguments			
	id	real	surface index
	fname	string	filename
	x	real	x position
	y	real	y position
	w	real	width
	h	real	height
	param	real	compression parameters

`FS_d3d_model_load(ind, fname)`

Loads model into index

		Type	Description
Return		**void**	
Arguments			
	ind	real	model index
	fname	string	filename

`FS_d3d_model_save(ind, fname)`

Saves model

		Type	Description
Return		**void**	
Arguments			
	id	real	surface index
	fname	string	filename

`FS_ini_open(fname)`

Opens ini file

		Type	Description
Return		**void**	
Arguments			
	fname	string	filename

FS_ini_close()

Closes ini file

		Type	Description
Return		**void**	
Arguments			

FS_ini_read_string(section, key, def)

Reads a string from ini file

		Type	Description
Return		**string**	**returns the value or def**
Arguments			
	section	string	section
	key	string	key
	def	string	default value

FS_ini_read_real(section, key, def)

Read real from ini file

		Type	Description
Return		**real**	**returns the value or def**
Arguments			
	section	string	section
	key	string	key
	def	real	default value

FS_ini_write_string(section, key, val)

Writes a string to ini file

		Type	Description
Return		**void**	
Arguments			
	section	string	section
	key	string	key
	def	string	default value

FS_ini_write_real(section, key, val)

Writes real to ini file

		Type	Description
Return		**void**	
Arguments			
	section	string	section
	key	string	key
	val	real	value

FS_ini_key_exists(section, key)

Whether key exists in ini

		Type	Description
Return		**real**	**whether the key exists**
Arguments			
	section	string	section
	key	string	key

FS_ini_key_delete(section, key)

Deletes the given key from the ini

		Type	Description
Return		**void**	
Arguments			
	section	string	section
	key	string	key

FS_ini_section_exists(section)

Whether key exists in ini

		Type	Description
Return		**real**	**whether the section exists**
Arguments			
	section	string	section

FS_ini_section_delete(section)

Deletes the given key from the ini

		Type	Description
Return		**void**	
Arguments			
	section	string	section

FS_ini_open_ext(ini, fname)

Opens ini file with extended options (multiple files can be opened at once)

		Type	Description
Return		**real**	**ini-file handle**
Arguments			
	fname	string	filename

FS_ini_close_ext(ini)

Closes ini file

		Type	Description
Return		**void**	
Arguments			
	ini	**real**	**ini handle**

`FS_ini_read_string_ext(ini, section, key, def)`

Reads a string from ini file

		Type	Description
Return		**string**	**returns the value or def**
Arguments			
	ini	real	ini handle
	section	string	section
	key	string	key
	def	string	default value

`FS_ini_read_real_ext(ini, section, key, def)`

Read real from ini file

		Type	Description
Return		**real**	**returns the value or def**
Arguments			
	ini	real	ini handle
	section	string	section
	key	string	key
	def	real	default value

`FS_ini_write_string_ext(ini, section, key, val)`

Writes a string to ini file

		Type	Description
Return		**void**	
Arguments			
	ini	real	ini handle
	section	string	section
	key	string	key
	def	string	default value

`FS_ini_write_real_ext(ini, section, key, val)`

Writes real to ini file

		Type	Description
Return		**void**	
Arguments			
	ini	real	ini handle
	section	string	section
	key	string	key
	val	real	value

FS_ini_key_exists_ext(ini, section, key)

Whether key exists in ini

		Type	Description
Return		**real**	**whether the key exists**
Arguments			
	ini	real	ini handle
	section	string	section
	key	string	key

FS_ini_key_delete_ext(ini, section, key)

Deletes the given key from the ini

		Type	Description
Return		**void**	
Arguments			
	ini	real	ini handle
	section	string	section
	key	string	key

FS_ini_section_exists_ext(ini, section)

Whether key exists in ini

		Type	Description
Return		**real**	**whether the section exists**
Arguments			
	ini	real	ini handle
	section	string	section

FS_ini_section_delete_ext(ini, section)

Deletes the given key from the ini

		Type	Description
Return		**void**	
Arguments			
	ini	real	ini handle
	section	string	section

FS_xml_open(fname)

Opens xml file

		Type	Description
Return		**void**	**xml file index**
Arguments			
	fname	string	filename

FS_xml_open_ext(fname, whatespace)

Opens xml file with specified whitespace handling

		Type	Description
Return		**real**	**xml file index**
Arguments			
	fname	string	filename
	whitespace	real	whitespace handling

FS_xml_close(xml)

Closes xml file

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file

FS_xml_get_node_type(xml, node)

Gets the type of the given node

		Type	Description
Return		**real**	**type**
Arguments			
	xml	real	xml file
	node	real	node handle

FS_xml_node_make_element(xml, node)

Converts node to element

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	node	real	node handle

FS_xml_same_node(node_left, node_right)

Tests node or element handles

		Type	Description
Return		**real**	**whether left & right point to same node**
Arguments			
	node_right	real	node handle
	node_left	real	node handle

FS_xml_root_element(xml)

Returns topmost element in xml file

		Type	Description
Return		**real**	**handle to topmost element**
Arguments			
	xml	real	xml file

`FS_xml_num_elem(xml, parent_node)`

Returns number of child elements

		Type	Description
Return		**real**	**number of child elements**
Arguments			
	xml	real	xml file
	parent_node	real	node handle

`FS_xml_num_node(xml, parent_node)`

Returns number of child nodes

		Type	Description
Return		**real**	**number of child nodes**
Arguments			
	xml	real	xml file
	parent_node	real	node handle

`FS_xml_elem_first(xml, parent_node)`

Returns first child element

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	parent_node	real	node handle

`FS_xml_elem_last(xml, parent_node)`

Returns last child element

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	parent_node	real	node handle

`FS_xml_elem_next(xml, elem)`

Returns next element

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	elem	real	element handle

`FS_xml_elem_prev(xml, elem)`

Returns previous element

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	elem	real	element handle

`FS_xml_named_elem_first(xml, parent_node, name)`

Returns first child element

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	parent_node	real	node handle
	name	string	element name

`FS_xml_named_elem_last(xml, parent_node, name)`

Returns last child element with the given name

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	parent_node	real	node handle
	name	string	element name

`FS_xml_named_elem_next(xml, elem)`

Returns next element with the same name

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	elem	real	element handle

`FS_xml_named_elem_prev(xml, elem)`

Returns previous element with the same name

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	elem	real	element handle

`FS_xml_node_first(xml, parent_node)`

Returns first child node

		Type	Description
Return		**real**	**node handle**
Arguments			
	xml	real	xml file
	parent_node	real	node handle

`FS_xml_node_last(xml, parent_node)`

Returns last child node

		Type	Description
Return		**real**	**node handle**
Arguments			
	xml	real	xml file
	parent_node	real	node handle

`FS_xml_node_next(xml, node)`

Returns next node

		Type	Description
Return		**real**	**node handle**
Arguments			
	xml	real	xml file
	elem	real	node handle

`FS_xml_node_prev(xml, node)`

Returns previous node

		Type	Description
Return		**real**	**node handle**
Arguments			
	xml	real	xml file
	node	real	node handle

`FS_xml_find_elem(xml, path)`

Finds the element at position path

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	path	strin	path where to look

`FS_xml_find_elem(xml, parent_elem, path)`

Finds the element at position path

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	path	string	path where to look

`FS_xml_parent_elem(xml, node)`

Returns parent element

		Type	Description
Return		**real**	**element handle**
Arguments			
	xml	real	xml file
	node	real	node handle

`FS_xml_get_elem_name(xml, elem)`

Gets the name of the given element

		Type	Description
Return		**string**	**element name**
Arguments			
	xml	real	xml file
	elem	real	element handle

`FS_xml_get_elem_data(xml, elem)`

Gets the data of the given element

		Type	Description
Return		**string**	**element data**
Arguments			
	xml	real	xml file
	elem	real	element handle

`FS_xml_get_node_raw_data(xml, node)`

Gets the value of the given node – interpretation depends on node type

		Type	Description
Return		**string**	**node value**
Arguments			
	xml	real	xml file
	elem	real	node handle

`FS_xml_set_elem_name(xml, elem, name)`

Sets the name of the given element

		Type	Description
Return		**string**	**element name**
Arguments			
	xml	real	xml file
	elem	real	element handle
	name	string	new element name

`FS_xml_get_elem_data(xml, elem, val)`

Sets the data of the given element

		Type	Description
Return		**string**	**element data**
Arguments			
	xml	real	xml file
	elem	real	element handle
	val	string	new element value

`FS_xml_set_node_raw_data(xml, node, val)`

Sets the value of the given node – interpretation depends on node type

		Type	Description
Return		**string**	**node value**
Arguments			
	xml	real	xml file
	node	real	node handle
	val	string	new node value

`FS_xml_insert_begin_elem(xml, parent_elem, name, value)`

Inserts a new element at the begin under the given parent

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	name	string	new element name
	val	string	new element value

`FS_xml_insert_end_elem(xml, parent_elem, name, value)`

Inserts a new element at the end under the given parent

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	name	string	new element name
	val	string	new element value

`FS_xml_insert_elem(xml, parent_elem, after_node, name, value)`

Inserts a new element after the given node, under the given parent

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	after_node	real	node handle
	name	string	new element name
	val	string	new element value

`FS_xml_insert_begin_node(xml, parent_elem, type, value)`

Inserts a new node at the begin under the given parent

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	type	real	new node type
	val	string	new node value

`FS_xml_insert_end_node(xml, parent_elem, type, value)`

Inserts a new node at the end under the given parent

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	type	real	new node type
	val	string	new node value

`FS_xml_insert_elem(xml, parent_elem, after_node, type, value)`

Inserts a new node after the given node, under the given parent

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	after_node	real	node handle
	type	real	new node type
	val	string	new node value

FS_xml_delete_node(xml, parent_elem, node)

Deletes given node (and all child nodes)

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	element handle
	node	real	node handle

FS_xml_delete_node(xml, parent_node)

Clear all child nodes

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	parent_elem	real	node handle

FS_xml_num_attributes(xml, elem)

Gets number of attributes of an element

		Type	Description
Return		**string**	**attribute value**
Arguments			
	xml	real	xml file
	elem	real	element handle

FS_xml_get_attribute(xml, elem, name)

Gets attribute value of attribute with given name

		Type	Description
Return		**string**	**attribute value**
Arguments			
	xml	real	xml file
	elem	real	element handle
	name	string	attribute name

FS_xml_set_attribute(xml, elem, name, value)

Sets/Adds attribute with the given name to the given value

		Type	Description
Return		**void**	
Arguments			
	xml	real	xml file
	elem	real	element handle
	name	string	attribute name
	value	string	attribute value

`FS_xml_delete_attribute(xml, elem, name)`

Deletes attribute with given name

		Type	Description
Return		**string**	**attribute value**
Arguments			
	xml	real	xml file
	elem	real	element handle
	name	string	attribute name

`FS_xml_attribute_first(xml, parent_elem)`

Returns first attribute

		Type	Description
Return		**real**	**attribute handle**
Arguments			
	xml	real	xml file
	parent_elem	real	element handle

`FS_xml_attribute_last(xml, parent_elem)`

Returns last attribute

		Type	Description
Return		**real**	**attribute handle**
Arguments			
	xml	real	xml file
	parent_elem	real	element handle

`FS_xml_attribute_next(xml, attribute)`

Returns next attribute

		Type	Description
Return		**real**	**attribute handle**
Arguments			
	xml	real	xml file
	attribute	real	attribute handle

`FS_xml_attribute_prev(xml, attribute)`

Returns previous attribute

		Type	Description
Return		**real**	**attribute handle**
Arguments			
	xml	real	xml file
	attribute	real	attribute handle

`FS_xml_attribute_get_name(xml, attribute)`

Returns name of attribute

		Type	Description
Return		**string**	**attribute name**
Arguments			
	xml	real	xml file
	attribute	real	attribute handle

`FS_xml_attribute_get_value(xml, attribute)`

Returns value of attribute

		Type	Description
Return		**string**	**attribute value**
Arguments			
	xml	real	xml file
	attribute	real	attribute handle