XML Library 1.1.6

Sons of Thunder Software
CUSTOM SOFTWARE DEVELOPMENT AND CONSULTING SERVICES

Introduction

The XML Library allows users of MetaCard[1] and Revolution[2] to read and manipulate XML documents. The XML Library is fully compliant with the W3C specifications for XML, which can be found at http://www.w3.org/TR/REC-xml. The XML Library comes in three editions:

- **Basic Edition:** A freeware version that provides the ability to read and parse multiple XML documents. This version does not provide writing capabilities (i.e. you can't change attribute values, create or delete nodes, etc.) or DTD support and the library comes as a locked stack.
- **Standard Edition:** A commercial version that provides the ability to read, parse and write multiple XML documents. This version does not come with DTD support. The library comes as an unlocked stack so you can modify the scripts for custom behaviors.
- **Professional Edition:** A full-featured commercial version that provides read, parse and write capabilities, along with full DTD support. Like the Standard version, the library comes as an unlocked stack so you can modify the scripts for custom behaviors. (Note: This version has not been released as of this writing, so only functions for the Basic and Standard Editions are described in this document.)

Technical Support/Contact Info

The XML Library was developed by Sons of Thunder Software (http://www.sonsothunder.com/). If there are any technical support issues, comments or questions, please send them via email to xmllib@sonsothunder.com.

White Space Symbols

Depending on what is being discussed, there may be a need to show white space characters (spaces, tabs, line feeds and carriage returns) in this document. For the purposes of clarity, the following symbols will be used:

· space   » tab   ↓ line feed   ¬ carriage return

XML Sample

To show how the XML methods (functions) operate, all of the examples will be based on this XML sample:

```
<?xml version="1.0"?>
<products>
  <!-- The widgets below are all from Australia -->
  <?runAction-checkStock?>
  <widgets>
    <widget id="1" name="Thingimajig" color="navy">
      <![CDATA[The <b>NEW</b> Thingimajig!
Comes in 5 new colors!]]>
    </widget>
    <widget id="3" name="Geegaw" color="red">
      The old Geegaw. Anyone want to buy one?
    </widget>
  </widgets>
</products>
```

[1] Must be 2.4.2 or higher. The XML Library makes heavy use of Perl-compatible regular expressions, which were first introduced in MetaCard 2.4.2.
[2] Must be a version of Revolution based on the MetaCard 2.4.2 engine or greater. See (1) above.

Using the Library

To use the XML Library (in any version), simply attach it to your stack through the use of the `start using` command. Assuming that the XML Library is in the default directory, you would attach it as follows:

```
on openStack
   start using "XML Lib.mc"
end openStack
```

Parsing XML, the **gXMLData** Array and *NodeIDs*

When you pass a set of XML to the XML Library via the XMLLoadData method, the data will be examined and processed – each node will be validated, given a node ID number (in the form *documentNumber*.*nodeNumber*), and will have specific node properties assigned to it (see Node Atttributes, below). This combination of *nodeID* and *nodeAttribute* defines a "slot" in a global array called **gXMLData**; an example looks like this:

```
gXMLData["1.5cont"]
```

The *nodeID* is `1.5` (document 1, node 5) and the *nodeAttribute* is `cont` (for "content").

When parsing the XML, end tags and white space are not recognized as nodes unto themselves, but are a part of other nodes. Using the XML Sample above (and assuming this is the first document parsed), the *nodeIDs* are assigned as follows:

```
1.0    (the document itself)
1.1    <?xml version="1.0"?>

1.2    <products>
1.3      <!-- The widgets below are all from Australia -->
1.4      <?runAction-checkStock?>
1.5      <widgets>
1.6        <widget id="1" name="Thingimajig" color="navy">
1.7          <![CDATA[The <b>NEW</b> Thingimajig!
       Comes in 5 new colors!]]>
           </widget>
1.8        <widget id="3" name="Geegaw" color="red">
1.9          The old Geegaw. Anyone want to buy one?
           </widget>
         </widgets>
       </products>
```

Node Types

Each node has a *node type*, which is one of its attributes (see Node Properties, below), and is represented by a type code. The types supported by the XML Library are as follows:

Name	Type Code	Opens With	Closes With	Description/Example
XML Document	`DOC`			A reference to the XML Document itself.
XML Declaration	`XDEC`	`<?xml`	`?>`	This is a specialized Processing Instruction that identifies that the document is an XML document, as well as whether the document contains supporting documents or not and what method of encoding was used for it. `<?xml version="1.0" encoding="UTF-8" standalone="true"?>`

Processing Instruction	`PROC`	`<?`	`?>`	A node that contains any special instructions that the parsing application may use in order to control the behavior of the parsing application. Can contain any characters between the opening and closing of the tags (except `?>`, which would indicate the close of the tag). `<?runAction-checkStock?>`
Comment	`CMNT`	`<!--`	`-->`	A node that contains comments. This is skipped over by most XML parsing applications. It can contain almost any characters between the opening and closing of the tag, including carriage returns and tabs. `<!-- The widgets below are all from Australia -->`
Text	`TEXT`			This node contains text, and may not contains < or &, and all white space will be normalized when the text of this node is retrieved (see Normalizing Data, below). `The old Geegaw. Anyone want to buy one?`

CDATA Section	`CDAT`	`<![CDATA[`	`]]>`	A special node that contains text contents with special characters such as < , &, etc. that a TEXT node cannot contain. When this data is retrieved, it is retrieved in a non-normalized state (see Normalizing Data, below).
				`<![CDATA[The <b>NEW</b> Thingimajig!`
				`        Comes in 5 new colors!]]>`

Element	`ELEM`	`<`*`name`*	`>`	This node represents an XML element, which may contain attributes and values. This kind of ELEM node also may contain children, and will end with </*name*>.
				`<widget id="3" name="Geegaw" color="red">`
				`</widget>`

Empty Element	`ELEM`	`<`*`name`*	`/>`	This is an ELEM node that does not contain children; it is in a self-enclosing tag. `<doohickey/>`

Node Formats

Each node type must conform to a specific format according to the World Wide Web Consortium (W3C). The formats for each node type are listed below, and are based on the symbolic approach provided by the W3C, but modified for clarity. To assist in reading these formats, keep the following rules in mind:

- The format is made up of *tokens*, each token is indicated by being underlined. This is similar to parameters for functions. For example, `xml` is a literal string, whereas `Letter` is a token.
- The format contains spaces for readability, but does not mean to include those actual spaces in the format. If white space (spaces, tabs, linefeeds or carriage returns) is supported in the format, it is identified with the `S` token (see *Base Particles*, below).
- Tokens or strings may be grouped together with parentheses ().
- If a vertical bar (|) is used within the parentheses, it means that each token or string is one of a series of options and you may use only *one* of the options. For example, `(white|red|green)` means you can use either `white` **or** `red` **or** `green`.)
- Ranges of characters are defined within brackets ([ ]), and may optionally use a hyphen (-) for a range. For example, `[A-Z]` means any capital letter from A through Z; `[A-Za-z]` means any letter (upper- or lower-case), `[ace]` means `a`, `c` or `e`. If preceded by a caret (^), this means "not", so `[^ace]` means `not a`, `not c` and `not e`. If letters are grouped by parentheses *within* the brackets, it indicates a *phrase*, not a set of characters, so `[^(ace)]` means `not ace`.

- Any token, string or group followed by a question mark (?) means that only 0 or 1 occurrences of that token/token group can exist (basically it means that it is optional).
- Any token, string or group followed by an asterisk (*) means that 0 or more occurrences of that token/token group can exist.
- Any token, string or group followed by a plus symbol (+) means that 1 or more occurrences of that token/token group can exist.

Base Particles

The following tokens and their actual values are used as components (what W3C calls *particles*) of the name formats:

Token	Description	Value	Example(s)
`S`	Whitespace	`(space|tab|linefeed|return)+`	
`Eq`	Equals	`S? = S?`	`=`
`Letter`	Any Letter	`[A-Za-z]`	`N`
`Digit`	Any Number	`[0-9]`	`4`
`Any`	Any character	*`any character`*	`$` `k`
`qt`	Quotation Mark[3]	`"`	`"`
`Name`	Name[4]	`(Letter | _ | :)(Letter | Digit | . | - | _ | :)*`	`_hello` `A3-Choice`

`Reference`	Character or Entity Reference	`(&#[0-9]+; | &#x[0-9a-fA-F]+; | & Name ;)`	`&#84;`
			`&#x0d;`
			`&Fred;`

`AttValue`	Attribute Value	`qt (Any [^%&*qt] | Reference)* qt`	`"www.abc.com"`
			`"&#84;"`

`EncDecl`	Encoding Declaration	`S encoding Eq qt [A-Za-z] ([A-Za-z0-9._-])* qt`	`encoding="UTF-8"`
`SDDecl`	Standalone Declaration	`S standalone Eq qt (yes | no) qt`	`standalone="no"`

An Example

To see how this is interpreted, the following format:

```
< Name (S Name Eq AttValue)* S? />
```

is interpreted as:

Item *(examined item is in bold)*	Interpretation
`<` `Name (S Name Eq AttValue)* S? />`	"A less-than symbol…

[3] The W3C supports both the single quote (') as well as the double quote (") as a quotation mark. This version of the XML Library only supports the double quote ("), and will not recognize the single quote (') as a quotation mark.

[4] The W3C extends the support of characters in the `Name` token to include additional Unicode characters (represented in the W3C specs as `CombiningChar` and `Extender`). These extensions are not supported in the XML Library.

< **Name** (S Name Eq AttValue)* S? />	… followed by a string where the first character is a letter, an underscore or a colon, followed by zero or more letters, digits, periods, hyphens, underscores or colons…
< Name **(S Name Eq AttValue)*** S? />	… followed by 0 or more name-attribute value pairs, each pair consisting of…
< Name (**S** Name Eq AttValue)* S? />	… one or more white space characters…
< Name (S **Name** Eq AttValue)* S? />	… followed by a string where the first character is a letter, an underscore or a colon, followed by zero or more letters, digits, periods, hyphens, underscores or colons…
< Name (S Name **Eq** AttValue)* S? />	… followed by 0 or more white space characters, an equals sign (=) and 0 or more white space characters…
< Name (S Name Eq **AttValue**)* S? />	… followed by a quotation mark, then either (a) 0 or more characters of any kind (except a less-than sign, ampersand or quotation mark) or (b) a character or entity reference, and finally followed by another quotation mark.
< Name (S Name Eq AttValue)* **S?** />	This is then followed by 0 or more white space characters[5]…
< Name (S Name Eq AttValue)* S? **/>**	… and finally terminated with a forward slash and greater-than symbol.

For clarity, here is a list of strings that either match or don't match this format (and if it doesn't match, why it doesn't):

String (white space shown)	Match	Not Match	Why
`<product/>`	✓		
`<·product/>`		✓	The first Name token cannot be preceded by white space.
`<product·src···=»"widget.jpg"/>`	✓		Remember S means 1 *or more* white space characters
`<product·/>`	✓		
`<product·val="me&you"/>`		✓	AttValue cannot contain an ampersand (&).
`<product·src="widget.jpg"·id=24/>`		✓	AttValue needs to be surrounded with quotes.
`<product·src="widget.jpg"·id="24"/>`	✓		

Node Formats

Now that you have a foundation for interpreting the node formats, here they are for the different node types:

Node Type	Node Format
`DOC`	*n/a*

[5] Technically S? is read like this: `((space|tab|return|linefeed)+)?`, which is the same as `(space|tab|return|linefeed)*`, or "0 or more white space characters".

`XDEC`	`<?xml (S version Eq "1.0") (EncDecl)? (SDDecl)? S? ?>`
`PROC`	`<? Name (Any [^(?>:)])* ?>` *Note: Name cannot be any form of the word "xml".*
`CMNT`	`<!-- (Any [^(--)])* -->` *Note: Cannot end in* `--->`
`TEXT`	`(Any [^<&])*`
`CDAT`	`(Any [^(]]>)])*`
`ELEM`	`< Name (S Name Eq AttValue)* S? >` *node contents go here* `< /Name S? >`
`ELEM (empty)`	`< Name (S Name Eq AttValue)* S? />`

Node Properties

Each node has a set of *node properties* that govern what they are and how their data is stored in the **gXMLData** array. These node properties are as follows:

`type`	Node type	The type of node, which is one of the following: • DOC XML Document • XDEC XML Declaration • PROC Processing Instruction • CMNT Comment • CDAT CDATA Section • ELEM Element (or Empty Element) • TEXT Text
`strt`	Start tag	Everything between "<" and ">" for the node. TEXT nodes do not have a `strt` node property.
`name`	Name	Used by ELEM only, indicates the text between the "<" and its terminator (either ">" for normal elements without attributes, "/>" for empty elements without attributes, or " " for elements/empty elements *with* attributes.
`attl`	Attribute list	A return-delimited list of attribute names owned by this node.
`prnt`	Parent ID	The *nodeID* of the parent of this node.
`kids`	Child list	A return-delimited list of *nodeIDs* indicating the children of this node.
`cont`	Contents	Everything that the node contains, including white space. If one node has children, a pointer to each child is indicated with [>> x] where "x" is the *nodeID* of the child node. Note that for ELEM nodes, this includes both the start tag and end tag of the node. Also note that white space that precedes the start tag of the node is owned by the node's *parent*, not by the node itself.
`root`	Root node	Used by DOC only, this contains the *nodeID* of the root ELEM node.
`last`	Last used node	Used by DOC only, this contains the last used node number. When new nodes are created, the node number assigned is the next number after `last`, and `last` is incremented accordingly. When a node is deleted, its node number is removed, but `last` is not changed. For this reason, `last` only identifies the last used node number, **not** the total number of nodes that are owned by the document.

`docs`	Document list	Used by the XML Library itself, it returns a return-delimited list of active document numbers.

These node properties are concatenated to the *nodeID* as a key to the **gXMLData** array, and their values are stored there. For example, using the XML Sample above, node 1.6 would have the following data stored:

`gXMLData["1.6type"]`	`ELEM`
`gXMLData["1.6strt"]`	`<widget id="1" name="Thingimajig" color="navy">`
`gXMLData["1.6name"]`	`widget`

`gXMLData["1.6attl"]`	`id`
	`name`
	`color`

`gXMLData["1.6prnt"]`	`1.5`
`gXMLData["1.6kids"]`	`1.7`

`gXMLData["1.6cont"]`	`<widget id="1" name="Thingimajig" color="navy">`
	`[>> 1.7]`
	`</widget>`

In contrast, the document itself would have these node properties:

`gXMLData["1.0type"]`	`DOC`
`gXMLData["1.0strt"]`	
`gXMLData["1.0name"]`	
`gXMLData["1.0attl"]`	
`gXMLData["1.0prnt"]`	

`gXMLData["1.0kids"]`	`1.1`
	`1.2`

`gXMLData["1.0cont"]`	`[>> 1.1]`
	
	`[>> 1.2]`

`gXMLData["1.0root"]`	`1.2`
`gXMLData["1.0last"]`	`9`

And the XML Library keeps track of the number of open documents. Suppose there are three documents that were loaded, but the second one was deleted with a call to XMLDeleteDocument. This is the result:

`gXMLData["docs"]`	`1`
	`3`

Normalizing Data

When data is retrieved with the XML Library, the type of node and the content that is being retrieved will determine if the text is *normalized* or not. Normalizing data performs the following transformations on the retrieved text:

- Any leading or trailing white space (space, tab, linefeed or returns) is discarded
- All white space characters are converted to spaces (i.e. tabs convert to spaces, etc.)
- All sequences of spaces are replaced with a single space

What this does is convert a chunk of text that looks like this (white space shown):

```
»»This·is·a·bunch·of¬
text·with····↓
·a·bunch·of···spaces·and¬
other·»·white space
```

to this:

```
This·is·a·bunch·of·text·with·a·bunch·of·spaces·and·other·white space
```

In general, retrieved values will be normalized, with the major exception being the content of CDATA Section nodes (node type `CDAT`).

Putting It All Together

You can see how nodes are identified and stored by using the xml_dump utility method (with *showInvisibles* turned on), which displays the node number, node type and raw node contents of each node, in sequential order. Using it on our XML Sample produced the following results:

```
[1.0:DOC][>>·1.1]¬†
¬†
[>>·1.2]
[1.1:XDEC]<?xml·version="1.0"?>
[1.2:ELEM]<products>¬†
» [>>·1.3]¬†
» [>>·1.4]¬†
» [>>·1.5]¬†
</products>
[1.3:CMNT]<!--·The·widgets·below·are·all·from·Australia·-->
[1.4:PROC]<?runAction-checkStock?>
[1.5:ELEM]<widgets>¬†
» » [>>·1.6]¬†
» » [>>·1.8]¬†
» </widgets>
[1.6:ELEM]<widget·id="1"·name="Thingimajig"·color="navy">¬†
» » » [>>·1.7]¬†
» » </widget>
[1.7:CDAT]<![CDATA[The·<b>NEW</b>·Thingimajig!·¬†
Comes·in·5·new·colors!]]>
[1.8:ELEM]<widget·id="3"·name="Geegaw"·color="red">¬†
» » » [>>·1.9]¬†
» » </widget>
[1.9:TEXT]The·old·Geegaw.·Anyone·want·to·buy·one?
```

As you can see, if you replace the pointers [>> x] with the actual nodes, you will end up with the original XML. Take a close look at who "owns" the white space – you will see that a node only owns white space that is between its starting tag and its ending tag. Anything prior to its starting tag is "owned" by its parent. Look at the first two nodes in the document (1.1 and 1.2). The return and linefeed characters (¬†) after node 1.1 (the XDEC) and between node 1.1 and 1.2 (the root ELEM) are "owned" by the document (1.0).

This may cause a kind of strange view of a node – for example, look at node 1.5. It's content is this:

```
<widgets>¬†
» » [>> 1.6]¬†
» » [>> 1.8]¬†
» </widgets>
```

Although the `<widgets>` and `</widgets>` line up properly in the composite XML document, the tab character (») that precedes `<widgets>` is not owned by this node, but by its parent (1.2), as can be seen by this portion of node 1.2's content:

```
» [>> 1.5]¬†
```

Since the tab that precedes the ending tag of 1.5 (`</widgets>`) is between the starting tag and the ending tag of the node, it is owned by the node itself (and not its parent).

Keep this in mind if you ever need to manipulate the raw contents of nodes in the **gXMLData** array.

Complete Example

Now that you have seen in the previous section how nodes are ordered and numbered, the following code will *create* the XML data of our XML Sample (note that you will need the Standard Edition to be able to run this code since it creates documents, nodes and attributes). You may wish to refer to this example as you read the next section where all the methods are laid out in detail; it may help in your understanding of how to get started with the XML Library.

```
on mouseUp
  -- reset everything (note: this clears *all* XML documents from memory)
  xml_reset
  -- set a callback to me if there are any problems
  xml_setCallback (long id of me),"xmlError"

  -- create the document, and its main nodes
  put XMLCreateDocument("products") into tDoc
  get XMLAppendChild("1.2","CMNT","","The widgets below are all from Australia")
  get XMLAppendChild("1.2","PROC","runAction-checkStock")
  put XMLAppendChild("1.2","ELEM","widgets") into tWidgets

  -- create the first child node of <widgets>
  put XMLAppendChild(tWidgets,"ELEM","widget") into tWidget
  get XMLCreateAttribute(tWidget,"id","1")
  get XMLCreateAttribute(tWidget,"name","Thingimajig")
  get XMLCreateAttribute(tWidget,"color","navy")
  get XMLAppendChild(tWidget,"CDAT","","The <b>NEW</b> Thingimajig!" & cr & \
    "Comes in 5 new colors!")

  -- create the second child node of <widgets>
  put XMLAppendChild(tWidgets,"ELEM","widget") into tWidget
  get XMLCreateAttribute(tWidget,"id","3")
  get XMLCreateAttribute(tWidget,"name","Geegaw")
  get XMLCreateAttribute(tWidget,"color","red")
  get XMLAppendChild(tWidget,"TEXT","","The old Geegaw. Anyone want to buy one?")

  -- display results
  answer "Display actual XML or see the node numbers?" with "XML" or "Nodes"
  if it = "XML" then
    put XMLGetNode("1.0",true,false) into fld "result"
  else
    put xml_dump(tDoc) into fld "result"
  end if
  xml_clearCallback
end mouseUp

on xmlError
  global gXMLError
  answer "An error was generated: " & gXMLError
  put empty into gXMLError
  exit to Metacard
end xmlError
```

List of Methods

Note: The support for the different editions of the XML Library are indicated with the following graphics on each method page: Basic Std Pro

XMLLoadData

Basic **Std** **Pro**

Summary

This method loads XML data into the global array **gXMLData** and verifies that it is well-formed.

Syntax

`XMLLoadData(`*`xmlData[, silentErrors]`*`)` ⇨ *`docNumber`*

Arguments

`xmlData`	The XML data that comes from any container (although usually it is straight from a file).
`silentErrors`	Optional. Can be either `true`, `false` or left blank. If `true`, the XML Library will not provide an error dialog box if there were any errors found in attempting to load the XML data. If `false` or left blank, the XML Library will provide an error message automatically if any errors were found in loading the document.

Possible Returned Values

`docNumber`	The document number for the parsed XML document.
`error`	An error was found during parsing. Check **gXMLError** for the error found (especially if *`silentErrors`* is `true`).

Description

Calling this method will parse the data passed in *`xmlData`* into an XML node structure in memory and stored in the global **gXMLData**. As multiple documents can be stored and referenced, the first XML document parsed is given a *`docNumber`* of 1, and each subsequent document loaded with **XMLLoadData** is given the next incremental number.

As each node is parsed, it is assigned a *nodeNumber* – once again starting with 1 for the first node parsed, and incrementing by 1 for each additional node parsed. Each node that is parsed is given an *node identifier* (*nodeID*), which is in the format *`docNumber`.nodeNumber*, where *`docNumber`*.0 indicates the document itself, *docID*.1 is the first node parsed, and so on.

Example

The following example loads XML from a user-selected file on disk:

```
on mouseUp
  local tDocNum
  answer file "Pick an XML file:" with filter "XML Files (*.xml)"&cr&"*.xml"
  if it <> "" then
    put XMLLoadData(url ("file:" & it)) into tDocNum
    -- If first time, tDocNum = "1"
  end if
end mouseUp
```

See also

xml_dump, xml_expand

XMLCreateDocument

Std Pro

Summary

This method creates a new XML document in the global array **gXMLData** .

Syntax

`XMLCreateDocument(`*`rootNodeName`*`)` ➪ *`docNumber`*

Arguments

rootNodeName	The name of the root node.

Possible Returned Values

docNumber	The document number for the newly created XML document.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

When executed, this method will create a new XML document, with an XML Declaration (XDEC) node, and a root node whose name is *rootNodeName*, and this data will be added to the global array **gXMLData**. The document is assigned the next available document number, and the root node is created as an empty element. From this point, you can use XMLCreateNode, XMLAppendChild and XMLCreateAttribute to fill out the new document.

Example

The following example:

```
on mouseUp
  get XMLCreateDocument("myRoot")
end mouseUp
```

… creates the following XML document:

```
<?xml version="1.0"?>
<myRoot/>
```

See also

XMLCreateNode, XMLAppendChild, XMLCreateAttribute

XMLDeleteDocument

Basic Std Pro

Summary

This method deletes an XML document from the global array **gXMLData** .

Syntax

`XMLDeleteDocument(`*`docNumber`*`)` ➪ *`docNumberList`*

Arguments

docNumber	The number of the document to delete.

Possible Returned Values

docNumberList	A return-delimited list of document numbers representing all of the documents that are still loaded in memory after the document *docNumber* has been deleted.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method will delete a document from memory, freeing up the space that it and its children took in the gXMLData array.

Example

Assuming that three documents had been created (and none had been subsequently deleted), the following example:

```
on mouseUp
  get XMLDeleteDocument(2)
end mouseUp
```

… would return:

```
1
3
```

See also

XMLLoadData, XMLCreateDocument, XMLGetDocuments

XMLGetDocuments

Basic Std Pro

Summary

This method retrieves a list of XML documents from the global array **gXMLData** .

Syntax

```
XMLGetDocuments() ⇨ docNumberList
```

Arguments

none

Possible Returned Values

docNumberList	A return-delimited list of document numbers representing all of the documents that are loaded in memory and have not been deleted with XMLDeleteDocument.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method will retrieve the list of document numbers created via calls to XMLLoadData or XMLCreateDocument. The list is return-delimited, with one document number on each line. Note that if you have deleted any documents via a call to XMLDeleteDocument, you may not have a continuous sequence of document numbers.

Example

Assuming that three documents had been created, the following example:

```
on mouseUp
  get XMLGetDocuments()
end mouseUp
```

… would return:

```
1
2
3
```

See also

XMLLoadData, XMLCreateDocument, XMLDeleteDocument

XMLCreateNode

Std Pro

Summary

This method creates a new node in the global array **gXMLData**, either as a sibling or child of an existing node.

Syntax

`XMLCreateNode(`*`where,nodeID,nodeType,nodeName[,nodeContents]`*`)` ➪ *`newNodeID`*

Arguments

`where`	The location of where to create the node. Can be one of the following constants: `before` Creates the new node as a sibling to *nodeID*, but in a position prior to *nodeID*. That is, if *nodeID* is the second child of its parent, the new node will be created as the second child, and *nodeID* will move down to become the third child. `after` Creates the new node as a sibling to *nodeID*, but in a position after *nodeID*. That is, if *nodeID* is the second child of its parent, the new node will be created as the third child, and any existing child that was originally the third child will move down to become the fourth child. `child` (Read: "as a child of") Creates a new node that becomes a child of *nodeID*. If *nodeID* has no children, the new node created will become the only child of *nodeID*. If *nodeID* has children, the new node created will become the last child of *nodeID* (i.e. if *nodeID* has three children, the new node created would be the fourth child).
`nodeID`	The reference node to which the new node will be created. This may not be the node ID of a DOC node (if you want to create nodes at the root level, you will need to create it as a sibling of an existing root-level node using `before` or `after`).
`nodeType`	The type of node to be created. (See Node Properties for a list of appropriate type codes.) You may not create DOC nodes; to create documents, use XMLCreateDocument.
`nodeName`	The name of the node to be created. This is only necessary for ELEM or PROC nodes, and can be left blank for any other node type. (If supplied, it will be ignored for non-ELEM and non-PROC nodes.)
`nodeContents`	Optional. This is the contents of the node. The content goes between the start tag and end tag of the node type. For example, a CDATA node's contents would appear between the `<![CDATA[` start tag and the `]]>` end tag.

Possible Returned Values

`newNodeID`	The node ID of the node that was created; the result of incrementing the `last` node property of the document by one (i.e. if **gXMLData("1.0last")** was `20` before this method was executed, the new node ID would be `1.21`, and **gXMLData("1.0last")** would now be `21`).
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method creates a new node, either as a child or a sibling of an existing node. You can only create children of ELEM nodes, although you can create a sibling of any node. Nodes that are created are validated for conformity before being added; if a nodes contains invalid data, an error will result. If you create a new ELEM node, it is created as an empty element (i.e. self-enclosed). If you add a node as a child to an empty element, it will change to be a normal element with start and end tags, with the newly created child inbetween.

Note that *nodeName* needs to follow the restrictions of the node format `Name` (see Node Formats, above).

Example 1: "Before"

The following example:

```
on mouseUp
  answer XMLCreateNode("before","1.5","ELEM","doohickeys","")
end mouseUp
```

… would create a new ELEM node and would make our XML example look like this…

```
1.0    (the document itself)
1.1    <?xml version="1.0"?>

1.2    <products>
1.3      <!-- The widgets below are all from Australia -->
1.4      <?runAction-checkStock?>
1.10     <doohickeys/>
1.5      <widgets>
1.6        <widget id="1" name="Thingimajig" color="navy">
1.7          <![CDATA[The <b>NEW</b> Thingimajig!
       Comes in 5 new colors!]]>
           </widget>
1.8        <widget id="3" name="Geegaw" color="red">
1.9          The old Geegaw. Anyone want to buy one?
           </widget>
         </widgets>
       </products>
```

… and would display `1.10` in the resulting dialog box.

Example 2: "After"

The following example:

```
on mouseUp
  answer XMLCreateNode("after","1.5","CDATA","Go","")
end mouseUp
```

… would create a new CDATA node and would make our XML example look like this…

```
1.0    (the document itself)
1.1    <?xml version="1.0"?>

1.2    <products>
1.3      <!-- The widgets below are all from Australia -->
1.4      <?runAction-checkStock?>
1.10     <doohickeys/>
1.5      <widgets>
1.6        <widget id="1" name="Thingimajig" color="navy">
1.7          <![CDATA[The <b>NEW</b> Thingimajig!
       Comes in 5 new colors!]]>
           </widget>
1.8        <widget id="3" name="Geegaw" color="red">
1.9          The old Geegaw. Anyone want to buy one?
```

```
            </widget>
          </widgets>
1.11      <?Go?>
        </products>
```

… and would display `1.11` in the resulting dialog box.

Example 3: "Child"

The following example:

```
on mouseUp
  answer XMLCreateNode("child","1.10","CMNT","","This is a comment")
end mouseUp
```

… would create a new Comment node and would make our XML example look like this…

```
1.0     (the document itself)
1.1     <?xml version="1.0"?>

1.2     <products>
1.3       <!-- The widgets below are all from Australia -->
1.4       <?runAction-checkStock?>
1.10      <doohickeys>
1.12        <!--This is a comment-->
          </doohickeys>
1.5       <widgets>
1.6         <widget id="1" name="Thingimajig" color="navy">
1.7           <![CDATA[The <b>NEW</b> Thingimajig!
        Comes in 5 new colors!]]>
            </widget>
1.8         <widget id="3" name="Geegaw" color="red">
1.9           The old Geegaw. Anyone want to buy one?
            </widget>
          </widgets>
1.11      <?Go?>
        </products>
```

… and would display `1.12` in the resulting dialog box.

See also

XMLAppendChild, XMLDeleteNode

XMLDeleteNode

Std Pro

Summary

This method deletes a node from the global array **gXMLData** .

Syntax

```
XMLDeleteNode(nodeID) ➩ “”
```

Arguments

nodeID	The node to be deleted.

Possible Returned Values

(empty)	The node was successfully deleted.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method will delete the node supplied in *nodeID*.

Example

The following example:

```
on mouseUp
  get XMLDeleteNode("1.5")
end mouseUp
```

… will delete node 1.5. Note that nodes are not renumbered.

See also

XMLCreateNode, XMLAppendChild

XMLGetNode

Basic Std Pro

Summary

This method retrieves the raw XML for a specified node (and optionally its children).

Syntax

```
XMLGetNode(nodeID[,includeChildren[,normalize]]) ⇨ nodeContents
```

Arguments

nodeID	The node that will be retrieved.
includeChildren	Optional. Must be either `true` or `false`; `false` is used by default if this argument is omitted. If `true`, the XML of all children (recursively) of *nodeID* will be retrieved . If `false`, only the XML of *nodeID* will be retrieved.
normalize	Optional. Must be either `true` or `false`; `true` is used by default if this argument is omitted. If `true`, the XML retrieved is artificially normalized according to the rules in the section Normalizing Data, above. If `false`, the XML retrieved will not be normalized; it will be returned “raw”.

Possible Returned Values

nodeContents	The retrieved XML data.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method gets the raw XML from the node supplied in *nodeID*.

If *includeChildren* is false or not provided, what is retrieved is the start tag of the node supplied in *nodeID* (e.g. the `strt` node property). If *includeChildren* is true, what is retrieved is the start and end tags of the node supplied in *nodeID*, along with all of its children (recursive).

If *normalize* is false or not provided, the data is retrieved exactly as it was read, i.e. in its raw form. If *normalize* is true, then the data returned is *artificially* normalized by passing the data through the utility function xml_normalize (i.e.even data contained in CDATA will be normalized).

Example 1: No Children

The following example:

```
on mouseUp
  get XMLGetNode("1.5")
end mouseUp
```

… returns:

```
<widgets>
```

Example 2: With Children

The following example:

```
on mouseUp
  get XMLGetNode("1.5","true")
end mouseUp
```

… returns:

```
<widgets>
  <widget id="1" name="Thingimajig" color="navy">
    <![CDATA[The <b>NEW</b> Thingimajig!
       Comes in 5 new colors!]]>
  </widget>
  <widget id="3" name="Geegaw" color="red">
    The old Geegaw. Anyone want to buy one?
  </widget>
</widgets>
```

Example 3: Normalized

The following example:

```
get XMLGetNode("1.7","false","false")
```

… returns:

```
<![CDATA[The <b>NEW</b> Thingimajig!
Comes in 5 new colors!]]>
```

whereas:

```
get XMLGetNode("1.7","false","true")
```

… returns:

```
<![CDATA[The <b>NEW</b> Thingimajig! Comes in 5 new colors!]]>
```

See also

XMLGetNodeName, XMLGetNodeType, XMLGetRoot

XMLGetNodeName Basic Std Pro

Summary

This method retrieves the `name` node property from a node.

Syntax

`XMLGetNodeName(`*`nodeID`*`)` ⇨ *`nodeName`*

Arguments

nodeID	The node from which the name will be retrieved.

Possible Returned Values

nodeName	The name of the node supplied in *nodeID*.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the `name` node property from the node supplied in *nodeID*. Note that only PROC and ELEM node types have a `name` node property; calling this method on any other node type will return an empty string (“”).

Example

The following example:

```
on mouseUp
  get XMLGetNodeName("1.8")
end mouseUp
```

… returns:

```
widget
```

See also

XMLGetNode, XMLGetNodeType, XMLGetRoot

XMLGetNodeType

Basic Std Pro

Summary

This method retrieves the `type` node property from a node.

Syntax

```
XMLGetNodeType(nodeID) ➪ nodeType
```

Arguments

nodeID	The node from which the name will be retrieved.

Possible Returned Values

nodeType	The node type of the node supplied in *nodeID*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the `type` node property from the node supplied in *nodeID*. This node property is supported in all nodes, including the document itself. See the section Node Types for a complete list of the returnable node types.

Example

The following example:

```
on mouseUp
  get XMLGetNodeType("1.8")
end mouseUp
```

… returns:

```
ELEM
```

See also

XMLGetNode, XMLGetNodeName, XMLGetRoot

XMLGetRoot

Basic Std Pro

Summary

This method retrieves the `root` node property from a document node.

Syntax

`XMLGetRoot(`*`docNumber`*`)` ⇨ *`rootNodeID`*

Arguments

`docNumber`	The node from which the name will be retrieved.

Possible Returned Values

`rootNodeID`	The ID of the root node from the document supplied in *docNumber*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the node ID of the root node from the document supplied in *docNumber*.

Example

The following example:

```
on mouseUp
  get XMLGetRoot("1")
end mouseUp
```

… returns:

```
1.2
```

See also

XMLGetNode, XMLGetNodeName, XMLGetNodeType

XMLIsEmptyElement Basic Std Pro

Summary

This method determines whether an ELEM node is an Empty Element or not.

Syntax

`XMLIsEmptyElement(`*`nodeID`*`) ➪ {true|false}`

Arguments

nodeID	The node to be examined.

Possible Returned Values

`true`	The node is an Empty Element node.
`false`	The node is not an Empty Element node.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method checks an ELEM node to determine if it is an Empty Element (i.e. self-enclosing, as in `<myTag/>`) or not. This only works on ELEM nodes; attempts to use this on non-ELEM nodes will return an error in **gXMLError**.

Example

The following example:

```
on mouseUp
  get XMLIsEmptyElement("1.5")
end mouseUp
```

… returns:

```
false
```

XMLAppendChild

Std Pro

Summary

This method adds a child node to an ELEM node.

Syntax

`XMLAppendChild(nodeID,nodeType,nodeName[,nodeContents])` ⇨ *`newNodeID`*

Arguments

`nodeID`	The reference node to which the new child will be added. This may not be the node ID of a DOC node (if you want to create nodes at the root level, you will need to create it as a sibling of an existing root-level node using `before` or `after`).
`nodeType`	The type of node to be created. (See Node Properties for a list of appropriate type codes.) You may not create DOC nodes; to create documents, use XMLCreateDocument.
`nodeName`	The name of the node to be created. This is only necessary for ELEM or PROC nodes, and can be left blank for any other node type. (If supplied, it will be ignored for non-ELEM and non-PROC nodes.)
`nodeContents`	Optional. This is the contents of the node. The content goes between the start tag and end tag of the node type. For example, a CDATA node's contents would appear between the `<![CDATA[` start tag and the `]]>` end tag.

Possible Returned Values

`newNodeID`	The node ID of the node that was added; the result of incrementing the `last` node property of the document by one (i.e. if **gXMLData("1.0last")** was `20` before this method was executed, the new node ID would be `1.21`, and **gXMLData("1.0last")** would now be `21`).
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method creates a new node as a child of an existing node. This does the same thing as XMLCreateNode (passing `child` in the first parameter). For more information, see the Description under XMLCreateNode.

Note that *nodeName* needs to follow the restrictions of the node format `Name` (see Node Formats, above).

Example

The following example:

```
on mouseUp
  answer XMLAppendChild("1.10","CMNT","","This is a comment")
end mouseUp
```

… inserts a new node as a child of node 1.10. To see the results of calling this method, see Example 3 under XMLCreateNode.

See also

XMLCreateNode, XMLDeleteNode

XMLCountChildren

Basic Std Pro

Summary

This method returns the number of child nodes owned by *nodeID*.

Syntax

`XMLCountChildren(`*`nodeID`*`)` ➩ *`numberOfChildren`*

Arguments

nodeID	The node to be examined.

Possible Returned Values

numberOfChildren	The number of children owned by *nodeID*.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method returns the number of child nodes owned by *nodeID*. Although only ELEM and DOC nodes can have children, you can pass any node type to this method (it will return 0 for non-ELEM/DOC nodes). Note that this counts direct children, and is not recursive. If you wish to get a count on recursive nodes, you will need to call XMLCountChildren recursively for each child.

Example

The following example:

```
on mouseUp
  get XMLCountChildren("1.5")
end mouseUp
```

… returns:

```
2
```

See also

XMLCountNamedChildren, XMLHasChildren

XMLCountNamedChildren Basic Std Pro

Summary

This method returns the number of child nodes of a node that have a specific name.

Syntax

`XMLCountNamedChildren(nodeID,childName)` ➪ *`numberOfChildren`*

Arguments

`nodeID`	The node to be examined.
`childName`	The name of the ELEM to match.

Possible Returned Values

`numberOfChildren`	The number of children of *nodeID* whose `name` node property matches *childName*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method returns the number of child nodes owned by *nodeID* whose `name` node property matches *childName*.. Although only ELEM and DOC nodes can have children, you can pass any node type to this method (it will return 0 for non-ELEM/DOC nodes). Note that this counts direct children, and is not recursive. If you wish to get a count on recursive nodes, you will need to call XMLCountChildren recursively for each child.

Example

The following example:

```
on mouseUp
  get XMLCountNamedChildren("1.2","widgets")
end mouseUp
```

… returns:

```
1
```

See also

XMLCountChildren, XMLHasChildren

XMLGetChildren

Basic **Std** **Pro**

Summary

This method retrieves a list of child nodes of a node, optionally of a specific node type.

Syntax

`XMLGetChildren(nodeID[,childNodeType]) ➪ childList`

Arguments

nodeID	The node to be examined.
childNodeType	Optional. The node type of the child to retrieve. If left blank, it will retrieve all types of nodes.

Possible Returned Values

childList	A return-delimited list of node IDs representing the children of *nodeID*.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method returns the return-delimited list of child nodes owned by *nodeID*. If *childNodeType* is supplied, the list only contains node IDs whose type matches *childNodeType*. Although only ELEM and DOC nodes can have children, you can pass any node type to this method (it will return "" for non-ELEM/DOC nodes). Note that this retrieves a list of direct children, and is not recursive. If you wish to get a list on recursive nodes, you will need to call XMLGetChildren recursively for each child.

Example 1:

The following example:

```
on mouseUp
  get XMLGetChildren("1.2")
end mouseUp
```

… returns:

```
1.3
1.4
1.5
```

Example 2:

The following example:

```
on mouseUp
  get XMLGetChildren("1.2","CMNT")
end mouseUp
```

… returns:

```
1.3
```

See also

XMLGetFirstChild, XMLGetNamedChildren, XMLGetLastChild, XMLGetNextSibling, XMLGetParent, XMLGetPrevSibling

XMLGetFirstChild

Basic Std Pro

Summary

This method retrieves the node ID of the first child owned by *nodeID*, optionally of a specific node type..

Syntax

`XMLGetFirstChild(`*`nodeID[,childNodeType]`*`)` ⇨ *`childNodeID`*

Arguments

`nodeID`	The node to be examined.
`childNodeType`	Optional. The node type of the child to retrieve. If left blank, it will retrieve all types of nodes.

Possible Returned Values

`childNodeID`	The node ID of the first child owned by *nodeID*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the node ID of the first child owned by *nodeID*. If *nodeID* does not have any children, this method will return empty (“”). Although only ELEM and DOC nodes can have children, you can pass any node type to this method (it will return “” for non-ELEM/DOC nodes).

Example 1:

The following example:

```
on mouseUp
  get XMLGetFirstChild("1.5")
end mouseUp
```

… returns:

```
1.6
```

Example 2:

The following example:

```
on mouseUp
  get XMLGetFirstChild("1.2","ELEM")
end mouseUp
```

… returns:

```
1.5
```

See also

XMLGetChildren, XMLGetNamedChildren, XMLGetLastChild, XMLGetNextSibling, XMLGetParent, XMLGetPrevSibling

XMLGetNamedChildren

Basic Std Pro

Summary

This method returns a list of child nodes of a node that have a specific name.

Syntax

`XMLGetNamedChildren(`*`nodeID,childName`*`)` ⇨ *`childList`*

Arguments

`nodeID`	The node to be examined.
`childName`	The name of the ELEM to match.

Possible Returned Values

`childList`	A return-delimited list of node IDs representing the children of *nodeID* whose `name` node property matches *childName*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method returns the return-delimited list of child nodes owned by *nodeID* whose `name` node property matches *childName*. Although only ELEM and DOC nodes can have children, you can pass any node type to this method (it will return "" for non-ELEM/DOC nodes). Note that this retrieves a list of direct children, and is not recursive. If you wish to get a list on recursive nodes, you will need to call XMLGetNamedChildren recursively for each child.

Example

The following example:

```
on mouseUp
  get XMLGetNamedChildren("1.2","widgets")
end mouseUp
```

… returns:

```
1.5
```

See also

XMLGetChildren, XMLGetFirstChild, XMLGetLastChild, XMLGetNextSibling, XMLGetParent, XMLGetPrevSibling

XMLGetLastChild

Basic Std Pro

Summary

This method retrieves the node ID of the last child owned by *nodeID*, optionally of a specific node type..

Syntax

`XMLGetLastChild(`*`nodeID[,childNodeType]`*`)` ➪ *`childNodeID`*

Arguments

`nodeID`	The node to be examined.
`childNodeType`	Optional. The node type of the child to retrieve. If left blank, it will retrieve all types of nodes.

Possible Returned Values

`childNodeID`	The node ID of the last child owned by *nodeID*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the node ID of the last child owned by *nodeID*. If *nodeID* does not have any children, this method will return empty (“”). Although only ELEM and DOC nodes can have children, you can pass any node type to this method (it will return “” for non-ELEM/DOC nodes).

Example 1:

The following example:

```
on mouseUp
  get XMLGetLastChild("1.5")
end mouseUp
```

… returns:

```
1.8
```

Example 2:

The following example:

```
on mouseUp
  get XMLGetLastChild("1.2","CMNT")
end mouseUp
```

… returns:

```
1.3
```

See also

XMLGetChildren, XMLGetNamedChildren, XMLGetFirstChild, XMLGetNextSibling, XMLGetParent, XMLGetPrevSibling

XMLGetNextSibling

Basic **Std** **Pro**

Summary

This method retrieves the node ID of the next sibling of *nodeID*, optionally of a specific node type..

Syntax

```
XMLGetNextSibling(nodeID[,siblingNodeType]) ⇨ siblingNodeID
```

Arguments

nodeID	The node to be examined.
siblingNodeType	Optional. The node type of the next sibling to retrieve. If left blank, it will retrieve the next sibling, regardless of type.

Possible Returned Values

siblingNodeID	The node ID of the next sibling of *nodeID*.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the node ID of the next sibling of *nodeID*. If *nodeID* is the last child of its parent node, this method will return empty (“”). Note that although ELEM and DOC nodes are the only nodes that can have children, these children can be of any type, and therefore the siblings of children can be of any type as well.

Example 1:

The following example:

```
on mouseUp
  get XMLGetNextSibling("1.6")
end mouseUp
```

… returns:

```
1.8
```

Example 2:

The following example:

```
on mouseUp
  get XMLGetNextSibling("1.3","ELEM")
end mouseUp
```

… returns:

```
1.5
```

See also

XMLGetChildren, XMLGetNamedChildren, XMLGetFirstChild, XMLGetLastChild, XMLGetParent, XMLGetPrevSibling

XMLGetParent

Basic Std Pro

Summary

This method returns the node ID of the parent of *nodeID*.

Syntax

```
XMLGetParent(nodeID) ➪ parentNodeID
```

Arguments

nodeID	The node to be examined.

Possible Returned Values

parentNodeID	The node ID of the parent of *nodeID*.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method returns the node ID of the parent of the node supplied in *nodeID*. All nodes have a parent, with the exception of the DOC node, which will return empty (“”).

Example

The following example:

```
on mouseUp
  get XMLGetParent("1.1")
end mouseUp
```

… doesSomething

```
1.0
```

See also

XMLGetChildren, XMLGetNamedChildren, XMLGetFirstChild, XMLGetLastChild, XMLGetNextSibling, XMLGetPrevSibling

XMLGetPreviousSibling Basic Std Pro

Summary

This method retrieves the node ID of the previous sibling of *nodeID*, optionally of a specific node type..

Syntax

`XMLGetPreviousSibling(`*`nodeID[,siblingNodeType]`*`)` ⇨ *`siblingNodeID`*

Arguments

nodeID	The node to be examined.
siblingNodeType	Optional. The node type of the previous sibling to retrieve. If left blank, it will retrieve the previous sibling, regardless of type.

Possible Returned Values

siblingNodeID	The node ID of the previous sibling of *nodeID*.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the node ID of the previous sibling of *nodeID*. If *nodeID* is the first child, this method will return empty (“”). Note that although ELEM and DOC nodes are the only nodes that can have children, these children can be of any type, and therefore the siblings of children can be of any type as well.

Example 1:

The following example:

```
on mouseUp
  get XMLGetPreviousSibling("1.8")
end mouseUp
```

… returns:

```
1.6
```

Example 2:

The following example:

```
on mouseUp
  get XMLGetPreviousSibling("1.5","CMNT")
end mouseUp
```

… returns:

```
1.3
```

See also

XMLGetChildren, XMLGetNamedChildren, XMLGetFirstChild, XMLGetLastChild, XMLGetParent, XMLGetNextSibling

XMLHasChildren **Basic** **Std** **Pro**

Summary

This method determines whether *nodeID* has children or not.

Syntax

```
XMLHasChildren(nodeID[,nodeType]) ➪ {true|false}
```

Arguments

nodeID	The node to be examined.
nodeType	Optional. The node type of the child to check for. If left blank, it will check for any type of node.

Possible Returned Values

`true`	The node *nodeID* has children.
`false`	The node *nodeID* does not have children.
(empty)	The node *nodeID* is not an ELEM or DOC node.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method determines whether the node supplied in *nodeID* has children, and optionally if *nodeID* has a specific type of children. Although only ELEM and DOC nodes can have children, you can pass any node type to this method (it will return "" for non-ELEM/DOC nodes). Note that this only checks the direct children of *nodeID*, and not its children's children.

Example 1:

The following example:

```
on mouseUp
  get XMLHasChildren("1.5")
end mouseUp
```

… returns:

```
true
```

Example 2:

The following example:

```
on mouseUp
  get XMLGetChildren("1.5","CDAT")
end mouseUp
```

… returns:

```
false
```

See also

XMLCountChildren, XMLCountNamedChildren

XMLGetCDATA

Basic Std Pro

Summary

This method retrieves the CDATA contents of a CDAT node.

Syntax

`XMLGetCDATA(`*`CDATAnodeID`*`)` ➪ *`nodeContents`*

Arguments

CDATAnodeID	The CDAT node to be examined.

Possible Returned Values

nodeContents	The contents of the CDAT node (not normalized).
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the CDATA contents of a CDAT node, and will only work on CDAT nodes (it will return an error if any other node type is supplied). The contents are returned as is and not normalized (as it should be), including all tags, white space, etc. owned by the node. If you need to normalize the data (for whatever reason), you can pass the results of this method to the xml_normalize utility method.

Example

The following example:

```
on mouseUp
  get XMLGetCDATA("1.7")
end mouseUp
```

… returns:

```
The <b>NEW</b> Thingimajig!
        Comes in 5 new colors!
```

See also

XMLGetText, xml_normalize

XMLGetText

Basic Std Pro

Summary

This method retrieves the text content of a TEXT node.

Syntax

`XMLGetText(textNodeID[,normalize])` ➪ *nodeContents*

Arguments

textNodeID	The TEXT node being referenced.
normalize	Optional. Can be either `true`, `false` or left blank. If `true`, the results will be normalized. If `false`, the results will not be normalized. If left blank, the results will be normalized.

Possible Returned Values

nodeContents	The contents of the TEXT node.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the text contents of a TEXT node, and will only work on TEXT nodes (it will return an error if any other node type is supplied). The contents are returned normalized, unless you pass `false` for *normalize*.

Example

The following example:

```
on mouseUp
  get XMLGetText("1.9")
end mouseUp
```

… returns:

```
The old Geegaw. Anyone want to buy one?
```

See also

XMLGetCDATA

XMLCountAttributes

Basic **Std** **Pro**

Summary

This method returns the number of attributes owned by *nodeID*.

Syntax

`XMLCountAttributes(nodeID) ➪ numberOfAttribs`

Arguments

nodeID	The node being examined.

Possible Returned Values

numberOfAttribs	The number of attributes owned by *nodeID*.
(empty)	The node passed is not an ELEM or XDEC node.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method returns the number of attributes owned by *nodeID*. Only ELEM and XDEC nodes can have attributes, although you can pass any node type to this method (it will return empty in this case).

Example

The following example:

```
on mouseUp
  get XMLCountAttributes("1.6")
end mouseUp
```

… returns:

```
3
```

See also

XMLGetAttributes

XMLCreateAttribute Std Pro

Summary

This method creates a new attribute for the node supplied in *nodeID*.

Syntax

`XMLCreateAttribute(nodeID,attribName,attribValue) ⇨ ""`

Arguments

nodeID	The node ID of the reference node.
attribName	The name of the attribute to be created.
attribValue	The value for the attribute *attribName*.

Possible Returned Values

(empty)	The attribute was successfully created.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method creates a new attribute named *attribName* with the value *attribValue* for the node supplied in *nodeID*. Note that *attribName* needs to follow the restrictions of the node format `Name` and *attribValue* needs to follow the restrictions of the node format `AttValue` (see Node Formats, above).

Example

The following example:

```
on mouseUp
  get XMLCreateAttribute("1.2","season","spring")
  get XMLGetNode("1.2")
end mouseUp
```

… returns:

```
<products season="spring">
```

See also

XMLDeleteAttribute, XMLGetAttribute, XMLHasAttribute, XMLSetAttribute

XMLDeleteAttribute **Std** **Pro**

Summary

This method deletes an attribute supplied in *attribName* from the node supplied in *nodeID*.

Syntax

```
XMLDeleteAttribute(nodeID,attribName) ⇨ “”
```

Arguments

nodeID	The node ID of the reference node.
attribName	The name of the attribute to be deleted.

Possible Returned Values

(empty)	The attribute was successfully deleted.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method deletes the attribute *attribName* from the node supplied in *nodeID*. The attribute, its value and the extra space that my follow it is deleted cleanly from *nodeID*.

Example

The following example:

```
on mouseUp
  get XMLDeleteAttribute("1.6","name")
  get XMLGetNode("1.6")
end mouseUp
```

… returns:

```
<widget id="1" color="navy">
```

See also

XMLCreateAttribute, XMLGetAttribute, XMLHasAttribute, XMLSetAttribute

XMLGetAttribute

Basic Std Pro

Summary

This method retrieves the value of the attribute *attribName* for the node provided in *nodeID*.

Syntax

```
XMLGetAttribute(nodeID,attribName) ➪ attribValue
```

Arguments

nodeID	The node ID of the reference node.
attribName	The name of the attribute whose value is to be retrieved.

Possible Returned Values

attribValue	The value of the attribute *attribName*.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves the value of the attribute *attribName* from the node provided in *nodeID*. *attribValue* will be the string that is inside the quotation marks of the attribute's value. If *attribValue* has no value (i.e. it is something like `color=""`) then you will get empty in the result.

Example

The following example:

```
on mouseUp
  get XMLGetAttribute("1.6","name")
end mouseUp
```

… returns:

```
Thingimajig
```

See also

XMLCreateAttribute, XMLDeleteAttribute, XMLGetAttributes, XMLHasAttribute, XMLSetAttribute

XMLGetAttributes

Basic Std Pro

Summary

This method retrieves the list of attributes possessed by the node supplied in *nodeID*.

Syntax

`XMLGetAttributes(`*`nodeID`*`)` ➯ *`attributeList`*

Arguments

nodeID	The node ID of the reference node.

Possible Returned Values

attributeList	A return-delimited list of attribute names.
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method retrieves a return-delimited list of attributes possessed by the node supplied in *nodeID*, in the order they appear in the start tag of *nodeID*, from left to right. If *nodeID* does not have any attributes, this method will return empty (“”).

Example

The following example:

```
on mouseUp
  get XMLGetAttributes("1.6")
end mouseUp
```

… returns:

```
id
name
color
```

See also

XMLCountAttributes, XMLGetAttribute

XMLHasAttribute

Basic Std Pro

Summary

This method determines whether *nodeID* has the attribute *attribName* as one of its attributes.

Syntax

```
XMLHasAttribute(nodeID,attribName) ⇨ {true|false}
```

Arguments

`nodeID`	The node ID of the reference node.
`attribName`	The name of the attribute to check for.

Possible Returned Values

`true`	The node *nodeID* has the attribute *attribName* as one of its attributes.
`false`	The node *nodeID* does not have the attribute *attribName* as one of its attributes.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method determines whether the node supplied in *nodeID* has the attribute *attribName* as one of its attributes.

Note that only ELEM and XDEC nodes can have attributes; this method will return `false` if used on any other node type. It is therefore recommended that you verify that the node you are examining is an ELEM or XDEC node through the use of XMLGetNodeType before executing this method.

Example

The following example:

```
on mouseUp
  get XMLHasAttribute("1.6","name")
end mouseUp
```

… returns:

```
true
```

See also

XMLCreateAttribute, XMLDeleteAttribute, XMLGetAttribute, XMLGetAttributes, XMLSetAttribute

XMLSetAttribute Std Pro

Summary

This method sets the value of an existing attribute of *nodeID* to a new value.

Syntax

`XMLSetAttribute(`*`nodeID,attribName,attribValue`*`)` ⇨ "″

Arguments

`nodeID`	The node ID of the reference node.
`attribName`	The name of the attribute whose value is to be changed.
`attribValue`	The new value for *attribName*.

Possible Returned Values

`(empty)`	The attribute was successfully changed.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method changes the value of the attribute supplied in *attribName* of the node supplied in *nodeID* to the new value supplied in *attribValue*. *attribName* must already exist or an error will result.

Note that *attribValue* needs to follow the restrictions of the node format `AttValue` (see Node Formats, above).

Example

The following example:

```
on mouseUp
  get XMLSetAttribute("1.6","name","Doohickey")
  get XMLGetNode("1.6")
end mouseUp
```

… returns:

```
<widget id="1" name="Doohickey" color="navy">
```

See also

XMLCreateAttribute, XMLDeleteAttribute, XMLGetAttribute, XMLGetAttributes, XMLHasAttribute

xml_clearCallback

Basic Std Pro

New in Version

Summary

This method prevents subsequent errors that are discovered from triggering a callback message.

Syntax

```
xml_clearCallback
```

Arguments

```
none
```

Possible Returned Values

```
none
```

Description

After a callback has been set with xml_setCallback, executing this method will prevent subsequent errors that are discovered in parsing the XML from triggering a callback message.

Example:

To clear callbacks:

```
on mouseUp
  xml_clearCallback
end mouseUp
```

See also

xml_setCallback

xml_dump

Basic Std Pro

Summary

This utility method will provide a “dump” of all the nodes of a document, optionally displaying white space characters with symbols.

Syntax

`xml_dump(`*`docNumber[,showInvisibles]`*`)` ➪ *`arrayContent`*

Arguments

`docNumber`	The number of the document to “dump”.
`showInvisibles`	Optional. Can either be `true`, `false` or left blank. If `true`, white space characters are represented using symbols; if `false`, white space is displayed in its normal form (see White Space Symbols, above.)

Possible Returned Values

`arrayContent`	The raw node contents of each node (i.e. what is actually stored in the node property `cont` for each node in **gXMLData**), including node pointers.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This utility method is used to get a complete picture of a document’s contents by returning a “dump” of every node’s contents, in node order from lowest to highest node ID (the highest node ID is the same as the value returned from the document’s `last` node property).

The content (i.e. the `cont` node property of each node) contains everything: white space (which may or may not be converted to symbols – see below), node pointers (in the form `[>>` *nodeID*`]`) and textual content. Each node’s content in the output is preceded by the node ID and node type in the following format (this is known as the *bracketed identifier*):

`[`*nodeID*`:`*nodeType*`]`*nodeContents*

So node ID 1.1 would be displayed as:

```
[1.1:XDEC]<?xml version="1.0"?>
```

If *showInvisibles* is `true`, white space is displayed using the following rules:

- If the white space character is a space, replace it with a small bullet (·).
- If the white space character is a tab, leave the tab character alone and precede it with a double right angle bracket (»).
- If the white space character is a linefeed, leave the linefeed character alone and precede it with a dagger (†).
- If the white space character is a carriage return, leave the carriage return alone and precede it with a continuation symbol (¬).

So node ID 1.1 with *showInvisibles* = `true` would be displayed as:

```
[1.1:XDEC]<?xml·version="1.0"?>
```

Note that white space is “owned” by a node only when it falls between the left angle bracket of the node’s start tag, and the right angle bracket of the node’s end tag. Anything prior to the left angle bracket of the node’s start tag is owned by its parent. (For an example of this, see Putting It All Together, above.)

Example

The following example:

```
on mouseUp
  get xml_dump("1.0")
end mouseUp
```

… returns:

```
[1.0:DOC][>>·1.1]¬†
¬†
[>>·1.2]
[1.1:XDEC]<?xml·version="1.0"?>
[1.2:ELEM]<products>¬†
» [>>·1.3]¬†
» [>>·1.4]¬†
» [>>·1.5]¬†
</products>
[1.3:CMNT]<!--·The·widgets·below·are·all·from·Australia·-->
[1.4:PROC]<?runAction-checkStock?>
[1.5:ELEM]<widgets>¬†
» » [>>·1.6]¬†
» » [>>·1.8]¬†
» </widgets>
[1.6:ELEM]<widget·id="1"·name="Thingimajig"·color="navy">¬†
» » » [>>·1.7]¬†
» » </widget>
[1.7:CDAT]<![CDATA[The·<b>NEW</b>·Thingimajig!·¬†
Comes·in·5·new·colors!]]>
[1.8:ELEM]<widget·id="3"·name="Geegaw"·color="red">¬†
» » » [>>·1.9]¬†
» » </widget>
[1.9:TEXT]The·old·Geegaw.·Anyone·want·to·buy·one?
```

See also

xml_expand

xml_expand

Basic Std Pro

Summary

This method takes a node's content, and expands all of the node pointers, replacing them with actual data (effectively, recreating a chunk of XML content).

Syntax

`xml_expand(nodeID[,normalize])` ⇨ *`xmlContent`*

Arguments

nodeID	The ID of the node whose contents are to be expanded.
normalize	Optional. Can be either `true`, `false` or left blank. If `true`, the returned data will be normalized. If `false` or left blank, the returned data will not be normalized.

Possible Returned Values

xmlContent	The expanded chunk of XML content that references the descendants of *nodeID*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method takes a node's content (its `cont` node property), and replaces any node pointers with the `cont` of the node that it points to, and so on, until all of the node pointers for *nodeID* and its descendants have been expanded. To expand a whole document, pass the document node number (`1.0`, `2.0`, etc.) for *nodeID*.

The general purpose of this utility is to convert data held in the **gXMLData** global array into actual XML data that can be manipulated (written to a file, stored in a variable, etc.). In general, this data should be returned intact (that is, the *normalize* argument should be `false` or left blank), although there may be some instances where it is useful to have the data normalized (for more information, see Normalizing Data, above).

Example 1:

The following example:

```
on mouseUp
  get xml_expand("1.5")
end mouseUp
```

… will take the `cont` of node 1.5 (white space shown):

```
<widgets>¬†
» » [>>·1.6]¬†
» » [>>·1.8]¬†
» </widgets>
```

… and will expand node 1.6 (which in turn expands node 1.7) and node 1.8, resulting in the following (white space shown – note the rules of who "owns" the white space before the opening < of the *nodeID*):

```
<widgets>¬†
» » <widget id="1" name="Thingimajig" color="navy">¬†
» » » <![CDATA[The <b>NEW</b> Thingimajig!¬†
Comes in 5 new colors!]]>¬†
» » </widget>¬†
» » <widget id="3" name="Geegaw" color="red">¬†
» » » The old Geegaw. Anyone want to buy one?¬†
» » </widget>¬†
» </widgets>
```

Example 2:

The following example expands a whole document:

```
on mouseUp
  get xml_expand("1.0")
end mouseUp
```

See also

xml_dump

xml_getPath

Basic Std Pro

New in Version

Summary

This method retrieves a node and translates it to the XPATH format for use in other applications that need a node identifier in this format.

Syntax

xml_getPath(*nodeID*) ➪ *xPath*

Arguments

nodeID	The node ID number to examine.

Possible Returned Values

xPath	The path to the node supplied in *nodeID* in XPATH format
error	An error was found during execution. Check **gXMLError** for the error found.

Description

This method converts a node ID to an XPATH formatted node identifier. XPATH's format is as follows:

```
//root/parent/parent/child
```

If the there is more than one child node of the same name, the numeric index to that child is supplied in parentheses after the child name (see the example below).

Example 1:

Here's an example of getting a path to a node with no similarly-named siblings:

```
on mouseUp
  get xml_getPath("1.5")
end mouseUp
```

… returns:

```
//products/widgets
```

Example 2:

Here's an example of getting a path to a node *with* similarly-named siblings (uses the index number):

```
on mouseUp
  get xml_getPath("1.8")
end mouseUp
```

… returns:

```
//products/widgets/widget(2)
```

See also

n/a

xml_isDocument

Basic Std Pro

Summary

This method determines whether the document number provided in *docNumber* is valid or not.

Syntax

```
xml_isDocument(docNumber) ➯ {true|false}
```

Arguments

`docNumber`	The document number to examine. Note that this is the document *number*, not its node ID.

Possible Returned Values

`true`	The document number *docNumber* is a valid, parsed document.
`false`	The document number *docNumber* is not a valid, parsed document.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method checks the number supplied in *docNumber* against the list of parsed documents in **gXMLData** (effectively calling XMLGetDocuments) to see if that document number exists in the list. If it exists, this method returns `true`. If it does not exist, it returns `false`. Note that a document may not exist if it had previously been deleted through use of the XMLDeleteDocument method.

Example

If no documents have yet been parsed, the following example:

```
on mouseUp
  get xml_isDocument("1")
end mouseUp
```

… returns:

```
false
```

See also

xml_isNode, xml_isType, XMLGetDocuments, XMLDeleteDocument

xml_isNode

Basic Std Pro

Summary

This method determines whether the node ID provided in *nodeID* is valid or not.

Syntax

`xml_isNode(nodeID) ➪ {true|false}`

Arguments

nodeID	The node ID number to examine.

Possible Returned Values

`true`	The node ID provided in *nodeID* is a valid node ID.
`false`	The node ID provided in *nodeID* is not a valid node ID.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method checks the node ID supplied in *nodeID* to determine if it is valid by examining the `type` node property of *nodeID* and seeing if it has a value. Since all nodes must have a type, if this returns empty (“”), the node does not exist, otherwise it exists. If it exists, this method returns `true`. If it does not exist, it returns `false`. Note that a node may not exist if it had previously been deleted through use of the XMLDeleteNode method.

Example

Assuming no nodes had been deleted, this example:

```
on mouseUp
  get xml_isNode("1.1")
end mouseUp
```

… returns:

```
true
```

See also

xml_isDocument, xml_isType, XMLDeleteNode

xml_isType

Basic Std Pro

Summary

This method determines whether the string passed in *typeCode* is a valid type code or not, optionally including DOC in the list of valid type codes to compare against.

Syntax

`xml_isType(`*`typeCode[,includeDOC]`*`)` ⇨ `{true|false}`

Arguments

`typeCode`	The string which is to be evaluated.
`includeDOC`	Optional. If `true`, the DOC type code will be checked as well as the other valid type codes. If `false` or omitted, the DOC type code will not be checked.

Possible Returned Values

`true`	The string *typeCode* is a valid type code.
`false`	The string *typeCode* is not a valid type code.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method determines whether the string passed in *typeCode* is a valid type code or not by comparing it against a list of valid type codes (see Node Types, above, for a complete list of type codes). By default, the DOC type code is omitted from the list. If you wish to include the DOC type code for the purposes of comparison, pass `true` for *includeDOC*.

Example 1:

The following example:

```
on mouseUp
  get xml_isType("DOC")
end mouseUp
```

… returns:

```
false
```

Example 2:

The following example:

```
on mouseUp
  get xml_isType("DOC",true)
end mouseUp
```

… returns:

```
true
```

See also

xml_isDocument, xml_isNode

xml_libVersion

Basic Std Pro

Summary

This method retrieves the version of the XML Library.

Syntax

```
xml_getVersion() ➪ xmlLibVersionInfo
```

Arguments

```
none
```

Possible Returned Values

xmlLibVersionInfo	The version of the XML Library.

Description

This method retrieves the version number, the last modified date (in the format mm/dd/yyyy) and the library type (Basic, Standard or Professional) of the XML Library in a three line, return-delimited result in the form:

xmlLibVersion
lastModifiedDate
xmlLibType

Example

The following example:

```
on mouseUp
  get xml_libVersion()
end mouseUp
```

… returns:

```
1.0
05/13/2002
Standard
```

See also

n/a

xml_normalize

Basic Std Pro

Summary

This method normalizes any data string passed to it.

Syntax

`xml_normalize(`*`dataString`*`)` ➪ *`normalizedData`*

Arguments

`dataString`	The string of data to be normalized.

Possible Returned Values

`normalizedData`	The string *dataString*, after it has been normalized.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method normalizes any data string passed to it according to the rules found in Normalizing Data, above.

Example

The following example:

```
on mouseUp
  put "This is     a " & cr & tab & linefeed & "test." into tData
  get xml_normalize(tData)
end mouseUp
```

… returns:

```
This is a test.
```

See also

n/a

xml_reset

Basic Std Pro

Summary

This method clears out the **gXMLData** array.

Syntax

```
xml_reset
```

Arguments

```
none
```

Possible Returned Values

```
none
```

Description

This method clears out the data in the **gXMLData** array, effectively deleting all documents and nodes that were currently in memory. Use this with caution!

Example

The following example:

```
on mouseUp
  xml_reset
end mouseUp
```

… deletes the **gXMLData** array.

See also

n/a

xml_setCallback

Basic Std Pro

New in Version

Summary

This method installs a callback message to be sent whenever an error is encountered.

Syntax

```
xml_setCallback targetObject,messageName
```

Arguments

targetObject	A valid object identifier to act as the target of the message that will be sent.
messageName	The name of the message to be sent to *targetObject*.

Possible Returned Values

error	An error was found during execution. Check **gXMLError** for the error found.

Description

Basically what this method does is to send the message *messageName* to the object *targetObject* whenever an error is encountered by the XML Library, **after** gXMLError has been filled with the error message. This can be used, combined with the **exit to top** command, to provide a generic error trapping routine that can bail out of script processing if anything goes wrong (so you don't need to keep checking **gXMLError** after every XML method) (See Example 1).

Used withoug **exit to top** allows script processing to continue, perhaps for error logging (see Example 2).

To remove the callback, use xml_clearCallback.

Example 1:

To set up a callback if something goes wrong:

```
on mouseUp
  xml_setCallback (the long id of me),"BailOut"
end mouseUp

on BailOut
  answer "Sorry, an error has occurred."
  exit to top
end BailOut
```

Example 2:

To set up a callback for logging purposes:

```
on mouseUp
  xml_setCallback (the long id of me),"LogError"
end mouseUp

on LogError
  global gXMLError
  LogIt gXMLError  -- call custom error logging routine
  -- without 'exit to top', script processing continues
end LogError
```

See also

xml_clearCallback

xml_validateNode

Basic Std Pro

Summary

This method validates the contents of an XML data string against a node type to determine if it is well-formed or not.

Syntax

```
xml_validateNode(nodeType,dataString) ➯ {true|false}
```

Arguments

nodeType	The four character node type that is to be used in validation.
dataString	The XML data string that is to be validated.

Possible Returned Values

`true`	The data in *dataString* is well-formed for the type of node provided in *nodeType*.
`false`	The data in *dataString* is not well-formed for the type of node provided in *nodeType*.
`error`	An error was found during execution. Check **gXMLError** for the error found.

Description

This method is used to determine whether an examined piece of XML data matches the rules for well-formedness provided by the W3C by seeing if it matches the proper node format for the node type provided in *nodeType*. For a complete listing of node formats by node type, see Node Formats, above.

Example

Suppose you had this XML data stored in the variable *tempNodeData*:

```
<? fred ?>
```

And you wanted to see if it was a valid PROC node. You would issue:

```
get xml_validateNode("PROC",tempNodeData)
```

… which would return:

```
false
```

… because the node format for a PROC node is

```
<? Name (Any [^(?>)])* ?>
```

… and there is no white space allowed between the `<?` and the `Name`.

See also

n/a