XS-XML
Documentation Manual

© 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 then 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

This structure is related only to RAM mode. It stores data associated with an individual tag element (node). It is an object that is plays the role of an individual node. Its members are as follows:

- `unsigned int depth`

It denotes the node's hierarchical level (as discussed in *page number 2*).
Its data type is an unsigned integer.

- `unsigned int number_of_contents`

It denotes the number of PCDATA contents associated with the node.
Its data type is an unsigned integer.

- `unsigned int number_of_attributes`

It denotes the number of attributes associated with the node.
In the below example, there are two attributes.
Its data type is an unsigned integer.

- `char *node_name`

It denotes the tag element's (node's) name (label).
Its data type is a string (that is, a char pointer, `char *`, in C).

- `char **content`

It consists of an array of the node's PCDATA contents. Each array element consist of contents associated with (placed within) the same tag element (node), but are separated by descendant nodes within this node.

In the below example, the node `Main_Node` has two PCDATA contents, separated by `Sub_Node`.

```
<Main_Node  attribute_1=″value_1″  attribute_2=″value_2″>
      Data Number One
      <Sub_Node/>
      Data Number Two
</Main_Node>
```

Its data type is a string array (that is, a double char pointer, or a char pointer array, `char **`, in C).

- `char **attribute_name`

It consists of an array of the node's attribute names.
In the above example (in *page number 7*), there are two attributes,
named `attribute_1` and `attribute_2`, respectively.

Its data type is a string array (that is, a double char pointer, or a char pointer array, `char **`, in C).

- `char **attribute_value`

It consists of an array of the node's attribute values.
In the above example (in *page number 7*), there are two attributes,
valued as `value_1` and `value_2`, respectively.

Its data type is a string array (that is, a double char pointer, or a char pointer array, `char **`, in C).

- `Xsxml_Nodes *ancestor`

It denotes the ancestor node (as discussed in *page number 2*).
It is a pointer pointing to another structure of this very same type: `Xsxml_Nodes`.

For example, consider obtaining `Xsxml_Nodes *This_Node`'s ancestor's number of attributes:

```
This_Node->ancestor->number_of_attributes
```

- `Xsxml_Nodes *descendant`

It denotes the descendant node (as discussed in *page number 2*).
It is a pointer pointing to another structure of this very same type: `Xsxml_Nodes`.

For example, consider obtaining `Xsxml_Nodes *This_Node`'s descendant's
first PCDATA content:

```
This_Node->descendant->content[0]
```

- `Xsxml_Nodes *next_sibling`

It denotes the next sibling node (as discussed in *page number 2*).
It is a pointer pointing to another structure of this very same type: `Xsxml_Nodes`.

For example, consider checking whether or not `Xsxml_Nodes *This_Node`'s next sibling exists:

```
if (This_Node->next_sibling != NULL)
{
     /* Hooray! Next sibling exists. */
}
else
{
     /* Oops! Next sibling does not exist. */
}
```

- `Xsxml_Nodes *previous_sibling`

It denotes the previous sibling node (as discussed in *page number 2*).
It is a pointer pointing to another structure of this very same type: `Xsxml_Nodes`.

For example, consider obtaining `Xsxml_Nodes *This_Node`'s ancestor's previous sibling's descendant's next sibling's node name:

```
This_Node->ancestor->previous_sibling->descendant->next_sibling->node_name
```

2. Xsxml

This structure is related only to RAM mode. It stores parsing or compiling process's result. Post-parsing operation, it also stores an array of node pointers (`Xsxml_Nodes **`). Pre-compiling operation, it expects the initialization of exactly one node pointer in the array, pointing to the user-created outermost node's address. The `Xsxml` structure's members are as follows:

- `Xsxml_Result result`

This is the enumeration that contains the result code (as discussed in *page number 5*).
It is generated post-parsing or post-compiling operations.

- `char *result_message`

It accompanies the above mentioned result code.
It is generated post-parsing or post-compiling operations.
It contains supplementary information regarding the outcome of the parsing or compiling process.
Its data type is a string (that is, a char pointer, `char *`, in C).

- `unsigned int number_of_nodes`

It denotes the number of nodes generated during the parsing process.
It is generated post-parsing operation.
This member is ignored during the compiling process.

- `Xsxml_Nodes **node`

It consists of an array of pointers (double pointers) to nodes generated during the parsing process.
The number of array elements, and correspondingly, nodes, is equal to a value
as denoted by the above mentioned variable, `number_of_nodes`.

For the compiling process, it expects the initialization of exactly one node pointer in the array, pointing to the user-created outermost node's address.

3. Xsxml_Files

This structure is related only to FILE mode. It stores parsing or compiling process's result. Post-parsing operation, it also stores additional information relating to the extracted information.
The `Xsxml` structure's members are as follows:

- `Xsxml_Result result`

This is the enumeration that contains the result code (as discussed in *page number 5*).
It is generated post-parsing or post-compiling operations.

- `char *result_message`

It accompanies the above mentioned result code.
It is generated post-parsing or post-compiling operations.
It contains supplementary information regarding the outcome of the parsing or compiling process.
Its data type is a string (that is, a char pointer, `char *`, in C).

- `unsigned int number_of_nodes`

It denotes the number of nodes generated during the parsing process.
It is generated post-parsing operation.
This member is ignored during the compiling process.

- `char *node_directory_path`

It contains the directory path entered by the user through the input of the *second (last) argument* while calling the function `Xsxml_Files *xsxml_files_parse`.
Its data type is a string (that is, a char pointer, `char *`, in C).

- `char *node_file_name`

It contains the random file name prefix generated during the compiling process.
This file name prefix is common for the cluster of files associated with
the extraction of the user inputted XML data file.
Its data type is a string (that is, a char pointer, `char *`, in C).

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.

C version library's external functions

A. RAM mode

`1. Xsxml *xsxml_parse( ... )`

This function parses the inputted XML data file.

- **Output variables : 1**

  A structure of type `Xsxml` containing the extracted information
  and the function result.

- **Input arguments : 1**

  `i. const char *input_file_path`

  Its value should contain the full file path of the input XML data file,
  that is, the full directory path, followed by the file name, followed by
  the file extension (normally, `*.xml`).
  Its data type is a string (that is, a char pointer, `char *`, in C).

`2. size_t *xsxml_occurrence( ... )`

This function searches for nodes based on the input information provided,
and returns a list of node numbers matching that information.

- **Output variables : 1**

  A pointer poiting to an array of variables of `size_t` containing the node numbers.
  The first element (`output[0]`) contains the number of nodes that match
  the input criteria. The node numbers list start from index `1` to `N`,
  where `N = output[0]`.

  The type, `size_t`, is equivalent to the type, `unsigned int`.

- **Input arguments : 6**

  `i. Xsxml *xsxml_object`

  This is the user-generated or compiling process generated structure variable.

  `ii. char *tag_name`

  If the user wants to search against a particular tag element (node) name,
  then input the relevant string, else, it may be set to `NULL`.

iii. `char *attribute_name`

If the user wants to search against a particular tag element's (node's) attribute name, then input the relevant string, else, it may be set to `NULL`.

iv. `char *attribute_value`

If the user wants to search against a particular tag element's (node's) attribute value, then input the relevant string, else, it may be set to `NULL`.

v. `char *content`

If the user wants to search against a particular tag element's (node's) PCDATA content, then input the relevant string, else, it may be set to `NULL`.

It should be noted that, for the previous three criteria, the library searches in terms of equality, that is, if the two strings are equal, only then do they match; in this criterion, the library searches in terms of presence, that is, if the input string is present, as a whole, within one of the tag element's (node's) PCDATA contents, only then do they match.

vi. `Xsxml_Direction direction`

It determines the search direction (as discussed in *page number 6*).

3. `void xsxml_compile( ... )`

This function expects the user to create inter-connected nodes, and provide the function with the outermost node as the first element of the `Xsxml_Nodes **node` variable. This variable is a structure member of the inputted structure variable, `Xsxml *xsxml_object`. The function then, uses the nodes to compile the information and generate a valid XML file out of it.

- **Output variables : 0**
- **Input arguments : 6**

i. `Xsxml *xsxml_object`

This is the user-generated or compiling process generated structure variable.

ii. `const char *save_directory`

This is a string variable that should contain the full directory path into whch the user wants the generated XML file to be saved.

iii. `const char *save_file_name`

This is a string variable that should contain the generated XML file's file name, including the file extension (normally, `*.xml`).

iv. `unsigned int indentation`

It denotes the number of horizontal spaces that should be applied, per indentation.

v. `unsigned int vertical_spacing`

It denotes the number of vertical spaces (that is, new line characters) that should be applied, per line break.

vi. `Xsxml_Non_Alnum_Chars_Conversion content_conversion_mode`

It determines the format with which the non-alpha-numeric characters (that is, characters that are neither alphabetic nor numeric) must be written into the generated XML file. This applies only to the tag element's (node's) PCDATA contents. There are four distinct conversion modes, as discussed in *page number 6*.

4. `void xsxml_unset( ... )`

This function systematically deletes the input `Xsxml` object. It frees the data pointed by the `Xsxml_Nodes *` node pointers and the nodes themselves, and sets the node pointers to `NULL`. It frees all the data pointed by the various pointers and then deletes the pointers themselves. In the end, the `Xsxml` object itself is set to `NULL`.

- **Output variables : 0**
- **Input arguments : 1**

  i. `Xsxml **xsxml_object`

  This is a double pointer (a pointer pointing to another pointer) of the structure type `Xsxml`. So, for a typical single pointer variable, `Xsxml *user_object`, the user could pass its address as the input argument as follows: `&user_object`.
  In other words, the input argument is passed by reference.

B. FILE mode

1. Xsxml_Files *xsxml_files_parse(...)

This function parses the inputted XML data file.

- **Output variables : 1**

  A structure of type `Xsxml` containing the extracted information
  and the function result.

- **Input arguments : 2**

  `i. const char *input_file_path`

  Its value should contain the full file path of the input XML data file,
  that is, the full directory path, followed by the file name, followed by
  the file extension (normally, `*.xml`).
  Its data type is a string (that is, a char pointer, `char *`, in C).

  `ii. const char *temporary_directory_path`

  Its value should contain the full file path of the input XML data file,
  that is, the full directory path, followed by the file name, followed by
  the file extension (normally, `*.xml`).
  Its data type is a string (that is, a char pointer, `char *`, in C).

2. size_t *xsxml_files_occurrence(...)

This function searches for nodes based on the input information provided,
and returns a list of node numbers matching that information.

- **Output variables : 1**

  A pointer poiting to an array of variables of `size_t` containing the node numbers.
  The first element (`output[0]`) contains the number of nodes that match
  the input criteria. The node numbers list start from index `1` to `N`,
  where `N = output[0]`.

  The type, `size_t`, is equivalent to the type, `unsigned int`.

- **Input arguments : 6**

  `i. Xsxml_Files *xsxml_files_object`

  This is the user-generated or compiling process generated structure variable.

ii. `char *tag_name`

If the user wants to search against a particular tag element (node) name, then input the relevant string, else, it may be set to `NULL`.

iii. `char *attribute_name`

If the user wants to search against a particular tag element's (node's) attribute name, then input the relevant string, else, it may be set to `NULL`.

iv. `char *attribute_value`

If the user wants to search against a particular tag element's (node's) attribute value, then input the relevant string, else, it may be set to `NULL`.

v. `char *content`

If the user wants to search against a particular tag element's (node's) PCDATA content, then input the relevant string, else, it may be set to `NULL`.

It should be noted that, for the previous three criteria, the library searches in terms of equality, that is, if the two strings are equal, only then do they match; in this criterion, the library searches in terms of presence, that is, if the input string is present, as a whole, within one of the tag element's (node's) PCDATA contents, only then do they match.

vi. `Xsxml_Direction direction`

It determines the search direction (as discussed in *page number 6*).

3. `char *xsxml_files_property( ... )`

This function returns the string valued information associated with the input criteria provided: in terms of the node index, property name, and property index.

For example, the user could obtain the outermost node's node name by setting the node index to zero (0), the property name to `XSXML_PROPERTY_NODE_NAME`,
and the property index to `XSXML_PROPERTY_NONE`.

As another example, the user could obtain the second node's sixth attribute's value by setting the node index to one (1), the property name to
`XSXML_PROPERTY_ATTRIBUTE_VALUE`,
and the property index to five (5).

- **Output variables : 1**

  A string (that is, a char pointer, `char *`, in C) containing the information pertaining to the requested input criteria.

- **Input arguments : 4**

  `i. Xsxml_Files *xsxml_files_object`

  This is the user-generated or compiling process generated structure variable.

  `ii. size_t node_index`

  It denotes the tag element's (node's) index, which is zero-based, with the index zero (0) being attributed to the outermost node.

  `iii. Xsxml_Property property_name`

  It denotes the tag element's (node's) property name (label).
  The variable type is an enumeration, as discussed in *page number 5*.

  `iv. size_t property_index`

  It denotes one of the tag element's (node's) property's indices, which is zero-based. If the property is a singular variable, and not an array, then in that case, set the value to `XSXML_PROPERTY_NONE`.

`4. void xsxml_files_unset( ... )`

This function deletes the hidden files that had been generated by the library's parse operation, and that are associated with the input `Xsxml_Files` object.
It frees all the data pointed by the various pointers and then deletes the pointers themselves. In the end, the `Xsxml_Files` object itself is set to `NULL`.

- **Output variables : 0**

- **Input arguments : 1**

  `i. Xsxml_Files **xsxml_files_object`

  This is a double pointer (a pointer pointing to another pointer) of the structure type `Xsxml_Files`. So, for a typical single pointer variable, `Xsxml_Files *user_object`, the user could pass its address as the input argument as follows: `&user_object`.
  In other words, the input argument is passed by reference.

JavaScript (JS) version library's external functions

A. RAM mode

All four (4) functions are similar to their C version library's counterparts, with minor changes.

1. function xsxml_unset(...)

No changes.

2. async function xsxml_parse(...)

No changes in the function, or its input arguments and output variables, but, the way the function functions is slightly different. As it is an asynchronous (`async`) function, the above function will have to be called in the following manner:

```
xsxml_user_object = await xsxml_parse( ... )
```

3. function xsxml_occurrence(...)

No changes.

4. function xsxml_compile(...)

No changes in the function, or its output variables, but, two of the input arguments employed in this function's C version library's counterpart does not exist in this function. The two input arguments are:

- `save_directory`
- `save_file_name`

This is because, for security reasons, web browsers do allow direct access to the user's computer file system.

B. FILE mode

The FILE mode does not exist in the JavaScript (JS) version, and so, no related functions exist.

END OF DOCUMENT