XS-XML

© Melwyn Francis Carlo 2021 <carlo.melwyn@outlook.com>

XS-XML is a tiny, basic, single-file, and portable XML parser and compiler library. The parser reads an XML data file, and if valid, extracts the tag elements, their atrributes and content, and arranges them in the form of virtual nodes, each connecting itself to the other: either side-to-side, or top-to-bottom (in a hierarchical manner). The compiler, on the other hand, expects an input of the outermost node (which is, needless to say, connected to other nodes, which are furthermore connected to other nodes, and so on). It uses these nodes to arrange the data in an XML file format, and produces a valid XML file as a result. XS-XML stands for '*Extra small XML*'.

The library is available in two programming languages: C and JavaScript (JS).

The processing of all alphabetic input and data is case-sensitive. It works fine with simple XML files written as per the standards of *version 1.0* and encoding of *Unicode* (*UTF-8)*. But, being a basic library, it has its limitations. It does not work with comments, and discards them.

It also does not cater to the following features:

- XML schemas and DTDs
- Name prefixes
- XML namespaces (the xmlns attribute)

The above features are not necessarily used by the average user. For basic, non-sophisticated use (e.g. maintaining a simple database), the above features need not be utilised.
Nevertheless, it can still parse CDATA* and character entity references (CER)**.

---

* CDATA are used for placing illegal characters like the less-than (<) and ampersand (&) characters, within the XML content in their plain-form, like this:

```
<![CDATA[<&]]>
```

** CERs are used to repace ASCII and Unicode (UTF-8) characters in their plain-form.
For example, consider the less-than (<) character.
It can be represented in three different CER forms.

ASCII form	`<`
Decimal form	`&#60;`
Hexa-decimal form	`&#x3C;`

Consider the XML data below, as an example. Here, let us refer to the tag elements as *nodes*.

```
<Outermost_Node>

      <Node_1>
                  <Sub_Node_1>
      </Node_1>

      <Node_2>
                  <Sub_Node_2>
      </Node_2>

</Outermost_Node>
```

A valid XML file must contain only one outermost node. Now, let the outermost node be at, say, the hierarchical level zero. Then, the nodes *Node_1* and *Node_2*, both, will be at level one. Because they share the same hierarchical level, as well as the same ancestor (that is, *Outermost_Node*), they are siblings. Conversely, they are the descendants of *Outermost_Node*, as they are placed right within the outermost node.

Again, the two tag elements labelled *Sub_Node_1* and *Sub_Node_2* are at the same level two, but, they share different ancestors, *Node_1* and *Node_2*, respectively. Hence, they cannot be siblings.

In summary, the relationship between the above given nodes are as follows:

Sr. No.	**Nodes**	**Ancestor**	**Descendant**	**Previous Sibling**	**Next Sibling**
1.	Outermost_Node	*None*	Node_1	*None*	*None*
2.	Node_1	Outermost_Node	Sub_Node_1	*None*	Node_2
3.	Node_2	Outermost_Node	Sub_Node_2	Node_1	*None*
4.	Sub_Node_1	Node_1	*None*	*None*	*None*
5.	Sub_Node_2	Node_2	*None*	*None*	*None*

The previous sibling and the next sibling denote the sibling nodes above and below the given node, respectively. The ancestor and descendant denote the nearest nodes that are the given node's super and sub nodes, respectively.

That is exactly how the XS-XML library connects the XML information, either using pointers, or via the computer file system.

The C version of the XS-XML library can operate in either of the two modes:

- RAM mode
- FILE mode

The JavaScript (JS) version of the library operates only in the RAM mode. While, in C, that denotes the use of arrays, structures, and pointers, in JavaScript, it denotes the use of objects referencing each other.

The FILE mode, which, by its name, denotes the use of the file system, is not possible in the JavaScript version of the library. This is because, for security reasons, web browsers do allow direct access to the user's computer file system.

You may read the following after you have read the RAM and FILE mode descriptions, and the variables and functions descriptions.

For both RAM and FILE modes, there are *unset* functions***, which are important to execute after making use of the extracted XML information.

In RAM mode, the *unset* function frees memory space at the addresses pointed by the node pointers. If the program exits immediately after working on the extracted XML information, then, by default, most operating systems (OS) are capable to freeing the memory space themselves. The problem arises if the program does not exit for a very long time. Regardless, it is a good practice to utilise the respective *unset* functions.

In FILE mode, the *unset* function is especially important, as operating systems (OS) cannot delete the files in the computer file system by themselves when the program exits.

While the `FILE *tmpfile(void)` function provided by the `stdio.h` library exists, seemingly for this very purpose, there was reluctance to use it for two reasons:

- The temporary file deletes itself upon closing the file, and so, to access the file contents to-and-fro, the file handlers will have to be kept open. Unfortunately, there is a limit for each program on simultaneously keeping open multiple files.

- Each temporary file name is random. In case of debugging, it makes it more difficult to distinguish the files and locate the error-causing file. As per the method undertaken by the library, that relies on meaningful file names, it is easier to recognise a cluster of files sharing a similar file name.

*** For RAM mode, it is, `void xsxml_unset(Xsxml **xsxml_object);`
for FILE mode, it is, `void xsxml_files_unset(Xsxml_Files **xsxml_files_object)`.

RAM mode

The term *RAM mode* denotes the use of the RAM (*random access memory*) to hold the processed XML information. The RAM memory space is utilised for this purpose. The amount of memory space utilised depends on the size of the XML data file.

In RAM mode, the data is stored on the RAM as a cluster of arrays and pointers in C (or objects in JavaScript), pointing (or referencing) either to data or to other pointers (or objects) within the cluster.

To work with this mode, the user would have to have a sound knowledge of reading and manipulating arrays and pointers in case of C, and working with objects and arrays in case of JavaScript (JS).

FILE mode

The term *FILE mode* denotes the use of the computer file system to store the processed XML information. The RAM memory space is not utilised to hold the processed XML information in this mode; although RAM space in the magnitude of a few kilobytes (kB) will be utilised by the library itself to process the information. The amount of computer file system space utilised depends on the size of the XML data file.

In FILE mode, the data is stored as a cluster of hidden files either within the user-inputted directory, or within the application program's directory. The computer file system's temporary directory has not been used instead because of the limited file space allocated to that directory: a few tens to a few hundred megabytes (MB). This allocation is done during formatting or partitioning of the computer system.

Regardless, the computer system's temporary directory may be inputted manually into the `Xsxml_Files *xsxml_files_parse( ... )` function as the function's *second (last) argument*.

For Linux, the temporary directory is:	`/tmp/`
For Windows, it is:	`%temp%\`

The FILE mode is suitable in case of large XML data files. Here, individual data are stored in unique files, bearing unique file names.

Just as in the case of pointers to other nodes like ancestor and descendant nodes and so on, there exist files for that, too. Furthermore, each node is assigned a node number as a form of identification, so that files acting as pointers would store (as its contents) the node number of the node to which it is pointed. For easy recognizability and access, all files in a cluster share the same random prefix name, a fixed (and distinct) suffix name denoting the nature of the individual file's contents, and distinct node numbers.

To work with this mode, the user is not expected to have any specialist knowledge, apart from the basic workings of C functions.

C version library's external variables, structures and enumerations

---

A. There are no external variables.

B. There are four external enumerations:

1. Xsxml_Property

This enumeration is related only to FILE mode. It is used to obtain the extracted XML information's individual data. Each data is associated with one of the properties of a given tag element (node). All enumeration members are related to the function `char *xsxml_files_property( ... )`. The enumeration member `XSXML_PROPERTY_NONE` can only be used as an input to the function's *fourth argument*, while all the other members can be used only as an input to the function's *third argument*.

The property members and their values are as follows:

`XSXML_PROPERTY_NONE`	-1
`XSXML_PROPERTY_NODE_NAME`	0
`XSXML_PROPERTY_NODE_LEVEL`	1
`XSXML_PROPERTY_NUMBER_OF_CONTENTS`	2
`XSXML_PROPERTY_NUMBER_OF_ATTRIBUTES`	3
`XSXML_PROPERTY_CONTENT`	4
`XSXML_PROPERTY_ATTRIBUTE_NAME`	5
`XSXML_PROPERTY_ATTRIBUTE_VALUE`	6
`XSXML_PROPERTY_ANCESTOR`	7
`XSXML_PROPERTY_DESCENDANT`	8
`XSXML_PROPERTY_NEXT_SIBLING`	9
`XSXML_PROPERTY_PREVIOUS_SIBLING`	10

2. Xsxml_Result

This enumeration is related to both RAM and FILE modes. It is used to denote the result of the XML parsing and compiling operations. The result may be one of the following three: a complete success, failure due to errors related to file operations, and failure due to errors related to the XML data file content.

This enumeration serves as a result code, which is always accompanied by a `result_message` string (that is, a char pointer, `char *`, in C).

The property members and their values are as follows:

XSXML_RESULT_SUCCESS	1
XSXML_RESULT_FILE_FAILURE	-1
XSXML_RESULT_XML_FAILURE	-2

3. Xsxml_Direction

This enumeration is related to both RAM and FILE modes. It is used to denote the search direction while using one of the two functions: `size_t *xsxml_occurrence( ... )`, or, `size_t *xsxml_files_occurrence( ... )`. It is used as an input to the *sixth (last) argument* of both functions.

The search can either be forward, from top-to-bottom, or backward, from bottom-to-top.

The property members and their values are as follows:

XSXML_DIRECTION_FORWARD	1
XSXML_DIRECTION_BACKWARD	-1

4. Xsxml_Non_Alnum_Chars_Conversion

This enumeration is related only to RAM mode. It is used to determine the conversion mode of each of the tag element's (node's) non-alpha-numeric contents during the compilation process. The characters which are neither alpha or numeric may be converted in one of the four ways: keeping it as it is, converting it to the decimal form of character entry references (CERs), converting it to the hexa-decimal form of CERs, or placing them into CDATA tags.

The enumeration members are to be used as input to the function's fourth argument, while all the other members can be used only as an input to the `void xsxml_compile( ... )` function's *sixth (last) argument*.

The property members and their values are as follows:

XSXML_NO_CONVERSION	0
XSXML_CER_DECIMAL_CONVERSION	1
XSXML_CER_HEXA_DECIMAL_CONVERSION	2
XSXML_CDATA_CONVERSION	3

C. There are three external structures:

1. Xsxml_Nodes

TBA

2. Xsxml

TBA

3. Xsxml_Files

TBA

JavaScript (JS) version library's external variables, structures and enumerations

---

A. There are no external variables.

B. There are four external enumerations, similar to the C version.

C. There are only two external structures, similar to the C version:

   1. **Xsxml_Nodes**
   2. **Xsxml**

   The **Xsxml_Files** structure does not exist in the JavaScript (JS) version.
   This is because the FILE mode does not exist in the JavaScript (JS) version.

END OF DOCUMENT

A parse operation requires the input of the XML file path (not just the file name).

A parse operation returns an Xsxml object which contains the following members:

The parse operation's result code
The parse operation's result message
The number of tag elements (nodes)
An array of tag element (node) objects
A tag element (node) object contains the following members:

A pointer to this node's ancestor node, if it exists, or a NULL pointer
A pointer to this node's descendant node, if it exists, or a NULL pointer
A pointer to this node's next sibling node, if it exists, or a NULL pointer
A pointer to this node's previous sibling node, if it exists, or a NULL pointer
The hierarchical level (depth) of this node
The number of distinct PCDATA contents within this node [3]
The number of attributes associated with this node
The tag element (node) name (label)
An array of PCDATA contents within this node [3]
An array of attribute names (labels) associated with this node
An array of attribute values associated with this node
[3] PCDATA contents separated by other tag elements (nodes) within a given node are considered as different (distinct) contents within the same node.

In FILE mode, data is stored in the file system as a cluster of hidden files. The storage is similar as in RAM mode, where each data is stored in a unique file, bearing a unique file name.

In case of pointers to other nodes, as in ancestor and descendant nodes and so on, there are files for that too; here, the file would store the node number of the node to which it is pointed. Because all the files in the cluster share a common prefix name, it makes it easier to locate the files based on node numbers.

In FILE mode, a parse operation requires the input of the XML file path (not just file name), as well as a temporary directory path (optional). If the temporary directory path is not needed, then input a NULL pointer, or an empty string "" instead; this will cause the parse operation to use the same directory as the XML file.

The occurrence function provides with an array of node indices that match the input description of tag name, attribute name, attribute value, and PCDATA content. If you wish to skip an input, simply provide a NULL pointer instead. Another input 'direction' states whether the search be from top to bottom (XSXML_DIRECTION_FORWARD) or from bottom to top (XSXML_DIRECTION_BACKWARD). The first element of the resultant array contains the number of resultant nodes.

A FILE mode parse operation returns an Xsxml_Files object which contains the following members:

The parse operation's result code
The parse operation's result message
The number of tag elements (nodes)
The temporary directory path
The common file name prefix
The FILE mode, too, contains the occurrence function that takes in the same input and provides the same output, although, their inner workings are different.

Furthermore, the FILE mode also contains the property function, which, upon providing it with a node index, a property name, and optionally, a property index (only if more than one exists), retrieves the corresponding node information.