Simple XML Writer Module

Document Revision History

Version	Date	Author	Description
0.1	15/11/2001	S. Edwards	Initial Version
0.2	17/11/2001	S. Edwards	Changes for version 0.2 of software added.

Document Summary

The purpose of this document is to describe the functions which are available via this Tcl Namespace. It also includes some simple examples for demonstration purposes.

Table of Contents

Introduction

XML is a popular data representation format that can be considered "human readable". However care must be taken when creating such files to ensure that the contents conform to the required standards.

The Simple Xml Writer module has been written to be used in conjunction with the Simple Xml parser utility. It is a Tcl namespace that offers a small set of procedures which can be used to generate very simple XML files.

The module will ensure that any XML file generated will conform to the required standards to ensure it can be read with either the Simple Xml Parser, or any other XML parser.

This namespace was written for several reasons:

- Tcl Only - many other implementations require a `C' component to be compiled. Although this aids performance it does mean that certain environments where I work would not permit it.
- Education Exercise - it allowed me to develop an understanding how such software might work.

Functionality Overview

The aim of this namespace is to be as simple as possible. Hence the number of procedures available is very limited. Thus, the typical series of steps to generate an XML file would be:

1. Initiate Writer descriptor
2. Start new element
3. Write attributes (optional)
4. Write data (optional)
5. Either repeat 2-4, or Pop current element
6. Finalise Writer descriptor

Although limited functionality exists, the utility does allow nested elements to be written, (as required by the standard - since only a single top level entity should exist). Obviously the exact definition of the files created really depends on how you use the functions. Yet however you use them, whenever a "finalize" is called for the Writer descriptor the procedures will try to ensure that the file only contains conformant information.

Creating a Writer descriptor

The first step prior to using the utility is to create a "descriptor", which can be used to indicate what information should be stored in what file. This descriptor is used in the same way as a file descriptor, as a "handle" to a particular file that is being created.

By using "descriptors" the environment allows multiple files to be generated simultaneously from the same program, (if you really want to...)

The "init" procedure is used to create a new descriptor, and is typically called as shown below:

```
source sxml_writer.tcl
set myfd [sxml::writer::init test_file.xml]
if { $myfd < 0 } {
     puts stderr "[sxml::writer::get_error $myfd]"
     exit 10
}
```

Notice that the return value from the "init" call above is tested to ensure it is not negative. A negative return value indicates an error. When such an error is returned the "get_error" procedure can be called to return a string containing more information about the particular problem.

However if the return value is 0 or greater than the call has been successful and the value indicates the "descriptor" for this file. Using the default options will ensure that an existing file will not be over written. If you want to allow an existing file to be over-written add an additional argument to the "init" call:

```
set myfd [sxml::writer::init test_file.xml 1]
```

Creating a New Element

Once a new descriptor has been returned it can be used to write elements to. To start a new element the "element" procedure is called. The standard use of the "element" routine is to immediately create the top-level entity once the file has been opened. If you wish to create a top level entity called "Jobs", using the descriptor used in the previous code section you could do the following:

```
set x [sxml::writer::element $myfd "Jobs"]
if { $x < 0 } {
     puts stderr "[sxml::writer::get_error $myfd]"
     exit 10
}
```

Notice again that the return value from the call is kept and tested. If an error occurs then the value will be negative. In such a case the "get_error" can be used to return a string with the actual error condition explained.

Calls to this function can be nested to created the required structure of the XML file. More information and examples follow later in the file.

Adding Attribute Information to current Element

Once an element has been created any attributes for that element should be sent prior to another element being started. This is necessary to ensure the interface as well as the code is kept as simple as possible.

For each attribute you wish to add to the latest element added a call to the "attribute" procedure should be made. The procedure takes the descriptor, the attribute name, and the value to assign to the attribute. For example to set the attribute "date" to "15.01.2001" and attribute "time" to "15:02", the following code could be used:

```
sxml::writer::attribute $myfd "date" "15.01.2001"
set err [sxml::writer::attribute $myfd "time" "15:02"]
if { $err < 0 } {
     puts stderr "[sxml::writer::get_error $myfd]"
      exit 10
}
```

In this example we don't check the return code of each call to attribute - if the first one fails, so will the second (for whatever reason). Hence if you know how many attributes you are going to set, simple perform the error checking on the last attribute.

There is no limit to the number of attributes you may add. However, there are limitations on the name of the attributes:

- No two attributes for the same element can use the same name. If the same name is used more than once then the attribute will be assigned the value givein in the last call.
- The name given for an attribute must be a single word, and only contain letters, and underscore characters. The present version of the code does not check to see if the attribute name given conforms to the required standards.

Adding Data Information to current Element

As with attributes, as soon as an element has been defined it is possible to add data to it. Adding data to the current element is just as simple as setting attributes - actually simpler still! When adding data the "data" procedure can be called one or more times. Every time it is called the data specified in the call is concatenated for the current element.

For example to add two lines of "text" to the current element, the following code might be used:

```
sxml::writer::data $myfd "This is a single line of text"
set x [sxml::writer::data $myfd "\nThis is 2nd line of
text"]
if { $x < 0 } {
      puts stderr "[sxml::writer::get_error $myfd]"
      exit 10
}
```

As with attributes you might want to only check the last call to "data" for an attribute, if you know when the last call will be. Notice that the second call to the routine includes a leading "\n" character. This ensures the data contains a new line - the utiity will indent the first line automatically, but will add no futher formatting. If you want to add white space or line feeds, you must do this.

Finishing the current Element

When you have added all required attributes and data for the lastly opened element you need to indicate to the utilities that this element is complete. This will ensure that two things occur:

- The utility will write this element out to disk. If this element is top the top-level element, then all preceeding headers for other elements not already written are written to the file as well.
- It allows the programmer to call the "element" procedure again afterwards to create a new element at the same level as the previous one, rather than being nested.

Finishing the current element is simply a matter of calling the "pop_element" routine, which simply requires the "descriptor" for the file being written. Of course if the format of the file being generated include several levels of elements, you may need several calls to this procedure occasionally.

```
set x [sxml::writer::pop_element $myfd]
if { $x < 0 } {
       puts stderr "[sxml::writer::get_error $myfd]"
       exit 10
}
```

Finishing the File

Finally once all information has been written to the file then you need to ensure the procedure "finalize" is called to write all remaining information to the file, and close the file. Again this is straightforward:

```
set x [sxml::writer::finalize $myfd]
if { $x < 0 } {
     puts stderr "[sxml::writer::get_error $myfd]"
     exit 10
}
```

Using Attribute Directives

As of version 0.2 of the software it is possible to set attribute directives to influence the module in various ways. Presently only one attribute is supported "compact". Attributes are set using the "set_attr" call. As usual the return code indicates whether an error has occured if the value is negative.

Each attribute can take a range of values, though since only "compact" is supported at present a list of attributes and supported values is not included in this document. The "compact" attribute supports values 0 and 1, turning it off and on respectively. Hence to turn on the "compact" attribute, (it is turned off by default), using the following command:

```
set  x [sxml::writer::set_attr $myfd compact 1]
if { $x < 0 } {
     puts stderr "[sxml::writer::get_error $myfd]"
     exit 10
}
```

Character Mappings

When specifying attribute values or data for a particular element the utility will check the contents of the string and replace certain characters to ensure the file will meet the required standards. At present the following character mapping occurs automatically:

Character	Converted To
&	&amp;
<	&lt;
>	&gt;
"	&quot;

When reading back the data using an XML parser the reverse mapping will occur, returning the original contents.

Appendix A: Complete Example Program

The program shown below can be found in the "examples" directory in the distribution, and is called "example_1.tcl". This should be run from the examples directory as follows:

```
./example_1.tcl
```

This is a dummy program, but shows references to all the calls. The complete code is included here as an example:

```
#!/usr/bin/tclsh

# Very simple example - please run from within the "examples" directory

if { ! [file exists ../sxml_writer.tcl] } {
       puts stderr "Error: Please CD to the \"examples\" directory to run example!"
       exit 1
}

source ../sxml_writer.tcl

file delete test_2.xml

set x [sxml::writer::init example_1.xml]
if { $x < 0 } {
       puts stderr [sxml::writer::get_error $x]
       exit 1
}

set myfd $x
set x [sxml::writer::element $myfd "batch"]
if { $x < 0 } {
       puts stderr [sxml::writer::get_error $myfd]
       exit 2
}
set x [sxml::writer::data $myfd "Top object data"]
set x [sxml::writer::attribute $myfd version 2.1]
set x [sxml::writer::data $myfd " More tobj stuff"]

set x [sxml::writer::element $myfd "job"]
if { $x < 0 } {
       puts stderr [sxml::writer::get_error $myfd]
       exit 2
}

set x [sxml::writer::attribute $myfd name "Job Bloggs"]
set x [sxml::writer::attribute $myfd age 44]
if { $x < 0 } {
       puts stderr [sxml::writer::get_error $myfd]
       exit 3
}

set x [sxml::writer::pop_element $myfd]

set x [sxml::writer::element $myfd "job2"]
if { $x < 0 } {
       puts stderr [sxml::writer::get_error $myfd]
       exit 2
}
set x [sxml::writer::attribute $myfd name "Simon Edwards"]
set x [sxml::writer::attribute $myfd age 30]
set x [sxml::writer::data $myfd "This is my data"]

set x [sxml::writer::pop_element $myfd]
set x [sxml::writer::data $myfd "\nJustin \"Ethan\" luke"]
set x [sxml::writer::finalize $myfd]
if { $x < 0 } {
```

```
      puts stderr [sxml::writer::get_error $myfd]
      exit 2
}

exit 0
```

The output from this file is as follows:

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

<batch
  version="2.1"
>
  <job
    name="Job Bloggs"
    age="44"
  >

  </job>
  <job2
    name="Simon Edwards"
    age="30"
  >
    This is my data
  </job2>
  Top object data More tobj stuff
Justin &amp;quot;Ethan&amp;quot; luke
</batch>
```

Appendix B: Using the "compact" Attribute

In the examples directory two additional programs are present, "example_2.tcl" and "example_2_nocompact.tcl". The two programs are identical, apart from the first sets the "compact" attribute, and the second does not. To save space the programs are not listed here, but the output is.

Firstly the output from the "example_2.tcl" program, (in a file called example_2.xml):

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

<batch version="2.1">
  <job>
    <name>Job Bloggs</name>
    <age>44</age>
  </job>
  <job>
    <name>Simon Edwards</name>
    <age>30</age>
  </job>
</batch>
```

Secondly the same data written without the compact setting, (the default):

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

<batch
  version="2.1"
>
  <job>
    <name>
      Job Bloggs
    </name>
    <age>
      44
    </age>

  </job>
  <job>
    <name>
      Simon Edwards
    </name>
    <age>
      30
    </age>

  </job>

</batch>
```