XBinder

XML Schema Compiler
Version 1.0.x
C/C++ User's Manual

Objective Systems, Inc. - March 2004

The software described in this document is furnished under a license agreement and may be used only in accordance with the terms of this agreement.

Copyright Notice

Copyright © 2003–2004 Objective Systems, Inc.

All Rights Reserved

This document may be distributed in any form, electronic or otherwise, provided that it is distributed in its entirety and that the copyright and this notice are included.

Author's Contact Information:

Comments, suggestions, and inquiries regarding XBinder may be submitted via electronic mail to info@obj-sys.com.

CHANGE HISTORY

Date	Author	Version	Description
7/23/2003	ED		Initial version
12/31/2003	ED	0.5.x	Add information on new and changed features.
03/08/2004	ED	1.0.x	Add information on C++ bindings. Move run-time reference information to a separate document.

Table of Contents

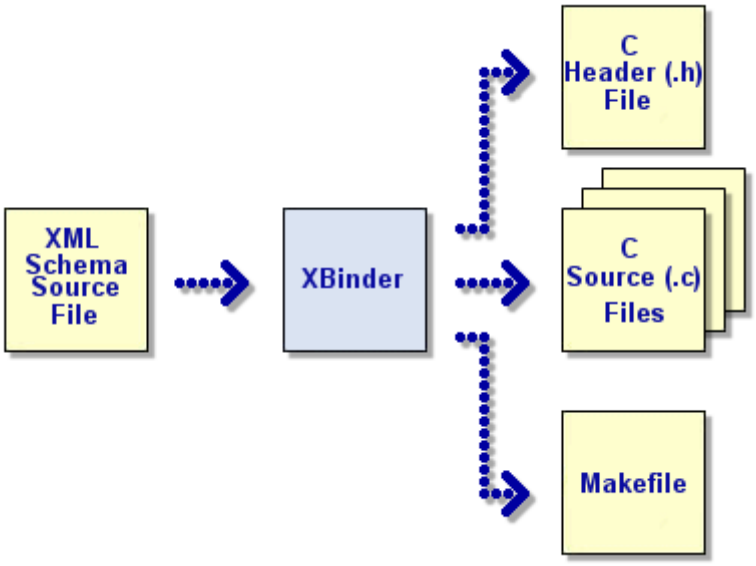
XBinder Overview

The XBinder code generation tool translates an XML Schema Definitions (XSD) or Web Services Definition Language (WSDL) source file into computer language C or C++ source files. These source files contain an application programming interface (API) that allows programatic data to be encoded (marshalled) to XML format and decoded (unmarshalled) to programatic variables. Each variable is of a type that corresponds to a type, element, or attribute defined within the XML schema document.

Each XSD source file results in the generation of the following C/C++ language files:

- An include (.h) file containing C typedefs or C++ classes that represent each of the XSD types and global elements contained within the XSD source file, and
- Multiple C/C++ source (.c/.cpp) files containing encode, decode, and utility functions. One encode and decode function is generated for each XSD type. Utility functions may be generated for a given type or element to initialize, print, or generate test data.
- An optional makefile to build the generated code.

These files, when compiled and linked with the XBinder run-time encode/decode function library, provide a complete package for working with XML encoded data.



XBinder is compliant with the 2001 version of the W3C XML Schema standard (http://www.w3.org/2001/XMLSchema). The encode API functions generate valid, well-formed XML messages that are consistent with the encoding procedures described in the standard. The decode API options are capable of decoding an instance of an XML instance that complies with the schema definition.

Using XBinder

Running XBinder from the Command-line

The XBinder distribution contains a command-line compiler executable as well as a graphical user interface (GUI) wizard (Windows version only) that can aid in the specification of compiler options. This section describes how to run the command-line version; the operation of the GUI is described in the online help files built into the wizard.

To test if XBinder was successfully installed, enter `xbinder` with no parameters as follows (note: if you have not updated your PATH variable, you will need to enter the full pathname):

```
xbinder
```

You should observe the following display (or something similar):

```
XBinder Compiler, Version 1.0.x
Copyright (c) 2002-2004 Objective Systems, Inc. All Rights Reserved.

Usage: xbinder <filename> options

   <filename>          XML schema source file name

  options:
   -xml                generate XML encode/decode functions
   -soap               add logic to format/parse SOAP headers
   -trace              add trace diag msgs to generated code
   -c                  generate C code
   -c++ or -cpp        generate C++ code
   -config <file>      specify configuration file
   -nodecode           do not generate decode functions
   -noencode           do not generate encode functions
   -warnings           Output compiler warning messages
   -o <directory>      Output file directory
   -I <directory>      Import file directory
   -all                Compile all dependent files
   -w32                Generate Windows code (default = GNU)
   -pdu <element>      Designate element to be a PDU
   -elemCasing <value> Set element name case to lower/upper
   -typeCasing <value> Set type name case to lower/upper
   -useNSPfx           Use namespace prefixes in C/C++ code
   -modularize         Handle included schemas as separate modules

  additional code generation options:
   -genMake            generate makefile
   -genTest            generate test code
```

```
-genWriter          generate writer test program
-genReader          generate reader test program
-genFree            generate memory free functions (C only)
-genPrint [<filename>] or
-print [<filename>] generate print functions
-usePDU <element>  use PDU for writer/reader test program
```

To use the compiler, at a minimum, a single XSD or WSDL source file must be provided along with at least one set of encoding rules and a target output language. The current version of XBinder supports the generation of 'C' (-c option) or C++ (-cpp option) source code and the generation of code to encode/decode XML (-xml). It is anticipated that additional language bindings will be added in the future (C# and Java), and additional binary encoding rules (such as ASN.1 BER and PER) to allow more compact and lightweight messages to be produced for a given schema.

The source file specification can be a full pathname or only what is necessary to qualify the file. If directory information is not provided, the user's current default directory is assumed. Multiple source filenames may be specified on the command line to compile a set of files. The wildcard characters '*' and '?' are also allowed in source filenames (for example, the command 'xbinder *.xsd -c -xml' will compile all XSD files in the current working directory).

The following table lists all of the command line options in alphabetical order and what they are used for:

Option	**Argument**	**Description**
-all	None	This option is used to tell the compiler to generate code for all dependent files in a given compilation. This includes the main XSD files specified on the command line as well as all imported and included schema files.
-c	None	Generate C source code.
-c++ -cpp	None	Generate C++ source code.
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. This is similar to the 'binding schema' used with some other XML data binding applications.

Option	Argument	Description
-elemCasing	lower or upper	This option is used to change the case of the first letter in element names in the C or C++ code from what is specified in the XSD file. This option is typically used when the XSD file contains type, element, and/or attribute names that are the same. It provide a way to make the name disambiguous in the generated C or C++ code. Typically, element names are set to lower case. See *-typeCasing* for changing the case of type names.
-genfree	None	This option instructs the compiler to generate memory free utility functions. Memory free functions are C functions that allow all memory within a given structure to be freed. It is possible to free memory without these functions by using the *rtxMemFree* function on a context. This frees all memory held by the context. But some applications require the capability to free the memory associated with a given structure. That is what the functions generated using this option do.
-genmake	None	This option is used to generate a makefile to build the generated code. The makefile is compatible with either the GNU make utility or the Visual Studio nmake utility depending on the setting of the -w32 command line option described next.
-genreader	None	This option is used to generate a complete reader program similar to what can be found in the sample subdirectory. This program will read an encoded XML document from an input file, decode it, and the print the decoded field values to standard output.
-gentest	None	This option instructs the compiler to generate test utility functions. Test functions populate an instance of each global element defined within a schema with random test data to be encoded. The functions provide a good template for writing code to populate the generated programatic variables. The functions are written to <modulename>Test.c files where <modulename> is the base name of the XSD source file that was parsed.

Option	Argument	Description
-genwriter	None	This option is used to generate a complete writer test program similar to what can be found in the sample subdirectory. This program will populate a record with test data, encode the data into XML, and then write the encoded record to an output file.
-I	<directory>	This option is used to specify a directory that the compiler will search for XSD <import> and <include> items. Multiple –I qualifiers can be used to specify multiple directories to search.
-modularize	None	This option is used when XSD <include> directives are included in a schema to tell the processor to put the generated code in separate output files based on the include file names. The default behavior if this is not used is to include all of the code in the main file that is including the definitions. This can only be used if the included files can be successfully compiled on their own (i.e. are not dependent on definitions from the parent module).
-noencode	None	This option suppresses the generation of encode functions.
-nodecode	None	This option suppresses the generation of decode functions.
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-pdu	<element>	This option tells the compiler to recognize the given global element as a protocol data unit (PDU). A PDU is a main message type in the module for which encode and decode functions are generated. By default, the compiler only recognizes non-referenced global elements as PDU's. This allows this default behavior to be overridden.
-print	None	This option instructs the compiler to generate print utility functions. Print functions are debug functions that allow the contents of generated type variables to be written to stdout. The functions are written to <modulename>Print.c files where <modulename> is the base name of the XSD source file that was parsed.

Option	Argument	Description
-soap	None	Add logic to generated code to add or parse SOAP envelope, body, and fault tags in XML messages. This allows the messages to be used in a SOAP client or server application.
-trace	None	This option is used to tell the compiler to add trace diagnostic messages to the generated code. These messages cause printf statements to be added to the generated code to print entry and exit information into the generated functions. This is a debugging option that allows encode/decode problems to be isolated to a given production processing function. Once the code is debugged, this option should not be used as it adversely affects performance.
-typeCasing	lower or upper	This option is used to change the case of the first letter in type names in the C or C++ code from what is specified in the XSD file. This option is typically used when the XSD file contains type, element, and/or attribute names that are the same. It provides a way to make the name disambiguous in the generated C or C++ code. Typically, type names are set to upper case. See *-elemCasing* for changing the case of element names.
-useNSPfx	None	This option instructs the compiler to add the namespace prefixes defined in the XSD source files to the generated C/C++ names. The format of the names generated when this is specified is <prefix>_<name>. This is useful when an XSD specification consists of multiple shemas defined in multiple namespaces and the same names are used for entities across the specifications but within different namespaces. This prevents name collisions in the generated code at the expense of creating more verbose names.
-warnings	None	Output information on compiler generated warnings.
-w32	None	This option causes the compiler to make adjustments to the code generation that are specific to the Windows operating system. For example the backslash character (\) is used as a path separator instead of forward slash (/). The format of the generated makefile is also in Visual Studio nmake format (see -genmake above).

Option	Argument	Description
-xml	None	Generate encode/decode functions that marshall programatic data to and from XML format.

Compiling and Linking Generated Code

C/C++ source code generated by the XBinder compiler can be compiled using any ANSI standard C or C++ compiler. The only additional option that must be set is the inclusion of the ASN.1 C/C++ header file include directory with the –I option.

When linking a program with compiler-generated code, it is necessary to include the XBinder common run-time library (*osysrt*), the XBinder XML run-time library (*osysrtxml*) and, for decoding, a third-party XML parser library. When including an XML parser library, it is also necessary to link with an object file that provides a common abstraction layer to different vendor implementations. The distribution contains an object file to interface with the EXPAT XML parser (http://www.expat.org/) and the libxml2 XML parser library (http://xmlsoft.org). These files are named *rtXmlExpatIF.obj* and *rtXmlLibxml2IF.obj* respectively (note: different variations of this object file exist for the different library configurations described below). It is possible to create your own implementation of this interface file if linking with a different XML parser library is desired. Source code is provided for the default implementations which can be used as a guide for writing your own implementation.

For static linking on Windows systems, the names of the library files are *osysrt_a.lib* and *osysrtxml_a.lib*. On UNIX/Linux, the library names are *libosysrt.a* and *libosysrtxml.a*. The library files and XML library interface object files are located in the *lib* subdirectory. For UNIX, the –L switch should be used to point to the subdirectory path and *–losysrtxml* and *–losysrt* used to link with the libraries. For Windows, the –LIBPATH switch should be used to specify the library path.

There are several other variations of the C/C++ run-time library files and XML parser library interface files for Windows. The following table summarizes what options were used to build each of these variations:

Library Files	Description
osysrt_a.lib *osysrtxml_a.lib* *rtXml<lib>IF_a.obj*	Static single-threaded libraries. These are built with the –ML option. These are not thread-safe. However, they provide the smallest footprint of the different libraries. (Note: <lib> would be replace with the name of the XML parser library to be used. For example, *rtXmlExpatIF_a.obj* for the EXPAT library.)
osysrt.lib *osysrtxml.lib* *rtXml<lib>IF.obj*	DLL libraries. These are used to link against the DLL versions of the run-time libraries (*osysrt.dll*, etc.)

Library Files	Description
osysrtmt_a.lib *osysrtxmlmt_a.lib* *rtXml<lib>IFmt_a.obj*	Static multi-threaded libraries. These libraries were built with the –MT option. They should be used if your application contains threads and you wish to link with the static libraries (note: the DLL's are also thread-safe).
osysrtmd_a.lib *osysrtxmlmd_a.lib* *rtXml<lib>IFmd_a.obj*	DLL-ready multi-threaded libraries. These libraries were built with the –MD option. They allow linking additional object modules in with the XBinder run-time modules to produce larger DLL's.

For dynamic linking on UNIX/Linux, a shared object version of each run-time library is included in the lib subdirectory. This file typically has the extension .so (for shared object) or .sl (for shared library). See the documentation for your UNIX compiler to determine how to link using these files (it varies for different types of UNIX systems). Typically, if a shared object version of a library exists in the linker library path, the linker will choose it over the static archive library version. So if you want to link with the static libraries, it is usually sufficient to move the shared object files somewhere else (or delete them).

The XBinder distribution contains some utilities to make the creation of build scripts easier. These utilities are as follows:

- A *-genmake* command line option to generate a sample makefile to build the files generated by a specific compilation, and
- Make include files (file ending with extension *.mk*) that contain common symbols for many of the options described above.

The *–genmake* option will cause a makefile to be created that will compile all of the generated source files into object files using the configured C or C++ compiler (the compiler is configured in the **platform.mk** file, see below). This makefile will contain dependencies for all included header files. The default generated makefile will be compatible with the GNU make utility and should be portable to most UNIX/Linux systems. The *–w32* command line switch can be used to generate a makefile that is compatible with the Microsoft Visual Studio *nmake* utility.

The two primary make include files are **platform.mk** and **xmlparser.mk**. The **platform.mk** files contains all of the common definitions for a particular platform. These include the C or C++ compiler and linker to be used and the compile/link options. The **xmlparser.mk** file contains common definitions for interfacing with an XML parser library. It is possible to change XML parser library implementations by simply changing the definitions in this file.

See the *makefile* in any of the sample subdirectories of the distribution for an example of what must be included to build a program using generated source code.

Getting Started with C or C++ Sample Programs

To begin using XBinder to generate C source code, one should start with the sample programs. These are located in the *c/sample* subdirectory of the installation. A good sample program to get started with is the *Employee* sample program. This program contains an XML schema file that describes an employee personnel record.

To run this sample program from the command-line interface, the following procedure should be followed:

1. Open an MS-DOS or other command shell window.
2. Change directory (cd) to the employee sample directory:

   ```
   cd c/sample/Employee
   ```

   Note: this assumes the starting point is the XBinder installation root directory.
3. Execute the `nmake` (Windows) or `make` (Linux/UNIX) utility program to build the program:

   ```
   nmake
   ```

   Note: nmake is a make utility program that comes with the Microsoft Visual C++ compiler. It may be necessary to execute the batch file *vcvars32.bat* that comes with Visual C++ in order to set up the environment variables to use this utility.
4. This should cause the XBinder compiler to be invoked to compile the employee.xsd XML schema file. It will then invoke the configured C compiler to compile the generated C file and test drivers. The result should be a writer.exe and reader.exe program file which, when invoked, will encode and decode a sample employee record.
5. Invoke writer from the command line:

   ```
   writer
   ```
6. This will generate an encoded record and write it to a disk file. By default, the file generated is message.xml. The test program has a number of command line switches that provide encoding options. To view the switches, enter writer ? and a usage display will be shown.
7. Invoke reader from the command line:

   ```
   reader
   ```
8. This will read the disk file that was just created by the writer program and decode its contents. The resulting decoded data will be written to standard output. The test program has a number of command line switches that provide decoding options. To view the switches, enter reader ? and a usage display will be shown.

The procedure to run a C++ sample program is the same except that you would start in one of the cpp/ sample directories. The same procedure applies: execute the make utility and then run the writer and reader programs.

Getting Started with your own XML Schema

The quickest way to get up and running with your own XML schema file or set of files is to let XBinder generate a sample program for you. This is done using the -genwriter and/or -genreader command-line options. These options cause a complete writer and/or reader program to be generated and, when used in combination with -genmake, cause a makefile to be generated to build the entire project.

For example, suppose you had a schema file named mySchema.xsd that you wanted to generate encoders and decoders for. The following command could be used to generate a complete set of C source files for this schema (note: C++ source files could be generated simply by changing -c to -c++ in this command):

```
xbinder mySchema.xsd -c -xml -print -genwriter -genreader -genmake
```

After generation is complete, all that needs to be done is execution of the generated makefile with the make utility program and all of the files will be compiled and linked to form reader and writer executable files. Note that if this is being done on Windows, the -w32 option should be added to generate a makefile that is compatible with the Visual C++ nmake utility.

The generated files can now act as a template on which you can base your own development. The generated mySchemaTest.c (or .cpp) file contains the complete logic necessary to populate a data record corresponding to your schema. The file in this case contains random data. This code generation can be taken a step further if you have a sample of an XML instance that matches you schema (say, for example, mySchemaInstance.xml). In this case, you can add `-gentest mySchemaInstance.xml` to the command line and test source code will be generated that populates a structure with data from the test instance.

Generated C/C++ Source Code

Header (.h) File

The generated C/C++ include file contains a section for each XSD type defined in the XSD source file. In general, there is a one-to-one correspondence between types defined in the XSD file and generated C type or C++ class definitions. In some cases, however, extra types/classes are generated to support certain XSD types. This occurs on XSD complex type definitions when the element nesting level is greater than two (see the section on Complex Types for details).

In addition to XSD types, code is also generated for global element definitions. If no other type references a global element, it is considered to be a main message element (also known as a **protocol data unit** or **PDU**). These elements are encoded into the main XML documents or messages that make up the given specification. An entry point encode and decode function is generated for each of these elements. The header file contains the function prototypes for these functions.

C Code Generated for XSD Types

If C code generation is selected, the following items are generated for each XSD type:

- Choice tag constants (xsd:choice type only)
- C type definition
- Encode function prototype
- Initialization function prototype
- Other function prototypes depending on selected options (for example, print)

A sample section from a C header file is as follows:

```
/*************************************************************/
/*                                                           */
/*  Name                                                     */
/*                                                           */
/*************************************************************/

typedef struct EXTERN Name {
   OSXMLSTRING givenName;
   OSXMLSTRING initial;
   OSXMLSTRING familyName;
} Name;

EXTERN int XmlET_Name
  (OSCTXT* pctxt, Name* pvalue,
   const OSUTF8CHAR* elemName, const OSUTF8CHAR* nsPrefix);

EXTERN int Init_Name (OSCTXT* pctxt, Name* pvalue);

EXTERN void Print_Name
   (const char* name, Name* pvalue);
```

This corresponds to the following XSD type definition:

```
   <xsd:complexType name="Name">
      <xsd:sequence>
         <xsd:element name="givenName" type="xsd:string"/>
         <xsd:element name="initial" type="xsd:string"/>
         <xsd:element name="familyName" type="xsd:string"/>
      </xsd:sequence>
   </xsd:complexType>
```

In this case, the *Name* C struct typedef corresponds to the *Name* XSD complex type definition.

The *XmlET_Name* function prototype is the XML encode function for the type. The type has no decode function because decoding is handled by SAX handler functions.

The *Init_Name* function prototype is the declaration of the type's initialization function. This function is called to initialize a variable of the type before encoding or decoding. It initializes all fields to zero or to the field's fixed or default value as specified in the XSD source file.

The *Print_Name* function prototype is for a print utility function. This is an optional function that was generated by using the –print command line qualifier. It prints the contents of a variable of the generated type to the standard output device.

C Code Generated for XSD Global Elements

At the end of the header file are the function prototypes corresponding to global elements that are not referenced by any other type definitions. These are global elements that are not used in any other type definitions via the *ref* attribute (for example, <complexType name="SomeType" ref="SomeGlobalElement"/>. A sample global element section is as follows:

```
/**
 * Global element functions.  These functions encode or
 * decode complete XML documents.  They are generated for global
 * elements that are either:
 *
 * 1) not referenced by any other types, or
 * 2) explicitly declared to be a PDU using the -pdu command line
 *    option, or
 * 3) explicitly declared to be a PDU using the <isPDU/>
 *    configuration file element.
 */
EXTERN int XmlE_personnelRecord
  (OSCTXT* pctxt, PersonnelRecord* pvalue);

EXTERN int XmlD_personnelRecord
  (OSCTXT* pctxt, PersonnelRecord* pvalue);
```

In this case, the global element function prototypes correspond to the following global element declaration in the XSD file:

```
   <xsd:element name="personnelRecord" type="PersonnelRecord"/>
```

This element is not referenced by any other types in the specification. An encode and decode function prototype is generated for the declaration. See the section on *Calling Generated C Encode and Decode Functions* for a step-by-step description on how to call these functions.

The following sections describe the specific mappings of XSD types to C types.

C++ Code Generated for XSD Types

For C++, a class definition is generated for each XSD type. This class is derived from either the *OSBaseType* run-time class or from a descendent of this class. The class may contain a constructor for initialization of member variables and a destructor to free dynamic memory held by the class. Method declarations will also be generated instead of C function prototypes for encoding, decoding, printing, and generation of test data. For some types, additional helper methods may also be declared (for example, enumerated type definitions contain a *toString* method declaration).

A sample section from a C++ header file corresponding to the XSD Name type defined above is as follows:

```
/*************************************************************/
/*                                                           */
/*  Name                                                     */
/*                                                           */
/*************************************************************/

class EXTERN Name : public OSBaseType {
public:
   OSXMLStringClass givenName;
   OSXMLStringClass initial;
   OSXMLStringClass familyName;

   int encodeXML (OSCTXT* pctxt,
      const OSUTF8CHAR* elemName, const OSUTF8CHAR* nsPrefix);

   void print (const char* name);
} ;
```

If you compare this to what was generated for C above, you will notice that all of the items are now encapsulated within a class definition. This includes the element declarations and well as the functions which are now methods in the class.

C++ Code Generated for XSD Global Elements

Special classes called **control classes** are generated for global elements that are not referenced by any other type definitions. These are global elements that are not used in any other type definitions via the *ref* attribute (for example, <complexType name="SomeType" ref="SomeGlobalElement"/>.

The purpose of a control class is to act as a control interface for encoding or decoding complete XML documents or messages. This class allows a message buffer or stream object to be associated with an XSD type class. Once this association is made, methods can be invoked from within the class to serialize data to and from the type class and the buffer or stream.

A sample global element section is as follows:

```
class EXTERN personnelRecord_CC : public OSXSDGlobalElement {
protected:
   PersonnelRecord& mValue;

public:
   personnelRecord_CC (PersonnelRecord& value);
   personnelRecord_CC
     (OSMessageBufferIF& msgbuf, PersonnelRecord& value);
   ~personnelRecord_CC();

   // standard encode/decode methods (defined in base class):
   // int encode ();
   // int decode ();

   // stream encode/decode methods:
   int encodeTo (OSMessageBufferIF& msgbuf);
   int decodeFrom (OSMessageBufferIF& msgbuf);

   void print (const char* name);
} ;
```

Namespace Considerations

In XML and XML Schema, namespaces are frequently used to ensure the uniqueness of entity names across schema boundaries. By default, XBinder does not use the namespace information when generating names for types, elements, and attributes in the C or C++ code. This is done to provide shorter and more concise names, but it sometimes leads to collisions and ambiguous names in the code.

There are a number of methods that can be used to remove this ambiguity. These are described below.

Use of the -useNSPfx Command Line Switch

Specifying *-useNSPfx* on the command line when compiling a set of XSD specifications will cause the namespace prefixes to be added to the generated C or C++ names. This will ensure that no naming collisions will occur (this is only true, of course, if the XSD specifications being compiled are valid in their use of namespaces). However, the generated C or C++ names will be longer as they will be of the form <*prefix*>_<*localName*> where prefix is the defined namespace prefix and localName is the local name for the item.

One thing to be aware of when using *-useNSPfx* is that prefixes for a given namespace URI can change across schemas. For this reason, it is recommended that all schemas that make up a project be compiled at once to ensure that the same prefix is used for a given name. This can be done by either including all of the schema filenames to be compiled on the command line at once, or by using the *-all* switch to

instruct the compiler to compile all included and/or imported schemas. The prefix that is used for a given name is the first one encountered during the compilation process. If you know that namespace prefix names are maintained in a consistent manner across schemas (i.e. the same prefix is always used to describe a given URI), then it is OK to compile the schemas individually with this option.

Use of the -typeCasing and -elemCasing Command Line Switches

Global element and type names may be the same in a given schema. While this may be a questionable programming practice from a logical point of view, it is legal and it will cause problems in XBinder generated code because the generated names will clash. This can even be the case within the same namespace; therefore, use of the *-useNSPfx* option cannot be used to solve this problem.

The *-typeCasing* and/or *-elemCasing* options provide a quick and easy way to fix these names in all compiled schemas. By setting one or the other (or both) to different case values (upper or lower), you can ensure that no name collisions of this sort will occur. The typcial convention when using these switches is to set element name case to lower and type name case to upper.

Use of the <prefix> Configuration File Setting

More specific control of naming problems can be achieved by using the <prefix> configuration file setting. This allows individual entities within a schema specification to be targeted for name alteration. It is a good alternative when you only have a few name clashes and do not want to add the verbosity to all names introduced by the *-useNSPfx* switch.

See the section on configuration file use for specifics on how to use a configuration file to customize the compilation process. Using <prefix> in a configuration file causes the specified prefix name to be prepended to the generated C or C++ name. This will make the name of the targeted item different in the generated code from another entity having the same name.

XSD Simple Type to C/C++ Type Mappings

XSD built-in simple type declarations are mapped directly to C types defined in the **rtxSysTypes.h** run-time header file. The general mapping of each XSD simple type to a C type is as follows:

XSD Built-In Type	C Type (in rtxSysTypes)	C Type (base)
anyURI	OSXMLSTRING	unsigned char*
base64Binary	OSDynOctStr	struct
boolean	OSBOOL	unsigned char
byte	OSINT8	char
date	OSXMLSTRING	unsigned char*
dateTime	OSXMLSTRING	unsigned char*
decimal	OSREAL	double
double	OSREAL	double
duration	OSXMLSTRING	unsigned char*
ENTITIES	OSRTDList	linked list struct
ENTITY	OSXMLSTRING	unsigned char*
float	OSREAL	double
gDay	OSXMLSTRING	unsigned char*
gMonth	OSXMLSTRING	unsigned char*
gMonthDay	OSXMLSTRING	unsigned char*
gYear	OSXMLSTRING	unsigned char*
gYearMonth	OSXMLSTRING	unsigned char*
hexBinary	OSDynOctStr	struct
ID	OSXMLSTRING	unsigned char*
IDREF	OSXMLSTRING	unsigned char*
IDREFS	OSRTDList	linked list struct
integer	OSINT32	int

XSD Built-In Type	C Type (in rtxSysTypes)	C Type (base)
int	OSINT32	int
language	OSXMLSTRING	unsigned char*
long	OSINT64	long (64-bit integer type)
Name	OSXMLSTRING	unsigned char*
NCName	OSXMLSTRING	unsigned char*
negativeInteger	OSINT32	int
NMTOKEN	OSXMLSTRING	unsigned char*
NMTOKENS	OSRTDList	linked list struct
nonNegativeInteger	OSUINT32	unsigned int
nonPositiveInteger	OSINT32	int
normalizedString	OSXMLSTRING	unsigned char*
positiveInteger	OSUINT32	unsigned int
short	OSINT16	short
string	OSXMLSTRING	unsigned char*
time	OSXMLSTRING	unsigned char*
token	OSXMLSTRING	unsigned char*
unsignedByte	OSUINT8	unsigned char
unsignedShort	OSUINT16	unsigned short
unsignedInt	OSUINT32	unsigned int
unsignedLong	OSUINT64	unsigned long (64-bit)

For C++, class wrappers are added around each of these types when they are used in simple type declarations. In most cases, these classes contain a single public member variable called *value* that holds the value of the type. They also contain a constructor and assignment operator for setting the value.

The following sections provide more detail on these mappings.

Character String Types

XSD defines many kinds of character string types including `string, normalizedString,` and `token`. All of these XSD types are mapped to an OSXMLSTRING type. This internal type represents a UTF-8 character string. The definition of this type in **rtxSysTypes.h** is as follows:

```
typedef struct OSXMLSTRING {
   OSBOOL cdata;
   const OSUTF8CHAR* value;
} OSXMLSTRING;
```

The *cdata* member of this structure is a flag indicating whether or not the value is to be encoded as an XML CDATA section. The value member is a pointer to the string to be encoded. The underlying C type for the OSUTF8CHAR type is `unsigned char`. This allows the entire UTF-8 character range to be represented as positive numbers.

For C++, the built-in OSXMLSTRING structure is extended to form an XML string class as follows:

```
class EXTERNRTX OSXMLStringClass :
public OSXMLSTRING, public OSBaseType {
 public:
   /**
    * The default constructor creates an empty string.
    */
   OSXMLStringClass();
   ...
} ;
```

This makes the data members from the C type available in the C++ case as well. It also adds constructors and other methods to allow the member variables to be initialized and manipulated.

The general mapping is as follows:

XSD type:

```
<xsd:simpleType name="TypeName">
   <restriction base="xsd:string"/>
</xsd:simpleType>
```

Generated C code:

```
typedef OSXMLSTRING TypeName;
```

Generated C++ code:

```
class TypeName : public OSXMLStringClass {
   ...
} ;
```

In this case, `xsd:string` refers to the XSD string base type and all other types that are derived from it. A variable of the type can be populated with a simple string literal cast to a const OSUTF8CHAR* variable as follows:

```
TypeName strval;
strval.cdata = FALSE;
strval.value = (const OSUTF8CHAR*) "my string";
```

String-based types may be further restricted through the use of facets such as length, minLength, maxLength, and pattern. These have no effect on the generated C type definition. Constraint checks are added to the generated C encoders and decoders to ensure values of the type are within the specified constraint bounds.

Enumerated Type

Another facet that is frequently applied to XSD string-based types is **enumeration**. This results in the generation of a C enum typedef that enumerates all of the identifiers that can be used in the type. The actual C typedef that is generated for the item is of type OSUINT16. The reasons for not directly using the generated enum type are:

- The use of OSUINT16 provides for a consistent size of the data variable. Use of enum can produce different sized variables on different platform/compiler combinations, and
- The variable is capable of storing enumerated types that were not defined in the original set. This makes the type extensible in the event a newer version of the schema is produced that contains additional enumeration items that were not in the original version.

The general mapping is as follows:

XSD type:

```
<xsd:simpleType name="TypeName">
   <restriction base="xsd:string">
      <xsd:enumeration value="enum1"/>
      <xsd:enumeration value="enum2"/>
      ...
      <xsd:enumeration value="enumN"/>
   </xsd:restriction>
</xsd:simpleType>
```

Generated C code:

```
typedef enum {
   TypeName_enum1 = 0,
   TypeName_enum2 = 1,
   ...
   TypeName_enumN = N - 1,
} TypeName_ENUM;

typedef OSUINT16 TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
public:
   enum {
      enum1 = 0,
      enum2 = 1,
      ...
      enumN = N - 1,
   } ;
   OSUINT16 value;
   ...
} ;
```

Note that for C, *TypeName* is used on the enumerated identifiers as a namespace mechanism in order to prevent name clashes if two or more enumerated types use the same identifier names. In this case, the type name may only be a partial fragment of the full name to keep the names shorter. This is not a problem in C++ as the class provides a namespace for the enumeration constants defined within (for example, *enum1* would be referenced as *TypeName::enum1* outside the class).

Integer Types

XSD defines several integer types including `integer, byte, unsignedByte, positiveInteger,` etc.. Each of these types is mapped to a C type depending on the following factors:

- The name of the type (for example, unsignedByte is mapped to a different type - OSUINT8 - than integer – OSINT32,
- Value range facets (minInclusive, maxInclusive, minExclusive, maxExclusive) that are applied to the type.

By default, an `xsd:integer` with no constraints results in the generation of an “OSINT32” type which is standard C signed 32-bit integer type. The general mapping is as follows:

XSD type:

```
<xsd:simpleType name="TypeName">
   <restriction base="xsd:integer"/>
</xsd:simpleType>
```

Generated C code:

```
typedef OSINT32 TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
   OSINT32 value;
   ...
} ;
```

Value range facets will alter the C type used to represent a given integer value. The smallest integer type that can hold the constrained value will always be used. For example, the following declaration declares an integer to hold a value between 2 and 10 (inclusive):

```
<xsd:simpleType name="Int_2_to_10">
        <xsd:restriction base="xsd:integer">
           <xsd:minInclusive value="2"/>
           <xsd:maxInclusive value="10"/>
        </xsd:restriction>
     </xsd:simpleType>
```

In this case, a byte type (unsigned char) could be used to hold the value because it must be between 2 and 10 (a signed byte could also be used but an unsigned value is always used whenever negative numbers are not required). Other value ranges would cause different integer types to be used that provide the most efficient amount of storage.

The following table shows the types that would be used for the different range values:

Min Lower Bound	Max Upper Bound	C Type (rtx)	C Type (base)
-128	127	OSINT8	char (signed 8-bit int)
0	255	OSUINT8	unsigned char (unsigned 8-bit number)
-32768	32767	OSINT16	short (signed 16-bit int)
0	65535	OSUINT16	unsigned short (unsigned 16-bit int)
-2147483648	2147483647	OSINT32	int (signed 32-bit integer)
0	4294967295	OSUINT32	unsigned int (unsigned 32-bit integer)

Real Number Types

XSD defines the following types that are mapped to the C double type:

- float
- double
- decimal

A double is always used because it provides the maximum precision to hold numbers for all of the types above.

The general mapping is as follows:

XSD type:

```
<xsd:simpleType name="TypeName">
   <restriction base="xsd:float"/>
</xsd:simpleType>
```

Generated C code:

```
typedef OSREAL TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
   OSREAL value;
   ...
} ;
```

Binary String Types

XSD defines the following types that are mapped to a C binary type structure:

- hexBinary
- base64Binary

The type of structure used depends on whether or not a length facet is applied to the type. If a length facet is not used, or the length is a very large value (> 32K), a built-in type containing a pointer to a dynamic memory buffer is used to hold the binary data. The definition of this type in **rtxSysTypes.h** is as follows:

```
typedef struct OSDynOctStr {
   OSUINT32     numocts;
   const OSOCTET* data;
} OSDynOctStr;
```

The *numocts* member holds the length of the binary string and the *data* member holds the actual data.

For C++, a built-in class definition is used that extends this structure:

```
class OSDynOctStrClass : public OSDynOctStr, public OSBaseType {
 public:
   ...
} ;
```

This class allows the base members to be accessed just as they were in the C case, but provides initialization through constructors and other utility methods as well.

If a length facet is used that restricts the size of the binary string to a value less than 32K, a custom type is generated that contains a static array to hold the data. The general form of this type is as follows:

```
typedef struct TypeName {
   OSUINT32     numocts;
   OSOCTET      data[length];
} TypeName;
```

In this case, *TypeName* would be the name of the type defined in the XSD specification and length would be the value of the length facet.

In the case of C++, a class is generated:

```
class TypeName : public OSBaseType {
   OSUINT32     numocts;
   OSOCTET      data[length];
    ...
} ;
```

The general mappings for each case are as follows:

Dynamic Case (no length facet):

XSD type:

```
<xsd:simpleType name="TypeName">
   <restriction base="xsd:hexBinary"/>
</xsd:simpleType>
```

Generated C code:

```
typedef OSDynOctStr TypeName;
```

Generated C++ code:

```
class TypeName : public OSDynOctStrClass, OSBaseType
{
   ...
} ;
```

Static Case (length restricted to 32K or less):

XSD type:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:hexBinary">
      <xsd:length value="length"/>
   </xsd:restriction>
</xsd:simpleType>
```

Generated C code:

```
typedef struct TypeName {
   OSUINT32    numocts;
   OSOCTET    data[length];
} TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
   OSUINT32    numocts;
   OSOCTET    data[length];
   ...
} ;
```

Note: in the static case, the *maxLength* facet will cause the same code to be generated with *maxLength* used for the size of the array.

In the case of the *xsd:any* type, it is interesting to note that a binary string is used instead of a text string (the XML encoding for any is simply the fully formed XML text string). The reason for doing it this way is in anticipation of future support for binary encoding rules. The use of a binary field would allow the data to be maintained in its native form whether text or binary encoding was used.

Boolean Type

The `xsd:boolean` type is mapped to a C unsigned char that is allowed to have the value zero for FALSE and any other value for TRUE. The general mapping is as follows:

XSD type:

```
<xsd:simpleType name="TypeName">
   <xsd:restriction base="xsd:boolean"/>
</xsd:simpleType>
```

Generated C code:

```
typedef OSBOOL TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
   OSBOOL value;
   ...
} ;
```

Union Type

An `xsd:union` type is used to specify that one of several simple types can be used for a specific value. This type is mapped to a C structured type that contains an identifier for the selected type and a union of all of the possible types. Atomic types (i.e. those that use a single processor storage unit such as integer or Boolean) are stored as values in the union whereas compound or structured types (such as the structure used to represent a hexBinary type) are stored as pointers.

The general mapping is as follows:

XSD type:

```
<xsd:simpleType name="TypeName">
   <xsd:union memberTypes="Type1 … TypeN"/>
</xsd:simpleType>
```

Generated C code:

```
/* choice tag constants */
#define T_TypeName_type1   1
...
#define T_TypeName_typeN   N

typedef struct TypeName {
   OSUINT16 t;
   union {
      /* t = 1 */
      Type1 type1;
      ...
      /* t = N */
      TypeN typeN;
   } u;
} TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
public:
   // tag constants
   enum {
      T_type1 = 1,
      ...
      T_typeN = N,
   } ;
   OSUINT16 t;
   union {
      /* t = 1 */
      Type1 type1;
      ...
      /* t = N */
      TypeN typeN;
   } u;
   ...
} ;
```

Notes:

1. Where typename begins with a lowercase letter above (for example, *Type1* is shown as *type1* in places), it means the actual typename is used with the first letter set to lowercase.
2. The choice tag constants (T_*TypeName_type*) are the identifiers of each of the particular values in the union. The selected value is stored in the *t* member variable of the generated structure. In the case of C++, the tag values are in the form of an *enum* construct within the class containing enumerations of the form *T_type*.
3. The member variables in the union may be stored as values (if atomic) or as pointers to a value of the item (if structured).

4. The generated C++ class contains additional methods to get, set, or query the union value. These are in the form of *get_type*, *set_type*, and *is_type* respectively.

List Type

An `xsd:list` type is used to model a space-separated list of values of a given type. This type is mapped to a linked-list type. The built-in `OSRTDList` type (run-time doubly linked list) is the type used for repeating sequences such as this. This list type can be used with the **rtxDList** run-time functions for building and manipulating lists. See the section on linked list run-time functions for more details.

In the case of C++, the built-in OSRTDListClass type is extended. This class extends the C OSRTDList structure and adds constructors and methods for adding, finding, and removing items from the list. The generated C++ code contains versions of these methods that correspond to the specific type of the element within the list.

The general C and C++ mapping for an XSD list type is as follows:

XSD type:

```
<xsd:simpleType name="TypeName">
   <xsd:list itemType="Type"/>
</xsd:simpleType>
```

Generated C code:

```
typedef OSRTDList TypeName;
```

Generated C++ code:

```
class TypeName : public OSRTDListClass {
   ...
} ;
```

The one exception to this mapping occurs when the referenced item type is an enumeration. In this case, a structure is generated with each enumerated item represented as a single bit. This is a more compact structure that is easier to work with for specifying enumerated items and for validation to make sure there are no duplicates in the list. The mapping for this special case is as follows:

XSD type:

```
<xsd:simpleType name="EnumType">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="enum1"/>
      <xsd:enumeration value="enum2"/>
      ...
      <xsd:enumeration value="enumN"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="TypeName">
   <xsd:list itemType="EnumType"/>
</xsd:simpleType>
```

Generated C code:

```
typedef struct TypeName {
   unsigned enum1Bit : 1;
   unsigned enum2Bit : 1;
   ...
   unsigned enumNBit : 1;
   OSRTDList* _extItems;
} TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
public:
   unsigned enum1Bit : 1;
   unsigned enum2Bit : 1;
   ...
   unsigned enumNBit : 1;
   OSRTDListClass* _extItems;

   ...
} ;
```

Each of the bit fields in this type represents a declared enumeration item in the XSD definition. The *_extItems* field is added for extensibility purposes (i.e. if an unknown item is received it is added to this list). This construct will be used if a declared enumerated type is referenced (as is the case above) or if the list type contains an anonymous type with an enumeration list.

XSD Complex Type to C/C++ Type Mappings

XSD complex type declarations are mapped to one or more C structured types or C++ classes. The actual mappings are influenced by several factors including the level of nesting of complex type structures within other complex types, facets that are applied to complex type groups and elements, and attributes that are added to the types.

The equivalent C type and C++ class definitions for each of the various XSD complex types follow.

SEQUENCE

The XSD SEQUENCE type <xsd:sequence> is a complex type consisting of a series of element definitions. These elements can reference other XSD types including other complex types. The elements must appear in the order they are declared in XML instances of this type.

In its simplest form, an XSD sequence consists of a series of element definitions that reference other types. The equivalent C type and C++ class mapping for this is a structure that contains the equivalent type mapping for each of the elements as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:sequence>
      <xsd:element name="elem1" type="Type1"/>
      <xsd:element name="elem2" type="Type2"/>
      ...
      <xsd:element name="elemN" type="TypeN"/>
   </xsd:sequence>
</xsd:complexType>
```

Generated C code:

```
typedef struct TypeName {
   Type1  elem1;
   Type2  elem2;
   ...
   TypeN  elemN;
} TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
public:
   Type1  elem1;
   Type2  elem2;
   ...
   TypeN  elemN;
} ;
```

Optional Elements

Elements within a sequence definition may be declared to be optional by using the *minOccurs="0"* facet. This indicates that the element is not required in the encoded message. An additional construct is added to the generated code to indicate whether an optional element is present in the message or not. This construct is a bit structure placed at the beginning of the generated sequence structure or class. This structure always has variable name 'm' (for 'mask') and contains single-bit elements of the form '*elemName*Present' as follows:

```
struct {
   unsigned elemName1Present : 1,
   unsigned elemName2Present : 1,
   ...
} m;
```

In this case, the elements included in this construct correspond to only those elements marked as optional (i.e. with *minOccurs="0"* facet) within the sequence group definition. If a sequence contains no optional elements, the entire construct is omitted.

For example, the following XSD sequence definition declares one optional and one required element:

```
<xsd:complexType name="SeqWithOptElem">
   <xsd:sequence>
      <xsd:element name="reqElem" type="xsd:string"/>
      <xsd:element name="optElem" type="xsd:integer" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
```

The C type that is generated for this XSD type is as follows:

```
typedef struct SeqWithOptElem {
   struct {
      unsigned optElemPresent : 1;
   } m;
   const OSXMLSTRING reqElem;
   OSINT32 optElem;
} SeqWithOptElem;
```

In this case, if the *optElemPresent* flag is set to FALSE in a variable of this type, the contents of the *optElem* field will not be included in an encoded XML instance of the type. Similarly, after decoding, the *optElemPreseent* flag can be tested to see if the message that was decoded contained this element. If this value is FALSE, the contents of the *optElem* field in the variable are undefined.

The C++ case is similar except that the mask structure is contained within the generated C++ class definition:

```
class SeqWithOptElem : public OSBaseType {
```

```
public:
   struct {
      unsigned optElemPresent : 1;
   } m;
   const OSXMLStringClass reqElem;
   OSINT32 optElem;
   ...
} ;
```

The constructors for this class (not shown) will set all bits in the mask to zero.

Repeating Elements

Elements within a sequence definition may be declared to be repeating by using the *minOccurs* and/or *maxOccurs* facets. In this case, a C or C++ list or array container type structure is used instead of a C/C++ element type definition. This container holds a series of objects of the element type.

If the C element type is a simple type and the maximum number of elements (maxOccurs) is less than or equal to 10,000, then an array type of the following form is used:

```
struct {
   OSUINT32 n;
   ElemType elem[maxOccurs];
}
```

In this definition, *n* is used to hold the count of element occurrences to be encoded (or that were decoded), and data holds the actual element data values.

If either of the above conditions is not true, a linked list type is used to hold a dynamic list of the data objects. This type is *OSRTDList* (run-time doubly linked list). It is defined in **rtxDList.h** as follows:

```
/* Doubly-linked list types */

typedef struct _OSRTDListNode {
   const void* data;
   struct _OSRTDListNode* next;
   struct _OSRTDListNode* prev;
} OSRTDListNode;

typedef struct _OSRTDList {
   OSUINT32 count;
   OSRTDListNode* head;
   OSRTDListNode* tail;
} OSRTDList;
```

There is a complete set of functions available for adding, deleting, and traversing lists of this type available in the run-time library. See the **rtxDList.h File Reference** section for documentation on these functions.

For C++, there is a corresponding class definition (*OSRTDListClass*) that extends the *OSRTDList* structure and contains constructors and methods for adding, removing, and finding items in the list.

The following example shows a sequence with two repeating elements. The first will cause an array type to be generated, the second, a list:

```
<xsd:complexType name="SeqWithArrayAndList">
   <xsd:sequence>
      <xsd:element name="anArray" type="xsd:integer"
         maxOccurs="10"/>
      <xsd:element name="aList" type="SomeOtherType"
         maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

The C type that is generated for this XSD type is as follows:

```
typedef struct SeqWithArrayAndList {
   struct {
      OSUINT32 n;
      OSINT32  elem[10];
   } anArray;
   /* List of SomeOtherType */
   OSRTDList aList;
} SeqWithOptElem;
```

Note that a comment is added to the generated C structure before the list declaration to indicate what type of objects the list is to contain.

In the case of C++, a constructor is added to the generated array structure to initialize the number of elements to zero. An inline class is generated for the list variable that extends the *OSRTDListClass* and adds methods to append items to the list and retrieve items from the list:

```
class SeqWithArrayAndList : public OSBaseType {
public:
   struct anArray_array {
      OSUINT32 n;
      OSINT32 elem[10];
      anArray_array() { n = 0; }
   } anArray;
   /* List of SomeOtherType */
```

```
   class aList_list : public OSRTDListClass {
    public:
      void append (const SomeOtherType* pdata) {
         OSRTDListClass::append ((const void*)pdata);
      }
      void append (SomeOtherType* pdata, OSBOOL ownMemory=FALSE) {
         OSRTDListClass::append ((void*)pdata, ownMemory);
      }
      const SomeOtherType* getItem (int idx) {
         return (const SomeOtherType*) OSRTDListClass::getItem (idx);
      }
   } aList;
   ...
} ;
```

Nested Types

It is possible to nest other XSD sequence or choice content model groups within another sequence. For example, it is possible to nest a sequence definition within another sequence definition as follows:

```
<xsd:complexType name="A">
   <xsd:sequence>
      <xsd:element name="x" type="xsd:string"/>
      <xsd:sequence minOccurs="0">
         <xsd:element name="y" type="xsd:integer"/>
        <xsd:element name="z" type="xsd:boolean"/>
      </xsd:sequence>
   </xsd:sequence>
</xsd:complexType>
```

In this example, the type has three elements – x , y, and z. A nested SEQUENCE is used with the y and z elements to indicate the group is optional.

The XBinder compiler recursively pulls all of the nested content model groups (i.e. the embedded sequence and choice definitions) out of the sequence type to form a series of types that contain only a single level of elements. The names of the newly formed types are of the form *BaseTypeName_LxS* where *BaseTypeName* is the name of the main type, *L* is the nesting level, and *S* is the sequential position of the element within the construct. For example, in the definition above, the following two C types are generated to model the XSD type above:

```
typedef struct A_1x2 {
   OSINT32  y;
   OSBOOL   z;
} A_1x2;

typedef struct A {
   struct {
      unsigned _seq2Present : 1;
   } m;
   const OSUTF8CHAR* x;
   A_1x2 _seq2;
} A;
```

In this case, XBinder created the type *A_1x2* to represent the inner sequence. It then added the *_seq2* element to the main C type using this type. This allows all of the elements in the inner sequence to be managed as a group in the generated code. This is particularly useful if the element group is optional or repeating.

The C++ generated code is similar except that items are in the form of class definitions instead of structures.

Any Element

An element in a sequence can be declared using the *xsd:any* keyword to indicate that an element of any type can be present in that position. An example of this type of construct is as follows:

```
<xsd:complexType name="SeqWithAny">
   <xsd:sequence>
      <xsd:element name="a" type="xsd:string"/>
      <xsd:any processContents="lax"/>
   </xsd:sequence>
</xsd:complexType>
```

In this case, the element *a* is followed by another element with any name and of any type. The *processContents=”lax”* attribute tells a schema processor to do lax validation processing on the element in this position – something that is of no concern to the XBinder compiler.

The generated C type definition for this type is as follows:

```
typedef struct SeqWithAny {
   OSXMLSTRING a;
   OSAnyElement _any;
} SeqWithAny;
```

C++ is similar except that the standard class pattern is used:

```
class SeqWithAny : public OSBaseType {
public:
   OSXMLStringClass a;
   OSAnyElementClass _any;
   ...
} ;
```

In this case, the compiler has inserted an *OSAnyElement* typed element to represent the any field. This contains UTF-8 character string fields for the element name and value.

An example code snippet that could be used to populate a C variable of this type for encoding is as follows:

```
const OSUTF8CHAR* anyData = (OSUTF8CHAR*)
   ”<anyData>this is test data</anyData>”;

SeqWithAny testSeq;

testSeq.a.value = (const OSUTF8CHAR*)”test string”;
```

```
testSeq._any.name  = (const OSUTF8CHAR*)"anyData";
testSeq._any.value =
       (const OSUTF8CHAR*)"this is test data";
...
```

ALL

The XSD ALL type <xsd:all> is a complex type consisting of a series of element definitions. These elements can reference other XSD types including other complex types. The main difference between this construct and a sequence is the elements can appear in any order (in a sequence, they must appear in the order they were declared).

The C type definition that is generated for an ALL is identical to that for a SEQUENCE above except for the addition of an order array. This array is added to control the order in which the elements are encoded. It appears as a special element within the generated C structure or C++ class as follows:

```
typedef struct TypeName {
   elements ...

   /* encoding control */
   OSUINT8 _order[n];
} TypeName;
```

The C initialization function for the type or C++ constructor will set this array to sequential order. A user can then alter this order if they would like to encode the elements in a different order. Also, on decode, the order the elements were received in is preserved in this array. That way, if the instance is reencoded, the elements will appear in the same order as in the original instance.

CHOICE

The XSD CHOICE type <xsd:choice> is a complex type consisting of a series of element definitions from which one may be selected to include in a message instance. It is converted into a C or C++ structured type containing an integer for the choice tag value (t) followed by a union (u) of all of the equivalent types that make up the CHOICE elements.

The tag value is simply a sequential number starting at one for each alternative in the CHOICE. For C, a #define constant is generated for each of these values. The format of this constant is T_*TypeName_elemName* where *TypeName* is the name of the XSD complexType and *elemName* is the name of the CHOICE alternative. For C++, an enumerated type is added to the class with enumerations of the form *T_elemName*.

The union of choice alternatives is made of the equivalent C or C++ type definition followed by the element name for each of the elements. The rules for element generation are essentially the same as was described for SEQUENCE above. Constructed types or elements that map to C structured types are

pulled out and temporary types are created. Names for elements that are not named (for an inline content group for example) names are automatically generated when needed.

The general mapping is as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:choice>
      <xsd:element name="elem1" type="Type1"/>
      <xsd:element name="elem2" type="Type2"/>
      ...
      <xsd:element name="elemN" type="TypeN"/>
   </xsd:choice>
</xsd:complexType>
```

Generated C code:

```
/* choice tag constants */
#define T_TypeName_elem1   1
#define T_TypeName_elem2   2
...
#define T_TypeName_elemN   N

typedef struct TypeName {
   OSUINT16 t;
   union {
      /* t = 1 */
      Type1 elem1;
      /* t = 2 */
      Type1 elem2;
      ...
      /* t = N */
      TypeN elemN;
   } u;
} TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
public:
   enum {
      T_elem1   1
      T_elem2   2
      ...
      T_elemN   N
   };
   OSUINT16 t;
   union {
      /* t = 1 */
      Type1 elem1;
      /* t = 2 */
      Type1 elem2;
      ...
      /* t = N */
      TypeN elemN;
   } u;
   ...
} ;
```

In most cases, the generated elements within the C union construct will be pointers to dynamic variables rather than inline static value references. The only exception to this rule is if the referenced type of the element is a simple, atomic type such as an integer. The reason for using pointers is to keep the size of the structures small (otherwise, it will be sized to fit the largest possible variable size even if that alternative is not being used) and to avoid problems with C++ constructor invocations if C++ types with constructors are referenced within the union.

Choice elements cannot be optional or repeating. Instead, the entire choice group itself could be labeled as optional or repeating within another construct (a sequence, for example). It is possible to nest other XSD sequence or choice content model groups within another choice. The rules for handling this are as described in the handling of nested types for sequence above.

Generated C++ Get/Set Methods

For C++, methods are generated to assist the user in getting, setting, or querying the choice construct variable. These methods are of the form *get_elemName*, *set_elemName*, and *is_elemName* where *elemName* would be replaced with the name of the element. The *get* method will return a pointer to the choice item only if it is the selected item; otherwise it will return null. The *is* method returns a boolean value of true if the element is the selected element of false otherwise. The *set* method sets the element to the given value and selects it by setting the *t* value.

C Example

The following is a common example of a choice construct with a nested sequence. This allows element a or element b or both elements to be present in an XML instance of the type:

```
<xsd:complexType name="AOrBOrBothType">
   <xsd:choice>
      <xsd:sequence>
         <xsd:element name="a" type="xsd:string"/>
         <xsd:element name="b" type="xsd:string"
            minOccurs="0"/>
      </xsd:sequence>
      <xsd:element name="b" type="xsd:string"/>
   </xsd:choice>
</xsd:complexType>
```

The generated C code for this type is as follows:

```
typedef struct AOrBOrBothType_1x1 {
   struct {
      unsigned bPresent : 1;
   } m;
   OSXMLSTRING a;
   OSXMLSTRING b;
} AOrBOrBothType_1x1;

/* choice tag constants */
#define T_AOrBOrBothType__seq1  1
#define T_AOrBOrBothType_b      2

typedef struct AOrBOrBothType {
   OSUINT16 t;
   union {
      /* t = 1 */
      AOrBOrBothType_1x1 *_seq1;
      /* t = 2 */
      OSXMLSTRING* b;
   } u;
} AOrBOrBothType;
```

In this case, XBinder created the type *AOrBOrBothType_1x1* to represent the inner sequence. It then added the *_seq1* element to the main C type using this type. A user populating the structure would use the *_seq1* element to specify element a or both and would use the *b* element to specify choice b.

C++ Example

The C++ code generated for the example schema above is as follows:

```
class AOrBOrBothType_1x1 : public OSBaseType {
public:
   struct {
      unsigned bPresent : 1;
   } m;
```

```
   OSXMLStringClass a;
   OSXMLStringClass b;

   ...
} ;

class AOrBOrBothType : public OSBaseType {
public:
   // tag constants
   enum {
      T__seq1 = 1,
      T_b = 2
   } ;
   OSUINT16 t;
   union {
      /* t = 1 */
      AOrBOrBothType_1x1 *_seq1;
      /* t = 2 */
      OSXMLStringClass *b;
   } u;

   ...

   inline AOrBOrBothType_1x1* get__seq1 () {
      return u._seq1;
   }
   inline OSBOOL is__seq1 () {
      return (t == T__seq1);
   }
   void set__seq1 (const AOrBOrBothType_1x1& value);

   inline OSXMLStringClass* get_b () {
      return u.b;
   }
   inline OSBOOL is_b () {
      return (t == T_b);
   }
   void set_b (const OSXMLStringClass& value);
} ;
```

This shows the generated get/set methods as well as the generated member variables in the class. If the user wanted to set the class to the nested sequence value, the *set__seq1* method could be used. If the user wanted to determine if the *b* element was selected in the class and then get the value, the following code snippet could be used (*object* is assumed to be an instance of the `AOrBOrBothType` class):

```
if (object.is_b()) {
   OSXMLStringClass* value = object.get_b();
}
```

Attributes

The XSD ComplexType syntax allows for the specification of attributes that can be added to the start element tag for an XML instance of the type. XBinder handles attributes the same way it does normal elements. They are added as typed fields to the C struct or C++ class definition for the complex type.

The general mapping is as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:group>
      ...
   </xsd:group>
   <xsd:attribute name="attr1" type="Type1"/>
   <xsd:attribute name="attr2" type="Type2"/>
   ...
   <xsd:attribute name="attrN" type="TypeN"/>
</xsd:complexType>
```

Generated C code:

```
typedef struct TypeName {
   group type definition..

   /* attributes */
   Type1  attr1;
   Type2  attr2;
   ...
   TypeN  attrN;
} TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
public:
   group type definition..

   /* attributes */
   Type1  attr1;
   Type2  attr2;
   ...
   TypeN  attrN;

   ...
} ;
```

In the definition above, *group* can be any content model group type (sequence, all, choice, or group). It is an optional item – it is possible to omit the group completely to form a type with empty content that only contains attributes.

Attributes are optional by default and are handled in the same way as optional elements. A bit is added to the optional bit mask at the beginning of the structure with the name *attr*Present (where *attr* is the attribute name). This bit is set to true if the attribute is to be added to a message instance or false if it is to be omitted.

Attributes that contain a default or fixed value are handled by modifying the initialization and/or encode/decode functions. A default value will be handled by adding a statement to the initialization function for the type to set the attribute to the default value. The user can later override this value in order to change it in a message instance.

A fixed value also causes a statement to be added to the generated C initialization function or C++ constructor to set the attribute to the given fixed value. Unlike default value, it is not possible to override this value. The generated encode function contains hard-coded logic to ignore the value in the type variable and encode the fixed value. On the decode side, the incoming value will be checked to make sure it equals the fixed value. If not, an error will be flagged and the value set to the fixed value in the typed variable.

ComplexContent

The XSD ComplexContent type <xsd:complexContent> is used to create a modified version of a base type through extension or restriction mechanisms. It is similar in concept to creating derived types in Java or C++. ComplexContent is handled differently depending on whether C or C++ code is being generated. For C, the type is converted into a C structured type containing a base element (base) and, optionally, an extensions element (ext) and additional attribute elements. The extension element will only appear if the extension mechanism is used to add additional elements to an existing content model group (sequence, all, or choice).

For C++, the inheritance mechanism is used to extend the base class. Extension elements will appear directly in the derived class.

Element Extension

The general mapping for *complexContent* with an extension element group is as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:complexContent>
      <xsd:extension base="BaseType">
         <xsd:group>
            <xsd:element name="elem1" type="Type1"/>
            <xsd:element name="elem2" type="Type2"/>
            ...
            <xsd:element name="elemN" type="TypeN"/>
         </xsd:group>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

Generated C code:

```
typedef struct TypeName_1x2 {
   group type definition..
} TypeName_1x2;

typedef struct TypeName {
   BaseType base;
   TypeName_1x2 ext;
} TypeName;
```

Generated C++ code:

```
class TypeName : public BaseType {
public:
   group type definition..
} ;
```

Notes:

1. *group* in the extension group definition above can be any content model group type (sequence, all, choice, or group).
2. In the case of C, the extension group is pulled out to form the temporary type (*TypeName*_1x2). The internals of this type depend on the content group type.
3. In the case of C++, the *BaseType* is extended to form a derived class. The contents of the

extension group are then added directly to the body of the derived class.

Example: Extension Elements
The following complexContent type contains a choice of two additional elements that were not defined in the base type (*ProductType*):

```
<xsd:complexType name="ShirtType">
   <xsd:complexContent>
      <xsd:extension base="ProductType">
         <xsd:choice>
            <xsd:element name="size" type="SizeType"/>
            <xsd:element name="color" type="ColorType"/>
         </xsd:choice>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

The following are the C typedefs that are generated for this definition:

```
#define T_ShirtType_1x2_size   1
#define T_ShirtType_1x2_color  2

typedef struct EXTERN ShirtType_1x2 {
   OSUINT16 t;
   union {
      /* t = 1 */
      SizeType size;
      /* t = 2 */
      ColorType color;
   } u;
} ShirtType_1x2;

typedef struct EXTERN ShirtType {
   ProductType base;
   ShirtType_1x2 ext;
} ShirtType;
```

In the case of C++, a new class is derived from the *ProductType* base class:

```
class EXTERN ShirtType : public ProductType {
public:
   // tag constants
   enum {
      T_size = 1,
      T_color = 2
   } ;
   OSUINT16 t;
   union {
```

```
        /* t = 1 */
        SizeType *size;
        /* t = 2 */
        ColorType *color;
    } u;
    ...
} ;
```

In this case, the contents of the extension are generated directly in the derived class. A temporary type is not created.

Attribute Extension

It is possible to extend a base type to contain additional attributes. In this case, the additional attribute definitions are added to the structure generated for the *complexContent* type.

The general mapping for complexContent with extension attributes is as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:complexContent>
      <xsd:extension base="BaseType">
         <xsd:attribute name="attr1" type="Type1"/>
         <xsd:attribute name="attr2" type="Type2"/>
         ...
         <xsd:attribute name="attrN" type="TypeN"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

Generated C code:

```
typedef struct TypeName {
   BaseType base;

   /* attributes */
   Type1 attr1;
   Type2 attr2;
   ...
   TypeN attrN;
} TypeName;
```

Generated C++ code:

```
class TypeName : public BaseType {
   /* attributes */
   Type1 attr1;
   Type2 attr2;
   ...
   TypeN attrN;

   ...
} ;
```

In this case, the attributes are handled the same as they were in the *Attributes* section above. If any are optional, an optional bit mask is added at the beginning of the *complexContent* structure. Logic to handle fixed and default values is added to the initialization and encode/decode functions.

Restrictions

It is possible to restrict elements and attributes in an existing content model group by using the *restriction* element. For either elements or attributes, it is possible to exclude optional items from the derived content model. It is also possible to restrict wildcards (*any* or *anyAttribute*) to contain values of a given type. It is also possible to further restrict facets such as *minOccurs* and *maxOccurs* to specify a narrower range than was defined in the base type.

All of these restriction types are handled in the C or C++ code generated to initialize, encode, and/or decode the restricted types. The type used to hold the data is not altered – it is the original base type specified in the *complexContent* definition. In order to maintain compatibility with the extension logic show above, the same general form of the type is used. In the case of C, this is a struct with a *base* element. For C++, it is a new class derived from the base class.

The general mapping is as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:complexContent>
      <xsd:restriction base="BaseType">
         ...
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```

Generated C code:

```
typedef struct TypeName {
   BaseType base;
} TypeName;
```

Generated C++ code:

```
class TypeName : public BaseType {
    ...
} ;
```

No matter what the content is, the generated type always consists of only a single element of the base type.

SimpleContent

The XSD *simpleContent* type <xsd:simpleContent> is used to create a modified version of a base type through extension or restriction mechanisms. It is similar in concept to creating derived types in Java or C++. The *simpleContent* type is converted into a C structured type containing a base element (base) and, optionally, additional attributes specified through the extension mechanism.

For C++, a derived class is created that extends the class generated for the base type. The exception to this rule is if the type being extended is a simple atomic type such as an integer. In this case, the class generated is derived from the default base class type (*OSBaseType*) and a member variable called *value* is added for the simple built-in type.

Extensions

The only purpose of simple content extensions is to add attributes to an existing base type. In this case, a C struct is generated containing a base element (base) of the type being extended and attribute elements for the additional attributes. For C++, the attributes are added to the derived class.

The general mapping is as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:simpleContent>
      <xsd:extension base="BaseType">
         <xsd:attribute name="attr1" type="Type1"/>
         <xsd:attribute name="attr2" type="Type2"/>
         ...
         <xsd:attribute name="attrN" type="TypeN"/>
      </xsd:extension>
   </xsd:simpleContent>
</xsd:complexType>
```

Generated C code:

```
typedef struct TypeName {
   BaseType base;

   /* attributes */
   Type1 attr1;
   Type2 attr2;
   ...
   TypeN attrN;
} TypeName;
```

Generated C++ code:

```
class TypeName : public BaseType {
   /* attributes */
   Type1 attr1;
   Type2 attr2;
   ...
   TypeN attrN;

   ...
} ;
```

In this case, *BaseType* can only be a built-in XSD simple type or a type that references a simple type.

Restrictions

Simple content restrictions are used to restrict the simple content and/or attributes of a complex type. As was the case for *complexContent* restrictions described above, this does not result in the generation of a new type. The generated type is a C struct containing a single element named *base* or a C++ class that extends the base type. All restriction processing to make sure the content and attributes are within the defined parameters is done in the generated initialization, encode, and decode functions.

The general mapping is as follows:

XSD type:

```
<xsd:complexType name="TypeName">
   <xsd:simpleContent>
      <xsd:restriction base="BaseType">
         ...
      </xsd:restriction>
   </xsd:simpleContent>
</xsd:complexType>
```

Generated C code:

```
typedef struct TypeName {
   BaseType base;
} TypeName;
```

Generated C++ code:

```
class TypeName : public BaseType {
   ...
} ;
```

Group

The XSD *group* type <xsd:group> is used to create a reusable content model group. This is similar in concept to the creation of a standalone type and is handled in the C or C++ language mapping as such. A group declaration is translated into a C type or C++ class definition. This type definition is then used in places where the group is referenced.

The general mapping is as follows:

XSD type:

```
<xsd:group name="TypeName">
   XSD content group definition ..
</xsd:group>
```

Generated C code:

```
typedef struct TypeName {
   group type definition..
} TypeName;
```

Generated C++ code:

```
class TypeName : public OSBaseType {
   group type definition..
} ;
```

Configuration File

The default bindings of source schema components to a C/C++ types as presented above may not meet the requirements of all applications. In such cases, the default bindings can be customized by using a configuration file. This is sometimes refered to as a *binding schema* in similar products. A configuration file contains binding declarations which are specified by a *binding language*, the syntax and semantics of which are defined in this section.

Binding Language

The binding language is an XML based language that defines constructs referred to as *binding declarations*. A binding declaration can be used to customize the default binding between an XML schema component and its C/C++ representation.

The schema for binding declarations is defined in the namespace `http://www.obj-sys.com/XBConfig`.

Binding Declaration

The configuration file format enables customized binding without requiring modification of the source schema. The schema component to which the binding declaration applies must be identified explicitly. Minimally, a configuration file is of the following format.

```
<bindings version=”1.0">
   <schemaBindings schemaLocation = "xsd:anyURI">
      <nodeBindings node = "xsd:string">*
          <node bindings declaration>
      <nodeBindings>
   </schemaBindings>
</bindings>
```

The *schemaBindings* node has the attribute *schemaLocation* to refer to a schema. The *nodeBindings* node has the attribute *node* to construct a reference to a node within the schema. The binding declaration is applied to this node by the binding compiler. The attribute values are interpreted as follows:

schemaLocation: A URI reference to an XML schema document.

node: An XPath 1.0[1] expression that identifies the schema node within a schema with which to associate binding declarations.

1.*XML Path Language (XPath) Version 1.0 (*http://www.w3.org/TR/xpath)

An example of a configuration file can be found in the section “Configuration File Example”.

Version Attribute

The normative binding schema specifies a global `version` attribute. This is used to identify the version of the binding declarations. For example, a future version of this specification may use the version attribute to specify backward compatibility. For this version of the specification, the `version` must always `"1.0"`. If any other version is specified, the configuration file will be skipped.

The `version` attribute must be specified in the root element `<bindings>` in the configuration file:

```
<bindings version="1.0" ... />
```

Configuration File Language Overview

A binding declaration customizes the default binding of a schema element to a C/C++ representation. The binding declaration defines one or more customization values each of which customizes a part of C/C++ representation.

Scope

When a customization value is defined in a binding declaration, it is associated with a *scope*. A scope of a customization value is the set of schema elements to which it applies.

The defined scopes are as follows:

- **global scope:** A customization value defined in `<bindings>` has *global scope*. A global scope covers all the schema elements in the source schema and (recursively) any schemas that are included or imported by the source schema.
- **schema scope:** A customization value defined in `<schemaBindings>` has *schema scope*. A schema scope covers all the schema elements in the target namespace of a schema.
- **node scope:** A customization value defined in `<nodeBindings>` has *node scope.* A node scope covers all schema elements that reference the type definition, the global declaration or the local declaration.

A customization value defined in one scope is inherited for use in a binding declaration covered by another scope as shown by the following inheritance hierarchy:

- A schema element in schema scope inherits a customization value defined in global scope.
- A schema element in node scope inherits a customization value defined in schema or global scope.

Likewise, a customization value defined in one scope can override a customization value inherited from another scope as shown below:

- A value in schema scope overrides a value inherited from global scope.
- A value in node scope overrides a value inherited from schema scope or global scope.

Global <bindings> Declaration

The customization values in the "<bindings>" binding declaration have global scope. These affect all elements within all schemas defined in the compilation project.

Usage

```
<bindings version="1.0">
       [<prefix>xs:token</prefix>]
       [<schemaBindings>. . .</schemaBindings>]
       ...
</bindings>
```

The following attributes are defined for the <bindings> node:

version:	See the section "Version Attribute" above for details.

The following customization elements may be defined within the global scope:

prefix:	This is used to specify a prefix that is prepended to all XML names including type names and global element names to form C/C++ type and variables names. It should be a legal C/C++ identifier.
schemaBindings:	This is used to identify individual schemas for schema scope binding declarations (see Section "<schemaBindings> Declaration"). It can be specified multiple times, but once per schema.

<schemaBindings> Declaration

The customization values in <schemaBindings> binding declarations have schema scope. These apply to all elements within the referenced XML schema document.

Usage

```
<schemaBindings schemaLocation="xs:anyURI">
       [<prefix>xs:token</prefix>]
       [<sourceFile>xs:anyURI</sourceFile>]
       [<nodeBindings>. . .<nodeBindings>]
       ...
</schemaBindings>
```

The following attributes are defined for <schemaBindings> node:

schemaLocation: A URI reference to a schema. It can be either a schema file path or a URL as it is used in *import* or *include* statements. In the latter case, *sourceFile* should be specified to map the schema URL to an actual schema file. XBinder does not have the capability to automatically reference schemas remotely; therefore, any imported or included schemas must have been downloaded in advance and be present on the user's computer.

The following customization values are defined in schema scope:

prefix: This is used to specify a prefix that is prepended to all XML names including type names and global element names to form C/C++ type and variables names. It should be a legal C/C++ identifier.

sourceFile: The actual schema file path. XBinder does not have the capability to automatically reference schemas remotely; therefore, any imported or included schemas must have been downloaded in advance and be present on the user's computer. This element is used to map a schema URL to a file on the local system.

nodeBindings: Node scope binding declarations (see Section "<nodeBindings> Declaration"). This element can be specified multiple times, but only once per definition.

<nodeBindings> Declaration

The customization values in the <nodeBindings> binding declaration have node scope. These refer to individual type or element definitions within a schema. It is also possible to reference local elements within complex types for customization.

Usage

```
<nodeBindings node="xs:string">
       [<prefix>xs:token</prefix>]
       [<array [maxSize="xs:nonNegativeInteger"/>]]
       [<isBigInteger/>]
       [<isDynamic/>]
       [<ctype> string | numeric </ctype>]
       [<nodeBindings>. . .<nodeBindings>]
       ...
</nodeBindings>
```

The following attributes are defined for <nodeBindings> node:

node: An XPath 1.0 expression that identifies the schema node within the referenced schema with which to associate binding declarations.

The following customization values are defined in node scope:

prefix: This is used to specify a prefix that is prepended to all XML names including type names and global element names to form C/C++ type and variables names. It should be a legal C/C++ identifier.

array: This specifies that an array should be used instead of a linked list for repeated elements. The *maxSize* attribute specifies the maximum size of the array. The default value if not specified is 100.

isBigInteger: This specifies that this type will be used to store an integer larger than the C or C++ *int* type on the given system (normally 32 bits) or even the 64-bit integer type if supported (*long long*, or *__int64*). A C UTF-8 string type (OSUTF8CHAR*) will be used to hold a textual representation of the value.

This qualifier can be applied to either an integer or complex type. In the latter case, all integer elements within the complex type are flagged as big integers.

isDynamic: This indicates that dynamic storage (i.e., pointers) should be used everywhere within the generated types where use could result in lower memory consumption.

ctype This is used to specify a specific C type be used in place of the default definition generated by the XBinder compiler. Currently, this can be used only for date and time types. XBinder currently generates string types for *date*, *time* and *dateTime* types. It is possible to use a built-in C structure for these types instead of strings. In this case *ctype* should contain the value *numeric* for the appropriate nodes.

nodeBindings: Nested *nodeBindings* declarations to allow more accurate references to enclosed elements such as local elements inside groups (*sequence*, *all*, *choice*, *group*, etc).

Configuration File Example

The following is an example of a configuration file for a framework consisting of two schemas:

```
<bindings version="1.0">
  <schemaBindings
   schemaLocation=
      "http://www.oasis-open.org/committees/ebxml-msg/schema/envelope.xsd">
     <sourceFile>C:\XBinder\dev\xsd\SOAP\envelope.xsd</sourceFile>

     <prefix>SOAP_</prefix>

     <nodeBindings node="//xsd:element[@name='myGlobalElem']">
        <prefix>GE_</prefix>
```

```
         <ctype>numeric</ctype>
      </nodeBindings>
   </schemaBindings>

   <schemaBindings schemaLocation="core-schema.xsd">

      <nodeBindings node="//xsd:element[@name='Manifest']">
         <prefix>Dsig</prefix>
      </nodeBindings>

      <nodeBindings node="//xsd:complexType[@name='Dss_Parms']">
         <nodeBindings node=".//xsd:element[@name='p']">
            <isBigInteger/>
         </nodeBindings>
         <nodeBindings node=".//xsd:element[@name='q']">
            <isBigInteger/>
         </nodeBindings>
         <nodeBindings node=".//xsd:element[@name='g']">
            <isBigInteger/>
         </nodeBindings>
      </nodeBindings>
   </schemaBindings>
</bindings>
```

Generated C Encode/Decode Functions

XBinder generates C encode functions to transform data from a populated C structure into an XML message instance. This process is known as *marshalling* or *serialization* in similar products. It generates decode functions to parse data from an XML message instance and store the data in a variable of the generated C structure. This is known as *unmarshalling* or *deserialization* in other applications.

The following sections describe procedures for using the XBinder generated functions encode and decode XML data.

Preparing C Data Variables for Encoding

Before data can be encoded, the C structure for a given data type must be populated. In most cases, this involves the simple assignment of data items to the elements within the structure. In some cases, however, dynamic memory pointers are involved. It is necessary to know how dynamic memory works in the run-time in order to populate these fields.

Dynamic Memory Management

The XBinder run-time uses several different memory management schemes in order to provide flexibility in handling different types of application's memory requirements. The following are the schemes available in this release:

- Standard
- Nibble-Allocation
- Custom

The standard memory allocation algorithm simply maps XBinder run-time function calls directly to the C standard run-time memory functions *malloc, free,* and *realloc*. (Note: in some environments such as some embedded RTOS's. the *realloc* function may not be available. The built-in run-time provides a custom implementation of this function using *malloc* and *free* for these cases). The advantages of standard management are simplicity and space-optimization. The primary disadvantage is performance — frequent calls to *malloc* and *free* can be detrimental to performance.

The nibble-allocation algorithm is designed to improve performance in the case where frequent requests for small amounts of memory are made. This is typical of many data-binding applications due to unconstrained types being declared within the schema. The way this algorithm works is large blocks of memory are allocated up front and then split up to provide memory for smaller allocation requests. This reduces the number of calls required to the C *malloc* and *free* functions.

Finally, it is possible for a user to build in his or her own custom management by implementing the functions defined within the standard XBinder run-time memory management interface.

The main entry points to the memory management system for users are the *rtxMemAlloc, rtxMemFree, rtxMemFreePtr,* and *rtxMemRealloc* functions. These are the functions that should always be used for doing memory management — not the built-in C memory functions.

Populating Generated Structure Variables for Encoding

Prior to calling a compiler generated encode function, a variable of the type generated by the compiler must be populated. This is normally a straightforward procedure — just plug in the values to be encoded into the defined fields. However, things get more complicated when more complex, constructed structures are involved. These structures frequently contain pointer types which means memory management issues must be dealt with.

There are two alternatives for managing memory for these types:

1. Allocate the variables on the stack and plug the address of the variables into the pointer fields,
2. Use the **rtxMemAlloc** and **rtxMemFreePtr** run-time library functions or their associated macros.

Allocating the variables on the stack is an easy way to get temporary memory and have it released when it is no longer being used. But one has to be careful when using additional functions to populate these types of variables. A common mistake is the storage of the addresses of automatic variables in the pointer fields of a passed-in structure. An example of this error is as follows (assume A, B, and C are other structured types):

```
typedef struct {
   A* a;
   B* b;
   C* c;
} Parent;

void fillParent (Parent* parent)
{
   A aa;
   B bb;
   C cc;

   /* logic to populate aa, bb, and cc */
   ...

   parent->a = &aa;
   parent->b = &bb;
   parent->c = &cc;
}

main ()
{
   Parent parent;

   fillParent (&parent);

   encodeParent (&parent);   /* error: pointers in
                                parent reference memory
```

```
                                        that is out of scope */
    …
}
```

In this example, the automatic variables aa, bb, and cc go out of scope when the *fillParent* function exits. Yet the parent structure is still holding pointers to the now out of scope variables (this type of error is commonly known as “dangling pointers”).

Using dynamic memory for the variables solves this problem. The *rtxMemAlloc* call can be used to allocate memory for each of the dynamic fields. The *rtxMemFree* function is used to release all memory held within the context at once. This is typically done after the populated variable is encoded. The *rtxMemFreePtr* function can be used to free an individual memory element.

It is recommended that these functions be used instead of the standard C memory management functions so that if the underlying memory management scheme is changed (see *Dynamic Memory Management* above) all memory handling within the application is changed to the new scheme without any recoding being required.

Accessing Encoded Message Components

After a message has been encoded, the user must obtain the start address and length of the message in order to do further operations with it. Before a message can be encoded, the user must describe the buffer the message is to be encoded into by specifying a message buffer start address and size. There are three different types of message buffers that can be described:

1. **static**: this is a fixed-size byte array into which the message is encoded
2. **dynamic**: in this case, the encoder manages the allocation of memory to hold the encoded message
3. **stream**: in this case, the encoder writes then encoded data directly to an output stream

The static buffer case is generally the better performing case because no dynamic memory allocations are required. However, the user must know in advance the amount of memory that will be required to hold an encoded message. There is no fixed formula to determine this number. XML encoding involves the additions of tags and attributes and other decorations to the provided data that will increase the size beyond the initial size of the populated data structures. The way to find out is either by trial-and-error (an error will be signaled if the provided buffer is not large enough) or by using a very large buffer in comparison to the size of the data.

In the dynamic case, the buffer description passed into the encoder is a null buffer pointer and zero size. This tells the encoder that it is to allocate memory for the message. It does this by allocating an initial amount of memory and when this is used up, it expands the buffer by reallocating. This can be an expensive operation in terms of performance – especially if a large number of reallocations are required. For this reason, run-time helper functions are provided that allow the user to control the size increment of buffer expansions. See the *C/C++ Run-Time Library Reference Manual* for a description of these functions.

In either case, after a message is encoded, it is necessary to get the start address and length of the message. In the static buffer case for XML, the start address of the message is simply the start address

of the buffer. But in the dynamic case, a function call is required to get the start address of the message after encoding is complete. The *rtXmlGetEncBufPtr* is provided for this purpose.

A stream message buffer can also be used for XML encoding. Special encode functions are generated for this purpose when the *–stream* command line switch is used. In this case, data is written directly to an output device such as a file or socket.

Generated XML Encode Functions

Standard XML C encode functions are generated when the *-xml* switch is specified on the command line (the other option is stream-oriented XML encoder functions which are generated when both *–xml* and *–stream* are specified). For each generated C type, a C XML encode function is generated. This function will convert a populated C variable of the given type into an XML encoded message.

Generated C Function Format and Calling Parameters

Generated encode functions are written to a .c file with a name of the following format:

```
<xsdFileName>Enc.c
```

where <*xsdFileName*> is the base name of the XSD file being parsed. For example, if code is being generated for file *x.xsd*, encode functions for each type and global element defined in the specification will be written to *xEnc.c*.

The format of the name of each generated XML encode function is as follows:

```
[<ns>]XmlET_<typeName>
```

where `<typeName>` is the name of the C type for which the function is being generated and `<ns>` is an optional namespace setting that can be used to disambiguate element names from multiple sources (note: this should not be confused with XML namespaces which are different).

The calling sequence for each encode function is as follows:

```
status = <encodeFunc>
   (OSCTXT* pctxt, <name>[*] value, const OSUTF8CHAR* elemName,
    const OSUTF8CHAR* nsPrefix);
```

In this definition, <encodeFunc> denotes the encode function name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of encode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her or her program.

The `value` argument contains the value to be encoded or holds a pointer to the value to be encoded. This variable is of the type generated from the XSD type. The object is passed by value if it is an atomic

XSD simple type such as boolean, integer, etc.. It is passed using a pointer reference if it is a structured type value (in this case, the name will be *pvalue* instead of *value*). Check the generated function prototype in the header file to determine how this argument is to be passed for a given function.

The *elemName* argument is used to pass an XML element name for the type. This name is what is included in the <name> </name> brackets used to delimit an XML item. If a null pointer (0) is passed in for this argument, then no name wrapper is added to encoded XML item.

The *nsPrefix* argument is used to specify a namespace prefix. If this value is null or empry, no prefix is added to element name. If a prefix is given, a qualified element name of the form *nsPrefix:elemName* is generated.

The function result variable `stat` returns the status of the encode operation. Status code 0 (zero) indicates the function was successful. A negative value indicates encoding failed. Return status values are defined in the "rtxErrCodes.h" include file. The error text and a stack trace can be displayed using the *rtxErrPrint* function.

Generated C Encode Functions for Global Elements

For each global element defined within an XSD specification, a special encode function is generated. This is identical to the encode function for XSD types described above except that the name is formed using the element name instead of the type name and the function does not contain an *elemName* argument. In this case, *elemName* is set to the name specified in the XSD global element definition. The encode function name prefix in this case is “XmlE_” instead of “XmlET_” in order to avoid name clashes when types and global elements have the same name.

These functions are the normal entry points when encoding complete XML message instances. All of the sample programs use a global element definition to define the top-level message to be encoded for a particular application.

Procedure for Calling a Generated C Encode Function

The encode function generated for an XSD global element definition is the normal entry point for encoding an XML document. The general procedure for calling a global element encode function is as follows:

1. Prepare a context variable for encoding
2. Initialize an encode message buffer or stream to receive the encoded XML data
3. Populate the data variable with data to be encoded
4. Call the appropriate compiler-generated encode function to encode the message
5. If a message buffer was used, get the start pointer and length of the encoded message

Before a C XML encode function can be called; the user must initialize a context variable. This is a variable of type OSCTXT. This variable holds all of the working data used during the encoding of a message. The context variable is declared as a normal automatic variable within the top-level calling function. **It must be initialized before use**. This can be accomplished by using the *rtXmlInitContext* function:

```
OSCTXT ctxt;       /* context variable */

if (rtXmlInitContext (&ctxt) != 0) {
   /* initialization failed, could be a license problem */
   printf ("context initialization failed (check license)\n");
   return -1;
}
```

The next step is to specify an encode buffer or stream into which the message will be encoded. This is accomplished by calling the *rtXmlSetEncBufPtr* run-time function (for a message buffer) or one of the *rtxStream* functions to create an output stream. If a message buffer is to be used, the user has the option to either pass the address of a buffer and size allocated in his or her program (referred to as a static buffer), or set these parameters to zero and let the encode function manage the buffer memory allocation (referred to as a dynamic buffer). Better performance can normally be attained by using a static buffer because this eliminates the high-overhead operation of allocating and reallocating memory.

After initializing the context and populating a variable of the structure to be encoded, an encode function can be called to encode the message. If the return status indicates success, the run-time library function *rtXmlGetEncBufPtr* can be called to obtain the start address of the encoded message. In the static case, this is simply the start address of the static buffer. In the dynamic case, this function will return the pointer to the allocated memory buffer. The memory allocated for a dynamic buffer will be freed when either the context is freed (*rtxFreeContext*) or all memory associated with the context is released (*rtxMemFree*) or the buffer memory is explicity released (*rtxMemFreePtr*).

In the stream case, the pointer to the encoded message generally cannot be obtained since the message has already been written to the stream. The only thing necessary to do in this case is to close the stream after encoding is complete. Use the *rtxStreamClose* function which should be called before the *rtxFreeContext* function.

A program fragment that could be used to encode an employee record is as follows:

```
#include "employee.h"

#define MAXMSGLEN 1024

int main (int argc, char** argv)
{
   PersonnelRecord employee;
   OSCTXT          ctxt;
   OSOCTET         msgbuf[MAXMSGLEN];
   int             i, stat;
   const char* filename = "message.xml";

   /* Init context */

   stat = rtXmlInitContext (&ctxt);
```

```
if (0 != stat) {
   printf ("Context initialization failed.\n");
   rtxErrPrint (&ctxt);
   return stat;
}

/* Populate structure of generated type */

Init_PersonnelRecord (&ctxt, &employee);

... logic to populate structure here ...

/* Encode */

stat = rtXmlSetEncBufPtr (&ctxt, msgbuf, sizeof(msgbuf));

if (0 == stat)
   stat = XmlE_personnelRecord (&ctxt, &employee);

if (0 == stat) {
   printf ("encoded XML message:\n");
   printf (msgbuf);
   printf ("\n");
}
else {
   printf ("Encoding failed\n");
   rtxErrPrint (&ctxt);
   return stat;
}

... logic to process encoded message (write to file, etc.) ...

rtxFreeContext (&ctxt);
```

This example used a static message buffer. The encoded XML text will reside in the msgbuf message buffer when the procedure complete.

A program fragment that could be used to encode an employee record into a stream is as follows:

```
#include "rtxsrc/rtxStreamFile.h"
#include "employee.h"

#define MAXMSGLEN 1024

int main (int argc, char** argv)
{
   PersonnelRecord employee;
   OSCTXT        ctxt;
```

```
   OSOCTET             msgbuf[MAXMSGLEN];
   int                 i, stat;
   const char* filename = "message.xml";

   /* Init context */

   stat = rtXmlInitContext (&ctxt);
   if (0 != stat) {
      printf ("Context initialization failed.\n");
      rtxErrPrint (&ctxt);
      return stat;
   }

   /* Populate structure of generated type */

   Init_PersonnelRecord (&ctxt, &employee);

   ... logic to populate structure here ...

   /* Encode directly to output stream */

   stat = rtxStreamFileCreateWriter (&ctxt, filename);
   if (0 != stat) {
      printf ("Stream initialization failed.\n");
      rtxErrPrint (&ctxt);
      return stat;
   }

   stat = XmlE_personnelRecord (&ctxt, &employee);

   if (0 == stat) {
      printf ("encoded XML message:\n");
      printf (msgbuf);
      printf ("\n");
   }
   else {
      printf ("Encoding failed\n");
      rtxErrPrint (&ctxt);
      return stat;
   }
   rtxStreamClose (&ctxt);
   rtxFreeContext (&ctxt);
   return 0;
}
```

Generated XML Decode Functions

The code generated to decode XML messages uses off-the-shelf XML parser software to parse the XML documents to be decoded. This software contains a common interface known as the *Simple API for XML (or SAX)* that is a de-facto standard that is supported by most parsers. XBinder generates an implementation of the content handler interface defined by this standard. This implementation receives the parsed XML data and uses it to populate the structures generated by the compiler.

The default XML parser used is the EXPAT parser (http://www.expat.org). This is a lightweight, open-source parser that was implemented in C. XBinder generates C SAX handler functions that are called from the SAX interface of this framework to decode XML data into the generated typed data structures. The interface was designed to be generic so that other XML parsers could be easily substituted. An interface to the GNOME LibXML2 parser (http://xmlsoft.org) is also available. Interfacing to other parsers requires only building an abstraction layer to map the common interface to the vendor's interface.

XBinder generates code to implement the following functions defined in the SAX content handler interface:

```
startElement

characters

endElement
```

The interface defines other methods that can be implemented as well, but these are sufficient to decode XML encoded data.

Generated C Function Format and Calling Parameters

Generated decode functions are written to a .c file with a name of the following format:

```
<xsdFileName>Dec.c
```

where <*xsdFileName*> is the base name of the XSD file being parsed. For example, if code is being generated for file *x.xsd*, decode functions for each global element defined in the specification will be written to *xDec.c*. In addition, the SAX handler functions that are invoked by the underlying XML parser software are written to a file with a name of the following format:

```
<xsdFileName>SAX.c
```

The format of the name of each generated C XML decode function is as follows:

```
[<ns>]XmlD_<elemName>
```

where <elemName> is the name of the XSD global element for which the function is being generated and <ns> is an optional namespace setting that can be used to disambiguate element names from multiple sources (note: this should not be confused with XML namespaces which are different).

The calling sequence for each decode function is as follows:

```
status = <decodeFunc> (OSCTXT* pctxt, <typeName>* pvalue);
```

In this definition, <decodeFunc> is the name of the decode function described above and <typeName> is the name of the generated C type definition for the global element.

The pctxt argument is used to hold a context pointer to keep track of decode parameters. This is a basic "handle" variable that is used to make the function reentrant so that it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type

declared somewhere in his or her program. The variable must be initialized using the *rtSaxInitContext* run-time function before use.

The `pvalue` argument is a pointer to a variable to hold the decoded result. This variable is of the type generated for the XSD type of the global element. The decode function will automatically allocate dynamic memory for variable length fields within the structure. This memory is tracked within the context structure and is released when the context structure is freed.

The function result variable `status` returns the status of the decode operation. Status code zero indicates the function was successful. A negative value indicates decoding failed. Return status values are defined in the "rtxErrCodes.h" include file. The reason text and a stack trace can be displayed using the *rtxErrPrint* function.

Procedure for Calling C Decode Functions

There are four steps to calling a compiler-generated C XML decode function:

1. Prepare a context variable for decoding;
2. Open a stream;
3. Call the appropriate compiler-generated decode function to decode the message;
4. Free the context after use of the decoded data is complete to free allocated memory structures

Before a C XML decode function can be called; the user must initialize a context variable. This is a variable of type OSCTXT. This variable holds all of the working data used during the decoding of a message. The context variable is declared as a normal automatic variable within the top-level calling function. **It must be initialized before use**. This can be accomplished by using the *rtXmlInitContext* function:

```
OSCTXT ctxt;        // context variable

if (rtXmlInitContext (&ctxt) != 0) {
   /* initialization failed, could be a license problem */
   printf ("context initialization failed (check license)\n");
   return -1;
}
```

The next step is to create a stream object within the context. This object is an abstraction of the input device from which the XML data will be read and parsed. Calling one of the following functions initializes the stream:

- *rtxStreamFileOpen*
- *rtxStreamFileAttach*
- *rtxStreamSocketAttach*
- *rtxStreamMemoryCreate*
- *rtxStreamMemoryAttach*

The *flags* parameter of these functions should be set to the OSRTSTRMF_INPUT constant value to indicate an input stream is being created.

A decode function can then be called to decode the message. If the return status indicates success (0), then the message will have been decoded into the given XSD type variable. The decode function may automatically allocate dynamic memory to hold variable length items during the course of decoding. This memory will be tracked in the context structure, so the programmer does not need to worry about freeing it. It will be released when the context is freed.

The final step of the procedure is to close the stream and free the context block. The function to close the stream is *rtxStreamClose*. The function to free the context is *rtxFreeContext*.

A program fragment that could be used to decode an employee record is as follows:

```
#include employee.h        /* include file generated by XBinder */

main ()
{
    int     stat;
    OSCTXT  ctxt;
    PersonnelRecord employee;
    const char* filename = "message.xml";

    /* Step 1: Init context structure */

    if (rtXmlInitContext (&ctxt) != 0) return -1;

    /* Step 2: Open a stream */

    stat = rtxStreamFileOpen (&ctxt, filename, OSRTSTRMF_INPUT);
    if (stat != 0) {
        rtxErrPrint (&ctxt);
        return -1;
    }

    /* Step 3: decode the record */

    stat = XmlD_personnelRecord (&ctxt, &employee);
    if (stat == 0) {
        if (trace) {
            printf ("Decode of PersonnelRecord was successful\n");
            printf ("Decoded record:\n");
            Print_PersonnelRecord ("Employee", &employee);
        }
    }
    else {
        printf ("decode of PersonnelRecord failed\n");
        rtxErrPrint (&ctxt);
```

```
        rtxStreamClose (&ctxt);
        return -1;
    }

    /* Step 4: Close the stream and free the context. */

    rtxStreamClose (&ctxt);
    rtxFreeContext (&ctxt);

    return 0;
}
```

Generated Print Functions

The *–print* option causes print functions to be generated. These functions can be used to print the contents of variables of generated types.

The generated print functions are written to a .c file with a name of the following format:

```
<xsdFileName>Print.c
```

where <*xsdFileName*> is the base name of the XSD file being parsed. For example, if code is being generated for file *x.xsd* and –print is specified, print functions will be written to *xPrint.c*.

The format of the name of each generated print function is as follows:

```
[<ns>]Print_<typeName>
```

where <typeName> is the name of the XSD type for which the function is being generated and <ns> is an optional namespace setting that can be used to disambiguate names from multiple sources (note: this should not be confused with XML namespaces which are different). Note that print routines are generated for each type within a specification making it possible to print the contents of any typed variable (some generated functions are only generated for global elements).

The calling sequence for each generated print function is as follows:

```
<printFunc> (const char* name, <typeName>* pvalue)
```

In this definition, <printFunc> denotes the formatted function name defined above.

The *name* argument is used to hold the top-level name of the variable being printed. It is typically set to the same name as the *pvalue* argument in quotes (for example, to print an employee record, a call to ‘Print_PersonnelRecord (“employee”, &employee) might be used).

The *pvalue* argument is used to pass a pointer to a variable of the item to be printed.

The code snippet in the section entitled *Procedure for Calling C Decode Functions* contains an example of calling a generated print function. If a successful status is returned from calling the decode function, the contents of the decoded variable are printed:

```
stat = XmlD_personnelRecord (&ctxt, &employee);
if (stat == 0) {
   if (trace) {
      printf ("Decode of PersonnelRecord was successful\n");
      printf ("Decoded record:\n");
      Print_PersonnelRecord ("employee", &employee);
   }
}
```

Generated Test Functions

The *–genTest* option causes test functions to be generated. These functions can be used to populate variables of generated types with random test data. They have two main purposes:

1. To allow testing of the application code with a wide-variety of test data, and
2. To provide a code template for users to use to write code to populate variables

The second item is quite useful to users because generated data types can become very complex as the schemas become more complex. It is sometimes difficult to figure out how to navigate all of the lists and pointers. Using *–genTest* can provide code that simply has to be modified to accomplish the population of a data variable with any type of data.

The generated test functions are written to a .c file with a name of the following format:

```
<xsdFileName>Test.c
```

where <*xsdFileName*> is the base name of the XSD file being parsed. For example, if code is being generated for file *x.xsd* and *-test* is specified, test functions will be written to *xTest.c.*

The format of the name of each generated test function is as follows:

```
[<ns>]Test_<elemName>
```

where <elemName> is the name of the XSD global element for which the function is being generated and <ns> is an optional namespace setting that can be used to disambiguate names from multiple sources (note: this should not be confused with XML namespaces which are different). Note that test routines are generated only for global elements within a specification.

The calling sequence for each generated test function is as follows:

```
<typeName>* pvalue = <testFunc> (OSCTXT* pctxt)
```

In this definition, <testFunc> denotes the formatted function name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of dynamic memory allocation parameters. This is a basic "handle" variable that is used to make the function reentrant so that it can be used in an asynchronous or threaded application. The user is required to supply a pointer to a variable of this type declared somewhere in his or her program. The variable must be initialized using either the *rtxInitContext* or *rtXmlInitContext* run-time function before use.

The `pvalue` argument is a pointer to hold the populated data variable. This variable is of the type generated for the XSD type of the global element. The test function will automatically allocate dynamic memory using the run-time memory management for the main variable as well as variable length fields within the structure. This memory is tracked within the context structure and is released when the context structure is freed.

Other Generated Functions

In addition to the functions described above, the following other types of functions are generated as part of the code generation process:

- Initialization function
- Memory free function
- Utility functions based on data type

All of these common functions are applicable to both encode and decode operations and, as such, are written to the common base .c file. The format of the name of this file is as follows:

```
<xsdFileName>.c
```

where <*xsdFileName*> is the base name of the XSD file being parsed. For example, if code is being generated for file *x.xsd* and *-test* is specified, then the common functions will be written to *x.c*.

Initialization functions are for initializing a variable of a generated data type before use. This includes setting all fields that contain default or fixed values to the value specified in the schema. All other fields are set to zero. The format of an initialization functions name is as follows:

```
[<ns>]Init_<typeName>
```

where `<typeName>` is the name of the XSD type for which the function is being generated and `<ns>` is an optional namespace setting that can be used to disambiguate names from multiple sources (note: this should not be confused with XML namespaces which are different).

The calling sequence for each generated initialization function is as follows:

```
<initFunc> (OSCTXT* pctxt, <typeName>* pvalue)
```

In this definition, <initFunc> denotes the formatted function name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of global parameters. The `pvalue` argument is a pointer to a variable of the type to be initialized.

Memory free functions allow memory associated with a specific typed variable instance to be freed. Their use is not required to free memory — the run-time function *rtxMemFree* can be called directly with a context variable to free all memory associated with a context. There are applications, however, where freeing the memory contents of a specific variable are desirable.

Memory free functions are not generated for all types — only those that contain fields that use dynamic memory. This includes types that contain elements or attributes that reference other types that use dynamic memory. The format of a generated memory free function is as follows:

```
[<ns>]Free_<typeName>
```

where `<typeName>` is the name of the XSD type for which the function is being generated and `<ns>` is an optional namespace setting that can be used to disambiguate names from multiple sources (note: this should not be confused with XML namespaces which are different).

The calling sequence for each generated initialization function is as follows:

```
<freeFunc> (OSCTXT* pctxt, <typeName>* pvalue)
```

In this definition, <freeFunc> denotes the formatted function name defined above.

The `pctxt` argument is used to hold a context pointer to keep track of global parameters. The `pvalue` argument is a pointer to a variable of the type containing the memory to be freed.

Other utility or "helper" functions are type specific and designed to help the user work with the generated code. The following utility function are generated for the following types:

- Enumerated: <typeName>_toString and <typeName>_toEnum functions are generated to allow conversion from enumerated to string and vice-versa.
- List or array: repeating fields that result in the generation of an *OSRTDList* variable contain a <typeName>_Append function. This is used to append an instance of a typed variable to the list variable.

Generated Makefile

The *–genmake* option causes a makefile to be generated to assist in the C compilation of all of the generated C source files. This makefile contains a rule to invoke XBinder to regenerate the .c and .h files if the XSD source file changes. It also contains rules to compile all of the C source files. Header file dependencies are generated for all the C source files.

Two basic types of makefiles are generated:

1. A GNU compatible makefile. This makefile is compatible with the GNU make utility which is suitable for compiling code on Linux and many UNIX operating systems, and
2. A Microsoft Visual Studio compatible makefile. This makefile is compatible with the Microsoft Visual Studio *nmake* utility.

A GNU compatible makefile is produced by default, the Microsoft compatible file is produced when the *–w32* command line option is specified in addition to *–genmake.*

Both of these makefile types rely on definitions in the *platform.mk* make include file. This file contains parameters specific to different compiler and linker utilities available on different platforms. Typically, all the needs to be done to port to a different platform is to adjust the parameters in this file.

Generated C++ Class Methods

XBinder generates C++ classes for all types and global elements defined in an XML schema. Each class generated for a global element contains the main encode or decode methods required to serialize data to and from XML class member variables. Methods generated for types are used by the global element methods to accomplish the complete encoding or decoding of an XML document for the given element. Methods are also generated to help users construct and populate the generated type classes.

The following sections describe procedures for using the XBinder generated C++ framework to encode and decode XML data.

Preparing C++ Objects for Encoding

Before data can be encoded, an instance of the C++ class for a given data type must be populated. In most cases, this involves the simple assignment of data items to the elements within the structure and is very similar to the C case presented earlier. In some cases, however, dynamic memory pointers are involved. It is necessary to know how dynamic memory works in the run-time in order to populate these fields.

Dynamic Memory Management

In the case of C++, dynamic memory management is handled by the *new* and *delete* operators. In most cases, handling memory involves the common action of making sure to delete any pointer that was allocated with *new* when done with it. However, the generated C++ classes contain the concept of memory ownership which makes it possible for a user to assign ownership of memory to the container object being populated. This makes freeing the memory much easier. Instead of having to track each individual item, the class does it for you. Therefore, when the main container class is deleted at the end of use, all memory objects within that class that have had ownership assigned to the container are deleted as well.

In cases where memory ownership can be assigned to the container class, the class may contain assignment methods that contain a boolean *ownMemory* argument. If this argument is set to true, the class will assume the object being transferred has been allocated dynamically using the *new* operator and will invoke *delete* on the object from within its destructor. An example of a method with an

ownMemory argument is the *append* method generated for a repeating element class (this is taken from the *cpp/sample/simpleArray* sample program):

```
class SimpleArray : public OSBaseType {
public:
   /* List of OSXMLStringClass */
   class item_list : public OSRTDListClass {
    public:
      void append (const OSXMLStringClass* pdata) {
         OSRTDListClass::append ((const void*)pdata);
      }
      void append (OSXMLStringClass* pdata, OSBOOL ownMemory=FALSE) {
         OSRTDListClass::append ((void*)pdata, ownMemory);
      }
      const OSXMLStringClass* getItem (int idx) {
          return (const OSXMLStringClass*)
             OSRTDListClass::getItem (idx);
      }
   } item;
   ...
```

In this case, the item list is a list of strings. By using the second form of the append method to add an item to this list, memory ownership can be transferred to the list container. Then when the list is deleted, all of the objects with transferred ownership will be deleted as well.

There is also a *setOwnMemory* method defined in the *OSBaseType* base class. This method can also be used to transfer ownership of dynamic memory to the container class.

When an XML instance is decoded, the decoder automatically transfers memory ownership to the container objects. It is therefore not necessary for the user to worry about freeing memory for any of the items within a returned class instance. Deleting the object is all that is necessary to free the memory of all of the items within.

Populating Generated Class Instances for Encoding

Prior to calling a compiler generated encode function, an instance of the class generated by the compiler must be populated. All member variables within the generated classes are declared to be public, it is therefore possible to do direct assignments to populate the variables. Sometimes that variables are more complicated, however, and special assignment methods are generated to assist the use in populating the variables. These special cases are described below.

Atomic Simple Types

Atomic simple types include boolean, integer, double, decimal, and types derived from these base types. The classes generated for these types include a *value* member that can be assigned to directly. They also include a parameterized constructor that allows assignment through construction and an assignment operator that makes it possible to do assignment directly through the '=' sign without having to use the *value* member. This makes assignment compatible with the C case.

For example, the following simple integer type declaration:

```
<xsd:simpleType name="EmployeeNumber">
   <xsd:restriction base="xsd:integer"/>
</xsd:simpleType>
```

causes a class with the following constructors and assignment operator to be generated:

```
class EmployeeNumber : public OSBaseType {
public:
   OSINT32 value;
   EmployeeNumber ();
   EmployeeNumber (OSINT32 value);
   EmployeeNumber& operator= (OSINT32 value);
```

This makes it possible to assign an employee number in any of the following ways:

```
EmployeeNumber empno (33);
```

or

```
EmployeeNumber empno;
empno.value = 33;
```

or

```
EmployeeNumber empno;
empno = 33;
```

Character String Types

Character string types are derived from the built-in base class *OSXMLStringClass*. This class is in turn derived from the *OSXMLSTRING* C struct type which contains a *value* member character pointer variable.

An example of the constructors and assignment operators generated for a character string type is shown for the following definition:

```
<xsd:simpleType name="Date">
   <xsd:restriction base="xsd:string"/>
</xsd:simpleType>
```

The C++ constructors and assignment operators generated for this definition are as follows:

```
class EXTERN Date : public OSXMLStringClass {
public:
   Date ();
   Date (const OSUTF8CHAR* value);
   Date (const char* value);
   Date& operator= (const OSUTF8CHAR* value);
   Date& operator= (const char* value);
```

Note that constructors and operators are available that allow string to be specified as either standard C character string or as UTF-8 strings.

It is also possible to transfer ownership of memory for a string to the string class. This is done by using the *setOwnMemory* method defined in the *OSBaseType* base class. For example, the following series of calls will cause the allocated string variable to be deleted when the encapsulating *Date* class instance goes out of scope:

```
const char* myStringVar = new char [40];
strcpy (myStringVar, "22 Feb 2004");
Date myDate (myStringVar);
myDate.setOwnMemory (TRUE);
```

Enumerated Type

Classes generated for enumerated types contains *setValue* methods that allow the contained *value* member variable to be set. Two overloaded forms of this method are present: one that takes the enumerated identifier as a number and one that takes a string representation of the value. A method is also generated to allow the value to be retrieved as a number (*getValue*) or as a string (*toString*).

The signatures for these methods is as follows:

```
inline OSUINT16 getValue () const { return value; }
int setValue (OSUINT16 enumval);
int setValue (const OSUTF8CHAR* strval);
const OSUTF8CHAR* toString () const;
```

Binary String Types

Binary string are used to represent XSD *hexBinary*, *base64Binary*, and *any* data types. A dynamic binary string type (i.e. one that is not constraint by a length facet) is derived from the *OSDynOctStrClass* base class. This class allows value assignment through constructors and *copyValue* and *setValue* methods.

The constructors allow a binary string to be specified using data pointer and number of octets arguments. One form of the constructor contains an *ownMemory* argument that makes it possible to transfer the memory management of a dynamic string over to the binary string class. The *copyValue* method is used make a copy of the given string and assign it to the class. The *setValue* method sets the base member variables directly to the given arguments.

Content Group Types
Classes generated for <xsd:sequence> or <xsd:all> complex types contain a series of public element declarations for each of the elements in the declaration. These are populated using either direct assignment or the methods available in the element type classes. If atomic types are used for elements, the primitive type itself is used in the generated class, not a class derived from the type. For example, if a sequence contains an element declared to be an <xsd:integer>, the OSINT32 type is used for the member variable.

Classes generated for the <xsd:choice> complex type will contain methods to get, set, or query each of the choice selection elements. The format of these methods is *get_<name>*, *set_<name>*, or *is_<name>* where *<name>* would be replaced with the actual element name as defined in the schema.

Content model groups that repeat (i.e. have a *maxOccurs* facet with a value greater than one) cause a class to be generated that is derived from the *OSObjListClass* base class. This class contains *append* methods for adding elements to the list. An example of what the append method looks like is given in the 'Dynamic Memory Management' section above. This method allows memory ownership for the object being added to the list to be tranferred to the list object. Note that this can be done on a per-element basis making it possible to mix and match dynamic and static element declarations in a given list.

Message Buffer or Stream Classes

Message buffer or stream classes are used to describe the source from which a message is being decoded or the target to which a message is being encoded. The base interface for these classes is **OSMessageBufferIF**. Classes for message buffers or streams specific to encoding or decoding and for different encoding rules (for example, XML) are derived from this base class. An instance of one of these derived classes along with an instance of the class generated for a particular XSD type are needed to encode or decode a message.

Message buffers for encoding can be either static or dynamic. A static buffer is simply a byte array in memory. It is generally the better performing case because no dynamic memory allocations are required. However, the user must know in advance the amount of memory that will be required to hold an encoded message. There is no fixed formula to determine this number. XML encoding involves the additions of tags and attributes and other decorations to the provided data that will increase the size beyond the initial size of the populated data structures. The way to find out is either by trial-and-error (an error will be signaled if the provided buffer is not large enough) or by using a very large buffer in comparison to the size of the data. A static buffer is described using a message buffer class object by passing the byte array address and size to the constructor.

A dynamic buffer is specified by using the default constructor. This tells the encoder that it is to allocate memory for the message. It does this by allocating an initial amount of memory and when this is used up, it expands the buffer by reallocating. This can be an expensive operation in terms of performance, especially if a large number of reallocations are required. Special methods are provided that allow the initial and incremental allocation sizes to be tuned for better performance. See the run-time class reference guide for further details on this.

In either case, after a message is encoded, it is necessary to get the start address and length of the message. In the static buffer case for XML, the start address of the message is simply the start address

of the buffer. But in the dynamic case, a function call is required to get the start address of the message after encoding is complete. The *getMsgPtr* method is provided for this purpose.

Generated XML C++ Encode Methods

An XML C++ encode method named *encodeXML* is added to each class generated for an XSD type when the *-xml* switch is specified on the command line. This method will convert a populated instance of the class into an encoded XML message.

Generated Method Format and Calling Parameters

Generated encode method implementations are written to a .cpp file with a name of the following format:

```
<xsdFileName>Enc.cpp
```

where <*xsdFileName*> is the base name of the XSD file being parsed. For example, if code is being generated for file *x.xsd*, encode method implementations for each type and global element defined in the specification will be written to *xEnc.cpp*.

The format of the name of each generated XML encode method is *encodeXML*. The calling sequence is as follows:

```
status = <object>.encodeXML
   (OSCTXT* pctxt, const OSUTF8CHAR* elemName,
    const OSUTF8CHAR* nsPrefix);
```

In this definition, <object> denotes an object instance of the generated class.

The `pctxt` argument is used to hold a context pointer to keep track of encode parameters. This is a basic "handle" variable that is used to make the function reentrant so it can be used in an asynchronous or threaded application. For C++, the context is obtained from the message buffer or control class object (they share a common context). The *getCtxtPtr* method is available in either of these classes for obtaining a pointer to the context variable.

The *elemName* argument is used to pass an XML element name for the type. This name is what is included in the <name> </name> brackets used to delimit an XML item. If a null pointer (0) is passed in for this argument, then no name wrapper is added to encoded XML item.

The *nsPrefix* argument is used to specify a namespace prefix. If this value is null or empry, no prefix is added to element name. If a prefix is given, a qualified element name of the form *nsPrefix:elemName* is generated.

The method result variable returns the status of the encode operation. Status code 0 (zero) indicates success. A negative value indicates encoding failed. Return status values are defined in the "rtxErrCodes.h" include file. The error text and a stack trace can be displayed using the message buffer *printErrorInfo* method.

Generated C++ Encode Methods for Global Elements

For each global element defined within an XSD specification, a control class definition is generated. Within this control class are encode methods that can be used to generate a complete XML document. This is the typical entry point an application program would use to serialize elements into XML.

There are two different encode methods that can be used. There is a method named *encode* that is defined in the *OSXSDGlobalElement* base class. This method encodes the data in the class instance into the default message buffer or stream that was associated with the control class when it was created. The second method is *encodeTo* which allows a message buffer or stream to be specified as an argument. The data in the associated class instance is serialized out to this buffer or stream.

Procedure for Using the Generated C++ Encode Method

The procedure to encode an XML message using the generated C++ encode method is as follows:

1. If exceptions were not disabled via the _NO_EXCEPTIONS C++ compiler switch, open a 'try' block to use for message encoding.
2. Create an instance of the generated type class to hold the data to be encoded.
3. Create an instance of an output stream or message buffer object to which the encoded XML message will be written.
4. Create an instance of the generated global element control class to link the generated type class instance with the message buffer or stream instance.
5. Populate the generated type class instance created in step 2 with data to be encoded.
6. Invoke the control class *encode* method.
7. If encoding was successful (indicated by return status equal to zero), the start-of-message pointer can be obtained by calling the message buffer *getMsgPtr* method (note: this assumes encoding using a message buffer was done, if a stream was used, the message has already been written to the target).
8. If encoding failed, the message buffer or stream *printErrorInfo* method can be invoked to print the reason for failure.
9. Close the 'try' block opened in step 1 and, at a minimum, catch exceptions of type *OSRTLException*. The exception class *getStatus* method can be called to get the run-time error status code.

A program fragment that uses this procedure to encode an employee record is as follows:

```
#include "rtxmlsrc/rtXmlCppMsgBuf.h"
#include "employee.h"

int main (int argc, char** argv)
{
   int           i, stat;
   const char*  filename = "message.xml";
```

```
OSBOOL       trace = TRUE, verbose = FALSE;
const OSOCTET* msgptr;

// Create buffer and control objects

try {
   PersonnelRecord value;
   OSXMLEncodeBuffer buffer;
   personnelRecord_CC pdu (buffer, value);

   if (verbose)
      rtxSetDiag (pdu.getCtxtPtr(), 1);

   // Populate structure of generated type

   ... logic to populate structure here ...

   // Encode

   stat = pdu.encode();

   if (0 == stat) {
      if (trace) {
         msgptr = buffer.getMsgPtr();
         printf ("encoded XML message:\n");
         printf ((const char*)msgptr);
         printf ("\n");
      }
   }
   else {
      printf ("Encoding failed\n");
      buffer.printErrorInfo();
      return stat;
   }

   // Write the encoded message out to the output file

   buffer.write (filename);
}
catch (OSRTLException e) {
    printf
    ("Run-time exception occurred, status = %d\n", e.getStatus());
}
```

Generated XML C++ Decode Methods

The code generated to decode XML messages uses off-the-shelf XML parser software to parse the XML documents to be decoded. This software contains a common interface known as the *Simple API for XML (or SAX)* that is a de-facto standard that is supported by most parsers. XBinder generates an implementation of the content handler interface defined by this standard. This implementation receives the parsed XML data and uses it to populate the structures generated by the compiler.

The default XML parser used is the EXPAT parser (http://www.expat.org). This is a lightweight, open-source parser that was implemented in C. XBinder generates C++ SAX handler classes within the generated type class for a given XML schema type. The methods within these classes are called from the SAX interface of the XML parser framework to decode XML data into the generated typed data structures. The interface was designed to be generic so that other XML parsers could be easily substituted. An interface to the GNOME LibXML2 parser (http://xmlsoft.org) is also available. Interfacing to other parsers requires only building an abstraction layer to map the common interface to the vendor's interface.

XBinder generates code to implement the following SAX content handler methods:

```
startElement

characters

endElement
```

The interface defines other methods that can be implemented as well, but these are sufficient to decode XML encoded data.

Generated C++ Method Format and Calling Parameters

Generated decode and SAX method implementations are written to a .cpp file with a name of the following format:

```
<xsdFileName>Dec.cpp
```

where <*xsdFileName*> is the base name of the XSD file being parsed. For example, if code is being generated for file *x.xsd*, decode functions for each global element defined in the specification will be written to *xDec.cpp*.

The main method for decoding an XML document that corresponds to an XSD global element is the *decode* or *decodeFrom* methods. The *decode* method exists in the *OSXSDGlobalElement* base class as decodes a message from the message buffer or stream associated with the class to the XSD type object instance. A *decodeFrom* method is generated for each global element that is not referenced by any other types. This method takes as an argument a message buffer of stream reference. The method reads the XML message from this buffer or stream and decodes it into the XSD type object instance associated with the control class instance.

Procedure for Calling C++ Decode Methods

The procedure to invoke a C++ decode method is as follows:

1. If exceptions were not disabled via the _NO_EXCEPTIONS C++ compiler switch, open a 'try' block to use for message decoding.
2. Create an instance of the generated type class into which the XML message data is to be decoded.
3. Create an instance of an input stream or message buffer object from which the XML message to be decoded will be read.
4. Create an instance of the generated global element control class to link the generated type class instance with the message buffer or input stream instance.
5. Invoke the control class *decode* method.
6. If decoding was successful (indicated by return status equal to zero), the decoded data will now be available for use in the generated type variable. The generated *print* method can be called at this time to examine the contents of the data structure.
7. If decoding failed, the message buffer or stream *printErrorInfo* method can be invoked to print the reason for failure.
8. Close the 'try' block opened in step 1 and, at a minimum, catch exceptions of type *OSRTLException*. The exception class *getStatus* method can be called to get the run-time error status code.

A program fragment that could be used to decode an employee record is as follows:

```
#include "employee.h"
#include "rtxmlsrc/rtXmlCppMsgBuf.h"

int main (int argc, char** argv)
{
   int              i, stat;
   const char*  filename = "message.xml";
   OSBOOL       trace = TRUE, verbose = FALSE;

   try {
      OSFileInputStream in (filename);
      OSXMLDecodeBuffer decodeBuffer (in);
      PersonnelRecord value;
      personnelRecord_CC personnelRecord (decodeBuffer, value);

      if (verbose)
         rtxSetDiag (personnelRecord.getCtxtPtr(), 1);

      // Decode

      stat = personnelRecord.decode();

      if (0 == stat) {
         if (trace) {
            printf ("decoded XML message:\n");
            personnelRecord.print ("personnelRecord");
            printf ("\n");
         }
      }
      else {
         printf ("Decoding failed\n");
         decodeBuffer.printErrorInfo();
         return stat;
      }
   }
   catch (OSSAXException& e) {
      printf ("SAX exception occurred, status = %d, msg = %s\n",
         e.getStatus(), e.getMessage());
   }
   catch (OSRTLException e) {
       printf
       ("Run-time exception occurred, status = %d\n", e.getStatus());
   }
```

XBinder C Runtime Library

The XBinder C Runtime Library contains the low-level functions that are assembled by the XBinder compiler to accomplish the encoding and decoding of XML messages. This library also contains common functions for memory management, stream operations, linked list handling, and character text conversions.

The following libraries make up the XBinder run-time:

- **osysrtxml** – contains low-level functions to implement the encoding and decoding of standard XML messages for the various XML schema types.
- **osysrt** – contains common low-level functions for memory management, etc.

There are several variations of the C XML and common run-time library files for Windows. The following table summarizes what options were used to build each of these variations:

Library Files	Description
osysrt_a.lib *osysrtxml_a.lib*	Static single-threaded libraries. These were built with the –ML option. These are not thread-safe. However, they provide the smallest footprint of the different libraries.
osysrt.lib *osysrtxml.lib*	DLL libraries. These are used to link against the DLL versions of the run-time libraries (*osysrt.dll*, etc.)
osysrtmt_a.lib *osysrtxmlmt_a.lib*	Static multi-threaded libraries. These libraries were built with the –MT option. They should be used if your application contains threads and you wish to link with the static libraries (note: the DLL's are also thread-safe).
osysrtmd_a.lib *osysrtxmlmd_a.lib*	DLL-ready multi-threaded libraries. These libraries were built with the –MD option. They allow linking additional object modules in with the run-time modules to produce larger DLL's.

For dynamic linking on UNIX/Linux, a shared object version of each run-time library is included in the lib subdirectory. This file typically has the extension .so (for shared object) or .sl (for shared library). See the documentation for your UNIX compiler to determine how to link using these files (it varies for different types of UNIX systems).

A version of the libraries is available that contains run-time source code making it possible for the end-user to build customized versions that are further optimized or that use other non-standard compiler options.

XML Run-time Library Functions

The XML low-level C encode/decode functions are used to encoded and decode an XML instance of an XML schema typed variable. These functions are identified by their prefixes: rtXmlEnc for encode, rtXmlDec for decode, rtSax for SAX helper functions, and rtXml for utility functions. The following sections describe these functions.

XML C Encode Functions

The XML C low-level encode functions handle the XML encoding of simple XML schema data types. Calls to these functions are assembled in the C source code generated by the XBinder compiler to accomplish the encoding of complex structures. These functions are also directly callable from within a user's application program if the need to accomplish a low level encoding function exists.

The procedure to call a low-level encode function is the same as the procedure to call a compiler generated encode function described earlier. It is as follows:

1. The *rtXmlInitContext* function must first be called to initialize a context block structure.
2. Either a stream must be set up or a memory buffer specified to receive the encoded message. To set up a stream, one of the *rtxStream* functions must be called. To set up a memory buffer, the *rtXmlSetEncBufPtr* function is used.
3. The *rtXmlEncStartDocument* function is called to add the standard XML document header to the buffer.
4. Encode functions are then invoked to encode the XML data types.
5. The *rtXmlEncEndDocument* function is then called to complete the encoding.

If a stream was used, the encoded message will have been written to the output stream. If a memory buffer was used, the result of the encoding will start at the beginning of the buffer, or, if a dynamic buffer was used, can be obtained by calling *rtXmlGetEncMsgPtr*. The length of the encoded component can be obtained by calling the C standard library *strlen* function. The encoded stream is a standard UTF-8 null-terminated text string.

For example, the following code fragment could be used to encode a document with a single, boolean value.

```
OSOCTET buf[1000];
OSBOOL  boolValue = TRUE;  /* true */
OSCTXT  ctxt;
int msglen, stat;

rtXmlInitContext (&ctxt);
rtXmlSetEncBufPtr (&ctxt, buf, sizeof(buf));

stat = rtXmlEncStartDocument (&ctxt);
if (stat != 0) {
```

```
    rtxErrPrint (&ctxt);
    exit (-1);
}

stat = rtXmlEncBool (&ctxt, boolValue, "boolValue");
if (stat != 0) {
    rtxErrPrint (&ctxt);
    exit (-1);
}

stat = rtXmlEncEndDocument (&ctxt);
if (stat != 0) {
    rtxErrPrint (&ctxt);
    exit (-1);
}

msglen = strlen (buf);
```

The msglen variable now contains the length (in octets) of the encoded boolean value and the encoded data starts at the beginning of buf.

A complete reference to all of the built-in C XML encode functions is available in the *XBinder C/C++ Runtime Reference Manual*.

XML C Decode Functions

XML C low-level decode functions handle the transformation of XML simple type content into C type program variable data. Calls to these functions are assembled in the C SAX handler source code generated by the XBinder compiler to decode complex XML schema-based messages. They are normally invoked from within a generated SAX *endElement* function to parse buffered data that was collected in a SAX *characters* function.

These functions are also directly callable from within a user's application program if the need to decode a primitive data item exists. Note, however, that the low-level C decode functions only decode the data within XML tagged fields, not the tags themselves. Thus, it is not possible to directly decode a string such as `<myInt>10</myInt>` by calling these functions. It would only be possible to convert "10" into a C integer value. To parse the entire XML string, it would be necessary to invoke the XML parser with registered SAX handlers that could parse all of the items.

A complete reference to all of the built-in C XML encode functions is available in the *XBinder C/C++ Runtime Reference Manual*.

C Common Runtime Library

The C common run-time library contains common functions used by the XML C low-level encode/decode functions. These functions could be common to other applications as well. They are identified by their *rtx* prefixes. The following general categories of functions are provided:

- Context management functions
- Memory management functions
- Memory buffer management functions
- Diagnostic trace functions
- Error formatting and print functions
- Formatted printing functions
- Linked list utility functions
- Character string conversion utility functions

The following sections describe these functions.

Common Include Files

The common runtime library includes the following common header files:

- rtxSysTypes.h common type definitions
- rtxCommon.h common function prototypes
- rtxContext.h run-time context (OSCTXT) structure definition
- rtxErrCodes.h error code contants

rtxSysTypes.h

The *rtxSysTypes.h* header file contains all of the simple type definitions for character string data, integers, floating point types, binary types, etc. The following common type definitions are included:

```
typedef void            OSVoid;
typedef void*           OSVoidPtr;
typedef unsigned char   OSBOOL;
typedef signed char     OSINT8;
typedef unsigned char   OSUINT8;
typedef short           OSINT16;
typedef unsigned short  OSUINT16;
typedef int             OSINT32;
typedef unsigned int    OSUINT32;
typedef OSUINT8        OSOCTET;
typedef OSUINT8         OSUTF8CHAR;  /* UTF-8 character   */
typedef OSUINT16        OSUNICHAR;   /* Unicode character */
typedef OSUINT32        OS32BITCHAR;
typedef double          OSREAL;

/* binary string type */

typedef struct OSDynOctStr {
```

```
   OSUINT32     numocts;
   const OSOCTET* data;
} OSDynOctStr;

/* XML string */

typedef struct OSXMLSTRING {
   OSBOOL cdata;                /* encode as a CDATA section */
   const OSUTF8CHAR* value;
} OSXMLSTRING;
```

rtxCommon.h

The *rtxCommon.h* file contains all of the common function prototypes. This file also contains macro definitions for inline code that is used to improve performance. These macros are used in both the common runtime code and also added to generated code by the XBinder compiler.

All of the runtime functions defined within this file are documented in the common runtime function sections below.

rtxContext.h

The *rtxContext.h* file contains the definition of the runtime context block structure – OSCTXT. This structure is used in practically all runtime function calls. It provides a common work area for the functions to preserve state information needed in the encoding or decoding of messages. The definition is as follows:

```
typedef struct OSCTXT {       /* run-time context block  */
   OSBuffer     buffer;       /* data buffer             */
   OSBufSave    savedInfo;    /* saved buffer info       */
   OSErrInfo    errInfo;      /* run-time error info     */
   OSMemHeap*   pMemHeap;     /* memory heap             */
   OSUINT32     initCode;     /* initialization code     */
   OSUINT16     flags;        /* flag bits               */
   OSOCTET      level;        /* nesting level           */
   OSOCTET      state;        /* encode/decode process state */
   struct OSRTStream* pStream; /* input/output stream block */
   OSVoidPtrpAppInfo;         /* Application specific info */
} OSCTXT;
```

A brief description of what each of these fields is used for is as follows:

- **buffer –** This contains information on the memory buffer to which a message is being encoded or holds an in-memory copy of a message being decoded. This may also be used as a temporary buffer if stream-based encoding or decoding is being done. The *OSBuffer* structure contains a data pointer to the memory buffer itself as well as the current byte index and bit offset.

- **savedInfo –** This is used to save a copy of the buffer information in places where an alternate buffer may need to be substituted. Sometimes it is possible to save the buffer information on the stack, but there are instances where this variable is needed for that purpose.
- **errInfo –** This is a stack containing information on errors that were encountered in formatting or parsing a message. The *OSErrInfo* type contains a status code, run-time error parameter stack, and an error location stack that is used to save source file/line number information so that a trace stack can be provided in the error message print routine.
- **pMemHeap –** This is a pointer to the memory heap managed by the runtime software. This tracks all of the memory usage used in the encoding or decoding of a specific message. See the section on memory management functions for more details on this.
- **level –** This variable contains the current nesting level of the data within the current XML or other message type that is being encoded or decoded.
- **state –** This is used to hold the current processing state when parsing an XML message. It is primarily used in SAX parsing to determine if the last element parsed was a start tag, data, or end tag.
- **pStream –** This is used to hold information about an input or output stream if data is being directly read from or written to a stream. See the section on stream handling functions for more details on this.
- **pAppInfo –** This is reserved for application specific information.

Context Management Functions

Context management functions handle the allocation, initialization, and destruction of context variables (variables of type **OSCTXT**). These variables hold all of the working data used during the process of encoding or decoding a message. The context provides thread safe operation by isolating what would otherwise be global variables within this structure. The context variable is passed from function to function as a message is encoded or decoded and maintains state information on the encoding or decoding process.

The main functions in this group that a user should be aware of are the following:

- **rtxInitContext** - This is the first function that must be called to initialize a context block structure before it can be used in subsequent as an argument in subsequent function calls. This function initializes all internal variables within the context to their start values.
- **rtxInitContextBuffer** - This function associates a memory buffer with a context. This memory buffer can be used to as the target for encoding an XML message or as the source to read from for decoding a message.
- **rtxFreeContext** - This function is used to free all working memory held within the context. All memory alloocation done using memory management functions are tracked within the context. All of this memory can be released at once by calling this function. It should be the last function called when all work using a particular context variable is complete.

Other functions exist for doing further operations on contexts including copying and setting data within. A full description of all context management functions can be found in the *XBinder C/C++ Runtime Reference Manual*.

Memory Management Functions

Memory management functions handle the allocation and deallocation of dynamic memory. These functions form an abstraction layer above the standard C memory management functions *malloc, free,* and *realloc*. This block of functions can be replaced by the user with custom code to implement a different memory management scheme. For example, an embedded system application might want to use a fixed-sized static block from which to allocate.

The built-in memory management logic implements a nibble-allocation memory management algorithm that provides superior performance to calling *malloc* and *free* directly. This algorithm causes memory blocks to be allocated up front in larger sizes and then subsequently split up when future allocation requests are received. These blocks can be reset and reused in applications that are constantly allocating and freeing memory (for example, a decoder that constantly reads and decodes XML messages in a long running loop).

The key memory management function that a user might use are the following:

- **rtxMemAlloc** - This function allocates a block of memory in much the same way *malloc* would. The only difference from the user's perspective is that a pointer to a context structure is required as an argument. The allocated memory is tracked within this context.

- **rtxMemFreePtr** - This function releases the memory held by a pointer in much the same way the C *free* function would. The only difference from a user's perspective is that a pointer to a context structure is required as an argument. This context must have been used in the call to *rtxMemAlloc* at the time the memory was allocated.
- **rtxMemFree** - This function releases all memory held within a context.
- **rtxMemReset** - This functions resets all memory held within a context. The difference between this and the *rtxMemFree* function is that this function does not actually free the blocks that were previously allocated. It only resets the pointers and indexes within those blocks to allow the memory to be reused.
- **rtxMemRealloc** - This function works in the same way as the C *realloc* function. It reallocates an existing block of memory. As in the other cases above, a pointer to a context structure is a required argument.

Note that these memory management functions are only used in the generation of C code, not C++ (although a user can use them in a C++ application). For C++, the built-in *new* and *delete* operators are used to ensure constructors and destructors are properly executed.

For a full description of these and other memory management functions, see the *XBinder C/C++ Runtime Reference Manual*.

UTF-8 String Functions

The UTF-8 string functions handle string operations on UTF-8 encoded strings. This is the default character string data type used for encoded XML data. UTF-8 strings are represented in C as strings of unsigned characters (bytes) to cover the full range of possible single character encodings.

This group of functions encompasses functions for doing conversions to and from UTF-8 to Unicode as well as standard string manipuation functions such as exist in the C standard string library.

For a complete list and full description of all of the UTF-8 string functions, see the *XBinder C/C++ Runtime Reference Manual.*

Doubly-Linked List Utility Functions

The XBinder compiler will generate a mapping to the *OSRTDList* type for many kinds of repeating types. This is a linked list structure type. The doubly-linked list utility functions are common routines for working with linked lists of this type.

Functions are available to initialize, append, insert, remove, and find elements in lists. Some useful functions in this group are as follows:

- **rtxDListInit** - This function is used to initialize a linked list variable. This is first function that should be called before working with a linked list variable.
- **rtxDListAppend** - This function is used to append an item to a linked list. The normal procedure for populating a linked list variable is to first initialize it and then call this function to add items.

- **rtxDListInsert** - This function is used to insert an item into a specific location within a list.
- **rtxDListRemove** - This function is used to remove an item from a list.

For a complete list and full description of all of the doubly-linked list functions, see the *XBinder C/C++ Runtime Reference Manual.*

Error Formatting and Print Functions

Error formatting and print functions allow information about encode/decode errors to be added to a context block structure and then printed when the error is propagated to the top level.

The **LOG_RTERR** macro is inserted in the generated code by the compiler to record the position of an error in the code and store information on the error in the context structure.

The **OSASSERT** macro can be used to test an assertion in much the same as the standard C *assert* call. If the assertion is false, the macro will cause the program to exit and a printout showing the file and line number of failure along with the failed condition will be shown.

Other key error handling routines for printing error information are as follows:

- **rtxErrPrint** - This function prints a message to standard output containing the error information recorded in the context by calls to **LOG_RTERR**.
- **rtxErrLogUsingCB** - This function allows information on an error to be logged using a user defined callback function. It is useful in environments where printing to standard output is not always an option (for example, in a Windows GUI application or an embedded application).

For a complete list and full description of all of the error formatting and print functions, see the *XBinder C/C++ Runtime Reference Manual.*

Diagnostic trace functions

These functions add diagnostic tracing to the generated code to assist in finding where a problem might occur. Calls to these macros and functions are added when the `-trace` command-line argument is used. The diagnostic message can be turned on and off at both C compile and run-time. To C compile the diagnostics in, the _TRACE macro must be defined (-D_TRACE). To turn the diagnostics on at run-time, the `rtxSetDiag` function must be invoked with the `value` argument set to TRUE.

The key functions in this group are as follows:

- **rtxSetDiag** - This function is used to turn diagnostic tracing on or off at run-time.
- **rtxDiagEnabled** - This function is used to determine if diagnostic tracing is currently enabled for the specified context.
- **rtxDiagHexDump** - This function is used to print a diagnostics hex dump of a section of memory.
- **rtxDiagPrint** - This function is used to print a diagnostics message to `stdout` .

For a complete list and full description of all of the diagnostic trace functions, see the *XBinder C/C++ Runtime Reference Manual.*

Input/Output Data Stream Utility Functions

This group of functions is used to operate on input or output data streams. The decode functions generated by the XBinder compiler can read and decode from a stream that was created using these functions. A stream is an abstraction of some physical input medium such as a file, memory buffer, or socket interface.

The key functions in this group are as follows:

- **rtxStreamFileOpen** -This function opens a file for input or output stream access.
- **rtxStreamFileAttach** -This function attaches as existing open file to a stream to allow read or write access using the stream functions.
- **rtxStreamSocketAttach** -This function attaches an existing open TCPI/IP socket connection to a stream.
- **rtxStreamMemoryCreate** -This function creates a memory buffer and attaches it to a stream for read or write access.
- **rtxStreamMemoryAttach** -This function attaches an existing memory buffer (for example, a static byte array) to a stream for read or write access.
- **rtxStreamRead** -This function reads data from the input stream into a given memory buffer.
- **rtxStreamWrite** -This function writes data to an output stream.
- **rtxStreamFlush** -This function flushes the output stream and forces any buffered output octets to be written out.
- **rtxStreamClose** - This function closes the input or output stream and releases any system resources associated with the stream. For output streams this function also flushes all internal buffers to the stream.

For a complete list and full description of all of the stream input/output functions, see the *XBinder C/C++ Runtime Reference Manual.*

TCP/IP or UDP socket utility functions

This group of functions allows TCP/IP or UDP sockets to be set up for interprocess communications. These functions can be used in conjunction with the stream input/output functions described above to allow direct encoding and decoding of XML messages to and from socket connections.

The key functions in this group are as follows:

- **rtxSocketAccept** - This function accepts an incoming connection request on a socket.
- **rtxSocketBind** - This function associates a local address with a socket.
- **rtxSocketConnect** -This function establishes a connection on a specified socket.
- **rtxSocketCreate** - This function creates a new socket.
- **rtxSocketListen** - This function places a socket in a state where it is listening for an incoming connection.
- **rtxSocketRecv** -This function receives (reads) data from a connected socket.
- **rtxSocketWrite** - This function writes data to a socket connection.

For a complete list and full description of all of the stream input/output functions, see the *XBinder C/C++ Runtime Reference Manual*.

SOAP and HTTP utility functions

This group of functions provides basic Simple Object Access Protocol (SOAP) and Hypertext Transfer Protocol (HTTP) support to allow XML messages created/parsed with XBinder to be exchanged with SOAP endpoints (for example, with a web-service application).

- **rtxSoapInitConn** - Initialize a connection structure for use in communicating with a SOAP endpoint.
- **rtxSoapConnect** - Connect to a SOAP endpoint.
- **rtxSoapSendHttp** - Send an HTTP request to a SOAP endpoint.
- **rtxSoapRecvHttp** - Receive a response from a SOAP endpoint.
- **rtxSoapRecvHttpContent** - Receive HTTP content from a SOAP endpoint.

For a complete list and full description of all of the SOAP functions, see the *XBinder C/C++ Runtime Reference Manual*.

C++ Built-in Runtime Classes

C++ runtime classes are the foundation on which generated C++ code is built. Some of these classes such as the message buffer classes are for direct use in application programs. Others, such as the XSD type base classes, are used primarily by the XBinder as base classes for generated classes. The general categories of C++ built-in runtime classes are as follows:

- Context management class
- Message buffer classes
- Global element base class
- XSD type base classes

A cursory description of these classes follows. For a full description, see the *XBinder C/C++ Runtime Reference Manual*.

Context Management Class

The context management class (*OSContext*) manages an XBinder context structure used for C function calls. In general, for C++, this structure is hidden in that it is encapsulated within the message buffer and global element classes. The user needs to be aware of its existence, however, in cases where they want to call one of the C functions from within a C++ application. In these cases, the message buffer or global context class contains a method called *getCtxtPtr* that can be used to retrieve a pointer to the underlying context variable. Note that it does not matter from which class this method is invoked because the message buffer and global element classes share a common context variable.

The context is shared between the global element and message buffer classes by means of the *OSCtxtPtr* referenced counted pointer class. This class is used to maintain a reference count on the context so that it remains in scope as long as either a message buffer or global element class is in scope. A user can invoke the *getContext* method from either a message buffer or global element object in order to obtain a reference to this reference counter pointer object. This would allow them to hold onto the context after all message buffer or global element objects go out of scope should they have a specialized need to do this (for example, if they were using the C memory management facilities that use the managed heap that is stored within the context).

Message Buffer Classes

Message buffer classes describe the memory buffers into which XML message are encoded or from which XML messages are decoded. The main base class for all memory buffer derivations is *OSMessageBufferIF*. This is a pure virtual class that defines the interface all derived message buffer or stream classes must implement.

The base class for all in-memory message buffers is *OSMessageBuffer*. This too is abstract. It is used as the base for the *OSXMLMessageBuffer* class which is the base class for XML message encoding or decoding. From this, concrete XML encode (*OSXMLEncodeBuffer*) and decode (*OSXMLDecodeBuffer*) buffer classes are derived.

To encode an XML message, a user would need to describe the target buffer to which it is to be written. This is what the *OSXMLEncodeBuffer* class is used for. The default constructor allows a dynamic buffer to be setup which which the encoder will manage the memory for to ensure their is enough space for a given encode operation to succeed. Another constructor is available that allows a fixed-sized buffer to be specified by providing the start address and buffer size. If this buffer is not large enough to hold a given encoded message, a buffer overflow error is returned from the encode method that is using the buffer.

To decode an XML message, it must first be described to the decoder. The *OSXMLDecodeBuffer* class can be used for this purpose. A message can be decoded directly from a file or memory or from an input stream object. Constructors are available for specifying each of these inputs.

Global Element Base Class

The global element base class - *OSXSDGlobalElement* - is the base class from which generated XSD global element control classes are derived. These are the main entry points for encoding or decoding items within an XML schema specification. The control class derived from this class is typically constructed with a reference to a variable of the type to be encoded or decoded as well as the associated message buffer. For example, from the C++ employee sample program writer program is the following snippet of code:

```
PersonnelRecord value;
OSXMLEncodeBuffer buffer;
personnelRecord_CC pdu (buffer, value);
```

These three lines of code form the necessary associations to accomplish the encoding of an employee record. The global element declaration in employee.xsd file is the following:

```
<xsd:element name="personnelRecord" type="PersonnelRecord"/>
```

This declares the personnelRecord element to be of type PersonnelRecord. The XBinder compiler generates the *PersonnelRecord* C++ class for the *PersonnelRecord* XSD type. It also generates the *personnelRecord_CC* class for the *personnelRecord* XSD global element. The series of statements above bind an instance of this generated type class with an encode message buffer to accomplish the encoding of an instance of the *personnelRecord* global element.

XSD Type Base Classes

The XSD type base classes are the base classes from which XBinder-generated C++ classes for XSD types are derived. The main base class from which all XSD type classes are derived is the *OSBaseType* class. Generated classes for many types are derived directly from this. However, the following intermediate built-in classes are also present from which generated classes are also derived:

- **OSXMLStringClass** - This is the base class for XSD string types such as xsd:string, xsd: token, etc.. It is derived from the *OSBaseType* class as well as the C *OSXMLSTRING* structure. It provides member variables to hold a UTF-8 string value as well as a CDATA flag to indicate if the string should be encoded as a CDATA section.
- **OSDynOctStrClass** - This is the base class for the XSD *hexBinary* and *base64Binary* string types. It is derived from the *OSBaseType* class as well as the C *OSDynOctStr* structure. It provides member variables to hold binary data in native machine format.
- **OSAnyElementClass** - This is the base class for XSD *any* element declarations. It is derived from the *OSBaseType* class as well as the C *OSUTF8AnyElement* structure. It provides member variables to hold the name and value of the element as UTF-8 strings.
- **OSRTDListClass** - This is the base class for XSD repeating types that use linked lists. It is derived from the *OSBaseType* class as well as the C *OSRTDList* structure. It provides methods for adding, retrieving, and removing items from linked lists.
- **OSObjListClass** - This is the base class for XSD repeating types that hold objects in linked lists. It is similar to the *OSRTDListClass* described above except that the base type for items in the list is *OSBaseType*. This allows items in the list to be properly destructed when memory ownership for the items is transferred to the list object.
- **OSXSDDateTimeClass** - This is a utility class for operating on XSD date/time formats. It is derived from the *OSBaseType* class as well as the C *OSXSDDateTime* structure. Although it is currently not used in any generated classes (date/time classes are currently represented as strings), the class can be used to format or parse XSD date/time strings in application programs.

INDEX

R

S

T

W

X