STS XML Library 2.1

Sons of Thunder Software
CUSTOM SOFTWARE DEVELOPMENT AND CONSULTING SERVICES

Introduction

The STS XML Library ("XML Library") allows users of MetaCard[1] and Revolution[2] to read and manipulate XML documents, including specialized XML documents like RSS feeds. The XML Library is fully compliant with the W3C specifications for XML[3], which can be found at http://www.w3.org/TR/REC-xml. There are two 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.

(Note: A professional version supporting DTDs, Schemas, XMLPath, etc. is being considered, but has not yet been developed.)

Although Revolution comes with an XML parsing external (revXML.dll), there are many reasons to use the XML Library instead of revXML.dll. These are fully covered in Appendix III: Comparing the XML Library With Revolution's RevXML.DLL.

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

Using the Library

To use the XML Library, 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.rev"
end openStack
```

You may also desire to use the library as a substack of one of your stacks; to do that, set the `mainstack` of the XML Library to your stack, save your mainstack, and then use a slightly different `start using` command:

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

[1] Must be 2.4.2 or higher. The XML Library makes heavy use of PCRE, which was 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.
[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.

Secondary Libraries (“Plugins”)

Starting with Version 2.0, the XML Library ships with one or more secondary libraries that manipulate specific kinds of XML documents and use the methods that are in the XML Library to do so. For ease of discussion, these are referred to as “plugins”. For example, one plugin may manipulate RSS feeds, and another may manipulate XML exported by Microsoft Word.

Regardless of whether you use the Basic or Standard versions of the XML Library, the stacks that contain the plugin’s code is open for copying and pasting into your own stacks, or you can simply `start using` the library along with the XML Library and call the commands and functions as you need to.

XML Sample

To show how the XML methods (functions) operate, all of the examples will be based on this XML sample, which was picked because it contains all the different kinds of XML elements supported by the XML Library, and included in the distribution of the XML Library:

```
<?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>
```

Parsing XML

The XML Library can work with either precreated blocks of XML formatted data or can create those blocks itself. Major blocks of XML data are called *documents* and correspond to either an actual XML file on disk, XML data that is in a variable, or XML data that is created from scratch with the XML Library.

In order for the XML Library to be able to work with XML documents, it needs to walk through the data in an XML document and parse it out into smaller components, called *nodes*. Nodes generally correspond to a specific opening tag in an XML document (like `<product>`) or to the content that lies between opening and closing tags (like `The old Geegaw. Anyone want to buy one?` from the sample XML above).

Node Types

Each node has a *node type* which is represented by a type code. The types supported by the XML Library are:

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 NEW 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/>

NodeIDs

When the XML Library parses the nodes in an XML document, each node is validated and given a *nodeID* number (in the form *documentNumber.nodeNumber*), storing all of this node information in a global array called **gSTSXML_Data**.

The good thing is that the **gSTSXML_Data** array is "behind the scenes" — you generally don't access it or manipulate it directly, but instead you use pre-defined *methods* in the XML Library that in turn access the data in this array. (If you happen to be curious about the **gSTSXML_Data** array and how it stores data in its node attributes, you can review Appendix I: The gSTSXML_Data Array at the end of this document.)

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>
```

As you can see above, *nodeIDs* can be used to point to any kind of node type, and is not restricted in any way.

Node Paths

In addition to *nodeIDs*, nodes can also be referenced by using *node paths*, which is a /-delimited representation of where nodes exist in the tree. Node paths can only point to element (ELEM) nodes, and are normally represented in this format:

```
/rootNode[/parentNode1[/ ... [/parentNodeN[/childNode]]]]
```

When there is more than one child node that is named the same, a square open bracket followed by an index number and a square closing bracket (like "[1]") is used to differentiate one node from another. Listed below are all of the node paths from the XML Sample above:

```
/products
/products/widgets
/products/widgets/widget[1]
/products/widgets/widget[2]
```

Working with XML "Documents"

Since the XML Library allows you to have multiple XML documents in memory at once and manipulate them, an extension to node paths has been added to allow you to do this. You can precede your node path with the document number; if you don't provide a document number, it is assumed that you are referring to document #1. So the actual format used by the XML Library is this:

```
[documentNumber]/rootNode[/parentNode1[/ ... [/parentNodeN[/childNode]]]]
```

So supposed you had loaded the XML Sample into memory twice, so that the first instance was document #1 and the second instance was document #2. And you wanted to change the color of the first widget in both documents to "brown". Here's how you would do it:

```
get stsXML_SetAttribute("1/products/widgets/widget[1]","color","brown")
get stsXML_SetAttribute("2/products/widgets/widget[1]","color","brown")
```

If you generally only work with one XML document, you should call stsXML_reset before you load and parse the XML data with stsXML_LoadData(). Doing this will ensure that that the XML data is always parsed into document #1, and will allow you to use node paths without having to specify the preceding `[documentNumber]` in the path.

Note, however, that there are *some* methods that require you to pass a node number (such as stsXML_prettyPrint), which to access the whole document you can either pass the document node number (`1.0`, `2.0`, etc.) or you can pass the document number (`1`, `2`, etc.) instead (it will add the ".0" to turn it into a document node number).

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. When data is normalized, the following transformations are performed on the data:

- 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 nodes (node type `CDAT`).

Putting It All Together

You can see how nodes are identified and stored by using the stsXML_dump utility method (see below, 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 node 1.5, 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 **gSTSXML_Data** array (see Appendix I: The gSTSXML_Data Array at the end of this document.).

Trapping For Errors

Whenever you execute one of the methods in the XML Library, there is always a possibility that you might have typed something wrong, or that the node you want doesn't exist, etc. So it is important to know about errors and how to trap for them.

In the XML Library, if an error occurs, the method you called will return the string `error` (either as the returned value of the function you called, or in `the result` if it's a command that is called), and the global **gSTSXML_Error** is filled with the error string. If no error occurred, **gSTSXML_Error** will be empty and the method will return what it was supposed to return.

For example:

```
stsXML_reset
put stsXML_GetRoot(1) into tDoc2RootNode
```

This will error out because the call to stsXML_reset clears out any and all parsed documents from memory, and then the call to stsXML_GetRoot(1) is asking to get the node that is the root node of the first document (which doesn't exist). When you do this, you will get the error "Error: Document node 1 is not a valid document node." can check the result and process the error, like this:

```
stsXML_reset
put stsXML_GetRoot(1) into tDoc2RootNode
if tDoc2RootNode = "error" then
  answer gSTSXML_Error
  exit to top
end if
```

Of course, you don't need to answer the error and exit to top – you could call your own handler to handle the error if you like. The XML Library has that ability built in via the stsXML_setCallback method, which will automatically call a handler of your own in the object of your choice. This allows you to have a centralized method of handling errors, like this:

```
stsXML_reset
stsXML_setCallback (the long id of this stack),"DoXMLError"
put stsXML_GetRoot(1) into tDoc2RootNode
-- continue as normal, calling on the various XML Library methods you need to run
```

The script of the stack could include this handler:

```
on DoXMLError
  put gSTSXML_Error into tError
  answer "An error has occurred:" & CR & tError with "Continue" or "Abort"
  if it is "Abort" then exit to top
end DoXMLError
```

Complete Examples

Now that you have seen in the previous sections how nodes are ordered, numbered and referenced, and how to trap for errors when methods are called, the following examples will show you how to read and write XML data. (Note that writing XML requires using the Standard Edition of the XML Library since it creates documents, nodes and attributes.)

Using the Workshop

The easiest way to work with the capabilities of the XML Library is through the use of the Workshop, which is part of the XML Library stack, and can be accessed by opening the XML Library in Revolution, and then clicking the Workshop >> button at the bottom of the window.

Once in the Workshop, you can test out XML code by following these steps:

1. ***Add/Import XML Data:*** You can either type XML code from scratch into the Sample Text field, or you can click on the Open XML File... button, select an XML file on disk to open, and click "OK", which will import the XML data into the Sample Text field.
2. ***Load and Parse the XML Data*** into the **gSTSXML_Data** array by clicking on the stsXML_LoadData button. When it is done, it will display a status in the "Parse Time" area, showing how many nodes were parsed and how long it took. The XML Document number will be displayed in the "Results" field (so the first time you do this, it will show "1").

   If your XML data that you typed/imported is very long (say 1000 nodes or more), it may take a long time to parse. If you want to be shown the progress of the parsing, you can turn on the "Show Progress Bar (slower)" checkbox before you click the "stsXML_LoadData" button. If, on the other hand, you want to see how long it takes the XML Library to parse the data going as fast as it can, you would need to turn off the "Validate (slower") checkbox and the "Show Progress Bar (slower") checkbox.
3. ***Test Individual Methods:*** Once the data's been parsed, you can execute the XML Library methods on the data by clicking on any of the "method" buttons on the right side of the window.

   If a method requires parameters, you will be asked to provide them through one or more dialog boxes. If you need to provide a node descriptor, you can provide use a node ID or node path. Ultimately, the result (assuming there was no error) will be placed in the "Results" field.

   All of the methods described in the List of Methods section should be available to test as a button in the Workshop, with the exception of stsXML_setCallback and stsXML_clearCallback, which can't be actively tested in the Workshop. It is strongly recommended that you look at the definition of the method in the List of Methods section while you operate the Workshop, so that you will know what to provide, and what to expect as a result.

The XML Sample

The XML Sample (see XML Sample, above) will be used in Example #2 for reading XML, and it will also be used as a template for Example #3 where you learn how to create a new XML document.

The XML Sample is already precreated for you in the XML file called "Sample.xml" which is in the "Sample XML Files" folder that was included in the XML Library download, so you don't need to retype it.

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.

Similarly, as you read through each example below, you may want to switch back and forth between this and the descriptions of each method implemented in the next section.

Example 1: Getting a list of products

Suppose you wanted to get a list of products from an inventory file (`inventory.xml`) that looked like this:

```
<?xml version="1.0"?>
<products category="Snacks">
  <product>Pretzels</product>
  <product>Chips</product>
  <product>Peanuts</product>
  <product>Carrot Sticks</product>
</products>
```

Since most of the XML doc is ELEM nodes, you have your choice as to whether you want to use *node IDs* or *node paths* to retrieve the list of products. Shown below are *both* methods:

```
global gSTSXML_Error

on mouseUp
  -- The following line assumes the file 'inventory.xml' is in the same folder
  -- that 'the defaultFolder' points to
  put url ("file:inventory.xml") into tXMLData
  put GetProductList(tXMLData) into field 1
end mouseUp

on xmlError
  -- This is called when an error occurs due to stsXML_setCallback
  -- Simply display an answer dialog for now and stop processing
  answer "An error has occurred:" & cr & gSTSXML_Error
  exit to top
end xmlError

-- NODE ID Version
function GetProductList pXMLData
  -- reset everything (note: this clears *all* XML documents from memory)
  stsXML_reset
  -- set a callback to me if there are any problems
  stsXML_setCallback (long id of me),"xmlError"

  -- Since we're always going to work with just one document, we don't need
  -- to hold onto the document ID number that stsXML_LoadData generates
  -- which is why we're using 'get'. If you need to hold onto the document
  -- ID number, do 'put stsXML_LoadData(pXMLData) into tDocID' instead
  get stsXML_LoadData(pXMLData)

  -- Since you know the <product> tags are children of the root node,
  -- you need to get the node ID for the root node that you can then
  -- pass to stsXML_GetNamedChildren.
  put stsXML_GetRoot(1) into tProductsNode
  put stsXML_GetNamedChildren(tProductsNode,"product") into tChildList

  -- tChildList will contain a multiline variable that holds the node IDs
  -- for each child node of the root node whose tag is "product". You
  -- can now iterate through the children to get the name of each product
  put "" into tResult
  repeat for each line tKid in tChildList
    put stsXML_GetNodeData(tKid) & cr after tResult
  end repeat
  delete last char of tResult

  stsXML_clearCallback
  answer "The result is:" & cr & tResult
end GetProductList
```

```
-- NODE PATH Version
function GetProductList pXMLData
  stsXML_reset
  stsXML_setCallback (long id of me),"xmlError"
  get stsXML_LoadData(pXMLData)

  -- Since you're using node paths, and you know the root node is <products>,
  -- you don't need to use stsXML_GetRoot() to identify the root node. You could
  -- use stsXML_GetNamedChildren() as in the first example, but here's an
  -- alternate method that uses stsXML_CountNamedChildren:
  repeat with x = 1 to stsXML_CountNamedChildren("/products","product")
    put stsXML_GetNodeData("/products/product[" & x & "]") into line x of tResult
  end repeat

  stsXML_clearCallback
  answer "The result is:" & cr & tResult
end GetProductList
```

As you can see, in this instance, using node paths is easier and more straightforward than using node IDs. Keep in mind that node IDs must be used (for example when referring to non-ELEM nodes).

Since the rest of the examples all use ELEM nodes, the "node path" method will be used.

Example 2: Reading an Attribute

This example uses the XML Sample to demonstrate locating a node based on an attribute, and reading the value of an attribute from a node. It uses the same "callback" approach as Example #1, and assumes that you have the XML data in the variable (pXMLData) which is passed to the handler below.

In this handler, we're retrieving the value of the "color" attribute from the <widget> node whose "name" attribute is "Geegaw":

```
on GetGeegawColor pXMLData
  stsXML_reset
  stsXML_setCallback (long id of me),"xmlError"
  get stsXML_LoadData(pXMLData)

  -- Locate the node whose "name" attribute matches "Geegaw"
  -- Start looking in the <widgets> node and look through all children and grandchildren
  -- Note that this will find ANY node that has 'name="Geegaw"'; it doesn't have to be
  -- a <widget> node (although that's the only node in the XML Sample that matches)
  put stsXML_FindNodesByAttribute("name","Geegaw","/products/widgets","match") into tNode

  -- If you made a typo, or for some reason what you're looking for doesn't exist,
  -- you should check your result

  if tNode is not empty then
    -- Now that you have the node, get the value of the "color" attribute
    put stsXML_GetAttribute(tNode,"color") into tColor
    answer "The color is" && tColor
  else
    answer "The 'Geegaw' node could not be located based on your search criteria."
  end if
end GetGeegawColor
```

Example 3: Creating the XML Sample

The following code will recreate the XML Sample using the same "callback" method in Example #1, and shows you how to create an XML document from scratch. Note that to implement this example you will need the Standard Edition of the XML Library.

```
on mouseUp
  stsXML_reset
  stsXML_setCallback (long id of me),"xmlError"

  -- create the document, and its main nodes
  put stsXML_CreateDocument("products") into tDoc

  -- Since the CMNT and PROC nodes can't have children, there is no reason to store the
  -- node IDs created which is why "get" is used instead of "put".
  get stsXML_AppendChild("1.2","CMNT","","The widgets below are all from Australia")
  get stsXML_AppendChild("1.2","PROC","runAction-checkStock")
  put stsXML_AppendChild("1.2","ELEM","widgets") into tWidgets

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

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

  -- Finish up by removing the error callback
  stsXML_clearCallback
end mouseUp

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

While you are working with this Example, it is very beneficial to see what your XML document currently looks like. You can use the methods stsXML_dump (which will provide a "node dump" of the document), stsXML_expand (which will give you the raw XML), or stsXML_prettyPrint (which will give you formatted XML with node identifiers). To give you an idea of which one to choose, here's an example of the first part of the output from these three methods (cropped to the column boundary for clarity so data wouldn't wrap):

stsXML_dump(1,"true")	stsXML_expand(1.0,"true")	stsXML_prettyPrint(1.0)
[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· [1.4:PROC]<?runAction:·checkStock?> [1.5:ELEM]<widgets>¬†	<?xml version="1.0"?> <products> <!-- The widgets below are all from <?runAction: checkStock?> <widgets> <widget id="1" name="Thingimajig" <![CDATA[The NEW Thingim </widget> <widget id="3" name="Geegaw" colo </widgets> </products>	1.0 <DOC> 1.1 <?xml version="1.0"?> 1.2 <products> 1.3 <!-- The widgets below are a 1.4 <?runAction: checkStock?> 1.5 <widgets> 1.6 <widget id="1" name="Thin 1.7 <![CDATA[The NEW </widget> 1.8 <widget id="3" name="Gee 1.9 The old Geegaw. Anyone </widget>

List of Methods

Note: The support for the Standard edition of the XML Library is indicated with the following graphic on the appropriate method page: Std

For each method if a parameter is marked *nodeDesc* (short for "node descriptor") it means that you can use either a node ID or a node path. If a specific form is required, it will be listed as either *nodeID* or *nodePath*. Remember that a node path can only point to ELEM nodes, whereas a node ID can point to *any* node.

Loading Methods *13*
stsXML_LoadData 13

Document Management Methods *15*
stsXML_CreateDocument 15
stsXML_DeleteDocument 16
stsXML_GetDocuments 17

Data/Text Methods *18*
stsXML_ChildrenToArray 18
stsXML_GetCDATA 20
stsXML_GetNodeData 21
stsXML_GetText 23
stsXML_SetNodeData 24

Attribute Methods *25*
stsXML_AttributesToArray 25
stsXML_CountAttributes 27
stsXML_CreateAttribute 28
stsXML_DeleteAttribute 29
stsXML_GetAttribute 30
stsXML_GetAttributes 31
stsXML_HasAttribute 33
stsXML_SetAttribute 34

Search Methods *35*
stsXML_FindNodesByAttribute 35
stsXML_FindNodesByData 37

Node Methods *39*
stsXML_CreateNode 39
stsXML_DeleteNode 43
stsXML_GetNodeXML 44
stsXML_GetNodeName 46
stsXML_GetNodeType 47
stsXML_GetRoot 48
stsXML_IsEmptyElement 49

Parent/Child Methods *50*

stsXML_AppendChild 50
stsXML_CountChildren 53
stsXML_CountNamedChildren 54
stsXML_GetChildren 55
stsXML_GetFirstChild 57
stsXML_GetNamedChildren 58
stsXML_GetLastChild 60
stsXML_GetNextSibling 61
stsXML_GetParent 62
stsXML_GetPreviousSibling 63
stsXML_HasChildren 64

XML Utility Methods 65
stsXML_clearCallback 65
stsXML_dump 66
stsXML_expand 68
stsXML_expandPaths 70
stsXML_getNodeID(s) 71
stsXML_getNodePath(s) 72
stsXML_isDocument 74
stsXML_isNode 75
stsXML_isType 76
stsXML_libVersion 77
stsXML_normalize 78
stsXML_prettyPrint 79
stsXML_replaceEntities 81
stsXML_reset 82
stsXML_setCallback 83
stsXML_validateNode 84

*Appendix I: The **gSTSXML_Data** Array 85*

Appendix II: The W3C Specification for XML Data 87

Appendix III: Comparing the XML Library With Revolution's revXML.DLL 90

stsXML_LoadData

Summary

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

Syntax

`stsXML_LoadData(xmlData[, silentErrors][, validateXML])` → *docNumber*

Arguments

xmlData	The XML data that comes from any container (although usually it is 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.
validateXML	Optional. Can be either `true`, `false` or left blank. If `true` or left blank, the XML Library will run all XML nodes through validation routines and return errors if the XML document is not properly formed. If `false`, the XML Library will not validate any of the XML it is parsing. This will result in much faster loading times, but at the sake of validating the XML.

Possible Returned Values

docNumber	The document number for the parsed XML document.
`error`	An error was found during parsing. Check **gSTSXML_Error** 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 **gSTSXML_Data**. 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 stsXML_LoadData 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. Additionally, each element (ELEM) node is assigned a *node path* so that either node IDs or node paths can be used to identify a specific XML node. If *validateXML* is `true` or blank, these nodes are validated against the W3C XML specification.

Example 1:

The following example loads XML from a user-selected file on disk, with errors being reported normally and validation turned on:

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

```
end mouseUp
```

Example 2:

The following example loads XML from a field on the current card, with errors being reported silently and validation turned off:

```
on mouseUp
  local tDocNum
  answer file "Pick an XML file:" with filter "XML Files (*.xml)"&cr&"*.xml"
  if it <> "" then
    put stsXML_LoadData(url ("file:" & it),"true","false") into tDocNum
  end if
end mouseUp
```

See also

stsXML_dump, stsXML_expand, stsXML_prettyPrint

Version History

1.0 — Initial appearance as XMLLoadData.

2.0 — Added the ability to optionally validate the XML data being parsed.

stsXML_CreateDocument

Std

Summary

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

Syntax

`stsXML_CreateDocument(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 **gSTSXML_Error** 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 **gSTSXML_Data**. 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 stsXML_CreateNode, stsXML_AppendChild and stsXML_CreateAttribute to fill out the new document.

Example

The following example:

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

… creates the following XML document:

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

See also

stsXML_CreateNode, stsXML_AppendChild, stsXML_CreateAttribute

Version History

1.0 — Initial appearance as XMLCreateDocument.

stsXML_DeleteDocument

Summary

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

Syntax

`stsXML_DeleteDocument(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 **gSTSXML_Error** 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 **gSTSXML_Data** array.

Example

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

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

… would return:

```
1
3
```

See also

stsXML_LoadData, stsXML_CreateDocument, stsXML_GetDocuments

Version History

1.0 — Initial appearance as XMLDeleteDocument.

stsXML_GetDocuments

Summary

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

Syntax

`stsXML_GetDocuments()→` *`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 stsXML_DeleteDocument.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

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

Example

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

```
on mouseUp
  get stsXML_GetDocuments()
end mouseUp
```

… would return:

```
1
2
3
```

See also

stsXML_LoadData, stsXML_CreateDocument, stsXML_DeleteDocument

Version History

1.0 — Initial appearance as XMLGetDocuments.

stsXML_ChildrenToArray

Summary

This method retrieves the TEXT or CDAT child node contents from all of the ELEM children of an ELEM parent node and copies it into an array.

Syntax

`stsXML_ChildrenToArray(`*`nodeDesc`*`)` → *`arrayData`*

Arguments

`nodeDesc`	The ELEM parent node being referenced.

Possible Returned Values

`arrayData`	The array that contains the children and their contents.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method retrieves the contents of the TEXT or CDAT node from all children of the node provided in *parentNodeDesc* and puts the data into an array, where the key (array element name) is the name of the child node, and the value of the key is the retrieved TEXT/CDAT contents. The contents are returned normalized for TEXT nodes and of course not normalized for CDAT nodes.

Look at the examples for a better understanding of how this works; the all use the following XML that's been loaded as document #1:

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

Notice that this is a variation of the XML Sample, where every thing that was an attribute in the Sample is now a separate ELEM node. This is a common format to use, especially when dealing with data that comes from databases.

Example 1:

The following example:

```
on mouseUp
  put stsXML_ChildrenToArray("/products/widgets/widget[1]") into tWidgetA
  put the keys of tWidgetA
end mouseUp
```

… returns:

```
name
description
id
color
```

… so this:

```
on mouseUp
  put stsXML_ChildrenToArray("/products/widgets/widget[1]") into tWidgetA
  put tWidgetA["name"]
end mouseUp
```

… returns:

```
Thingimajig
```

… and this:

```
on mouseUp
  put stsXML_ChildrenToArray("/products/widgets/widget[1]") into tWidgetA
  put tWidgetA["description"]
end mouseUp
```

… returns:

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

Note that the CMNT node, although it is a child of `/products/widgets/widget[1]` is not an ELEM node, and so is skipped. Also note that Revolution does not return the keys in the order they are encountered in the XML, so although the order of the keys in the XML above is “id”, “name”, “color”, “description”, Revolution returned `the keys of tWidgetA` as “name”, “description”, “id”, “color”. Normally this shouldn’t make a difference as you’ll be retrieving the data from the array element.

See also

stsXML_GetCDATA, stsXML_GetNodeData, stsXML_GetText, stsXML_SetNodeData, stsXML_AttributesToArray, stsXML_normalize

Version History

2.0 — Initial appearance.

stsXML_GetCDATA

Summary

This method retrieves the CDATA contents of a CDAT node.

Syntax

`stsXML_GetCDATA(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 **gSTSXML_Error** 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 stsXML_normalize utility method.

NOTE: This method has been deprecated in deference to stsXML_GetNodeData, which does not require that you know the node type before retrieving the contents.

Example

The following example:

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

… returns:

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

See also

stsXML_ChildrenToArray, stsXML_GetText, stsXML_GetNodeData, stsXML_SetNodeData, stsXML_normalize

Version History

1.0 — Initial appearance as XMLGetCData.

2.0 — Deprecated in deference to stsXML_GetNodeData.

stsXML_GetNodeData

Summary

This method retrieves the TEXT or CDAT child node content from an ELEM parent node.

Syntax

`stsXML_GetNodeData(`*`parentNodeDesc[,normalize]`*`)` → *`nodeContents`*

Arguments

`parentNodeDesc`	A reference to the ELEM parent node of the TEXT or CDAT child node that is to be retrieved.
`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 specified node.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method retrieves the contents of a TEXT or CDAT node that is the child of the parent node provided in *parentNodeDesc*. The contents are returned normalized, unless you pass `false` for *normalize*. If *parentNodeDesc* is not an ELEM node, or does not have any TEXT or CDAT children, this returns empty ("").

Note that although this supercedes stsXML_GetText and stsXML_GetCDATA, it is different in that this points to the *parent* of the TEXT or CDATA node, whereas stsXML_GetText and stsXML_GetCDATA point to the TEXT/CDATA node itself.

Example 1:

The following example:

```
on mouseUp
  get stsXML_GetNodeData("/products/widgets/widget[2]")
end mouseUp
```

… returns:

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

Example 2:

The following example:

```
on mouseUp
  get stsXML_GetNodeData("/products/widgets/widget[1]")
end mouseUp
```

… returns:

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

See also

stsXML_ChildrenToArray, stsXML_SetNodeData, stsXML_GetText, stsXML_GetCDATA, stsXML_normalize

Version History

2.0 — Initial appearance.

stsXML_GetText

Summary

This method retrieves the text content of a TEXT node.

Syntax

`stsXML_GetText(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 **gSTSXML_Error** 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*.

NOTE: This method has been deprecated in deference to stsXML_GetNodeData, which does not require that you know the node type before retrieving the contents.

Example

The following example:

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

… returns:

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

See also

stsXML_ChildrenToArray, stsXML_GetCDATA, stsXML_GetNodeData, stsXML_SetNodeData, stsXML_normalize

Version History

1.0 — Initial appearance as XMLGetText.

2.0 — Deprecated in deference to stsXML_GetNodeData.

stsXML_SetNodeData

Summary

This method either changes the existing content of the TEXT or CDAT child node of a specified node, or creates one if it does not already exist.

Syntax

`stsXML_SetNodeData(`*`parentNodeDesc, childNodeContents[, isCDATA]`*`)` → *`newChildNodeID`*

Arguments

`parentNodeDesc`	The ELEM node that either contains or is to contain the TEXT or CDAT node contents provided in *childNodeContents*.
`childNodeContents`	The contents to be inserted or added. If it is prefaced with `<![CDATA[`, the contents are inserted or added as a CDATA child node to *parentNodeDesc*, and any value in *isCDATA* is ignored.
`isCDATA`	Optional. Can be either `true`, `false` or left blank. If `true`, the data in *childNodeContents* will assume to be implemented as CDATA. If `false` or left blank, the contents will assume to be implemented as TEXT (unless *childNodeContents* is prefaced by `<![CDATA[` (see above), in which case it is treated as CDATA regardless of the setting of *isCDATA*.

Possible Returned Values

`childNodeID`	The node ID of the child of *parentNodeDesc* that was created as a result of this method.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method is used to set or create the contents of an ELEM node. It first determines if there is an existing TEXT or CDAT node that is a child of the node referenced as *parentNodeDesc*. If there is, it deletes it and then adds a new TEXT or CDAT child to *parentNodeDesc.* Note that because of this, the node ID number for the replacement TEXT or CDAT will be different than the original node ID number before this method was executed.

Example

The following example:

```
on mouseUp
  get stsXML_SetNodeData("/products/widgets/widget[1]","This is a test")
end mouseUp
```

… returns:

```
1.10
```

which is the new ID number of the replaced node; the old CDATA node at 1.7 was deleted, and a new TEXT node (1.10) was inserted.

See also

stsXML_ChildrenToArray, stsXML_GetNodeData, stsXML_GetText, stsXML_GetCData, stsXML_normalize

Version History

2.0 — Initial appearance.

stsXML_AttributesToArray

Summary

This method takes the attributes owned by a node, along with their values, and returns them as an array.

Syntax

`stsXML_AttributesToArray(`*`nodeDesc`*`)` → *`arrayData`*

Arguments

`nodeDesc`	The node being examined.

Possible Returned Values

`arrayData`	An array of data where each attribute of *nodeDesc* is a key in the array, and the value of each attribute is the value of the array element.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method returns the attributes owned by *nodeDesc*, along with their values, as a Revolution array variable. Only ELEM nodes can use this function, although you can pass any node type to this method (it will return empty in this case).

Example

The following example (based on the XML Sample):

```
on mouseUp
  put stsXML_AttributesToArray("/products/widgets/widgets[1]") into tAttribA
  put the keys of tAttribA
end mouseUp
```

… returns:

```
name
id
color
```

… so this:

```
on mouseUp
  put stsXML_AttributesToArray("/products/widgets/widgets[1]") into tAttribA
  put tAttribA["name"]
end mouseUp
```

… returns:

```
Thingimajig
```

Note that Revolution does not return the keys in the order they are encountered in the XML, so although the order of the keys in the XML above is “id”, “name”, “color”, Revolution returned `the keys of tAttribA` as “name”, “id”, “color”. Normally this shouldn’t make a difference as you’ll be retrieving the data from the array element.

See also

stsXML_GetAttributes, stsXML_ChildrenToArray

Version History

2.0 — Initial appearance.

stsXML_CountAttributes

Summary

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

Syntax

`stsXML_CountAttributes(nodeDesc)` → *`numberOfAttribs`*

Arguments

`nodeDesc`	The node being examined.

Possible Returned Values

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

Description

This method returns the number of attributes owned by *nodeDesc*. 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 stsXML_CountAttributes("1.6")
end mouseUp
```

… returns:

```
3
```

See also

stsXML_GetAttributes

Version History

1.0 — Initial appearance as XMLCountAttributes.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_CreateAttribute

Std

Summary

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

Syntax

`stsXML_CreateAttribute(nodeDesc,attribName,attribValue)` → `""`

Arguments

nodeDesc	The node descriptor 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 **gSTSXML_Error** for the error found.

Description

This method creates a new attribute named *attribName* with the value *attribValue* for the node supplied in *nodeDesc*. 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 Appendix II, at the end of this document).

Example

The following example:

```
on mouseUp
  get stsXML_CreateAttribute("1.2","season","spring")
  get stsXML_GetNodeXML("/products")  -- or XMLGetNodeXML("1.2")
end mouseUp
```

… returns:

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

See also

stsXML_DeleteAttribute, stsXML_GetAttribute, stsXML_HasAttribute, stsXML_SetAttribute

Version History

1.0 — Initial appearance as XMLCreateAttribute.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_DeleteAttribute

Std

Summary

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

Syntax

`stsXML_DeleteAttribute(nodeDesc,attribName)` → "″

Arguments

nodeDesc	The node descriptor 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 **gSTSXML_Error** for the error found.

Description

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

Example

The following example:

```
on mouseUp
  get stsXML_DeleteAttribute("1.6","name")
  get stsXML_GetNodeXML("1.6")
end mouseUp
```

… returns:

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

See also

stsXML_CreateAttribute, stsXML_GetAttribute, stsXML_HasAttribute, stsXML_SetAttribute

Version History

1.0 — Initial appearance as XMLDeleteAttribute.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetAttribute

Summary

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

Syntax

`stsXML_GetAttribute(nodeDesc,attribName)` → `attribValue`

Arguments

nodeDesc	The node descriptor 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 **gSTSXML_Error** for the error found.

Description

This method retrieves the value of the attribute *attribName* from the node provided in *nodeDesc*. *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 stsXML_GetAttribute("1.6","name")
end mouseUp
```

… returns:

```
Thingimajig
```

See also

stsXML_CreateAttribute, stsXML_DeleteAttribute, stsXML_GetAttributes, stsXML_HasAttribute, stsXML_SetAttribute

Version History

1.0 — Initial appearance as XMLGetAttribute.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetAttributes

Summary

This method retrieves the list of attributes possessed by the node supplied in *nodeDesc*, as well as optionally retrieving the values for those attributes.

Syntax

`stsXML_GetAttributes(`*`nodeDesc[,getValues[,valueDelimiter]]`*`)` → *`attributeList`*

Arguments

`nodeDesc`	The node descriptor of the reference node.
`getValues`	Optional. Can be either `true`, `false` or left blank. If `true`, the attributes will be returned with their values, separated by the delimiter specified in *valueDelimiter*. If `false` or left blank, only the attributes themselves will be returned.
`valueDelimiter`	Optional. Can be a single character (or a constant that resolves to a single character) or left blank. If specified, the character provided in *valueDelimiter* will be used to separate the attribute from its value; if not specified, a tab character will be used.

Possible Returned Values

`attributeList`	A return-delimited list of attribute names with or without values.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

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

Example

The following example:

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

… returns:

```
id
name
color
```

The following example:

```
on mouseUp
  get stsXML_GetAttributes("/products/widgets/widget[1]","true","=")
end mouseUp
```

… returns:

```
id=1
name=Thinkgimajig
color=navy
```

See also

stsXML_CountAttributes, stsXML_GetAttribute, stsXML_AttributesToArray

Version History

1.0 — Initial appearance as XMLGetAttributes.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*; added ability to optionally retrieve the values of those attributes at the same time, and to provide what character to use as the delimiter between attribute name and attribute value.

stsXML_HasAttribute

Summary

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

Syntax

`stsXML_HasAttribute(nodeDesc,attribName)` → `{true|false}`

Arguments

nodeDesc	The node descriptor of the reference node.
attribName	The name of the attribute to check for.

Possible Returned Values

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

Description

This method determines whether the node supplied in *nodeDesc* 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 stsXML_GetNodeType before executing this method.

Example

The following example:

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

… returns:

```
true
```

See also

stsXML_CreateAttribute, stsXML_DeleteAttribute, stsXML_GetAttribute, stsXML_GetAttributes, stsXML_SetAttribute

Version History

1.0 — Initial appearance as XMLHasAttribute.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_SetAttribute

Summary

This method sets the value of an existing attribute of *nodeDesc* to a new value, optionally creating it if it doesn't exist.

Syntax

`stsXML_SetAttribute(`*`nodeDesc,attribName,attribValue[,createIfNotExist]`*`)` → `"″`

Arguments

nodeDesc	The node descriptor of the reference node.
attribName	The name of the attribute whose value is to be changed.
attribValue	The new value for *attribName*.
createIfNotExist	Optional. Can be either `true`, `false` or left blank. If `true`, the attribute specified in *attribName* will be created with the value specified in *attribValue* if it doesn't already exist. If `false` or left blank, an error will result if you try to set an attribute that doesn't already exist (this is for backwards compatibility).

Possible Returned Values

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

Description

This method changes the value of the attribute supplied in *attribName* of the node supplied in *nodeDesc* to the new value supplied in *attribValue*. If *attribName* doesn't already exist, an error will result unless *createIfNotExist* is `true`.

Note that *attribValue* needs to follow the restrictions of the node format `AttValue` (see Appendix II, at the end of this document).

Example

The following example:

```
on mouseUp
  get stsXML_SetAttribute("1.6","name","Doohickey")
  get stsXML_GetNodeXML("1.6")
end mouseUp
```

… returns:

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

See also

stsXML_CreateAttribute, stsXML_DeleteAttribute, stsXML_GetAttribute, stsXML_GetAttributes, stsXML_HasAttribute

Version History

1.0 — Initial appearance as XMLSetAttribute.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*; added ability to optionally create the attribute if it did not already exist.

stsXML_FindNodesByAttribute

Summary

This method locates all of the ELEM nodes in a section of a document which has a specific attribute that matches a specific search string.

Syntax

`stsXML_FindNodesByAttribute(`*`attribName,matchString[,nodeDesc][,searchType]`*`)` → *`nodeList`*

Arguments

`attribName`	The name of the attribute in which to search.
`matchString`	The string that you wish to find in the value of the attribute provided in *attribName*.
`nodeDesc`	Optional. Identifies the node in the document from which searching will begin; the nodes that are searched are *nodeDesc* and its descendants; if you want to search an entire document, provide the document node number or path. If it is left blank, it is assumed that you wish to search document #1 in its entirety (that is, it uses "1.0" as *nodeDesc*).
`searchType`	Optional. Determines how to locate the nodes. Can be either `starts [with]`, `ends [with]`, `contains`, `match[es]` or left blank. If it is left blank, `match` is used (which means that the attribute must be exactly the same as *matchString*).

Possible Returned Values

`nodeList`	A multi-line list of node IDs that matches the search.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method searches through all of the ELEM nodes in the document starting at *nodeDesc* (or the top of the first document if *nodeDesc* is empty) and goes through its descendants, looking to see if the value of the attribute specified in *attribName* is the same as, starts with, ends with, or contains *matchString*. If so, it puts the node ID of the matching node into a list and then returns the list when it is done. If there are no nodes that match the request, this method will return empty ("").

Note that you can use the utility method stsXML_getNodePath (or its plural alternative stsXML_getNodePaths) on the returned list if you want to convert your node IDs into node paths (see the examples below).

Example 1:

The following example:

```
on mouseUp
  put stsXML_FindNodesByAttribute("name","g","","contains") into tMatchingNodes
  -- This returned node 1.6, which matches "g" in "Thingimajig"
  -- and 1.8, which matches "g" in "Geegaw"
  put stsXML_getNodePaths(tMatchingNodes)
end mouseUp
```

… returns:

```
1/products/widgets/widget[1]
1/products/widgets/widget[2]
```

Example 2:

The following example:

```
on mouseUp
  put stsXML_FindNodesByAttribute("name","Geegaw") into tMatchingNodes
  -- This returned node 1.8 only
  put stsXML_getNodePaths(tMatchingNodes)
end mouseUp
```

… returns:

```
1/products/widgets/widget[2]
```

See also

stsXML_FindNodesByData, stsXML_getNodePath

Version History

2.0 — Initial appearance.

stsXML_FindNodesByData

Summary

This method locates all of the ELEM nodes in a section of a document which have TEXT or CDAT contents that match a specific search string.

Syntax

`stsXML_FindNodesByData(matchString[,nodeDesc][,searchType])` → `nodeList`

Arguments

matchString	The string that you wish to find in the TEXT or CDAT contents of matching nodes.
nodeDesc	Optional. Identifies the node in the document from which searching will begin; the nodes that are searched are *nodeDesc* and its descendants; if you want to search an entire document, provide the document node number or path. If it is left blank, it is assumed that you wish to search document #1 in its entirety (that is, it uses "1.0" as *nodeDesc*).
searchType	Optional. Determines how to locate the nodes. Can be either `starts [with]`, `ends [with]`, `contains`, `match[es]` or left blank. If it is left blank, `match` is used (which means that the data must be exactly the same as *matchString*).

Possible Returned Values

nodeList	A multi-line list of node IDs that matches the search.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method searches through all of the ELEM nodes in the document starting at *nodeDesc* (or the top of the first document if *nodeDesc* is empty) and goes through its descendants, looking to see if the node has a TEXT or CDAT child; if so, it searches within that child node to see if it starts with, ends with, contains, or is an exact match to *matchString*. If so, it puts the node ID of the matching parent node into a list and then returns the list when it is done. If there are no nodes that match the request, this method will return empty ("").

Note that this returns the *parent* ELEM node ID, not the child TEXT or CDAT node ID. To retrieve the data using the parent node ID, use stsXML_GetNodeData.

Note that you can use the utility method stsXML_getNodePath (or its plural alternative stsXML_getNodePaths) on the returned list if you want to convert your node IDs into node paths (see the example below).

Example:

The following example:

```
on mouseUp
  put stsXML_FindNodesByData("the","","contains") into tMatchingNodes
  -- This returned node 1.6, which matches "the" in "The <b>NEW</b> ..."
  -- and 1.8, which matches "the" in "The old Geegaw."
  put stsXML_getNodePaths(tMatchingNodes)
end mouseUp
```

… returns:

```
1/products/widgets/widget[1]
1/products/widgets/widget[2]
```

See also

stsXML_FindNodesByAttribute, stsXML_GetNodeData, stsXML_getNodePath

Version History

2.0 — Initial appearance.

stsXML_CreateNode

Std

Summary

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

Syntax

```
stsXML_CreateNode(where,nodeDesc,nodeType,nodeName[,nodeContents][,collapseNode])
→ 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 *nodeDesc*, but in a position prior to *nodeDesc*. That is, if *nodeDesc* is the second child of its parent, the new node will be created as the second child, and *nodeDesc* will move down to become the third child.
- `after` — Creates the new node as a sibling to *nodeDesc*, but in a position after *nodeDesc*. That is, if *nodeDesc* 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 *nodeDesc*. If *nodeDesc* has no children, the new node created will become the only child of *nodeDesc*. If *nodeDesc* has children, the new node created will become the last child of *nodeDesc* (i.e. if *nodeDesc* has three children, the new node created would be the fourth child).

nodeDesc — 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 Types for a list of appropriate type codes.) You may not create DOC nodes; to create documents, use stsXML_CreateDocument.

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.

collapseNode — Optional. This can be either true or false, and indicates whether to collapse the whitespace on the node created so instead of showing up like this:

```
<tag>
    data
</tag>
```

it shows up like this:

```
<tag>data</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 **gSTSXML_Data("1.0last")** was `20` before this method was executed, the new node ID would be `1.21`, and **gSTSXML_Data("1.0last")** would now be `21`). For more information on the `last` node property and how it relates to **gSTSXML_Data**, see Appendix I: The gSTSXML_Data Array.
`error`	An error was found during execution. Check **gSTSXML_Error** 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 node 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 Appendix II: The W3C Specification for XML Data, at the end of this document).

Example 1: "Before"

The following example:

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

… would create a new ELEM node and would make our XML Sample 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 stsXML_CreateNode("after","1.5","CDATA","Go","")
end mouseUp
```

… would create a new CDATA node and would make our XML Sample 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      <![CDATA[Go]]>
        </products>
```

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

Example 3: "Child"

The following example:

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

… would create a new Comment node and would make our XML Sample 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      <![CDATA[Go]]>
        </products>
```

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

See also

stsXML_AppendChild, stsXML_DeleteNode

Version History

1.0 — Initial appearance as XMLCreateNode.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

2.1 — Added *collapseNode* parameter.

stsXML_DeleteNode

Std

Summary

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

Syntax

`stsXML_DeleteNode(`*`nodeDesc`*`)` → `""`

Arguments

`nodeDesc`	The node to be deleted.

Possible Returned Values

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

Description

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

Example

The following example:

```
on mouseUp
  get stsXML_DeleteNode("/products/widgets")
end mouseUp
```

… will delete node 1.5 (the node number corresponding to `/products/widgets`). Note that nodes are <u>not</u> renumbered, and node numbers that are deleted are <u>never</u> reused.

See also

stsXML_CreateNode, stsXML_AppendChild

Version History

1.0 — Initial appearance as XMLDeleteNode.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetNodeXML

Summary

Originally called XMLGetNode, this method retrieves the raw XML for a specified node, and optionally its children. (See the Version History for the reason the name was changed.)

Syntax

`stsXML_GetNodeXML(`*`nodeDesc[,includeChildren[,normalize]]`*`)` → *`nodeContents`*

Arguments

`nodeDesc`	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 *nodeDesc* will be retrieved . If `false`, only the XML of *nodeDesc* 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 **gSTSXML_Error** for the error found.

Description

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

If *includeChildren* is false or not provided, what is retrieved is the start tag of the node supplied in *nodeDesc* (e.g. the `strt` node property – see Appendix I). If *includeChildren* is true, what is retrieved is the start and end tags of the node supplied in *nodeDesc*, 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 stsXML_normalize (i.e.even data contained in CDATA will be normalized).

Example 1: No Children

The following example:

```
on mouseUp
  get stsXML_GetNodeXML("1.5")
  -- or stsXML_GetNode("/products/widgets")
end mouseUp
```

… returns:

```
<widgets>
```

Example 2: With Children

The following example:

```
on mouseUp
  get stsXML_GetNodeXML("1.5","true")
  -- or stsXML_GetNode("/products/widgets","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 stsXML_GetNodeXML("1.7","false","false")
-- can’t use node paths; not an ELEM node
```

… returns:

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

whereas:

```
get stsXML_GetNodeXML("1.7","false","true")
-- can’t use node paths; not an ELEM node
```

… returns:

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

See also

stsXML_GetNodeName, stsXML_GetNodeType, stsXML_GetRoot

Version History

1.0 — Initial appearance as XMLGetNode.

2.0 — Renamed to stsXML_GetNodeXML to distinguish it from stsXML_GetNodeData.

stsXML_GetNodeName

Summary

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

Syntax

`stsXML_GetNodeName(`*`nodeDesc`*`)` → *`nodeName`*

Arguments

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

Possible Returned Values

`nodeName`	The name of the node supplied in *nodeDesc*.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method retrieves the `name` node property from the node supplied in *nodeDesc*. 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 (“”).

Note that although supplying a node path for *nodeDesc* would work, there is no reason to use node paths since to do that you’d already need to know the name of the element to which you are referring.

Example

The following example:

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

… returns:

```
widget
```

See also

stsXML_GetNodeXML, stsXML_GetNodeType, stsXML_GetRoot

Version History

1.0 — Initial appearance as XMLGetNodeName.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*, although as mentioned above, it is unlikely that you would use it.

stsXML_GetNodeType

Summary

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

Syntax

`stsXML_GetNodeType(`*`nodeDesc`*`)` → *`nodeType`*

Arguments

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

Possible Returned Values

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

Description

This method retrieves the `type` node property from the node supplied in *nodeDesc*. 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.

Note that although supplying a node path for *nodeDesc* would work, it would be unnecessary since all node paths can point to are ELEM nodes, and so you would already know that the result would be "ELEM".

Example

The following example:

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

… returns:

```
ELEM
```

See also

stsXML_GetNodeXML, stsXML_GetNodeName, stsXML_GetRoot

Version History

1.0 — Initial appearance as XMLGetNodeType.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*, although as mentioned above, it is unlikely that you would use it.

stsXML_GetRoot

Summary

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

Syntax

`stsXML_GetRoot(`*`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 **gSTSXML_Error** 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 stsXML_GetRoot("1")
end mouseUp
```

… returns:

```
1.2
```

See also

stsXML_GetNodeXML, stsXML_GetNodeName, stsXML_GetNodeType

Version History

1.0 — Initial appearance as XMLGetRoot.

stsXML_IsEmptyElement

Summary

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

Syntax

`stsXML_IsEmptyElement(nodeDesc)` → `{true|false}`

Arguments

nodeDesc	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 **gSTSXML_Error** 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 **gSTSXML_Error**.

Example

The following example:

```
on mouseUp
  get stsXML_IsEmptyElement("1.5")
  -- or stsXML_IsEmptyElement("/products/widgets")
end mouseUp
```

… returns:

```
false
```

See also

n/a

Version History

1.0 — Initial appearance as XMLIsEmptyElement.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_AppendChild

Std

Summary

This method adds a child node to an ELEM node.

Syntax

```
stsXML_AppendChild(nodeDesc,nodeType,nodeName[,nodeContents][,collapseNode]) ➪
newNodeID
```

Arguments

nodeDesc — 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 Types for a list of appropriate type codes.) You may not create DOC nodes; to create documents, use stsXML_CreateDocument.

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.

collapseNode — Optional. This can be either true or false, and indicates whether to collapse the whitespace on the node created so instead of showing up like this:

```
<tag>
    data
</tag>
```

it shows up like this:

```
<tag>data</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 **gSTSXML_Data("1.0last")** was `20` before this method was executed, the new node ID would be `1.21`, and **gSTSXML_Data("1.0last")** would now be `21`). For more information on the `last` node property and how it relates to **gSTSXML_Data**, see Appendix I: The gSTSXML_Data Array.

`error` — An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

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

Note that *nodeName* needs to follow the restrictions of the node format `Name` (see Appendix II: The W3C Specification for XML Data, at the end of this document).

Example

The following example:

```
on mouseUp
  answer stsXML_AppendChild("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 stsXML_CreateNode.

See also

stsXML_CreateNode, stsXML_DeleteNode

Version History

1.0 — Initial appearance as XMLAppendChild.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

2.1 — Added *collapseNode* parameter.

stsXML_CountChildren

Summary

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

Syntax

`stsXML_CountChildren(nodeDesc)` → *`numberOfChildren`*

Arguments

`nodeDesc`	The node to be examined.

Possible Returned Values

`numberOfChildren`	The number of children owned by *nodeDesc*.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method returns the number of child nodes owned by *nodeDesc*. 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 stsXML_CountChildren recursively for each child.

Example

The following example:

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

… returns:

```
2
```

See also

stsXML_CountNamedChildren, stsXML_HasChildren

Version History

1.0 — Initial appearance as XMLCountChildren.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_CountNamedChildren

Summary

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

Syntax

`stsXML_CountNamedChildren(`*`nodeDesc`*`,`*`childName`*`)` → *`numberOfChildren`*

Arguments

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

Possible Returned Values

`numberOfChildren`	The number of children of *nodeDesc* whose `name` node property (see Appendix I) matches *childName*.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method returns the number of child nodes owned by *nodeDesc* whose `name` node property (see Appendix I) 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 stsXML_CountChildren recursively for each child.

Example

The following example:

```
on mouseUp
  get stsXML_CountNamedChildren("/products/widgets","widget")
end mouseUp
```

… returns:

```
2
```

See also

stsXML_CountChildren, stsXML_HasChildren

Version History

1.0 — Initial appearance as XMLCountNamedChildren.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetChildren

Summary

This method retrieves a list of child nodes of a node, optionally of a specific node type, and optionally recursing through all descendants of the specified node.

Syntax

`stsXML_GetChildren(`*`nodeDesc[,childNodeType][,maximumLevel]`*`)` → *`childList`*

Arguments

`nodeDesc`	The node to be examined.
`childNodeType`	Optional. The node type of the child to retrieve. If left blank, or if the string `any` is used, it will retrieve all types of nodes.
`maximumLevel`	Optional. This is the number of levels of descendants below *nodeDesc* that should be returned by this method. If not supplied, one level of descendants (i.e. just the immediate children of *nodeDesc*) are returned. To return all descendants, pass `all` for *maximumLevel*, or a sufficiently large number (like 1000).

Possible Returned Values

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

Description

This method returns the return-delimited list of child nodes owned by *nodeDesc*. 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 method returns node IDs; if you want node paths, you can pass this through the stsXML_getNodePaths method, but be aware that this will only return node paths for ELEM nodes; any other node type cannot have paths, and will therefore not be in the resulting list.

Example 1:

The following example:

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

… returns:

```
1.3
1.4
1.5
```

Example 2:

The following example:

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

… returns:

```
1.3
```

Example 3:

The following example:

```
on mouseUp
  get stsXML_GetChildren("1.2","any","all")
end mouseUp
```

… returns:

```
1.3
1.4
1.5
1.6
1.7
1.8
1.9
```

See also

stsXML_GetFirstChild, stsXML_GetNamedChildren, stsXML_GetLastChild, stsXML_GetNextSibling, stsXML_GetParent, stsXML_GetPrevSibling

Version History

1.0 — Initial appearance as XMLGetChildren.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*; added the ability to specify a depth for the recursion through the descendants of *nodeDesc*.

stsXML_GetFirstChild

Summary

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

Syntax

`stsXML_GetFirstChild(nodeDesc[,childNodeType])` → `childNodeID`

Arguments

nodeDesc	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 *nodeDesc*.
error	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method retrieves the node ID of the first child owned by *nodeDesc*. If *nodeDesc* 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 stsXML_GetFirstChild("1.5")
end mouseUp
```

… returns:

```
1.6
```

Example 2:

The following example:

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

… returns:

```
1.5
```

See also

stsXML_GetChildren, stsXML_GetNamedChildren, stsXML_GetLastChild, stsXML_GetNextSibling, stsXML_GetParent, stsXML_GetPrevSibling

Version History

1.0 — Initial appearance as XMLGetFirstChild.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetNamedChildren

Summary

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

Syntax

`stsXML_GetNamedChildren(`*`nodeDesc,childName[,returnChildNumbers]`*`)` → *`childList`*

Arguments

`nodeDesc`	The node to be examined.
`childName`	The name of the ELEM to match.
`returnChildNumbers`	Optional. Can be either `true`, `false` or empty; determines whether to return node names with index numbers (if `true`) or node IDs (if `false` or empty).

Possible Returned Values

`childList`	If *returnChildNumbers* is `false`, it will return a return-delimited list of node IDs representing the children of *nodeDesc* whose `name` node property (see Appendix I) matches *childName*. If *returnChildNumbers* is `true`, it will return a return-delimited list of node names with each child's index number provided in brackets (see the examples below for the distinction).
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method returns the return-delimited list of child nodes (either node IDs or node names) owned by *nodeDesc* whose `name` node property (see Appendix I) 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 stsXML_GetNamedChildren recursively for each child.

Example 1:

The following example:

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

… returns:

```
1.5
```

Example 1:

The following example:

```
on mouseUp
  get stsXML_GetNamedChildren("/widgets","widget","true")
end mouseUp
```

… returns:

```
widget[1]
widget[2]
```

See also

stsXML_GetChildren, stsXML_GetFirstChild, stsXML_GetLastChild, stsXML_GetNextSibling, stsXML_GetParent, stsXML_GetPrevSibling

Version History

1.0 — Initial appearance as XMLGetNamedChildren.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*; added ability to optionally return the name and index numbers of all children that match *childName*, instead of node ID numbers.

stsXML_GetLastChild

Summary

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

Syntax

`stsXML_GetLastChild(nodeDesc[,childNodeType])` → `childNodeID`

Arguments

nodeDesc	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 *nodeDesc*.
error	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method retrieves the node ID of the last child owned by *nodeDesc*. If *nodeDesc* 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 stsXML_GetLastChild("1.5")
end mouseUp
```

… returns:

```
1.8
```

Example 2:

The following example:

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

… returns:

```
1.3
```

See also

stsXML_GetChildren, stsXML_GetNamedChildren, stsXML_GetFirstChild, stsXML_GetNextSibling, stsXML_GetParent, stsXML_GetPrevSibling

Version History

1.0 — Initial appearance as XMLGetLastChild.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetNextSibling

Summary

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

Syntax

`stsXML_GetNextSibling(nodeDesc[,siblingNodeType])` → *siblingNodeID*

Arguments

nodeDesc	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 *nodeDesc*.
error	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method retrieves the node ID of the next sibling of *nodeDesc*. If *nodeDesc* 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 stsXML_GetNextSibling("1.6")
end mouseUp
```

… returns:

```
1.8
```

Example 2:

The following example:

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

… returns:

```
1.5
```

See also

stsXML_GetChildren, stsXML_GetNamedChildren, stsXML_GetFirstChild, stsXML_GetLastChild, stsXML_GetParent, stsXML_GetPrevSibling

Version History

1.0 — Initial appearance as XMLGetNextSibling.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetParent

Summary

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

Syntax

`stsXML_GetParent(nodeDesc)` → *parentNodeID*

Arguments

nodeDesc	The node to be examined.

Possible Returned Values

parentNodeID	The node ID of the parent of *nodeDesc*.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

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

Example

The following example:

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

… doesSomething

```
1.0
```

See also

stsXML_GetChildren, stsXML_GetNamedChildren, stsXML_GetFirstChild, stsXML_GetLastChild, stsXML_GetNextSibling, stsXML_GetPrevSibling

Version History

1.0 — Initial appearance as XMLGetParent.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_GetPreviousSibling

Summary

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

Syntax

`stsXML_GetPreviousSibling(nodeDesc[,siblingNodeType])` → *siblingNodeID*

Arguments

nodeDesc	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 *nodeDesc*.
error	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method retrieves the node ID of the previous sibling of *nodeDesc*. If *nodeDesc* 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 stsXML_GetPreviousSibling("1.8")
end mouseUp
```

… returns:

```
1.6
```

Example 2:

The following example:

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

… returns:

```
1.3
```

See also

stsXML_GetChildren, stsXML_GetNamedChildren, stsXML_GetFirstChild, stsXML_GetLastChild, stsXML_GetParent, stsXML_GetNextSibling

Version History

1.0 — Initial appearance as XMLGetPreviousSibling.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_HasChildren

Summary

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

Syntax

`stsXML_HasChildren(nodeDesc[,nodeType])` → `{true|false}`

Arguments

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

Description

This method determines whether the node supplied in *nodeDesc* has children, and optionally if *nodeDesc* 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 *nodeDesc*, and not its children's children.

Example 1:

The following example:

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

… returns:

```
true
```

Example 2:

The following example:

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

… returns:

```
false
```

See also

stsXML_CountChildren, stsXML_CountNamedChildren

Version History

1.0 — Initial appearance as XMLHasChildren.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_clearCallback

Summary

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

Syntax

```
stsXML_clearCallback
```

Arguments

```
none
```

Possible Returned Values

```
none
```

Description

After a callback has been set with stsXML_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
  stsXML_clearCallback
end mouseUp
```

See also

stsXML_setCallback

Version History

1.1 — Initial appearance as xml_clearCallback.

stsXML_dump

Summary

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

Syntax

`stsXML_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 **gSTSXML_Data** (see Appendix I)), including node pointers.
`error`	An error was found during execution. Check **gSTSXML_Error** 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 (see Appendix I)).

The content (i.e. the `cont` node property of each node (see Appendix I)) contains everything: white space (which may or may not be converted to symbols – see below), node pointers (in the form `[>>` *nodeDesc*`]`) 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*):

`[`*nodeDesc*`:`*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; for more information on the **gSTSXML_Data** array, see Appendix I at the end of this document.)

Example

The following example:

```
on mouseUp
  get stsXML_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

stsXML_expand

Version History

1.0 — Initial appearance as xml_dump.

stsXML_expand

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

`stsXML_expand(`*`nodeDesc[,normalize]`*`)` → *`xmlContent`*

Arguments

`nodeDesc`	The node descriptor 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 *nodeDesc*.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method takes a node's content (its `cont` node property from the **gSTSXML_Data** array), 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 *nodeDesc* and its descendants have been expanded. To expand a whole document, pass the document node number for *nodeDesc*.

The general purpose of this utility is to convert data held in the **gSTSXML_Data** 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; for more information on the **gSTSXML_Data** array, see Appendix I at the end of this document.).

Example 1:

The following example:

```
on mouseUp
  get stsXML_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 *nodeDesc*):

```
<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 stsXML_expand("1.0")
end mouseUp
```

See also

stsXML_dump, stsXML_expandPaths, stsXML_prettyPrint

Version History

1.0 — Initial appearance as xml_expand.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_expandPaths

Summary

This method takes a node's content, and returns all the node paths from the specified node and below in the node hierarchy.

Syntax

`stsXML_expandPaths(`*`nodeDesc`*`)` → *`pathList`*

Arguments

`nodeDesc`	The node descriptor whose contents are to be expanded.

Possible Returned Values

`pathList`	A return-delimited list of node paths corresponding to the target node and all of its ELEM descendents.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method takes the node's current location in the node hierarchy, and recursively gets the children (ELEM type only) of the node, retrieving the paths for all the nodes along the way (the `path` node property from the **gSTSXML_Data** array (see Appendix I). To expand a whole document, pass the document node number (`1.0`, `2.0`, etc.) for *nodeDesc*.

Example 1:

The following example:

```
on mouseUp
  put stsXML_expandPaths("1.5")
  --you can also do:  put stsXML_expandPaths("1/products/widgets")
end mouseUp
```

… will put the following into the Message Box:

```
1/products/widgets/widget[1]
1/products/widgets/widget[2]
```

Example 2:

The following example expands a whole document:

```
on mouseUp
  put stsXML_expandPaths("1.0")
end mouseUp
```

… which put the following into the Message Box:

```
1/products
1/products/widgets
1/products/widgets/widget[1]
1/products/widgets/widget[2]
```

See also

stsXML_dump, stsXML_expand

Version History

2.0 — Initial appearance.

stsXML_getNodeID(s)

Summary

This method takes one ore more node paths and translates them to the node ID format for use in passing to other XML Library methods.

Syntax

```
stsXML_getNodeID(nodePath) → nodeID
stsXML_getNodeIDs(nodePathList) → nodeIDList
```

Arguments

nodePath	The node path to convert.
nodePathList	A return-delimited list of node paths to convert.

Possible Returned Values

nodeID	The node ID number for the path supplied in *nodePath.*
nodeIDList	A return-delimited list of node ID numbers for the paths supplied in *nodePathList.*
error	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method converts one or more node paths to node IDs. Note that only ELEM nodes can contain a path.

You can choose to use the singular or plural form of this method interchangeably; for example you may wish to use the plural form for clarity in your code if working with multiple nodes (but you don't have to if you don't want to).

Example:

Here's an example of getting the ID of a node:

```
on mouseUp
  get stsXML_getNodeID("/products/widgets/widget[1]")
end mouseUp
```

… returns:

```
1.6
```

See also

stsXML_getNodePath

Version History

2.0 — Initial appearance.

stsXML_getNodePath(s)

Summary

This method takes one ore more node IDs and translates them to the node path format for use in other applications that need a node identifier in this format, or for use in passing to other XML Library methods.

Syntax

```
stsXML_getNodePath(nodeID) → nodePath
stsXML_getNodePaths(nodeIDList) → nodePathList
```

Arguments

nodeID	A specific node ID number to convert.
nodeIDList	A return-delimited list of node IDs to convert.

Possible Returned Values

nodePath	The node path to the node supplied in *nodeID*.
nodePathList	A return-delimited list of node paths to the nodes supplied in *nodeIDList*.
error	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method converts one or more node IDs to node paths. If the there is more than one child node of the same name, the numeric index to that child is supplied in square brackets after the child name (see the example below).

You can choose to use the singular or plural form of this method interchangeably; for example you may wish to use the plural form for clarity in your code if working with multiple nodes (but you don't have to if you don't want to).

Example 1:

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

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

… returns:

```
/products/widgets
```

Example 2:

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

```
on mouseUp
  get stsXML_getNodePaths("1.6" & cr & "1.8")
end mouseUp
```

… returns:

```
/products/widgets/widget[1]
/products/widgets/widget[2]
```

See also

stsXML_getNodeId

Version History

1.1 — Initial appearance as xml_getPath; however it was determined that the format that was being returned was incorrect and not very usable (returned a path like `//root/parent/parent/child(index)` instead of the proper `/root/parent/parent/child[index]`, and was immediately deprecated in favor of the upcoming stsXML_getNodePath.

2.0 — Added to replace stsXML_getPath; added ability to accept multiple node IDs in a return-delimited list.

stsXML_isDocument

Summary

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

Syntax

`stsXML_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 **gSTSXML_Error** for the error found.

Description

This method checks the number supplied in *docNumber* against the list of parsed documents in **gSTSXML_Data** (effectively calling stsXML_GetDocuments) 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 stsXML_DeleteDocument method.

Example

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

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

… returns:

```
false
```

See also

stsXML_isNode, stsXML_isType, stsXML_GetDocuments, stsXML_DeleteDocument

Version History

1.0 — Initial appearance as xml_isDocument.

stsXML_isNode

Summary

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

Syntax

`stsXML_isNode(nodeDesc)` → `{true|false}`

Arguments

nodeDesc	The node ID or path number to examine.

Possible Returned Values

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

Description

This method checks the node supplied in *nodeDesc* to determine if it is valid by examining the `type` node property of *nodeDesc* and seeing if it has a value. Since all nodes must have a type, if the type for some reason 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 stsXML_DeleteNode method.

Example

Assuming no nodes had been deleted, this example:

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

… returns:

```
true
```

See also

stsXML_isDocument, stsXML_isType, stsXML_DeleteNode

Version History

1.0 — Initial appearance as xml_isNode.

2.0 — Added ability to accept a node path as well as a node ID for *nodeDesc*.

stsXML_isType

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

`stsXML_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 **gSTSXML_Error** 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 stsXML_isType("DOC")
end mouseUp
```

… returns:

```
false
```

Example 2:

The following example:

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

… returns:

```
true
```

See also

stsXML_isDocument, stsXML_isNode

Version History

1.0 — Initial appearance as xml_isType.

stsXML_libVersion

Summary

This method retrieves the version of the XML Library.

Syntax

`stsXML_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 stsXML_libVersion()
end mouseUp
```

… returns:

```
2.0
01/07/2006
Standard
```

See also

n/a

Version History

1.0 — Initial appearance as xml_libVersion.

stsXML_normalize

Summary

This method normalizes any data string passed to it.

Syntax

`stsXML_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 **gSTSXML_Error** 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 stsXML_normalize(tData)
end mouseUp
```

… returns:

```
This is a test.
```

See also

n/a

Version History

1.0 — Initial appearance as xml_normalize.

stsXML_prettyPrint

Summary

This method outputs a formatted block of XML along with node ID numbers.

Syntax

`stsXML_prettyPrint(`*`nodeDesc`*`)` → *`formattedXML`*

Arguments

`nodeDesc`	The node descriptor whose descendents and contents are to be expanded and annotated.

Possible Returned Values

`formattedXML`	The expanded chunk of XML content that references the descendants of *nodeDesc*.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method is similar to stsXML_expand, in that it expands a node and its descendents to reformulate the original XML. However this method prepends each node with the node number to make debugging easier, and cleans up the result so that it can be more easily viewed.

To expand a whole document, pass the document node number (`1.0`, `2.0`, etc.) for *nodeDesc*.

Example:

The following example:

```
on mouseUp
  get stsXML_prettyPrint("1.0")
end mouseUp
» » <widget id="1" name="Thingimajig" color="navy">¬†
» » » <![CDATA[The <b>NEW</b> Thingimajig!¬†
Comes in 5 new colors!]]>¬†
```

… will give you this:

```
1.0  »  <DOC>
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>
  »  »  </DOC>
```

See also

stsXML_dump, stsXML_expand, stsXML_expandPaths

Version History

2.0 — Initial appearance.

stsXML_replaceEntities

Summary

This method takes any data that is passed to it, and replaces any supported character entities that are found with their proper replacement.

Syntax

`stsXML_replaceEntities(dataString)` → *`newData`*

Arguments

`dataString`	The string of data to be processed.

Possible Returned Values

`newData`	The string *dataString*, after it has had supported character entities replaced with their proper characters.
`error`	An error was found during execution. Check **gSTSXML_Error** for the error found.

Description

This method replaces the character entities in the table below with their proper replacements. Note that this table is limited to a handful of the more common character entities. To convert other entities, you can use Revolution's `replace` command or `replaceText` function.

Entity	Replacement
"	"
&	&
<	<
>	>
–	–
—	—

Entity	Replacement
“	“
”	”
‘	‘
’	’
€	€
†	†

Entity	Replacement
‡	‡
©	©
®	®
™	™
£	£
¢	¢

Example

The following example:

```
on mouseUp
  put "This is a &lt;tag&gt;" into tData
  get stsXML_replaceEntities(tData)
end mouseUp
```

… returns:

```
This is a <tag>
```

See also

n/a

Version History

2.0 — Initial appearance.

stsXML_reset

Summary

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

Syntax

```
stsXML_reset
```

Arguments

```
none
```

Possible Returned Values

```
none
```

Description

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

Example

The following example:

```
on mouseUp
  stsXML_reset
end mouseUp
```

… deletes the **gSTSXML_Data** array.

See also

n/a

Version History

1.0 — Initial appearance as xml_reset.

stsXML_setCallback

Summary

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

Syntax

`stsXML_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 **gSTSXML_Error** 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** gSTSXML_Error 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 **gSTSXML_Error** after every XML method) (See Example 1). Used without **exit to top** allows script processing to continue, perhaps for error logging (see Example 2). To remove the callback, use stsXML_clearCallback.

Example 1:

To set up a callback if something goes wrong:

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

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

See also

stsXML_clearCallback

Version History

1.1 — Initial appearance as xml_setCallback.

stsXML_validateNode

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

`stsXML_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 **gSTSXML_Error** 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 Appendix II at the end of this document.

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 stsXML_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

Version History

1.0 — Initial appearance as xml_validateNode.

Appendix I: The **gSTSXML_Data** Array

About the gSTSXML_Data Array

When the XML Library parses the nodes in an XML document, each node is validated, given a *nodeID* number (in the form *documentNumber.nodeNumber*), and is assigned specific node properties.

The XML Library stores all of the data about each node that it parses in a global array called **gSTSXML_Data**. Each node is stored in the array by using a number of "slots" that correspond to a combination of its *nodeID* and one of its attributes. For example, the "slot" in the **gSTSXML_Data** array that corresponds to *nodeID* 1.5 (document #1, node #5) with the node attribute `cont` (for "content") would be stored in this "slot" of the **gSTSXML_Data** array:

```
gSTSXML_Data["1.5cont"]
```

Node Properties

Each node has a set of *node properties* that govern what they are and how their data is stored in the **gSTSXML_Data** 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 *nodeDesc* of the parent of this node.
`kids`	Child list	A return-delimited list of *nodeDescs* 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 *nodeDesc* 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 *nodeDesc* 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.
`path`	XML path	Used by ELEM only, this contains the node path that points to this node.
`indx`	XML path index	Used by ELEM only, this contains the index number for same-named siblings. For example, a node path of `/root/item` has an empty value for the `indx` property, whereas a node path of `/root/item[5]` has a value of "5" for the `indx` property.

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

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

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

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

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

gSTSXML_Data["1.6path"]	`/products/widgets/widget[1]`
gSTSXML_Data["1.6indx"]	`1`

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

gSTSXML_Data["1.0type"]	`DOC`
gSTSXML_Data["1.0strt"]	
gSTSXML_Data["1.0name"]	
gSTSXML_Data["1.0attl"]	
gSTSXML_Data["1.0prnt"]	
gSTSXML_Data["1.0kids"]	`1.1` `1.2`

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

gSTSXML_Data["1.0root"]	`1.2`
gSTSXML_Data["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 stsXML_DeleteDocument. This is the result:

gSTSXML_Data["docs"]	`1` `3`

Appendix II: The W3C Specification for XML Data

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[4]	"	"
`Name`	Name[5]	`(Letter | _ | :)(Letter | Digit`	`_hello`

[4] 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.

		`| . | - | _ | :)*`	`A3-Choice`
`Reference`	Character or Entity Reference	`(&#[0-9]+; | &#x[0-9a-fA-F]+; | & Name ;)`	`T` `
` `&Fred;`
`AttValue`	Attribute Value	`qt (Any [^%&*qt] | Reference)* qt`	`"www.abc.com"` `"T"`
`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…
`<` **`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[6]…
`< Name (S Name Eq AttValue)* S?` **`/>`**	… and finally terminated with a forward slash and greater-than symbol.

[5] 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.

[6] 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".

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 used by the XML Library:

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

Appendix III: Comparing the XML Library With Revolution's revXML.DLL

Overview

The STS XML Library was developed before Runtime Revolution came out with their own method of parsing/managing XML, the external DLL file called *revXML.dll*. As such, it is logical that there be some comparison between the two in order to make the proper decision as to which to use, or for the purposes of migrating from one to the other.

For that purpose, the comparison below is divided into two parts – the "product" itself (how it was built, what its strengths and weaknesses are) and then comparing the individual commands/functions/methods to see how they compare.

Comparing the Products

The table below summarizes a comparison of the two products:

Feature	STS XML Library	revXML.DLL
Written in	Transcript	C++
Supports reading XML data	Yes	Yes
Supports writing XML data	Yes (Standard only)	Yes
Supports DTDs (document type declarations)	No	Yes
Supports SOAP	No	Yes
Can handle multiple XML documents	Yes	Yes
Can accept and parse badly formatted XML	Yes (optional)	Yes (optional)[7]
Modifiable by the end user	Yes (Standard only)	No
Can manipulate node types other than element and text nodes (comments, processing instructions, XML declarations)[8]	Yes	No
Maintains original whitespace in XML when retrieving the XML tree (document) as text	Yes	No
Normalizes data node contents	Yes	No
Documentation	Comprehensive	Good
Includes sample XML	Yes	Yes
Requires additional file(s) to install	No	Yes

The XML Library, since it is written in Transcript is both modifiable (as long as you have the Standard version) and doesn't require any add-ons. However, since it *is* written in Transcript, it is not nearly as fast as the revXML.dll in parsing large XML documents. In fact, if you need to parse XML documents with thousands of nodes, the XML Library will most likely be too slow for your needs. It also doesn't support SOAP or DTDs, as the revXML.dll does; however the documentation for the revXML.dll (according to reports) is not as good as the docs you are reading now, and the XML Library does do a good job at attempting to keep the whitespace

[7] Although you can set a flag that tells the revXML.dll to only accept well-formed XML, it apparently doesn't consider the XML declaration (e.g. `<? xml version="1.0" ?>`) in its consideration of well- or poorly-formed XML. The XML Library does, however, so XML documents without the XML declaration will return a parsing error.

[8] The revXML.dll can manipulate CDATA nodes, but not easily. You need to create an XML template with your CDATA tags, and then use `revAppendXML` to add it to your document and then `revPutIntoXMLNode` to populate it.

from the original document intact when manipulating the data so that if you choose to write that XML back to a file, it is in a more human-readable form[9].

Also one thing to note is that for some reason, the revXML.dll does not normalize the contents of a node, so this node:

```
<info>This          is          a         bunch        of       data</info>
```

when retrieved with revXMLNodeContents returns:

```
This          is          a         bunch        of       data
```

whereas when the same data is retrieved with the XML Library's stsXML_GetNodeData, returns:

```
This is a bunch of data
```

Normalizing of data is the standard set forth by the W3C with the only exception being CDATA nodes, whose content is left alone. This may be a bug in the revXML.dll, but is worth noting here.

Comparing the Methods

Two tables will be provided for the purposes of comparing the revXML.dll and the XML Library.

The first shows the XML Library compared to the revXML.dll, with the focus on the revXML.dll, listing out all of its functions and comparing them against the XML Library to see if there are comparable matches:

revXML.DLL Function/Command	STS XML Library Method	Comments
revAddXMLNode	stsXML_CreateNode	XML Library can create a node as a sibling or a child of the referenced node; revXML.dll can only add a node after the last child node of the referenced node.
revAppendXML	—	
revCreateXMLTree	stsXML_LoadData	
revCreateXMLTreeFromFile	stsXML_LoadData	
revDeleteAllXMLTrees	stsXML_reset	
revDeleteXMLNode	stsXML_DeleteNode	
revDeleteXMLTree	stsXML_DeleteDocument	
revPutIntoXMLNode	stsXML_SetNodeData	
revSetXMLAttribute	stsXML_SetAttribute or stsXML_CreateAttribute	
revXMLAddDTD	—	
revXMLAttribute	stsXML_GetAttribute	
revXMLAttributes	stsXML_GetAttributes	
revXMLAttributeValues	—	

[9] Apparently there is an undocumented third parameter to the `revXMLText` command, that if "true" is passed, some whitespace is retained, although XML added with `revAddXMLNode` doesn't get this treatment and it is unclear just how much whitespace is actually retained.

revXML.DLL Function/Command	STS XML Library Method	Comments
revXMLChildContents	stsXML_GetNodeXML	revXML.dll lets you specify depth; XML Library returns either the specified node, or all the children under the node specified
revXMLChildNames	stsXML_GetNamedChildren	
revXMLFirstChild	stsXML_GetFirstChild	
revXMLMatchingNode	stsXML_FindNodesByAttribute	
revXMLNextSibling	stsXML_GetNextSibling	
revXMLNodeContents	stsXML_GetNodeData *or* stsXML_GetCDATA *or* stsXML_GetText	
revXMLNumberOfChildren	stsXML_CountChildren or stsXML_CountNamedChildren	revXML.dll can do both in same function
revXMLParent	stsXML_GetParent	
revXMLPreviousSibling	stsXML_GetPreviousSibling	
(revXMLRemoveAttribute)	stsXML_DeleteAttribute	*NOTE:* revXMLRemoveAttribute is not documented in Revolution as of the time of this writing (Rev v2.6.1), and may not actually work.[10]
revXMLRootNode	stsXML_GetRoot	
revXMLText	stsXML_GetNodeXML	In the XML Library, to get the entire document, you just supply the document number as the node
revXMLTree	—	
revXMLTrees	stsXML_GetDocuments	
revXMLValidateDTD	—	

The second table is the reverse of the first table, focusing instead on the XML Library, and listing out all of *its* methods and comparing them against the revXML.dll to see if there are comparable matches:

STS XML Library Method	revXML.DLL Function/Command	Comments
Loading Methods		
stsXML_LoadData	revCreateXMLTree or revCreateXMLTreeFromFile	
Document Management Methods		
stsXML_CreateDocument	—	revXML.dll cannot create the necessary XML declaration
stsXML_DeleteDocument	revDeleteXMLTree	
stsXML_GetDocuments	revXMLTrees	

[10] It was discovered by attaching the revXML.dll external to a stack and then asking for the `externalCommands` of the stack. It was in the list, but since there's no documentation on it, there's no way to use it.

STS XML Library Method	revXML.DLL Function/Command	Comments
Data/Text Methods		
stsXML_ChildrenToArray	—	
stsXML_GetCDATA	revXMLNodeContents	
stsXML_GetNodeData	revXMLNodeContents	
stsXML_GetText	revXMLNodeContents	
stsXML_SetNodeData	revPutIntoXMLNode	
Attribute Methods		
stsXML_AttributesToArray	—	
stsXML_CountAttributes	revXMLAttributes, then count the result	
stsXML_CreateAttribute	revSetXMLAttribute	
stsXML_DeleteAttribute	(revXMLRemoveAttribute)	The revXMLRemoveAttribute command is not documented and may not actually work; see footnote on previous page.
stsXML_GetAttribute	revXMLAttribute	
stsXML_GetAttributes	revXMLAttributes	
stsXML_HasAttribute	revXMLAttributes, then check the result to see if it contains the attribute you're looking for	
stsXML_SetAttribute	revSetXMLAttribute	
Search Methods		
stsXML_FindNodesByAttribute	revXMLMatchingNode	
stsXML_FindNodesByData	—	
Node Methods		
stsXML_CreateNode	revAddXMLNode	XML Library can create a node as a sibling or a child of the referenced node, and can create all different types of nodes (not just ELEM).
stsXML_DeleteNode	revDeleteXMLNode	
stsXML_GetNodeXML	revXMLChildContents revXMLText	revXML.dll lets you specify depth; XML Library returns either the specified node, or all the children under the node specified
stsXML_GetNodeName	—	Not necessary in revXML.dll since it works with paths and you need to know the name of the element ahead of time
stsXML_GetNodeType	—	revXML.dll does not work with any type other than ELEM.
stsXML_GetRoot	revXMLRootNode	
stsXML_IsEmptyElement	—	revXML.dll can't tell the difference between <tag/> and <tag></tag>.

STS XML Library Method	revXML.DLL Function/Command	Comments
Parent/Child Methods		
stsXML_AppendChild	revAddXMLNode	XML Library can create all different types of nodes (not just ELEM).
stsXML_CountChildren	revXMLNumberOfChildren	
stsXML_CountNamedChildren	revXMLNumberOfChildren	
stsXML_GetChildren	revXMLChildNames	
stsXML_GetFirstChild	revXMLFirstChild	
stsXML_GetNamedChildren	revXMLChildNames	
stsXML_GetLastChild	—	
stsXML_GetNextSibling	revXMLNextSibling	
stsXML_GetParent	revXMLParent	
stsXML_GetPreviousSibling	revXMLPreviousSibling	
stsXML_HasChildren	revXMLChildNames, then return if the count = 0	
Utility Methods		
stsXML_clearCallback	n/a	
stsXML_dump	n/a	
stsXML_expand	revXMLText	XML Library maintains whitespace
stsXML_expandPaths	n/a	revXML.dll does everything with paths
stsXML_getNodeID stsXML_getNodeIDs	n/a	revXML.dll does everything with paths
stsXML_getNodePath stsXML_getNodePaths	n/a	revXML.dll does everything with paths
stsXML_isDocument	revXMLTrees, then check to see if the document number is listed	
stsXML_isNode	—	
stsXML_isType	—	revXML.dll does not work with any types other than ELEM.
stsXML_libVersion	n/a	
stsXML_normalize	—	revXML.dll does not do *any* normalization of data
stsXML_replaceEntities		
stsXML_reset	revDeleteAllXMLTrees	
stsXML_setCallback	n/a	
stsXML_validateNode	—	