ASN1C

ASN.1 Compiler
Version 6.4
C/C++ User's Guide

Objective Systems, Inc. — February 2011

Author's Contact Information

Comments, suggestions, and inquiries regarding ASN1C may be submitted via electronic mail to info@obj-sys.com.

Table of Contents

Overview of ASN1C

The ASN1C code generation tool translates an Abstract Syntax Notation 1 (ASN.1) or XML Schema Definitions (XSD) source file into computer language source files that allow ASN.1 data to be encoded/decoded. This release of the compiler includes options to generate code in four different languages: C, C++, C#, or Java. This manual discusses the C and C++ code generation capabilities. The *ASN1C Java User's Manual* discusses the Java code generation capability. The *ASN1C C# User's Manual* discusses the C# code generation capability.

Each ASN.1 module that is encountered in an ASN.1 source file results in the generation of the following two types of C/C++ language files:

1. An include (.h) file containing C/C++ typedefs and classes that represent each of the ASN.1 productions listed in the ASN.1 source file, and

2. A set of C/C++ source (.c or .cpp) files containing C/C++ encode and decode functions. One encode and decode function is generated for each ASN.1 production. The number of files generated can be controlled through command-line options.

These files, when compiled and linked with the ASN.1 low-level encode/decode function library, provide a complete package for working with ASN.1 encoded data.

ASN1C works with the version of ASN.1 specified in ITU-T international standards X.680 through X.683 (ISO/IEC 8824). It generates code for encoding/decoding data in accordance with the following encoding rules:

- Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), or Canonical Encoding Rules (CER) as published in the ITU-T X.690 and ISO/IEC 8825-1 standards.

- Packed Encoding Rules (PER) as published in the ITU-T X.691 and ISO/IEC 8825-2 standards. Both aligned and unaligned variants are supported via a switch that is set at run-time.

- XML Encoding Rules (XER) as published in the ITU-T X.693 and ISO/IEC 8825-3 standards.

- Medical Device Encoding Rules (MDER) as published in the ISO/IEEE 11073 standards.

- Octet Encoding Rules (OER) as published in the NTCIP 1102:2004 standard.

Additional support for XML is provided in the form of an option to generate an equivalent XML Schema Definitions (XSD) file for a given ASN.1 specification. Encoders and decoders can then be generated using the -xml option to format or parse XML documents that conform to this schema. This level of support is closer to the W3C definition of XML then is the ITU-T X.693 XER definition. As of release version 6.0, it is possible to compile an XML schema definitions (XSD) file and generate encoders/decoders that can generate XML in compliance with the schema as well as binary encoders/encoders that implement the ASN.1 binary encoding rules.

The compiler is capable of parsing all ASN.1 syntax as defined in the standards. It is capable of parsing advanced syntax including Information Object Specifications as defined in the ITU-T X.681 standard as well as Parameterized Types as defined in ITU-T X.683. The compiler is also capable of using table constraints as defined in ITU-T X.682 to generate single-step encoders and decoders that can encode or decode multi-part messages in a single function call.

This release of the compiler contains a special command-line option - -asnstd x208 - that allows compilation of deprecated features from the older X.208 and X.209 standards. These include the ANY data type and unnamed fields in SEQUENCE, SET, and CHOICE types. This version can also parse type syntax from common macro definitions such as the OPERATION and ERROR macros in ROSE.

Using the Compiler

Running ASN1C from the Command-line

The ASN1C compiler distribution contains command-line compiler executables as well as a graphical user interface (GUI) wizard that can aid in the specification of compiler options. This section describes how to run the command-line version; the next section describes the GUI.

To test if the compiler was successfully installed, enter `asn1c` with no parameters as follows (note: if you have not updated your PATH variable, you will need to enter the full pathname):

```
asn1c
```

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

```
ASN1C Compiler, Version 6.4.x
Copyright (c) 1997-2011 Objective Systems, Inc. All Rights Reserved.

Usage: asn1c <filename> <options>

    <filename>          ASN.1 or XSD source filename(s).  Multiple filenames
                          may be specified.  * and ? wildcards are allowed.

  language options:
    -c                  generate C code
    -c++                generate C++ code
    -c#                 generate C# code
    -java               generate Java code
    -cldc               generate Java ME CLDC compatible code
    -xsd [<filename>]   generate XML schema definitions

  encoding rule options:
    -ber                generate BER encode/decode functions
    -cer                generate CER encode/decode functions
    -der                generate DER encode/decode functions
    -oer                generate OER encode/decode functions
    -mder               generate MDER encode/decode functions
    -per                generate PER encode/decode functions
    -xer                generate XER encode/decode functions
    -xml                generate XML encode/decode functions

  basic options:
    -asn1 [<file>]      generate pretty-printed ASN.1 source code
    -asnstd <std>       set standard to be used for parsing ASN.1
                        source file. Possible values - x208, x680, mixed
                        (default is x680)
    -compact            generate compact code
    -compat <version>   generate code compatible with previous
                        compiler version. <version> format is
                        x.x (for example, 5.3)
    -config <file>      specify configuration file
    -depends            compile main file and dependent IMPORT items
    -html               generate HTML marked-up version of ASN.1
    -I <directory>      set import file directory
    -lax                do not generate constraint checks in code
    -laxsyntax          do not do a thorough ASN.1 syntax check
    -list               generate listing
    -noContaining       do not generate inline type for CONTAINING <type>
    -nodecode           do not generate decode functions
    -noencode           do not generate encode functions
    -noIndefLen         do not generate indefinite length tests
    -noObjectTypes      do not gen types for items embedded in info objects
    -noOpenExt          do not generate open extension elements
    -notypes            do not generate type definitions
    -noxmlns            do not generate XML namespaces for ASN.1 modules
    -o <directory>      set output file directory (also '-srcdir <dir>')
    -libdir <directory> set output libraries directory
    -bindir <directory> set output binary directory
    -objdir <directory> set output object directory
    -pdu <type>         designate <type> to be a Protocol Data Unit (PDU)
                        (<type> may be "all" to select all type definitions)
    -usepdu <type>      specify a Protocol Data Unit (PDU) type for which
                        sample reader/writer programs and test code has to
                        be generated
    -print [<filename>] generate print functions
    -prtfmt details | bracetext  format of output generated by print
    -shortnames         reduce the length of compiler generated names
    -syntaxcheck        do syntax check only (no code generation)
    -trace              add trace diag msgs to generated code
    -[no]UniqueNames    resolve name clashes by generating unique names
                          default=on, use -noUniqueNames to disable
    -warnings           output compiler warning messages
    -nodatestamp        do not put date/time stamp in generated files

  C/C++ options:
    -array              use arrays for SEQUENCE OF/SET OF types
    -arraySize <size>   specify the size of the array variable
    -dynamicArray       use dynamic arrays for SEQUENCE OF/SET OF types
    -linkedList         use linked-lists for SEQUENCE OF/SET OF types
    -hfile <filename>   C or C++ header (.h) filename
                          (default is <ASN.1 Module Name>.h)
    -cfile <filename>   C or C++ source (.c or .cpp) filename
                          (default is <ASN.1 Module Name>.c)
    -genBitMacros       generate named bit set, clear, test macros
    -genFree            generate memory free functions for all types
    -hdrGuardPfx <pfx>  add prefix to header guard #defines in .h files
    -maxlines <num>     set limit of number of lines per source file
                        (default value is 50000)
    -noInit             do not generate initialization functions
    -noEnumConvert      do not generate conversion functions for enumerated
                           items (BER/CER/DER/PER only)
    -oh <directory>     set output directory for header files
    -static             generate static elements (not pointers)
    -cppNs <namespace>  add a C++ namespace to generated code (C++ only)

  C/C++ makefile/project options:
    -genMake [<filename>] generate makefile to compile generated code
    -genMakeDLL [<filename>] generate makefile to build DLL
    -genMakeLib [<filename>] generate makefile to build static library
    -make [<filename>]  same as -genMake as described above.
    -nmake [<filename>] generate Windows nmake file (same as -genMake -w32)
    -vcproj [version]   generate Visual Studio project files.
                          [version] is 2008, 2005, 2003. (Windows only)
    -builddll           generate makefile/project to build DLL
    -dll, -usedll       generate makefile/project to use DLL's
    -mt                 generate makefile/project to use multithreaded libs
    -w32                generate code for Windows 32-bit O/S (default=GNU)
    -w64                generate code for Windows 64-bit O/S (default=GNU)

  Java options:
    -compare           generate comparison functions
    -dirs              output Java code to module name dirs
    -genbuild          generate build script
    -genant            generate ant build.xml script
    -genjsources       generate <modulename>.mk for list of java files
    -getset            generate get/set methods and protected member vars
    -pkgname <text>    Java package name
    -pkgpfx <text>     Java package prefix
    -java4             generate code for Java 1.4

  C# options:
    -nspfx <text>      C# namespace prefix
    -namespace <text>  C# namespace name
    -dirs              output C# code to module name dirs
    -csfile <filename> generate one .cs file or one per module (*.cs)
    -gencssources      generate <modulename>.mk for list of C# files
    -genMake           generate makefile to build generated code

  pro options:
    -events            generate code to invoke SAX-like event handlers
    -stream            generate stream-based encode/decode functions
    -strict            do strict checking of table constraint conformance
    -tables            generate table constraint functions
    -table-unions      generate union structures for table constraints
    -param <name>=<value> create types from param types using given value
    -prtToStr [<filename>]
                       generate print-to-string functions (C/C++)
    -prtToStrm [<filename>]
                       generate print-to-stream functions (C/C++)
    -genTest  [<filename>]
                       generate sample test functions
    -reader            generate sample reader program
    -writer            generate sample writer program
    -compare [<filename>]
                       generate comparison functions (C/C++)
    -copy [<filename>] generate copy functions (C/C++)
    -maxcfiles         generate separate file for each function (C/C++)

  XSD options:
    -appinfo [<items>] generate appInfo for ASN.1 items
                         <items> can be tags, enum, and/or ext
                         ex: -appinfo tags,enum,ext
                         default = all if <items> not given
    -attrs [<items>]   generate non-native attributes for <items>
                         <items> is same as for -appinfo
    -targetns [<namespace>] Specify target namespace
                        <namespace> is namespace URI, if not given
                        no target namespace declaration is added
    -useAsn1Xsd         reference types in asn1.xsd schema
  Symbian options:
    -symbian [<items>] generate code for Symbian OS
                         <items> can be dll
                         e.g. -symbian dll
                         default = symbian application style code
```

Note that this usage summary shows all options for the pro version of ASN1C. Some of these options are not available in the basic version.

To use the compiler, at a minimum, an ASN.1 or XSD source file must be provided. 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. If a file extension is not provided, the default extension ".asn" is appended to the name. 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 `asn1c *.asn`code> will compile all ASN.1 files in the current working directory).

The source file(s) must contain ASN.1 productions that define ASN.1 types and/or value specifications. This file must strictly adhere to the syntax specified in ASN.1 standard ITU-T X.680. The -*asnstd x208* command-line option should be used to parse files based on the 1990 ASN.1 standard (x.208) or that contain references to ROSE macro specifications.

The following table lists all of the command line options and what they are used for. The options are shown in alphabetical order. Note that the Java and C# options are not shown here. They are shown in their respective documents.

Option	**Argument**	**Description**
-3gpp	None	This option is used to generate special code for table constraints in ASN.1 specifications that have a common pattern as found in many of the 3rd Generation Partnership Project (3GPP) specifications. Specifications having this pattern include NBAP, RANAP, S1AP, and X2AP. ASN1C can take advantage of this common pattern to generate more efficient code. **Note:** This option is deprecated. The new option that provides the same functionality is *-table-unions* which may be used to replace the *-3gpp -tables* option combination.
-appInfo	<items>	This option only has meaning when generating an XML schema definitions (XSD) file using the -xsd option. It instructs the compiler to generate an XSD application information section (<appinfo>) for certain ASN.1-only items. The items are speci-

Option	Argument	Description
		fied as a comma-delimited list. Valid values for items are tags, enum, and ext. <items> is an optional parameter. If it is not specified, it is assumed that application information should be produced for all three item classes: ASN.1 tags, ASN.1 enumerations, and extended elements.
-array	none	This option specifies that an array type will be used for SEQUENCE OF/SET OF constructs.
-arraySize	<size>	This option specifies the size of array variable.
-asn1	<filename>	This option causes pretty-printed ASN.1 to be generated to the given file name or to stdout if no filename was given. Besides the obvious use of providing neatly formatted ASN.1 source code, the tool is also useful for producing ASN.1 source code from XML schema document (XSD) files as well as producing trimmed specifications when <include> or <exclude> configuration directives are used.
-asnstd	x208 x680 mixed	This option instructs the compiler to parse ASN.1 syntax conforming to the specified standard. ‘x680’ (the default) refers to modern ASN.1 as specified in the ITU-T X.680-X.690 series of standards. ‘x208’ refers to the now deprecated X.208 and X.209 standards. This syntax allowed the ANY construct as well as unnamed fields in SEQUENCE, SET, and CHOICE constructs. This option also allows for parsing and generation of code for ROSE OPERATION and ERROR macros and SNMP OBJECTTYPE macros. The ‘mixed’ option is used to specify a source file that contains modules with both X.208 and X.680 based syntax.
-attrs	<items>	This option only has meaning when generating an XML schema definitions (XSD) file using the -xsd option. It instructs the compiler to generate non-native attributes for certain ASN.1-only items that cannot be expressed in XSD. The items are speci-

Option	Argument	Description
		fied as a comma-delimited list. Valid values for items are tags, enum, and ext. <items> is an optional parameter. If it is not specified, it is assumed that application information should be produced for all three item classes: ASN.1 tags, ASN.1 enumerations, and extended elements.
-ber	None	This option instructs the compiler to generate functions that implement the Basic Encoding Rules (BER) as specified in the X.690 ASN.1 standard.
-bindir	<directory>	This option is used in conjunction with the -genMake option to specify the name of the binary executable directory to be added to the makefile. Linked executable programs will be output to this directory.
-bitMacros	None	This option instructs the compiler to generate additional macros to set, clear, and test named bits in BIT STRING constructs. By default, only bit number constants are generated. Bit macros provide slightly better performance because mask values required to do the operations are computed at compile time rather than runtime.
-c	None	Generate C source code.
-c# or -csharp	None	Generate C# source code. See the *ASN1C C# User's Guide* for more information and options for generating C# code.
-c++ or -cpp	None	Generate C++ source code.
-cer	None	This option instructs the compiler to generate functions that implement the Canonical Encoding Rules (CER) as specified in the X.690 ASN.1 standard.
-cfile	[<filename>]	This option allows the specification of a C or C++ source (.c or .cpp) file to which all of the generated encode/decode functions will be written. If not specified, the default is to write to a series of .c or .cpp files based on the ASN.1 module name(s) of the documents being compiled.
-compact	None	This option instructs the compiler to generate more compact code at the expense of some con-

Option	Argument	Description
		straint and error checking. This is an optimization option that should be used after an application is thoroughly tested.
-compat	<versionNumber>	Generate code compatible with an older version of the compiler. The compiler will attempt to generate code more closely aligned with the given previous release of the compiler. <versionNumber> is specified as x.x (for example, -compat 5.2)
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. A full discussion of the contents of a configuration file is provided in the *Compiler Configuration File* section.
-cppns	<namespace>	This option is used to add a C++ namespace name to generated C++ files.
-depends	None	This option instructs the compiler to generate a full set of header and source files that contain only the productions in the main file being compiled and items those productions depend on from IMPORT files.
-der	None	This option instructs the compiler to generate functions that implement the Distinguished Encoding Rules (DER) as specified in the X.690 ASN.1 standard.
-dll	None	When used in conjunction with the *-genMake* command-line option, the generated makefile uses dynamically-linked libraries (DLLs in Windows, or .so files in UNIX) instead of statically-linked libraries.
-dynamicArray	none	This option specifies that a dynamic array type is to be used for SEQUENCE OF/SET OF constructs.
-genbuild	None	This option instructs the compiler to generate a build script when producing Java source code. The generated build script is either a batch file (Windows) or a shell script (UNIX).
-genCompare -compare	[<filename>]	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated compare functions will be written. Compare

Option	Argument	Description
		functions allow two variables of a given ASN.1 type to be compared for equality. The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Compare.c where <modulename> is the name of the module from the ASN.1 source file.
-genCopy -copy	[<filename>]	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated copy functions will be written. Copy functions allow a copy to be made of an ASN1C generated variable. For C++, they cause copy constructors and assignment operators to be added to generated classes. The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Copy.c where <modulename> is the name of the module from the ASN.1 source file.
-genFree	None	This option instructs the compiler to generate a memory free function for each ASN.1 production. Normally, memory is freed within ASN1C by using the *rtxMemFree* run-time function to free all memory at once that is held by a context. Generated free functions allow finer grained control over memory freeing by just allowing the memory held for specific objects to be freed.
-genMake	[<filename>]	This option instructs the compiler to generate a portable makefile for compiling the generated C or C++ code. If used with the -*w32* command-line option, a makefile that is compatible with the Microsoft Visual Studio *nmake* utility is generated; otherwise, a GNU-compatible makefile is generated. Note that the -*nmake* option may now be used instead of the -*make* -*w32* combination to produce a Visual Studio makefile.
-genMakeDLL	[<filename>]	This option instructs the compiler to generate a portable makefile for compiling the generat-

Option	Argument	Description
		ed C or C++ code into a Dynamic Link Library (DLL).
-genMakeLib	[<filename>]	This option instructs the compiler to generate a portable makefile for compiling the generated C or C++ code into a static library file.
-genPrint -print	[<filename>]	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated print functions will be written. Print functions are debug functions that allow the contents of generated type variables to be written to stdout. The <filename> argument to this option is optional. If not specified, the print functions will be written to <modulename>Print.c where <modulename> is the name of the module from the ASN.1 source file.
-genPrtToStr -prtToStr	[<filename>]	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated "print-to-string" functions will be written. "Print-to-string" functions are similar to print functions except that the output is written to a user-provided text buffer instead of stdout. This makes it possible for the use to display the results on different output devices (for example, in a text window). The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Print.c where <modulename> is the name of the module from the ASN.1 source file.
-genPrtToStrm -prtToStrm	[<filename>]	This option allows the specification of a C or C++ source (.c or .cpp) file to which generated "print-to-stream" functions will be written. "Print-to-stream" functions are similar to print functions except that the output is written to a user-provided stream instead of stdout. The stream is in the form of an output callback function that can be set within the run-time context making it possible to redirect output to any type of device.

Option	Argument	Description
		The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Print.c where <modulename> is the name of the module from the ASN.1 source file.
-genTables -tables	[<filename>]	This option is used to generate additional code for the handling of table constraints as defined in the X.682 standard. See the *Generated Information Object Table Structures* section for additional details on the type of code generated to support table constraints. **Note:** An alternaitve option for C/C++ is *-table-unions* which generates union structures for table constraints. These are generally easier to work with then the legacy void pointer approach used in this option.
-genTest	[<filename>]	This option allows the specification of a C or C ++ source (.c or .cpp) file to which generated "test" functions will be written. "Test" functions are used to populate an instance of a generated PDU type variable with random test data. This instance can then be used in an encode function call to test the encoder. Another advantage of these functions is that they can act as templates for writing your own population functions. The <filename> argument to this option is optional. If not specified, the functions will be written to <modulename>Test.c where <modulename> is the name of the module from the ASN.1 source file.
-hdrGuardPrefix	[<prefix>]	This option allows the specification of a prefix that will be used in the generated #defines that are added to header files to make sure they are only included once.
-hfile	[<filename>]	This option allows the specification of a header (.h) file to which all of the generated typedefs and function prototypes will be written. If not specified, the default is <modulename>.h where <modulename> is the name of the module from the ASN.1 source file.
-html	None	This option instructs the compiler to generated HTML markup for every compiled ASN.1

Option	Argument	Description
		file. This markup contains hyperlinks to all referenced items within the specifications. One HTML file is generated for each corresponding ASN.1 source file.
-I	<directory>	This option is used to specify a directory that the compiler will search for ASN.1 source files for IMPORT items. Multiple –I qualifiers can be used to specify multiple directories to search.
-java	None	Generate Java source code. See the *ASN1C Java User's Guide* for more information on Java code generation.
-lax	None	This option instructs the compiler to not generate code to check constraints. When used in conjunction with the *-compact* option, it produces the smallest code base for a given ASN.1 specification.
-laxsyntax	None	This option instructs the compiler to not do a thorough syntax check when compiling a specification and to generate code even if the specification contains non-fatal syntax errors. Use of the code generated in this case can have unpredictable results; however, if a user knows that certain parts of a specification are not going to be used, this option can save time.
-libdir	<directory>	This option is used in conjunction with the -genMake option to specify the name of the library directory to be added to the makefile.
-linkedList	none	This option specifies that a linked-list type is to be used for SEQUENCE OF/SET OF constructs.
-list	None	Generate listing. This will dump the source code to the standard output device as it is parsed. This can be useful for finding parse errors.
-maxcfiles	None	Maximize number of generated C files. This option instructs the compiler to generate a separate .c file for each generated C function. In the case of C++, a separate .cpp file is generated for each control class, type, and C function. This is a space optimization option - it can lead to smaller executable sizes by allowing the linker to only link in the required program module object files.

Option	**Argument**	**Description**
-maxlines	<number>	This option is used to specify the maximum number of lines per generated .c or .cpp file. If this number is exceeded, a new file is started with a "_n" suffix where "n" is a sequential number. The default value if not specified is 50,000 lines which will prevent the VC++ "Maximum line numbers exceeded" warning that is common when compiling large ASN.1 source files. Note that this number is approximate - the next file will not be started until this number is exceeded and the compilation unit that is currently being generated is complete.
-mder	None	This option instructs the compiler to generate functions that implement the Medical Device Encoding Rules (MDER) as specified in the IEEE/ISO 11073 standard.
-mt	None	When used in conjunction with the *-genMake* command-line option, the generated makefile uses multi-threaded libraries.
-nmake	[<filename>]	This option instructs the compiler to generate a Visual Studio compatible makefile. It is equivalent to using the *-genMake -w32* combination of command-line options.
-noContaining	None	This option suppresses the generation of inline code to support the CONTAINING keyword. Instead, a normal OCTET STRING or BIT STRING type is inserted as was done in previous ASN1C versions.
-nodecode	None	This option suppresses the generation of decode functions.
-noencode	None	This option suppresses the generation of encode functions.
-noIndefLen	None	This option instructs the compiler to omit indefinite length tests in generated decode functions. These tests result in the generation of a large amount of code. If you know that your application only uses definite length encoding, this option can result in a much smaller code base size.
-noInit	None	This option instructs the compiler to not generate an initialization function for each ASN.1 pro-

Option	Argument	Description
		duction. A variable of a generated structure can always be initialized by memset'ing the variable to zero. However, this is not usually the most efficient way to initialize a variable because if it contains large byte arrays, a significant amount of processing is required to set all bytes to zero (and they don't need to be). Initialization functions provide a smart alternative to memset'ing in that only what needs to be set to zero actually is. Note that previous versions of ASN1C did not generate initialization functions by default. The -*genInit* switch has been deprecated in favor of -*noInit*.
-noEnumConvert	None	This option suppresses the generation of utility functions in C and C++ that assist in converting enumerated values to strings and vice versa. XER and XML encodings are unaffected by this option, since conversions are necessary for encoding and decoding.
-noObjectTypes	None	This option suppresses the generation of application language types corresponding to ASN.1 types embedded within information object definitions.
-noOpenExt	None	This option instructs the compiler to not add an open extension element (*extElem1*) in constructs that contain extensibility markers. The purpose of the element is to collect any unknown items in a message. If an application does not care about these unknown items, it can use this option to reduce the size of the generated code.
-notypes	None	This options suppresses the generation of type definitions. It is used in conjunction with the -*events* options to generate pure parser functions.
-noxmlns	None	This option instructs the compiler not to insert XML namespace entries in generated XML documents. This includes `xmlns` attributes and prefixed names.
-nouniquenames	None	This option instructs the compiler not to automatically generate unique names to resolve

Option	**Argument**	**Description**
		name collisions in the generated code. Name collisions can occur, for example, if two modules are being compiled that contain a production with the same name. A unique name is generated by prepending the module name to one of the productions to form a name of the form <module>_<name>. Note that name collisions can also be manually resolved by using the *typePrefix*, *enumPrefix*, and *valuePrefix* configuration items (see the *Compiler Configuration File* section for more details). Previous versions of the compiler did not generate unique names by default. The compiler option -*uniquenames* has been deprecated in favor of -*nouniquenames*.
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-objdir	<directory>	This option is used in conjunction with the -genMake option to specify the name of the object file directory to be added to the makefile. Compiled object files will be output to this directory.
-oh	<directory>	This option is used to specify the name of a directory to which only the generated header files (*.h) will be written.
-oer	None	This option instructs the compiler to generate functions that implement the Octet Encoding Rules (OER) as specified in the NTCIP 1102:2004 standard.
-param	<name>=<value>	This option is used to instantiate all parameterized types with the ASN.1 modules that are being compiled with the given parameter value. In this declaration, <name> refers to the dummy reference in a parameterized type definition and <value> refers to an actual value.
-pdu	<typeName>	Designate given type name to be a "Protocol Definition Unit" (PDU) type. This will cause a C++ control class to be generated for the given type. By default, PDU types are determined to be

Option	Argument	Description
		types that are not referenced by any other types within a module. This option allows that behavior to be overridden. The "all" keyword may be specified for <typeName> to indicate that all productions within an ASN.1 module should be treated as PDU types.
-per	None	This option instructs the compiler to generate functions that implement the Packed Encoding Rules (PER) as specified in the ASN.1 standards.
-prtfmt	bracetext details	Sets the print format for generated print functions. The *details* option causes a line-by-line display of all generated fields in a generated structure to be printed. The *bracetext* option causes a more concise printout showing only the relevant fields in a C-like brace format. As of release version 6.0, *bractext* is the default (*details* was the default or only option in previous versions).
-shortnames	None	Generate a shorter form of an element name for a deeply nested production. By default, all intermediate names are used to form names for elements in nested types. This can lead to very long names for deeply nested types. This option causes only the production name and the last element name to be used to form a generated type name.
-static	None	This has the same effect as specifying the global *<storage> static </storage>* configuration item. The compiler will insert static elements instead of pointer variables in some generated structures.
-stream	None	This option instructs the compiler to generate stream-based encoders/decoders instead of memory buffer based. This makes it possible to encode directly to or decode directly from a source or sink such as a file or socket. In the case of BER, it will also cause forward encoders to be generated, which will use indefinite lengths for all constructed elements in a message.

Option	**Argument**	**Description**
		Note that stream and memory-buffer based encode/decode functions cannot be used/combined in any way. The two are mutually exclusive. If the -stream option is selected, then only stream-based run-time functions can be used with the generated code.
-strict	None	This option instructs the compiler to generate code for strict validation of table constraints. By default, generated code will not check for value field constraints. It should be noted that real world messages typically do not strictly follow value field table constraint definitions. Therefore, this option should be used with care.
-syntaxcheck	None	This option instructs the compiler to do a syntax check of the ASN.1 source files only. No code is to be generated.
-table-unions	None	This option instructs the compiler to generate union structures for table constraints in C/C++ instead of void pointers as is done when the *-tables* option is used. These are generally easier to use as they present all options in a user-friendly way.
-targetns	<namespace>	This option only has meaning when generating an XML schema definitions (XSD) file using the -xsd option. It allows specification of a target namespace. <namespace> is a namespace URI; if it is not provided, no target namespace declaration is added to the generated XSD file.
-trace	None	This option is used to tell the compiler to add trace diagnostic messages to the generated code. These messages cause print 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.

Option	**Argument**	**Description**
-usepdu	<PDU type name>	This option is used to specify the name of the Protocol Data Unit (PDU) type to be used in generated reader and writer programs.
-vcproj	2003 2005 2008 2010	This option instructs the compiler to generate Visual C++- or Visual Studio-compatible project files to compile generated source code. This is a Windows-only option. By passing one of the listed years, the compiler will generate a project that links against libraries provided for those versions of Visual Studio. For example, specifying 2008 will generate a project that links against libraries in the `*_vs2008` directory. Not specifying a year will cause the compiler to link against libraries compiled for Visual Studio 6.0. A custom build rule is generated that deletes the generated source files and then invokes ASN1C to regenerate them when a rebuild is done. Doing a clean operation will cause the generated source files to be deleted; a subsequent build will regenerate them. For Visual C++ 6.0 project files you can see this build rule by locating the first ASN.1 file under the Source Files for the project, right clicking it, choosing Settings... and then choosing the Custom Build tab. For Visual Studio 2005, 2008, and 2010 project files the procedure to see the custom rule is very similar to that for Visual C++ 6.0, except that you choose Properties when you right click on the first ASN.1 file, and then click on Custom Build Step or Custom Build Tool on the left.
-w32	None	This option is used with makefile and/or Visual Studio project generation to indicate the generated file is to be used on a Windows 32-bit system. In the case of makefile generation, this will cause a makefile to be generated taht si compatible with the Visual Studio nmake utility.

Option	Argument	Description
-w64	None	This option is similar to the *-w32* option documented above except that it specifies a Windows 64-bit system.
-warnings	None	Output information on compiler generated warnings.
-xer	None	This option instructs the compiler to generate functions that implement the XML Encoding Rules (XER) as specified in the X.693 ASN.1 standard.
-xml	None	This option instructs the compiler to generate functions that encode/ decode data in an XML format that is more closely aligned with World-Wide Web Consortium (W3C) XML schema. The *-xsd* option can be used in conjunction with this option to generate a schema describing the XML format.
-xsd	[<filename>]	This option instructs the compiler to generate an equivalent XML Schema Definition (XSD) for each of the ASN.1 productions in the ASN.1 source file. The definitions are written to the given filename or to <modulename>.xsd if the filename argument is not provided.

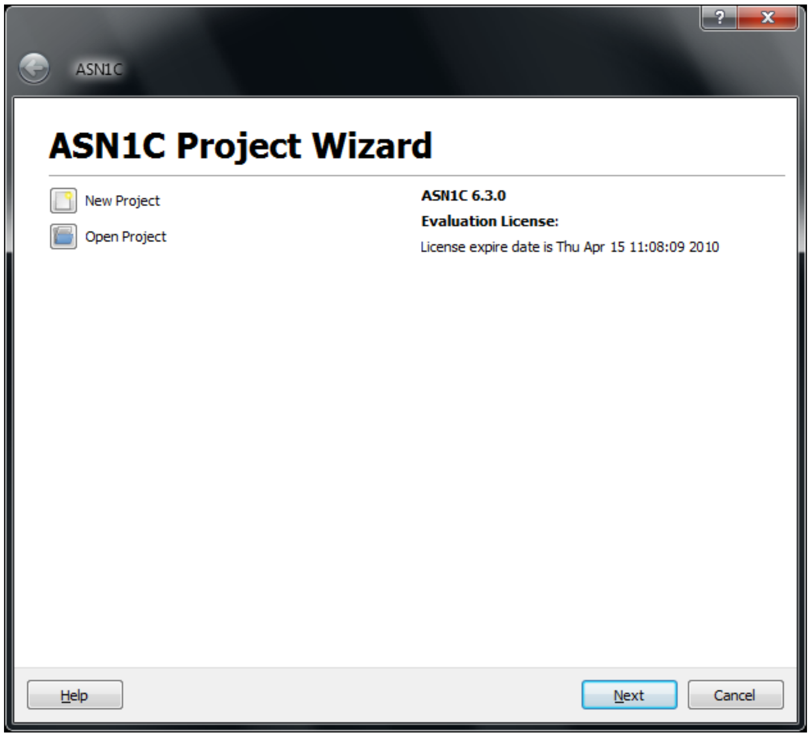
Using the GUI Wizard to Run ASN1C

ASN1C includes a graphical user interface (GUI) wizard that can be used as an alternative to the command-line version. It is a cross-platform GUI and has been ported to Windows and several UNIXes. The GUI makes it possible to specify ASN.1 files and configuration files via file navigation windows, to set command line options by checking boxes, and to get online help on specific options.

The Windows installation program should have installed an ‘ASN1C Compiler’ option on your computer desktop and an ‘ASN1C’ option on the start menu. The wizard can be launched using either of these items. The UNIX version should be installed in `ASN1C_INSTALL_DIR/bin`; no desktop shortcuts are created, so it will be necessary to create one or to run the wizard from the command-line.

Using Projects

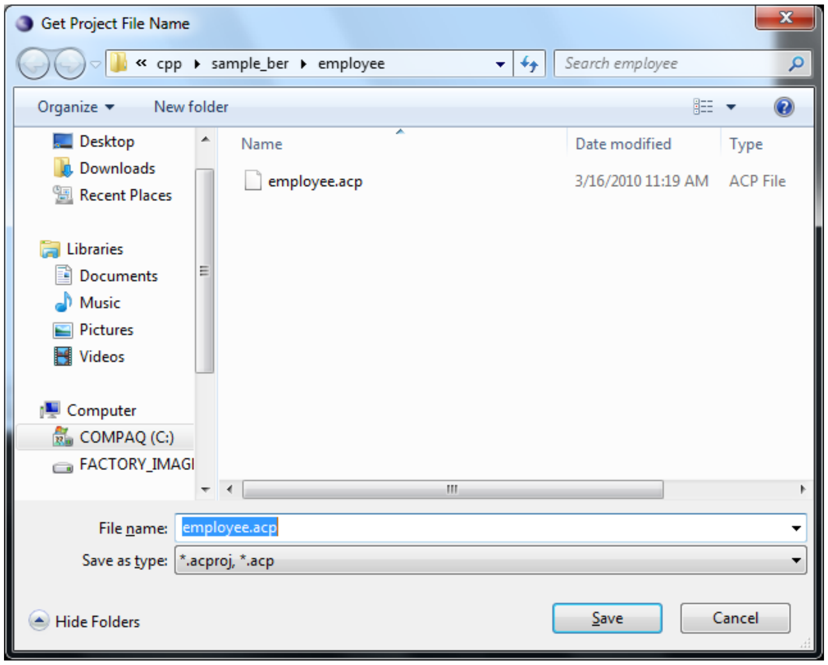
The wizard is navigated by means of *Next* and *Back* buttons. Following is the initial window:



ASN1C 6.3.0
Evaluation License:
License expire date is Thu Apr 15 11:08:09 2010

The status window will display the version of the software you have installed as well as report any errors upon startup that occur, such as a missing license file.

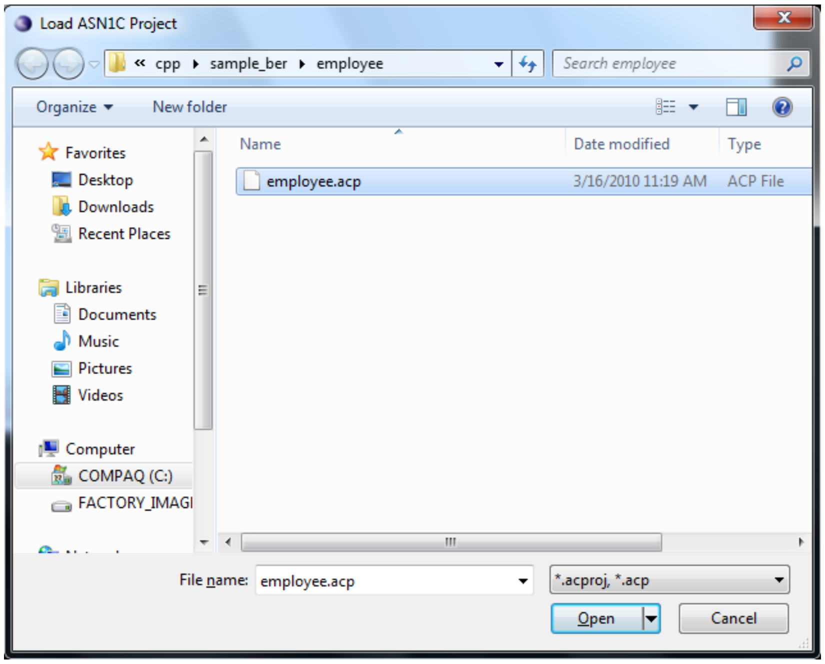
The Project Wizard will allow you to save your compilation options and file settings into a project file and retrieve them later. If you wish to make a new project, click the icon next to *Create a New Project*:



Name	Date modified	Type
employee.acp	3/16/2010 11:19 AM	ACP File

File name: employee.acp
Save as type: *.acproj, *.acp

Previously saved projects may be recalled by clicking the icon next to *Open an Existing Project*:



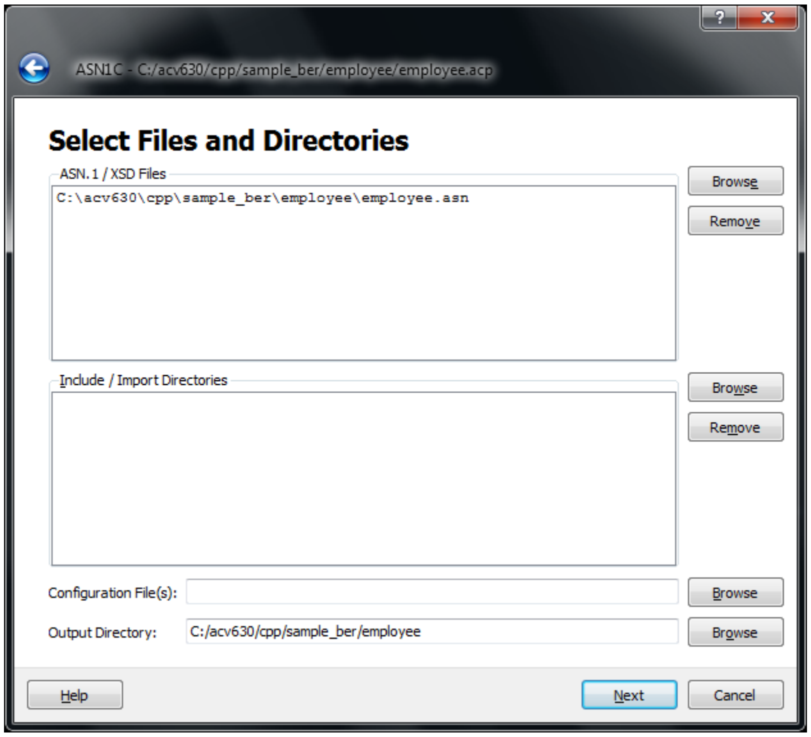
Name	Date modified	Type
employee.acp	3/16/2010 11:19 AM	ACP File

File name: employee.acp

*.acproj, *.acp

The project format has changed in ASN1C 6.3 to help accommodate the transition to Qt 4.5. Changes to the interface necessitated changes to the underlying project file format. Projects made with previous versions may be loaded with version 6.3, but new projects are incompatible with previous versions. Additional metadata are stored in the project file to help with version tracking.

Files may be added to a project in the following window:


ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp
Select Files and Directories
ASN.1 / XSD Files
Browse
Remove
Include / Import Directories
Browse
Remove
Browse
Browse
Help
Next
Cancel

ASN.1 / XSD Files: C:\acv630\cpp\sample_ber\employee\employee.asn

Configuration File(s):

Output Directory: C:/acv630/cpp/sample_ber/employee

In this window, the ASN.1 file or files to be compiled are selected. This is done by clicking the *Add* button on the right hand side of the top windows pane. A file selection box will appear allowing you to select the ASN.1 or XSD files to be compiled. Files can be *removed* from the pane by highlighting the entry and clicking the Remove button.

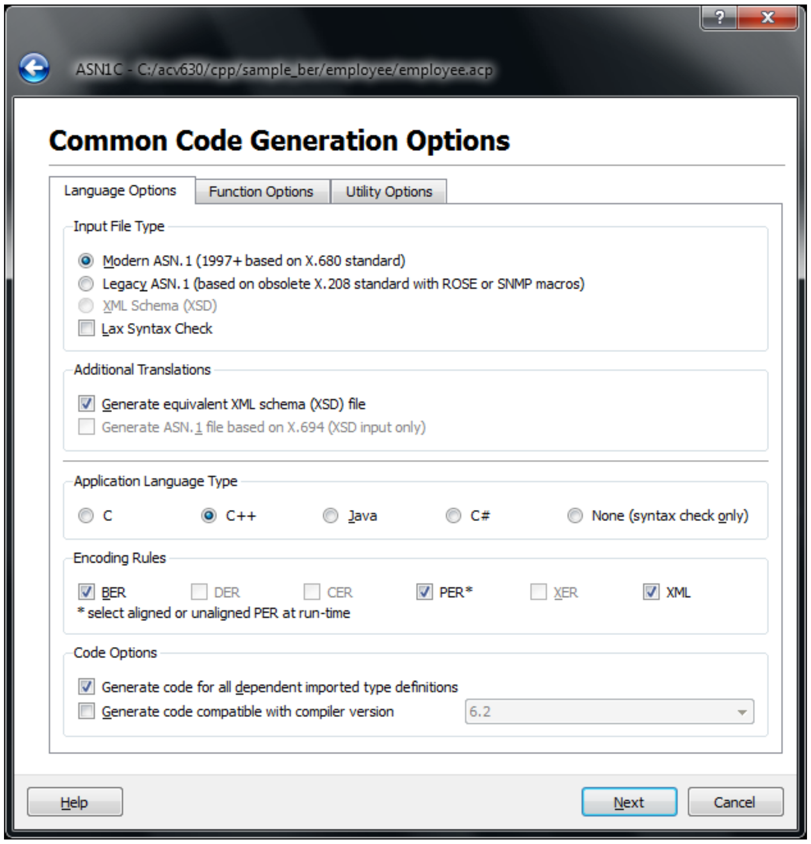
ASN.1 specifications and XML Schema Documents must not be compiled in the same project. Once an ".asn" file has been added, no ".xsd" files may be added.

Include directories are selected in a similar manner in the middle pane. These are directories the compiler will search for import files. By default, the compiler looks for files in the current working directory with the name of the module being imported and extension ".asn" or ".xsd". Additional directories can be searched for these files by adding them here.

User-defined configuration files are specified in the third pane. These allow further control of the compilation process. They are optional and are only needed if the default compilation process is to be altered (for example, if a type prefix is to be added to a generated type name). See the *Compiler Configuration File* section for details on defining these files.

Common Code Generation Options

Code generation options common to all language types are specified in the following tabbed window:


ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp

Common Code Generation Options

Input File Type

[✓ Modern ASN.1 (1997+ based on X.680 standard)]
Legacy ASN.1 (based on obsolete X.208 standard with ROSE or SNMP macros)
XML Schema (XSD)
Lax Syntax Check

Additional Translations

[✓ Generate equivalent XML schema (XSD) file]
Generate ASN.1 file based on X.694 (XSD input only)

Application Language Type

C [✓ C++] Java C# None (syntax check only)

Encoding Rules

[✓ BER] DER CER [✓ PER*] XER [✓ XML]
* select aligned or unaligned PER at run-time

Code Options

[✓ Generate code for all dependent imported type definitions]
Generate code compatible with compiler version 6.2

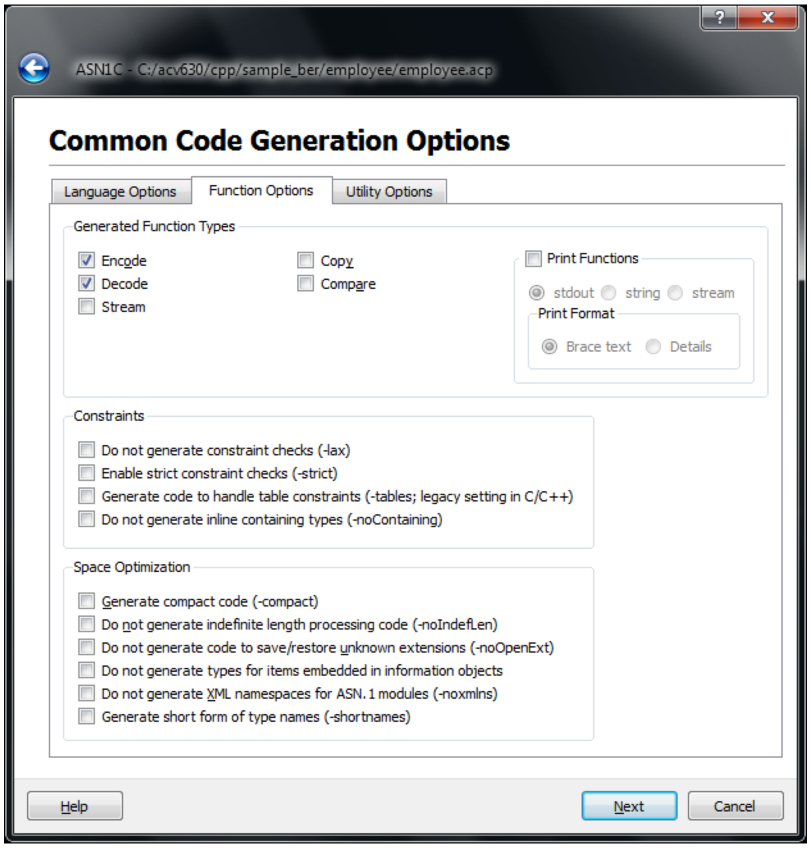
Language options, pictured above, encompass not only the output language choice, but also input specification type, encoding rules, and code compatibility options.

Certain options will be inactive (greyed out) depending on the file type selected. For example, if an XSD file is selected, the option *Generate ASN.1 file based on X.694* will be active and the option *Generate equivalent XML schema (XSD) file* will be inactive.

Checking *Generate code for all dependent imported type definitions* will cause the compiler to search and generate code for modules specified in the `IMPORTS` statement of an ASN.1 specification.

Basic encoding rules are selected by default. Only one of BER, DER, and CER can be checked at any time. XML and XER are also mutually exclusive options.

Generated function options are shown in the following tab:



ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp

Common Code Generation Options

Generated Function Types

- [✓] Encode
- [✓] Decode
- [ ] Stream
- [ ] Copy
- [ ] Compare
- [ ] Print Functions
  - [● stdout] string stream
  - Print Format: [● Brace text] Details

Constraints

- [ ] Do not generate constraint checks (-lax)
- [ ] Enable strict constraint checks (-strict)
- [ ] Generate code to handle table constraints (-tables; legacy setting in C/C++)
- [ ] Do not generate inline containing types (-noContaining)

Space Optimization

- [ ] Generate compact code (-compact)
- [ ] Do not generate indefinite length processing code (-noIndefLen)
- [ ] Do not generate code to save/restore unknown extensions (-noOpenExt)
- [ ] Do not generate types for items embedded in information objects
- [ ] Do not generate XML namespaces for ASN.1 modules (-noxmlns)
- [ ] Generate short form of type names (-shortnames)

The options in this tab control which functions are generated and what modifications are made to those functions.

By default, encoding and decoding functions are generated by the compiler. If the target application does not require encoding or decoding capabilities (for example, if it is only intended to read messages and does not need to write them), unchecking the corresponding checkbox will reduce the amount of code generated.

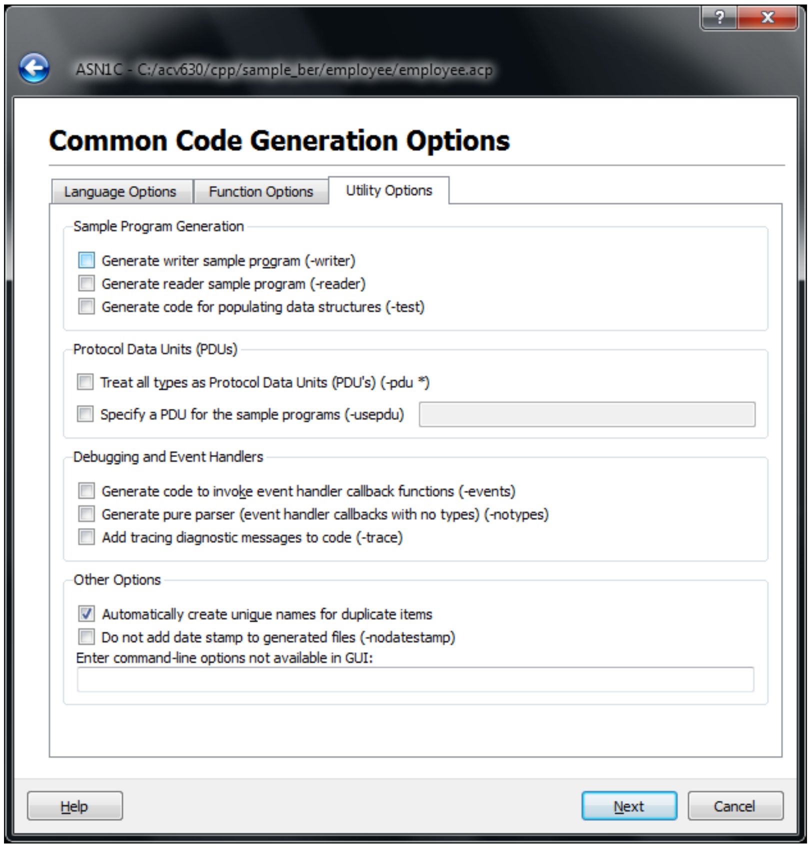
Check *Stream* to modify generated encode and decode functions to use streams instead of memory buffers. This allows encoding and decoding to a source or sink such as a file or socket. Stream-based encoding and decoding cannot be combined with buffer-based.

As an aid to debugging, *Print* functions may also be generated. Three different different types exist: print to stdout, print to string, and print to stream. These allow the contents of generated types to be printed to the standard output, a string, or a stream (such as a file or socket).

Constraint checking may be relaxed or tightened depending on selected options. Constraints may be ignored completely by checking *Do not generate constraint checks*. To tighten constraints, check *Enable strict constraint checks*. ASN1C supports decoding and encoding values described by table constraints; checking *Generate code to handle table constraints* will enable this behavior. This option is a legacy option for C and C++ code generation: generating table constraints in unions is the preferred method (see the following section).

To reduce the code footprint, several other options may be selected: *Generate compact code*, *Do not generate indefinite length processing code*, *Do not generate code to save/restore unknown extensions*, and *Do not generate types for items embedded in information objects* may all be used to reduce the amount of generated code. *Generate compact code* cannot be used in conjunction with *Generate compatible code*. If XML validation is not needed, check *Do not generate XML namespaces for ASN.1 modules*. This will result in a smaller codebase as well as smaller output XML data. Check *Generate short form of type names* if generated type names are too long for the target language.

The following tab provides options for generating utility functions and applications:



ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp

Common Code Generation Options

Sample Program Generation

- [ ] Generate writer sample program (-writer)
- [ ] Generate reader sample program (-reader)
- [ ] Generate code for populating data structures (-test)

Protocol Data Units (PDUs)

- [ ] Treat all types as Protocol Data Units (PDU's) (-pdu *)
- [ ] Specify a PDU for the sample programs (-usepdu)

Debugging and Event Handlers

- [ ] Generate code to invoke event handler callback functions (-events)
- [ ] Generate pure parser (event handler callbacks with no types) (-notypes)
- [ ] Add tracing diagnostic messages to code (-trace)

Other Options

- [✓] Automatically create unique names for duplicate items
- [ ] Do not add date stamp to generated files (-nodatestamp)

Enter command-line options not available in GUI:

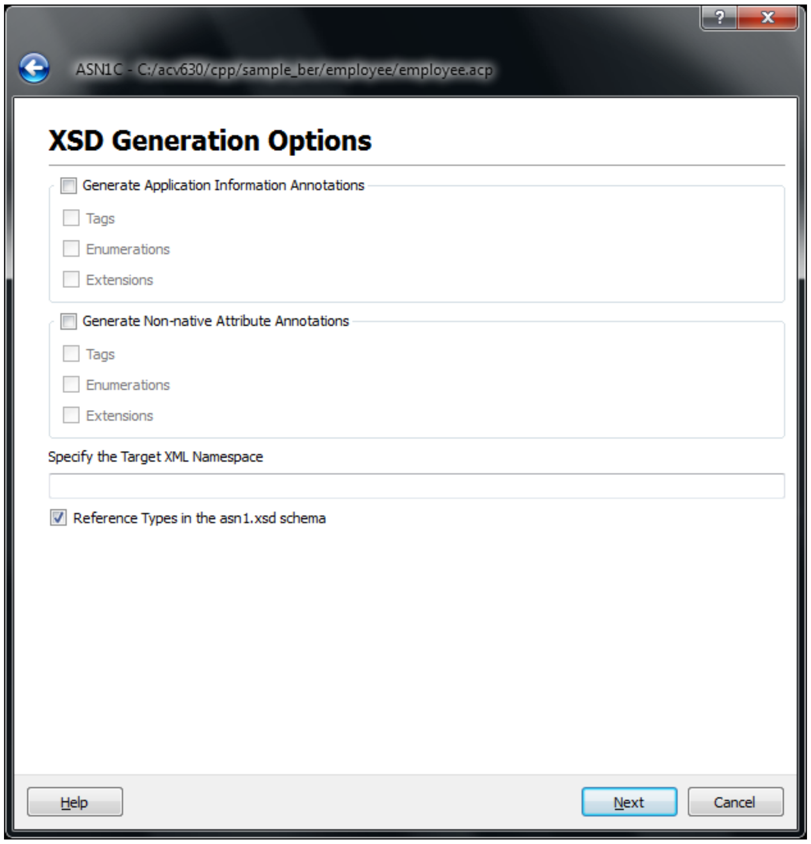
The *Sample Program Generation* frame allows you to generate boilerplate reader and writer applications as well as randomized test data for populating a sample encoded message. The items in the *Protocol Data Units* frame may be used in conjunction to select the appropriate PDU data type to be used in the sample programs.

The *Debugging and Event Handlers* frame contains options that generate code for adding trace diagnostics and event handling hooks into generated code. It is possible to generate a type parser by generating only an event handler and no data types for the decoded messages. This grants a great deal of flexibility in handling input data at the expense of generating pre-defined functions for most common encoding and decoding tasks. Users of embedded systems may find this useful as it will shrink the output considerably while allowing them fine control over decoding procedures.

The *Other Options* frame contains miscellaneous modifications to code output, including type name resolution (avoiding duplicate names), date stamp removal (useful when generated code will be stored in source control), and a line item for including any new command-line features not yet represented in the GUI.

XSD Options

If the *Generate equivalent XML schema (XSD) file* option was checked in the *Common Code Generation Options* screen, the following window will be presented for modifying the contents of the generated XSD:



ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp

XSD Generation Options

- [ ] Generate Application Information Annotations
  - [ ] Tags
  - [ ] Enumerations
  - [ ] Extensions
- [ ] Generate Non-native Attribute Annotations
  - [ ] Tags
  - [ ] Enumerations
  - [ ] Extensions

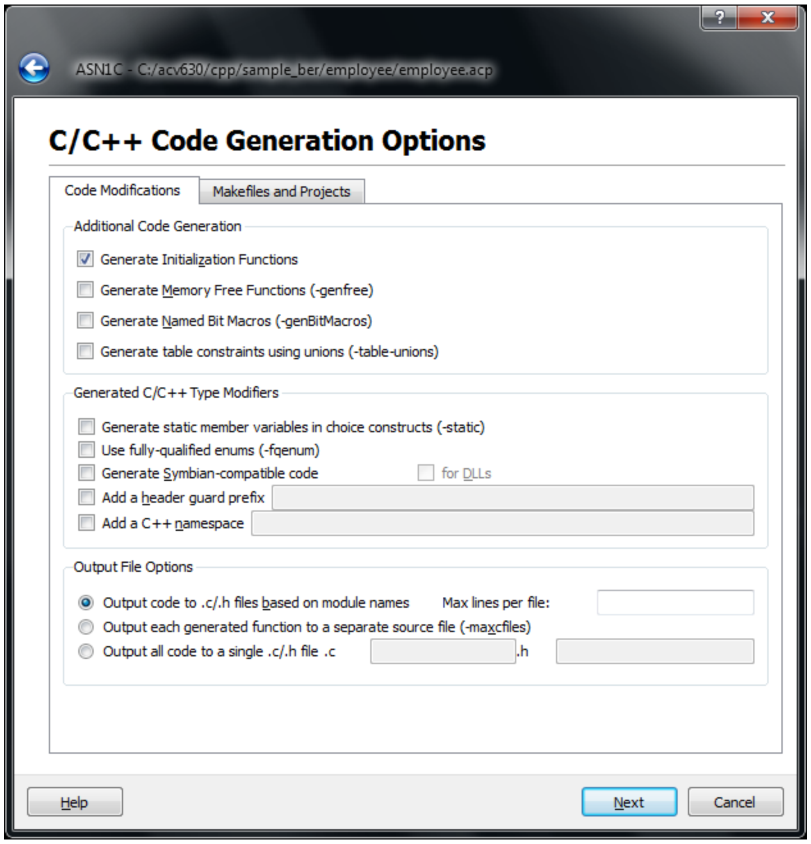
Specify the Target XML Namespace

- [✓] Reference Types in the asn1.xsd schema

These options are described in *Running ASN1C from the command-line*.

C/C++ Code Generation Options

The following windows describe the options available for generating C or C++ source code.



ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp

C/C++ Code Generation Options

Code Modifications | Makefiles and Projects

Additional Code Generation

- [✓] Generate Initialization Functions
- [ ] Generate Memory Free Functions (-genfree)
- [ ] Generate Named Bit Macros (-genBitMacros)
- [ ] Generate table constraints using unions (-table-unions)

Generated C/C++ Type Modifiers

- [ ] Generate static member variables in choice constructs (-static)
- [ ] Use fully-qualified enums (-fqenum)
- [ ] Generate Symbian-compatible code [ ] for DLLs
- [ ] Add a header guard prefix
- [ ] Add a C++ namespace

Output File Options

- (●) Output code to .c/.h files based on module names Max lines per file:
- ( ) Output each generated function to a separate source file (-maxcfiles)
- ( ) Output all code to a single .c/.h file .c .h

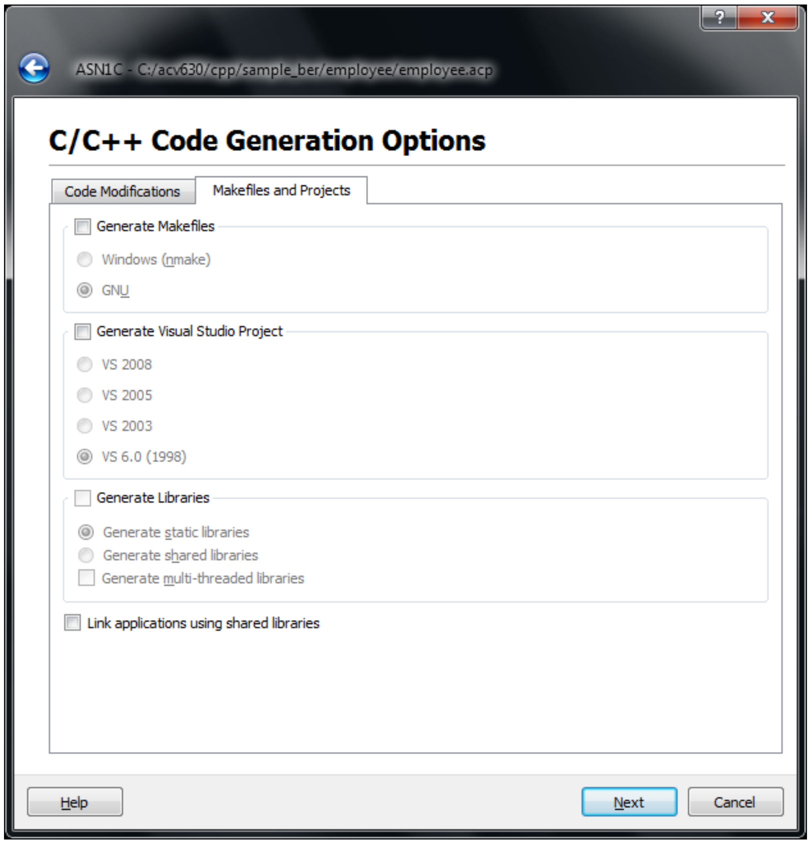
The first tab, *Code Modifications*, contains options for adding C/C++-specific function types, adding type modifiers, and changing how files are output.

By default, ASN1C generates initialization functions that are used by generated code to ensure that newly-constructed types have appropriate default values. Memory free functions are not generated by default, but may be added when users need to free specific types individually instead of relying on ASN1C's built-in memory management functions. Bit macro functions may also be generated for setting, clearing, and testing bits in `BIT STRING` productions.

Table constraints in C/C++ were modified for ASN1C version 6.2 to make their data structures easier to manipulate. The *table unions* option causes a `CHOICE`-like union structure to be generated for information object classes. In practice, the generated code is more compact, easier to read, and confers several advantages for developers (principally type safety and readability).

The *Generate static elements* option is used to add static elements to `CHOICE` constructs instead of pointer values. Using fully-qualified enumerated types will cause ASN1C to emit enumerated types prefixed by their parent module name. Enumerated types are often reused in different modules, and ASN1C's automatic name resolution is usually insufficient to disambiguate which type is which. This option only needs to be selected in C, since C++ enumerations are contained in classes.

The other options are described in greater detail in *Running ASN1C from the command-line*.


ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp

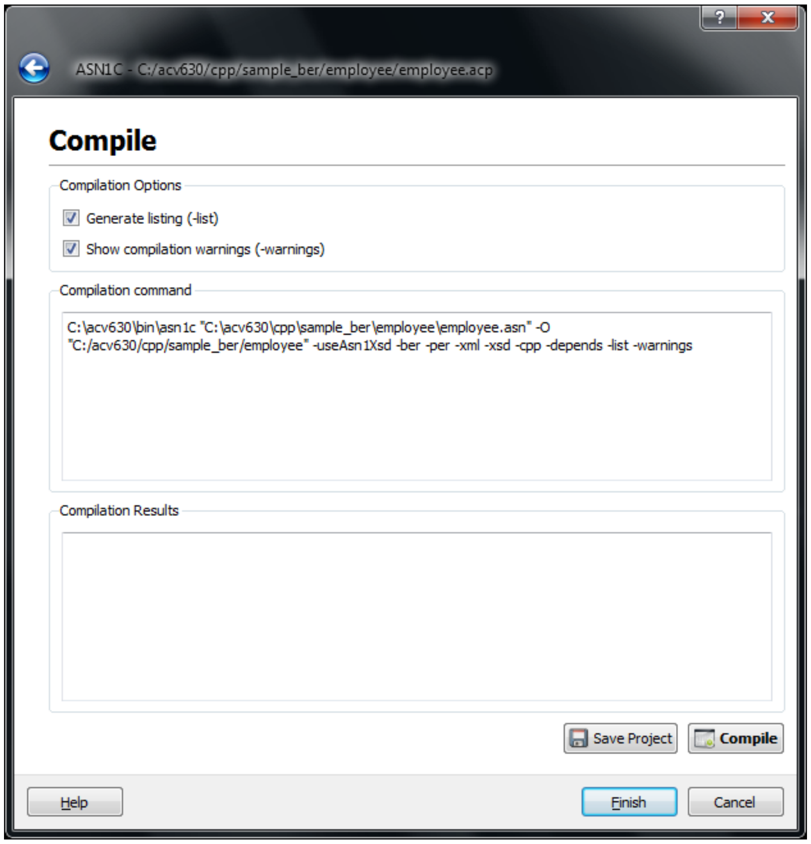
C/C++ Code Generation Options

- [ ] Generate Makefiles
  - ○ Windows (nmake)
  - ● GNU
- [ ] Generate Visual Studio Project
  - ○ VS 2008
  - ○ VS 2005
  - ○ VS 2003
  - ● VS 6.0 (1998)
- [ ] Generate Libraries
  - ● Generate static libraries
  - ○ Generate shared libraries
  - [ ] Generate multi-threaded libraries
- [ ] Link applications using shared libraries

The preceding window contains options for generating makefiles and Visual Studio projects. Makefile and project targets may be modified by selecting *Generate Libraries* or *Link applications using shared libraries*. If the latter option is checked, applications will be dynamically linked instead of statically linked.

Compilation

When all options have been specified, the final screen may be used to execute the compilation command:


ASN1C - C:/acv630/cpp/sample_ber/employee/employee.acp

Compile

Compilation Options

[✓] Generate listing (-list)

[✓] Show compilation warnings (-warnings)

Compilation command

```
C:\acv630\bin\asn1c "C:\acv630\cpp\sample_ber\employee\employee.asn" -O
"C:/acv630/cpp/sample_ber/employee" -useAsn1Xsd -ber -per -xml -xsd -cpp -depends -list -warnings
```

Compilation Results

Included in the window are the compiler command, an option to save the project, and the output from compilation. Selected options are reflected in the command line.

It is also possible to generate a printed listing of the input specifications. Warnings encountered during compilation will also be printed if the appropriate check box is marked.

Click *Finish* to terminate the program. The wizard will ask whether or not to save any changes made, whether a new project has been created or not.

Compiling and Linking Generated Code

C/C++ source code generated by the 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 ASN.1 run-time libraries. It is necessary to include at least one of the encoding rules libraries (asn1ber, asn1per, or asn1xer) as well as the common run-time functions library (asn1rt). See the *ASN1C C/C++ Run-time Reference Manual* for further details on these libraries.

For static linking on Windows systems, the name of the library files are *asn1ber_a.lib*, *asn1per_a.lib*, or *asn1xer_a.lib* for BER/DER/CER, PER, XER, or XML respectively, and *asn1rt_a.lib* for the common run-time components. On UNIX/Linux, the library names are *libasn1ber.a*, *libasn1per.a*, *libasn1xer.a*, *libasn1xml.a* and *libasn1rt.a*. The library files are located in the lib subdirectory. For UNIX, the –L switch should be used to point to the subdirectory path and - lasn1ber, -lasn1per, -lasn1xer, -lasn1xml and/or -lasn1rt 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 for Windows. The following table summarizes what options were used to build each of these variations:

Library Files	**Description**

`asn1rt_a.lib`
`asn1ber_a.lib`
`asn1per_a.lib`
`asn1xer_a.lib`
`asn1xml_a.lib`
	Static single-threaded libraries. These are built without -MT (multithreading) and -MD (dynamic link libraries) options. These are not thread-safe. However, they provide the smallest footprint of the different libraries.

`asn1rt.lib`
`asn1ber.lib`
`asn1per.lib`
`asn1xer.lib`
`asn1xml.lib`
	DLL libraries. These are used to link against the DLL versions of the run-time libraries (*asn1rt.dll*, etc.)

`asn1rtmt_a.lib`
`asn1bermt_a.lib`
`asn1permt_a.lib`
`asn1xermt_a.lib`
`asn1xmlmt_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. (The DLLs are also thread-safe.)
	
	In Visual Studio 2005 and greater, all libraries are multi-threaded by default, so these libraries are not available for those versions.

`asn1rtmd_a.lib`
`asn1bermd_a.lib`
`asn1permd_a.lib`
`asn1xermd_a.lib`
`asn1xmlmd_a.lib`
	DLL-ready multi-threaded libraries. These libraries were built with the –MD option. They allow linking additional object modules in with the ASN1C run-time modules to produce larger DLLs.

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.

Compiling and linking code generated to support the XML encoding rules (XER) is more complex then the other rules because XER requires the use of third-party XML parser software. This requires the use of additional include directories when compiling and libraries when linking. The C++ sample programs that are provided use the EXPAT XML parser (http://expat.sourceforge.net/). All of the necessary include files and binary libraries are included with the distribution for using this parser. If a different parser is to be used, consult the vendor's documentation for compile and link procedures.

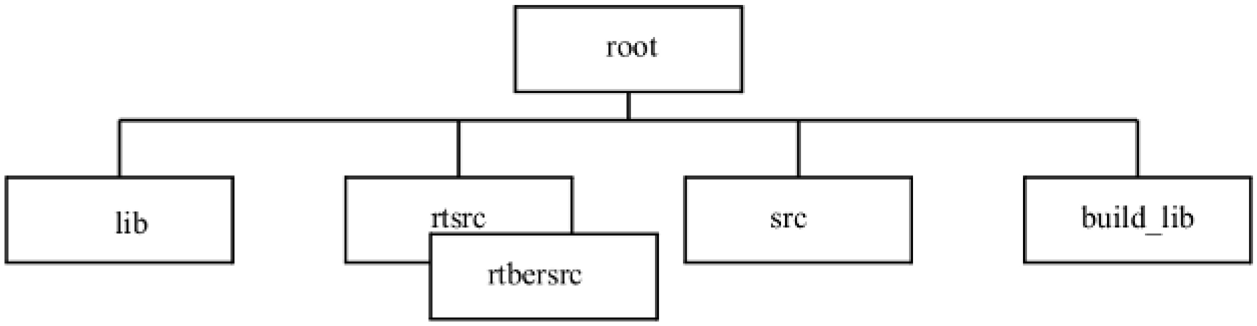
See the makefile in any of the sample subdirectories of the distribution for an example of what must be included to build a program using generated source code.

Porting Run-time Code to Other Platforms

The run-time source version of the compiler includes ANSI-standard source code for the base run-time libraries. This code can be used to build binary versions of the run-time libraries for other operating environments. Included with the source code is a portable makefile that can be used to build the libraries on the target platform with minimal changes. All platform-specific items are isolated in the *platform.mk* file in the root directory of the installation.

The procedure to port the run-time code to a different platform is as follows (note: this assumes common UNIX or GNU compilation utilities are in place on the target platform).

1. Create a directory tree containing a root directory (the name does not matter) and lib, src, rt*src, and build_lib subdirectories (note: in these definitions, * is a wildcard character indicating there are multiple directories matching this pattern). The tree should be as follows:



2. Copy the files ending in extension ".mk" from the root directory of the installation to the root directory of the target platform (note: if transferring from DOS to UNIX or vice-versa, FTP the files in ASCII mode to ensure lines are terminated properly).

3. Copy all files from the src and the different rt*src subdirectories from the installation to the src and rt*src directories on the target platform (note: if transferring from DOS to UNIX or vice-versa, FTP the files in ASCII mode to ensure lines are terminated properly).

4. Copy the makefile from the build_lib subdirectory of the installation to the build_lib subdirectory on the target platform (note: if transferring from DOS to UNIX or vice-versa, FTP the files in ASCII mode to ensure lines are terminated properly).

5. Edit the *platform.mk* file in the root subdirectory and modify the compilation parameters to fit those of the compiler of the target system. In general, the following parameters will need to be adjusted:

   a. cc: C compiler executable name

   b. ccc: C++ compiler executable name

c. `CFLAGS_`: Flags that should be specified on the C or C++ command line

The *platform.w32* and *platform.gnu* files in the root directory of the installation are sample files for Windows 32 (Visual C++) and GNU compilers respectively. Either of these can be renamed to *platform.mk* for building in either of these environments.

6. Invoke the makefile in the build_lib subdirectory.

If all parameters were set up correctly, the result should be binary library files created in the lib subdirectory.

Compiler Configuration File

In addition to command line options, a configuration file can be used to specify compiler options. These options can be applied not only globally but also to specific modules and productions.

A simple form of XML is used to format items in the file. XML was chosen because it is fairly well known and provides a natural interface for representing hierarchical data such as the structure of ASN.1 modules and productions. The use of an external configuration file was chosen over embedding directives within the ASN.1 source itself due to the fact that ASN.1 source versions tend to change frequently. An external configuration file can be reused with a new version of an ASN.1 module, but internal directives would have to be reapplied to the new version of the ASN.1 code.

At the outer level of the markup is the `<asn1config> </asn1config>` tag pair. Within this tag pair, the specification of global items and modules can be made. Global items are applied to all items in all modules. An example would be the `<storage>` qualifier. A storage class such as dynamic can be specified and applied to all productions in all modules. This will cause dynamic storage (pointers) to be used for any embedded structures within all of the generated code to reduce memory consumption demands.

The specification of a module is done using the `<module></module>` tag pair. This tag pair can only be nested within the top-level `<asn1config>` section. The module is identified by using the required `<name></name>` tag pair or by specifying the name as an attribute (for example, `<module name="MyModule">`). Other attributes specified within the `<module>` section apply only to that module and not to other modules specified within the specification. A complete list of all module attributes is provided in the table at the end of this section.

The specification of an individual production is done using the `<production></production>` tag pair. This tag pair can only be nested within a `<module>` section. The production is identified by using the required <name></name> tag pair or by specifying the name as an attribute (for example, `<production name="MyProd">`). Other attributes within the production section apply only to the referenced production and nothing else. A complete list of attributes that can be applied to individual productions is provided in the table at the end of this section.

When an attribute is specified in more than one section, the most specific application is always used. For example, assume a `<typePrefix>` qualifier is used within a module specification to specify a prefix for all generated types in the module and another one is used to a specify a prefix for a single production. The production with the type prefix will be generated with the type prefix assigned to it and all other generated types will contain the type prefix assigned at the module level.

Values in the different sections can be specified in one of the following ways:

1. Using the `<name>value</name>` form. This assigns the given value to the given name. For example, the following would be used to specify the name of the "H323-MESSAGES" module in a module section:

   ```
   <name>H323-MESSAGES</name>
   ```

2. Flag variables that turn some attribute on or off would be specified using a single `<name/>` entry. For example, to specify a given production is a PDU, the following would be specified in a production section:

   ```
   <isPDU/>
   ```

3. An attribute list can be associated with some items. This is normally used as a shorthand form for specifying lists of names. For example, to specify a list of type names to be included in the generated code for a particular module, the following would be used:

   ```
   <include types="TypeName1,TypeName2,TypeName3"/>
   ```

The following are some examples of configuration specifications:

```
<asn1config><storage>dynamic</storage></asn1config>
```

This specification indicates dynamic storage should be used in all places where its use would result in significant memory usage savings within all modules in the specified source file.

```
<asn1config>
   <module>
      <name>H323-MESSAGES</name>
      <sourceFile>h225.asn</sourceFile>
      <typePrefix>H225</typePrefix>
   </module>
   ...
</asn1config>
```

This specification applies to module ‘H323-MESSAGES’ in the source file being processed. For IMPORT statements involving this module, it indicates that the source file ‘h225.asn’ should be searched for specifications. It also indicates that when C or C++ types are generated, they should be prefixed with ‘H225’. This can help prevent name clashes if one or more modules are involved and they contain productions with common names.

The following tables specify the list of attributes that can be applied at all of the different levels: global, module, and individual production:

Global Level

These attributes can be applied at the global level by including them within the `<asn1config>` section:

Name	**Values**	**Description**
<events></events>	*defaultValue* keyword.	This configuration item is for use with Event Handling as described in a later section in this document. It is used to include a special event that is fired when a PER message is being parsed. This event occurs at the location a value should be present in the message but is not and a default value has been specified in the ASN.1 file for the element. In this case, the normal event sequence (startElement, contents, endElement) is executed using the default value.
<rootdir></rootdir>	<ASN1C root directory>	This configuration item is used to specify the root directory of the ASN1C installation for makefile or Visual Studio project generation. It is only needed if generation of these items is done outside of the ASN1C installation.
<storage></storage>	*dynamic*, *static*, *list*, *array*, or *dynamicArray* keyword.	If *dynamic* , it indicates that dynamic storage (i.e., pointers) should be used everywhere within the generated types where use could result in lower memory consumption. These places include the array element for sized SEQUENCE OF/SET OF types and all alternative elements within CHOICE constructs. If *static*, it indicates static types should be used in these places. In general, static types are easier to work with. If *list*, a linked-list type will be used for SEQUENCE OF/SET OF constructs instead of an array type. If *array*, an array type will be used for SEQUENCE OF/SET OF constructs. The maxSize attribute can be used in this case to specify the size of the array variable (for example, <storage maxSize="12"> array </storage>). If *dynamicArray*, a dynamic array will be used for SEQUENCE OF/SET OF constructs. A dynamic array is an array that uses dynamic storage for the array elements.

Module Level

These attributes can be applied at the module level by including them within a `<module>` section:

Name	**Values**	**Description**
<name> </name>	module name	This attribute identifies the module to which this section applies. Either this or the <oid> element/attribute is required.
<oid>	module OID (object identifier)	This attribute provides for an alternate form of module identification for the case when module name is not unique. For example, a given ASN.1 module may have multiple versions. A unique version of the module can be identified using the OID value.
<codename> </codename>	C/C++, Java, or C# name	This item specifies an alternate name for the module to be used in generated code. By default, the module name is used in the form it appears in the ASN.1 specification with hyphens converted to underscores.
<include types="names" values="names"/>	ASN.1 type or value names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be included in the generated code. By default, the compiler generates code for all types and values within a specification. This allows the user to reduce the size of the generated code base by selecting only a subset of the types/values in a specification for compilation. Note that if a type or value is included that has dependent types or values (for example, the element types in a SEQUENCE, SET, or CHOICE), all of the dependent types will be automatically included as well.
<include encoders="names"/>	ASN.1 type names specified as an attribute list.	This item allows a list of ASN.1 types to be included in the generated code for which only encode functions will be generated.
<include decoders="names"/>	ASN.1 type names specified as an attribute list.	This item allows a list of ASN.1 types to be included in the generated code for which only decode functions will be generated.
<include memfree="names"/>	ASN.1 type names specified as an attribute list.	This item allows a list of ASN.1 types to be included in the generated code for which only memory free functions will be generated.
<include importsFrom= "name"/>	ASN.1 module name(s) specified as an attribute list.	This form of the include directive tells the compiler to only include types and/or values in the generated code that are imported by the given module(s).

Name	**Values**	**Description**
<exclude types="names" values="names"/>	ASN.1 type or values names are specified as an attribute list	This item allows a list of ASN.1 types and/or values to be excluded in the generated code. By default, the compiler generates code for all types and values within a specification. This is generally not as useful as in *include* directive because most types in a specification are referenced by other types. If an attempt is made to exclude a type or value referenced by another item, the directive will be ignored.
<storage> </storage>	*dynamic*, *static*, *list*, *array*, or *dynamicArray* keyword.	The definition is the same as for the global case except that the specified storage type will only be applied to generated C and C++ types from the given module.
<sourceFile> </sourceFile>	source file name	Indicates the given module is contained within the given ASN.1 source file. This is used on IMPORTs to instruct the compiler where to look for imported definitions.
<prefix> </prefix>	prefix text	This is used to specify a general prefix that will be applied to all generated C and C++ names (note: for C++ types, the prefix is applied after the standard 'ASN1T_' prefix). This can be used to prevent name clashes if multiple modules are involved in a compilation and they all contain common names.
<typePrefix> </typePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated C and C++ typedef names (note: for C++, the prefix is applied after the standard 'ASN1T_' prefix). This can be used to prevent name clashes if multiple modules are involved in a compilation and they all contain common names.
<enumPrefix> </enumPrefix>	prefix text	This is used to specify a prefix that will be applied to all generated enumerated identifiers within a module. This can be used to prevent name clashes if multiple modules are involved in a compilation. (note: this attribute is normally not needed for C++ enumerated identifiers because they are already wrapped in a structure to allows the type name to be used as an additional identifier).

Name	**Values**	**Description**
<valuePrefix> </valuePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated value constants within a module. This can be used to prevent name clashes if multiple modules are involved that use a common name for two or more different value declarations.
<classPrefix> </classPrefix>	prefix text	This is used to specify a prefix that will be applied to all generated items in a module derived from an ASN.1 CLASS definition.
<objectPrefix> </objectPrefix>	prefix text	This is used to specify a prefix that will be applied to all generated items in a module derived from an ASN.1 Information Object definition.
<objectsetPrefix> </objectsetPrefix>	prefix text	This is used to specify a prefix that will be applied to all generated items in a module derived from an ASN.1 Information Object Set definition.
<noPDU/>	n/a	Indicates that this module contains no PDU definitions. This is normally true in modules that are imported to get common type definitions (for example, InformationFramework). This will prevent the C++ version of the compiler from generating any control class definitions for the types in the module.
<intCType>	byte, int16, uint16, int32, uint32, int64, string	This is used to specify a specific C integer type be used for all unconstrained integer types. By default, ASN1C will use the int32 (32-bit integer) type for all unconstrained integers.
<arcCType>	int32, int64	The is used to specify a specific C integer type be used for the arc types in Object Identifier definitions. By default, int32 (32-bit integer arc values) are generated.
<namespace> </namespace>	namespace URI	This is used to specify the target namespace for the given module when generating XSD and/or XML code. By default, the compiler will not include a targetNamespace directive in the generated XSD code (i.e. all items will not be assigned to any namespace). This option only has meaning when used with the - xml / -xsd command line options.
<hFile> </hFile>	C/C++ header filename	This is used to specify the name of a C/C++ header file to be used to store generated defini-

Name	Values	Description
		tions for the module. By default, the header file name is set to the ASN.1 name of the module with '.h' appended to the end.
<alias asn1name="name" codename="name"/>	ASN.1 to computer language name mapping	This item allows a name in the ASN.1 specification being compiled to be mapped to an alternate name in the generated computer language files. The primary use is to allow shorter names to be used in places where a combination of names may be very long. In this release, the only names that can be used in the alias statement are information object set names.

Production Level

These attributes can be applied at the production level by including them within a `<production>` section:

Name	Values	Description
<name> </name>	production name	This attribute identifies the production (type) to which this section applies. It is required.
<ctype>	byte, int16, uint16, int32, uint32, int64, string, chararray	This is used to specify a specific C integer or character string type be used in place of the default definition generated by ASN1C. In the case of integers, ASN1C will normally try and use the smallest integer type available based on the value or value range constraint on the integer type. If the integer is not constrained, the int32 (32-bit integer) type will be used. For character string, ASN1C will use a character string pointer (char*) by default. The 'chararray' item can be used on strings with size constrains to specify a static character array variable be used.
<enumPrefix> </enumPrefix>	prefix text	This is used to specify a prefix that will be applied to all generated enumerated identifiers within a module. This can be used to prevent name clashes if multiple modules are involved in a compilation. (note: this attribute is normally not needed for C++ enumerated identifiers because they are already wrapped in a structure to allows the type name to be used as an additional identifier).

Name	Values	Description
<format> </format>	base64, hex, xmllist	This is used to set format options specific to XER encoding. The base64 or hex alternative is used to set the output format that binary data in OCTET STRING variables is displayed in in XML markup. The xmllist alternative is used with SEQUENCE OF or SET OF types to denote that items should be displayed in XML space-separated list format as opposed to a using a separate element for each list item.
<isBigInteger/>	n/a	This is a flag variable (an ‘empty element’ in XML terminology) that specifies that this production will be used to store an integer larger than the C or C++ int type on the given system (normally 32 bits). A C string type (char*) will be used to hold a textual representation of the value. This qualifier can be applied to either an integer or constructed type. If constructed, all integer elements within the constructed type are flagged as big integers.
<isPDU/>	n/a	This is a flag variable that specifies that this production represents a Protocol Data Unit (PDU). This is defined as a production that will be encoded or decoded from within the application code. This attribute only makes a difference in the generation of C++ classes. Control classes that are only used in the application code are only generated for types with this attribute set.
<storage> </storage>	*dynamic*, *static*, *list*, *array*, or *dynamicArray* keyword.	The definition is the same as for the global case except that the specified storage type will only be applied to the generated C or C++ type for the given production.
<typePrefix> </typePrefix>	prefix text	This is used to specify a prefix that will be applied to all generated C and C++ typedef names (note: for C++, the prefix is applied after the standard ‘ASN1T_’ prefix). This can be used to prevent name clashes if multiple modules are involved in a compilation and they all contain common names.

Element Level

These attributes can be applied at the element level by including them within an `<element>` section:

Name	**Values**	**Description**
<name> </name>	element name	This attribute identifies the element within a SEQUENCE, SET, or CHOICE construct to which this section applies. It is required.
<ctype>	chararray	This is used to specify a specific C type be used in place of the default definition generated by ASN1C. In the case of elements, the only supported customization is for character string types which would normally be represented by a character pointer type (char*) to be changed to use static character arrays. This can only be done if the string type contains a size constraint.
<isOpenType/>	n/a	This flag variable specifies that this element will be decoded as an open type (i.e. skipped). Refer to the section on deferred decoding for further information. Note that this variable can only be used with BER, CER, or DER encoding rules.
<notUsed/>	n/a	This flag variable specifies that this element will not be used at all in the generated code. It can only be applied to optional elements within a SEQUENCE or SET, or to elements within a CHOICE. Its purpose is for production of more compact code by allowing users to configure out items that are of no interest to them.
<perEncoding> </perEncoding>	hex data	This variable allows a user to substitute a known binary PER encoding for the given element. This encoding will be inserted into the encoded data stream on encoding and skipped over on decoding. Its purpose is the production of more compact and faster code for PER by bypassing runtime calculations needed to encode or decode variable data.
<storage> </storage>	*dynamic*, *static*, *list*, *array*, or *dynamicArray* keyword.	The definition is the same as for the global case except that the specified storage type will only be applied to the generated C or C++ type for this element.

Compiler Error Reporting

Errors that can occur when generating source code from an ASN.1 source specification take two forms: syntax errors and semantics errors.

Syntax errors are errors in the ASN.1 source specification itself. These occur when the rules specified in the ASN.1 grammar are not followed. ASN1C will flag these types of errors with the error message ‘Syntax Error’ and abort compilation on the source file. The offending line number will be provided. The user can re-run the compilation with the ‘-l’ flag specified to see the lines listed as they are parsed. This can be quite helpful in tracking down a syntax error.

The most common types of syntax errors are as follows:

- Invalid case on identifiers: module name must begin with an uppercase letter, productions (types) must begin with an uppercase letter, and element names within constructors (SEQUENCE, SET, CHOICE) must begin with lowercase letters.

- Elements within constructors not properly delimited with commas: either a comma is omitted at the end of an element declaration, or an extra comma is added at the end of an element declaration before the closing brace.

- Invalid special characters: only letters, numbers, and the hyphen (-) character are allowed. The use of the underscore character (_) in identifiers is not allowed in ASN.1, but is allowed in C. Since C does not allow hyphens in identifiers, ASN1C converts all hyphens in an ASN.1 specification to underscore characters in the generated code.

Semantics errors occur on the compiler back-end as the code is being generated. In this case, parsing was successful, but the compiler does not know how to generate the code. These errors are flagged by embedding error messages directly in the generated code. The error messages always begin with an identifier with the prefix ‘%ASN-’, so a search can be done for this string in order to find the locations of the errors. A single error message is output to stderr after compilation on the unit is complete to indicate error conditions exist.

ASN.1 To C/C++ Mappings

Type Mappings

BOOLEAN

The ASN.1 BOOLEAN type is converted into a C type named *OSBOOL*. In the global include file *osSysTypes.h*, *OSBOOL* is defined to be an `unsigned char`.

ASN.1 production:

```
<name> ::= BOOLEAN
```

Generated C code:

```
typedef OSBOOL <name>;
```

Generated C++ code:

```
typedef OSBOOL ASN1T_<name>;
```

For example, if `B ::= [PRIVATE 10] BOOLEAN` was defined as an ASN.1 production, the generated C type definition would be `typedef OSBOOL B`. Note that the tag information is not represented in the type definition. It is handled within the generated encode/decode functions.

The only difference between the C and C++ mapping is the addition of the `ASN1T_` prefix on the C++ type.

INTEGER

The ASN.1 INTEGER type is converted into one of several different C types depending on constraints specified on the type. By default, an INTEGER with no constraints results in the generation of an OSINT32 type. In the global include file *osSysTypes.h*, `OSINT32` is defined to be an `int` which is normally a signed 32-bit integer value on most computer systems.

ASN.1 production:

```
<name> ::= INTEGER
```

Generated C code:

```
typedef OSINT32 <name>;
```

Generated C++ code:

```
typedef OSINT32 ASN1T_<name>;
```

Value range constraints can be used to alter the C type used to represent a given integer value. For example, the following declaration from the SNMP SMI specification would cause an OSUINT32 type (mapped to a C unsigned int) to be used:

```
Counter ::= [APPLICATION 1] IMPLICIT INTEGER (0..4294967295)
```

In this case, an `OSINT32` could not be used because all values within the given range could not be represented. 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	ASN1C Type	C Type
-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)

The C type that is used to represent a given integer value can also be altered using the "<ctype>" configuration variable setting. This allows any of the integer types above to be used for a given integer type as well as a 64-bit integer type. The values that can be used with <ctype> are: byte, int16, uint16, int32, uint32, and int64. An example of using this setting is as follows:

Suppose you have the following integer declaration in your ASN.1 source file:

```
MyIntType ::= [APPLICATION 1] INTEGER
```

You could then have ASN1C use a 64-bit integer type for this integer by adding the following declaration to a configuration file to be associated with this module:

```
<production>
   <name>MyIntType</name>
   <intCType>int64</intCType>
</production>
```

The `<intCType>` setting is also available at the module level to specify that the given C integer type be used for all unconstrained integers within the module.

Large Integer Support

In C and C++, the maximum size for an integer type is normally 64 bits (or 32 bits on some older platforms). ASN.1 has no such limitation on integer sizes and some applications (security key values for example) demand larger sizes. In order to accommodate these types of applications, the ASN1C compiler allows an integer to be declared a "big integer" via a configuration file variable (the <isBigInteger/> setting is used to do this - see the section describing the configuration file for full details). When the compiler detects this setting, it will declare the integer to be a character string variable instead of a C int or unsigned int type. The character string would then be populated with a character string representation of the value to be encoded. Supported character string representations are hexadecimal (strings starting with 0x), octal (strings starting with 0o) and decimal (no prefix).

For example, the following INTEGER type might be declared in the ASN.1 source file:

```
SecurityKeyType ::= [APPLICATION 2] INTEGER
```

Then, in a configuration file used with the ASN.1 definition above, the following declaration can be made:

```
<production>
   <name>SecurityKeyType</name>
   <isBigInteger/>
</production>
```

This will cause the compiler to generate the following type declaration:

```
typedef const char* SecurityKeyType
```

The SecurityKeyType variable can now be populated with a hexadecimal string for encoding such as the following:

```
SecurityKeyType secKey = "0xfd09874da875cc90240087cd12fd";
```

Note that in this definition the `0x` prefix is required to identify the string as containing hexadecimal characters.

On the decode side, the decoder will populate the variable with the same type of character string after decoding.

There are also a number of run-time functions available for big integer support. This set of functions provides an arbitrary length integer math package that can be used to perform mathematical operations as well as convert values into various string forms. See the *ASN1C C/C++ Common Run-time User's Manual* for a description of these functions.

BIT STRING

The ASN.1 BIT STRING type is converted into a C or C++ structured type containing an integer to hold the number of bits and an array of unsigned characters ("OCTETs") to hold the bit string contents. The number of bits integer specifies the actual number of bits used in the bit string and takes into account any unused bits in the last byte.

The type definition of the contents field depends on how the bit string is specified in the ASN.1 definition. If a size constraint is used, a static array is generated; otherwise, a pointer variable is generated to hold a dynamically allocated string. The decoder will automatically allocate memory to hold a parsed string based on the received length of the string.

In the static case, the length of the character array is determined by adjusting the given size value (which represents the number of bits) into the number of bytes required to hold the bits.

Dynamic Bit String

ASN.1 production:

```
<name> ::= BIT STRING
```

Generated C code:

```
typedef ASN1DynBitStr <name>;
```

Generated C++ code:

```
typedef ASN1TDynBitStr ASN1T_<name>;
```

In this case, different base types are used for C and C++. The difference between the two is the C ++ version includes constructors that initialize the value and methods for setting the value.

The *ASN1DynBitStr* type (i.e., the type used in the C mapping) is defined in the *asn1type.h* header file as follows:

```
typedef struct ASN1DynBitStr {
   OSUINT32 numbits;
   const OSOCTET* data;
} ASN1DynBitStr;
```

The *ASN1TDynBitStr* type is defined in the *asn1CppTypes.h* header file as follows:

```
struct ASN1TDynBitStr : public ASN1DynBitStr {
   // ctors
   ASN1TDynBitStr () : numbits(0) {}
   ASN1TDynBitStr (OSUINT32 _numbits, OSOCTET* _data);
   ASN1TDynBitStr (ASN1DynBitStr& _bs);
} ASN1TDynBitStr;
```

Note that memory management of the byte array containing the bit string data is the responsibility of the user. The wrapper class does not free the memory on destruction nor deep-copy the data when a string is copied.

Static (sized) BIT STRING

ASN.1 production:

```
<name> ::= BIT STRING (SIZE (<len>))
```

Generated C code:

```
typedef struct {
   OSUINT32 numbits;
```

```
      OSOCTET data[<adjusted_len>*];
   } <name>;
```

Generated C++ code:

```
   typedef struct <name> {
      OSUINT32 numbits;
      OSOCTET data[<adjusted_len>*];
      // ctors
      ASN1T_<name> ();
      ASN1T_<name> (OSUINT32 _numbits, const OSOCTET* _data);
   } ASN1T_<name>;

   * <adjusted_len> = ((<len> - 1)/8) + 1;
```

For example, the following ASN.1 production:

```
   BS ::= [PRIVATE 220] BIT STRING (SIZE (42))
```

Would translate to the following C typedef:

```
   typedef struct BS {
      OSUINT32 numbits;
      OSOCTET data[6];
   } BS;
```

In this case, six octets would be required to hold the 42 bits: eight in the first five bytes, and two in the last byte.

In the case of small-sized strings (less than or equal to 32 bits), a built-in type is used rather than generating a custom type. This built-in type is defined as follows:

```
   typedef struct ASN1BitStr32 {
      OSUINT32 numbits;
      OSOCTET data[4];
   } ASN1BitStr32;
```

The C++ variant (ASN1TBitStr32) adds constructors for initialization and copying.

Note that for C++, ASN1C generates special constructors and assignment operators to make populating a structure easier. In this case, two constructors were generated: a default constructor and one that takes numbits and data as arguments.

Named Bits

In the ASN.1 standard, it is possible to define an enumerated bit string that specifies named constants for different bit positions. ASN1C provides support for this type by generating symbolic constants and optional macros that can be used to set, clear, or test these named bits. These symbolic constants equate the bit name to the bit number defined in the specification. They can be used with the *rtBitSet*, *rtBitClear*, and *rtBitTest* run-time functions to set, clear, and test the named bits. In addition, generated C++ code contains an enumerated constant added to the control class with an entry for each of the bit numbers. These entries can be used in calls to the methods of the *ASN1CBitStr* class to set, clear, and test bits.

The *-genBitMacros* command line option can be used to generate macros to set, clear, or test the named bits in a bit string structure. These macros offer better performance then using the run-time functions because all calculations of mask and index values are done at compile time. However, they can result in a large amount of additional generated code.

For example, the following ASN.1 production:

```
NamedBS ::= BIT STRING { bitOne(1), bitTen(10) }
```

Would translate to the following if *-genBitMacros* was specified:

```
/* Named bit constants */

#define NamedBS_bitOne      1

#define SET_BS3_bitOne(bs) \
<code to set bit..>

#define CLEAR_BS3_bitOne(bs) \
<code to clear bit..>

#define TEST_BS3_bitOne(bs) \
<code to test bit..>

#define NamedBS_bitTen      10

#define SET_BS3_bitTen(bs) \
<code to set bit..>

#define CLEAR_BS3_bitTen(bs) \
<code to clear bit..>

#define TEST_BS3_bitTen(bs) \
<code to test bit..>

/* Type definitions */

typedef struct ASN1T_NamedBS {
   OSUINT32 numbits;
   OSOCTET data[2];
} NamedBS;
```

The named bit constants would be used to access the data array within the *ASN1T_NamedBS* type. If bit macros were not generated, the *rtxSetBit* function could be used to set the named bit *bitOne* with the following code:

```
NamedBS bs;
memset (&bs, 0, sizeof(bs));
rtxSetBit (bs.data, 10, NamedBS_bitOne);
```

The statement to clear the bit using *rtxClearBit* would be as follows:

```
rtxClearBit (bs.data, 10, NamedBS_bitOne);
```

Finally, the bit could be tested using *rtxTestBit* with the following statement:

```
if (rtxTestBit (bs.data, 10, NamedBS_bitOne) {
      ... bit is set
   }
```

Note that the compiler generated a fixed length data array for this specification. It did this because the maximum size of the string is known due to the named bits - it must only be large enough to hold the maximum valued named bit constant.

Contents Constraint

It is possible to specify a contents constraint on a BIT STRING type using the CONTAINING keyword. This indicates that the encoded contents of the specified type should be packed within the BIT STRING container. An example of this type of constraint is as follows:

```
ContainingBS ::= BIT STRING (CONTAINING INTEGER)
```

ASN1C will generate a type definition that references the type that is within the containing constraint. In this case, that would be INTEGER; therefore, the generated type definition would be as follows:

```
typedef OSINT32 ContainingBS;
```

The generated encoders and decoders would handle the extra packing and unpacking required to get this to and from a BIT STRING container. This direct use of the containing type can be suppressed through the use of the *-noContaining* command-line argument. In this case, a normal BIT STRING type will be used and it will be the users responsibility to do the necessary packing and unpacking operations to encode and decode the variable correctly.

ASN1CBitStr Control Class

When C++ code generation is specified, a control class is generated for operating on the target bit string. This class is derived from the ASN1CBitStr class. This class contains methods for operating on bits within the string.

Objects of this class can also be declared inline to make operating on bits within other ASN.1 constructs easier. For example, in a SEQUENCE containing a bit string element the generated type will contain a public member variable containing the ASN1T type that holds the message data. If one wanted to operate on the bit string contained within that element, they could do so by using the ASN1CBitStr class inline as follows:

```
ASN1CBitStr bs (<seqVar>.<element>);
bs.set (0);
```

In this example, <seqVar> would represent a generated SEQUENCE variable type and <element> would represent a bit string element within this type.

See the section on the *ASN1CBitStr* class in the *ASN1C C/C++ Common Run-time User's Manual* for details on all of the methods available in this class.

OCTET STRING

The ASN.1 OCTET STRING type is converted into a C structured type containing an integer to hold the number of octets and an array of unsigned characters (OCTETs) to hold the octet string contents. The number of octets integer specifies the actual number of octets in the contents field.

The allocation for the contents field depends on how the octet string is specified in the ASN.1 definition. If a size constraint is used, a static array of that size is generated; otherwise, a pointer variable is generated to hold a dynamically allocated string. The decoder will automatically allocate memory to hold a parsed string based on the received length of the string.

For C++, constructors and assignment operators are generated to make assigning variables to the structures easier. In addition to the default constructor, a constructor is provided for string or binary data. An assignment operator is generated for direct assignment of a null-terminated string to the structure (note: this assignment operator copies the null terminator at the end of the string to the data).

Dynamic OCTET STRING

ASN.1 production:

```
   <name> ::= OCTET STRING
```

Generated C code:

```
   typedef ASN1DynOctStr <name>;
```

Generated C++ code:

```
   typedef ASN1TDynOctStr ASN1T_<name>;
```

In this case, different base types are used for C and C++. The difference between the two is the C++ version includes constructors, assignment operators, and other helper methods that make it easier to manipulate binary data.

The *ASN1DynOctStr* type (i.e., the type used in the C mapping) is defined in the *asn1type.h* header file as follows:

```
typedef struct ASN1DynOctStr {
   OSUINT32 numocts;
   const OSOCTET* data;
} ASN1DynOctStr;
```

The *ASN1TDynOctStr* type is defined in the *ASN1TOctStr.h* header file. This class extends the C *ASN1DynOctStr* class and adds many additional constructors and methods. See the *C/C++ Common Run-time Reference Manual* for a complete description of this class.

Static (sized) OCTET STRING

ASN.1 production:

```
<name> ::= OCTET STRING (SIZE (<len>))
```

Generated C code:

```
typedef struct {
   OSUINT32 numocts;
   OSOCTET data[<len>];
} <name>;
```

Generated C++ code:

```
typedef struct {
   OSUINT32 numocts;
   OSOCTET data[<len>];
   // ctors
   ASN1T_<name> ();
   ASN1T_<name> (OSUINT32 _numocts,
                 const OSOCTET* _data);
   ASN1T_<name> (const char* cstring);
   // assignment operators
   ASN1T_<name>& operator= (const char* cstring);
} ASN1T_<name>;
```

Contents Constraint

It is possible to specify a contents constraint on an OCTET STRING type using the CONTAINING keyword. This indicates that the encoded contents of the specified type should be packed within the OCTET STRING container. An example of this type of constraint is as follows:

```
ContainingOS ::= OCTET STRING (CONTAINING INTEGER)
```

ASN1C will generate a type definition that references the type that is within the containing constraint. In this case, that would be INTEGER; therefore, the generated type definition would be as follows:

```
typedef OSINT32 ContainingOS;
```

The generated encoders and decoders would handle the extra packing and unpacking required to get this to and from an OCTET STRING container. This direct use of the containing type can be suppressed through the use of the *-noContaining* command-line argument. In this case, a normal OCTET STRING type will be used and it will be the users responsibility to do the necessary packing and unpacking operations to encode and decode the variable correctly.

ENUMERATED

The ASN.1 ENUMERATED type is converted into different types depending on whether C or C++ code is being generated. The C mapping is either a C enum or integer type depending on whether or not the ASN.1 type is extensible or not. The C++ mapping adds a struct wrapper around this type to provide a namespace to aid in making the enumerated values unique across all modules.

C Mapping

ASN.1 production:

```
   <name> ::= ENUMERATED (<id1>(<val1>), <id2>(<val2>), ...)
```

Generated code :

```
   typedef enum {
      id1 = val1,
      id2 = val2,
      ...
   } <name>_Root

   typedef OSUINT32 <name>;
```

The compiler will automatically generate a new identifier value if it detects a duplicate within the source specification. The format of this generated identifier is 'id_n' where id is the original identifier and n is a sequential number. The compiler will output an informational message when this is done. This message is only displayed if the -*warnings* qualifier is specified on the command line.

A configuration setting is also available to further disambiguate duplicate enumerated item names. This is the "enum prefix" setting that is available at both the module and production levels. For example, the following would cause the prefix "h225" to be added to all enumerated identifiers within the H225 module:

```
       <module>
          <name>H225</name>
          <enumPrefix>h225</enumPrefix>
       </module>
```

The -*fqenum* (fully-qualified enum) option may also be used to make C names unique. When specified, enumerated identifiers will be automatically prefixed with the enclosing type name. In the specification above, each of the identifiers would have the form "<name>_<id>". This can be useful in situations where common identifiers are often repeated in different types. This is not a problem in C++ because the identifiers are wrapped in a struct declaration which provides a namespace for the values (see the C++ section below for more details).

In addition to the generated type definition, helper functions are also generated to make it easier to convert to/from enumerated and string format. The signatures of these functions are as follows:

```
const OSUTF8CHAR* <name>_ToString (OSINT32 value);
int <name>_ToEnum (OSCTXT* pctxt, const OSUTF8CHAR* value, <name>* pvalue);
```

The first function would be used to convert an enumerated value into string form. The second would do the opposite - convert from string to enumerated.

C++ Mapping

ASN.1 production:

```
   <name> ::= ENUMERATED (<id1>(<val1>), <id2>(<val2>), ...)
```

Generated code :

```
struct <name> {
   enum Root {
      id1 = val1,
      id2 = val2,
      ...
   }
   [ enum Ext {
      extid1 = extval1,
      ...
   } ]
} ;

typedef OSUINT32 ASN1T_<name>
```

The struct type provides a namespace for the enumerated elements. This allows the same enumerated constant names to be used in different productions within the ASN.1 specification. An enumerated item is specified in the code using the <name>::<id> form.

Every generated definition contains a *Root* enumerated specification and, optionally, an *Ext* specification. The *Root* specification contains the root elements of the type (or all of the elements if it is not an extended type), and the *Ext* specification contains the extension enumerated items.

The form of the typedef following the struct specification depends on whether or not the enumerated type contains an extension marker or not. If a marker is present, it means the type can contain values outside the root enumeration. An OSUINT32 is always used in the final typedef to ensure a consistent size of an enumerated variable and to handle the case of unknown extension values.

NULL

The ASN.1 NULL type does not generate an associated C or C++ type definition

OBJECT IDENTIFIER

The ASN.1 OBJECT IDENTIFIER type is converted into a C or C++ structured type to hold the subidentifier values that make up the object identifier.

ASN.1 production:

```
<name> ::= OBJECT IDENTIFIER
```

Generated C code:

```
typedef ASN1OBJID <name>;
```

Generated C++ code:

```
typedef ASN1TObjId ASN1T_<name>;
```

In this case, different base types are used for C and C++. The difference between the two is the C ++ version includes constructors and assignment operators that make setting the value a bit easier.

The *ASN1OBJID* type (i.e., the type used in the C mapping) is defined in *asn1type.h* to be the following:

```
typedef struct {
   OSUINT32 numids; /* number of subidentifiers */
   OSUINT32 subid[ASN_K_MAXSUBIDS];/* subidentifier values */
} ASN1OBJID;
```

The constant *ASN_K_MAXSUBIDS* specifies the maximum number of sub-identifiers that can be assigned to a value of the type. This constant is set to 128 as per the ASN.1 standard.

The *ASN1TObjId* type used in the C++ mapping is defined in *ASN1TObjId.h*. This class extends the C *ASN1OBJID* structure and adds many additional constructors and helper methods. See the *ASN1C C/C++ Common Run-time Reference Manual* for more details.

RELATIVE-OID

The ASN.1 RELATIVE-OID type is converted into a C or C++ structured type that is identical to that of the OBJECT IDENTIFIER described above:

ASN.1 production:

```
<name> ::= RELATIVE-OID
```

Generated C code:

```
typedef ASN1OBJID <name>;
```

Generated C++ code:

```
typedef ASN1TObjId ASN1T_<name>;
```

A RELATIVE-OID is identical to an OBJECT IDENTIFIER except that it does not contain the restriction on the initial two arc values that they fall within a certain range (see the X.680 standard for more details on this).

REAL

The ASN.1 REAL type is mapped to the C type *OSREAL*. In the global include file *osSysTypes.h*, *OSREAL* is defined to be a *double*.

ASN.1 production:

```
ASN.1 production:
```

Generated C code:

```
typedef OSREAL <name>;
```

Generated C++ code:

```
typedef OSREAL ASN1T_<name>;
```

SEQUENCE

This section discusses the mapping of an ASN.1 SEQUENCE type to C. The C++ mapping is similar but there are some differences. These are discussed in the *C++ Mapping of SEQUENCE* subsection at the end of this section.

An ASN.1 SEQUENCE is a constructed type consisting of a series of element definitions. These elements can be of any ASN.1 type including other constructed types. For example, it is possible to nest a SEQUENCE definition within another SEQUENCE definition as follows:

```
A ::= SEQUENCE {
  x  SEQUENCE {
     a1 INTEGER,
     a2 BOOLEAN
  },
  y OCTET STRING (SIZE (10))
}
```

In this example, the production has two elements: `x` and `y`. The nested SEQUENCE x has two additional elements: `a1` and `a2`.

The ASN1C compiler first recursively pulls all of the embedded constructed elements out of the SEQUENCE and forms new internal types. The names of these types are of the form `<name>_<element-name1>_<elementname2>_ ... <element-nameN>`. For example, in the definition above, two temporary types would be generated: `A_x` and `A_y` (`A_y`is generated because a static OCTET STRING maps to a C++ struct type).

The general form is as follows:

ASN.1 production:

```
<name> ::= SEQUENCE {
   <element1-name> <element1-type>,
   <element2-name> <element2-type>,
   ...
}
```

Generated C code:

```
typedef struct {
   <type1> <element1-name>;
   <type2> <element2-name>;
   ...
} <name>;
```

- or -

```
typedef struct {
   ...
} <tempName1>

typedef struct {
   ...
} <tempName2>

typedef struct {
   <tempName1> <element1-name>;
   <tempName2> <element2-name>;
   ...
} <name>;
```

The `<type1>` and `<type2>` placeholders represent the equivalent C types for the ASN.1 types `<element1-type>` and `<element2-type>` respectively. This form of the structure will be generated if the internal types are primitive. `<tempName1>` and `<tempName2>` are formed using the algorithm described above for pulling structured types out of the definition. This form is used for constructed elements and elements that map to structured C types.

The example above would result in the following generated C typedefs:

```
typedef struct A_x {
   OSINT32 a1;
   OSBOOL a2;
} A_x;

typedef struct A_y {
   OSUINT32 numocts;
   OSOCTET data[10];
} A_y;

typedef struct A {
   A_x x;
   A_y y;
} A;
```

In this case, elements `x` and `y` map to structured C types, so temporary typedefs are generated.

In the case of nesting levels greater than two, all of the intermediate element names are used to form the final name. For example, consider the following type definition that contains three nesting levels:

```
X ::= SEQUENCE {
   a SEQUENCE {
      aa SEQUENCE { x INTEGER, y BOOLEAN },
      bb INTEGER
   }
}
```

In this case, the generation of temporary types results in the following equivalent type definitions:

```
X-a-aa ::= SEQUENCE { x INTEGER, y BOOLEAN }

X-a ::= SEQUENCE { aa X-a-aa, bb INTEGER }

X ::= SEQUENCE { X-a a }
```

Note that the name for the `aa` element type is `X-a-aa`. It contains both the name for `a` (at level 1) and `aa` (at level 2). The concatanation of all of the intermdeiate element names can lead to very long names in some cases. To get around the problem, the *-shortnames* command-line option can be used to form shorter names. In this case, only the type name and the last element name are used.

In the example above, this would lead to an element name of `X-aa`. The disadvantage of this is that the names may not always be unique. If using this option results in non-unique names, an `_n` suffix is added where `n` is a sequential number to make the names unique.

Note that although the compiler can handle embedded constructed types within productions, it is generally not considered good style to define productions this way. It is much better to manually define the constructed types for use in the final production definition. For example, the production defined at the start of this section can be rewritten as the following set of productions:

```
X ::= SEQUENCE {
   a1 INTEGER,
   a2 BOOLEAN
}

Y ::= OCTET STRING
A ::= SEQUENCE {
   X x,
   Y y
}
```

This makes the generated code easier to understand for the end user.

Unnamed Elements

Note

As of X.680, unnamed elements are not allowed: elements must be named. ASN1C still provides backward compatibility support for this syntax however.

In an ASN.1 SEQUENCE definition, the <element-name> tokens at the beginning of element declarations are optional. It is possible to include only a type name without a field identifier to define an element. This is normally done with defined type elements, but can be done with built-in types as well. An example of a SEQUENCE with unnamed elements would be as follows:

```
AnInt ::= [PRIVATE 1] INTEGER

Aseq ::= [PRIVATE 2] SEQUENCE {
          x          INTEGER,
          AnInt
}
```

In this case, the first element (x) is named and the second element is unnamed.

ASN1C handles this by generating an element name using the type name with the first character set to lower case. For built-in types, a constant element name is used for each type (for example, *aInt* is used for INTEGER). There is one caveat, however. ASN1C cannot handle multiple unnamed elements in a SEQUENCE or SET with the same type names. Element names must be used in this case to distinguish the elements.

So, for the example above, the generated code would be as follows:

```
typedef OSINT32 AnInt;
typedef struct Aseq {
   OSINT32 x;
   AnInt anInt;
} Aseq;
```

OPTIONAL keyword

Elements within a sequence can be declared to be optional using the OPTIONAL keyword. 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. This structure always has variable name 'm' and contains single-bit elements of the form '<element-name>Present' as follows:

```
struct {
   unsigned <element-name1>Present : 1,
   unsigned <element-name2>Present : 1,
   ...
} m;
```

In this case, the elements included in this construct correspond to only those elements marked as OPTIONAL within the production. If a production contains no optional elements, the entire construct is omitted.

For example, the production in the previous example can be changed to make both elements optional:

```
Aseq ::= [PRIVATE 2] SEQUENCE {
   x      INTEGER OPTIONAL,
          AnInt   OPTIONAL
}
```

In this case, the following C typedef is generated:

```
typedef struct Aseq {
   struct {
      unsigned xPresent : 1,
      unsigned anIntPresent : 1
   } m;
   OSINT32   x;
   AnInt     anInt;
} Aseq;
```

When this structure is populated for encoding, the developer must set the *xPresent* and *anIntPresent* flags accordingly to indicate whether the elements are to be included in the encoded message or not. Conversely, when a message is decoded into this structure, the developer must test the flags to determine if the element was provided in the message or not.

The generated C++ structure will contain a constructor if OPTIONAL elements are present. This constructor will set all optional bits to zero when a variable of the structured type is declared. The programmer therefore does not have to be worried about clearing bits for elements that are not used; only with setting bits for the elements that are to be encoded.

DEFAULT keyword

The DEFAULT keyword allows a default value to be specified for elements within the SEQUENCE. ASN1C will parse this specification and treat it as it does an optional element. Note that the value specification is only parsed in simple cases for primitive values. It is up to the programmer to provide the value in complex cases. For BER encoding, a value must be specified be it the default or other value.

For DER or PER, it is a requirement that no value be present in the encoding for the default value. For integer and boolean default values, the compiler automatically generates code to handle this requirement based on the value in the structure. For other values, an optional present flag bit is generated. The programmer must set this bit to false on the encode side to specify default value selected. If this is done, a value is not encoded into the message. On the decode side, the developer must test for present bit not set. If this is the case, the default value specified in the ASN.1 specification must be used and the value in the structure ignored.

Extension Elements

If the SEQUENCE type contains an open extension field (i.e., a ... at the end of the specification or a ..., ... in the middle), a special element will be inserted to capture encoded extension elements for inclusion in the final encoded message. This element will be of type *OSRTDList* and have the name *extElem1*. This is a linked list of open type fields. Each entry in the list is of type *ASN1OpenType*. The fields will contain complete encodings of any extension elements that may have been present in a message when it is decoded. On subsequent encode of the type, the extension fields will be copied into the new message.

The *-noOpenExt* command line option can be used to alter this default behavior. If this option is specified, the *extElem1* element is not included in the generated code and extension data that may be present in a decoded message is simply dropped.

If the SEQUENCE type contains an extension marker and extension elements, then the actual extension elements will be present in addition to the *extElem1* element. These elements will be treated as optional elements whether they were declared that way or not. The reason is because a version 1 message could be received that does not contain the elements.

Additional bits will be generated in the bit mask if version brackets are present. These are groupings of extended elements that typically correspond to a particular version of a protocol. An example would be as follows:

```
TestSequence ::= SEQUENCE {
   item-code    INTEGER (0..254),
   item-name    IA5String (SIZE (3..10)) OPTIONAL,
   ... ! 1,
   urgency      ENUMERATED { normal, high } DEFAULT normal,
   [[ alternate-item-code       INTEGER (0..254),
      alternate-item-name       IA5String (SIZE (3..10)) OPTIONAL
   ]]
}
```

In this case, a special bit flag will be added to the mask structure to indicate the presence or absence of the entire element block. This will be of the form "_v#ExtPresent" where # would be replaced by the sequential version number. In the example above, this number would be three (two would be the version extension number of the urgency field). Therefore, the generated bit mask would be as follows:

```
struct {
   unsigned item_namePresent : 1;
   unsigned urgencyPresent : 1;
   unsigned _v3ExtPresent : 1;
   unsigned alternate_item_namePresent : 1;
} m;
```

In this case, the setting of the *_v3ExtPresent* flag would indicate the presence or absence of the entire version block. Note that it is also possible to have optional items within the block (alternate-item-name).

C++ Mapping of SEQUENCE

The C++ mapping of an ASN.1 SEQUENCE type is very similar to the C mapping. However, there are some important differences:

1. As with all C++ types, the prefix ASN1T_ is added before the typename to distinguish the data class from the control class (the control class contains an ASN1C_ prefix).

2. A default constructor is generated to initialize the structure elements. This constructor will initialize all elements and set any simple default values that may have been specified in the ASN.1 definition.

3. If the *-genCopy* command line switch was specified, a copy constructor will be generated to allow an instance of the data contained within a PDU control class object to be copied.

4. Also if *-genCopy* was specified, a destructor is generated if the type contains dynamic fields. This destructor will free all memory held by the type when the object is deleted or goes out of scope.

SET

The ASN.1 SET type is converted into a C or C++ structured type that is identical to that for SEQUENCE as described in the previous section. The only difference between SEQUENCE and SET is that elements may be transmitted in any order in a SET whereas they must be in the defined order in a SEQUENCE. The only impact this has on ASN1C is in the generated decoder for a SET type.

The decoder must take into account the possibility of out-of-order elements. This is handled by using a loop to parse each element in the message. Each time an item is parsed, an internal mask bit within the decoder is set to indicate the element was received. The complete set of received elements is then checked after the loop is completed to verify all required elements were received.

SEQUENCE OF

The ASN.1 SEQUENCE OF type is converted into one of the following C/C++ types:

- A doubly-linked list structure (OSRTDList for C, or ASN1TSeqOfList, a class derived from OSRTDList, for C++)
- A structure containing an integer count of elements and a pointer to hold an array of the referenced data type (a dynamic array)
- A structure containing an integer count of elements and a fixed-sized array of the referenced data type (a static array)

The linked list option is the default for constructed types. An array is used for a sequence of primitive types. The allocation for the contents field of the array depends on how the SEQUENCE OF is specified in the ASN.1 definition. If a size constraint is used, a static array of that size is generated; otherwise, a pointer variable is generated to hold a dynamically allocated array of values. The decoder will automatically allocate memory to hold parsed SEQUENCE OF data values.

The type used for a given SEQUENCE OF construct can be modified by the use of a configuration item. The <storage> qualifier is used for this purpose. The *dynamicArray* keyword can be used at the global, module, or production level to specify that dynamic memory (i.e., a pointer) is used for the array. The syntax of this qualifier is as follows:

```
<storage>dynamicArray</storage>
```

The *array* keyword is used to specify that a static array is to be generated to hold the data. In this case, if the SEQUENCE OF production does not contain a size constraint, the *maxSize* attribute must be used to specify the maximum size of the array. For example:

```
<storage maxSize="100">array</storage>
```

If *maxSize* is not specified and the ASN.1 production contains no size constraint, then a dynamic array is used.

The *list* keyword can also be used in a similar fashion to specify the use of a linked-linked structure to hold the elements:

```
<storage>list</storage>
```

See the section entitled *Compiler Configuration File* for further details on setting up a configuration file.

Dynamic SEQUENCE OF Type

ASN.1 production:

```
<name> ::= SEQUENCE OF <type>
```

Generated C code:

```
typedef struct {
   OSUINT32 n;
   <type>* elem;
} <name>;
```

Generated C++ code:

```
typedef struct [ : public ASN1TPDU ] {
   OSUINT32 n;
   <type>* elem;
   ASN1T_<name>();
   [~ASN1T_<name>();]
} ASN1T_<name>;
```

Note that parsed values can be accessed from the dynamic data variable just as they would be from a static array variable; i.e., an array subscript can be used (ex: elem[0], elem[1]...).

In the case of C++, a constructor is generated to initialize the element count to zero. If the type represents a PDU type (either by default by not referencing any other types or explicitly via the -*pdu* command-line option), the *ASN1TPDU* base class is extended and a destructor is added. This destructor ensures that memory allocated for elements is freed upon destruction of the object.

Static (sized) SEQUENCE OF Type

ASN.1 production:

```
<name> ::= SEQUENCE (SIZE (<len>)) OF <type>
```

Generated C code:

```
typedef struct {
   OSUINT32 n;
   <type> elem[<len>];
} <name>;
```

Generated C++ code:

```
typedef struct {
   OSUINT32 n;
   <type> elem[<len>];
   } ASN1T_<name>;
```

List-based SEQUENCE OF Type

A doubly-linked list header type (OSRTDList) is used for the type definition if the list storage configuration setting is used (see above). This can be used for either a sized or unsized SEQUENCE OF construct. The generated C or C++ code is as follows:

Generated C code:

```
typedef OSRTDList <name>;
```

Generated C++ code:

```
typedef ASN1TSeqOfList ASN1T_<name>;
```

The type definition of the OSRTDList structure can be found in the osSysTypes.h header file. The common run-time utility functions beginning with the prefix rtxDList are available for initializing and adding elements to the list. See the *C/C++ Common Run-time Reference Manual* for a full description of these functions.

For C++, the *ASN1TSeqOfList* class is used, or, in the case of PDU types, the *ASN1TPDUSeqOfList* class. The *ASN1TSeqOfList* extends the C *OSRTDList* structure and adds constructors and other helper methods. The *ASN1TPDUSeqOfList* is similar except that it also extends the *ASN1TPDU* base class to add additional memory management capabilities needed by PDU types to automatically release memory on destruction. See the *ASN1CSeqOfList* section in the *C/C++ Common Run-time Reference Manual* for details on all of the methods available in this class.

Populating Linked-List Structures

Populating generated list-based SEQUENCE OF structures for the most part requires the use of dynamic memory to allocate elements to be added to the list (note that it is possible to use static elements for this, but this is unusual). The recommended method is to use the built in run-time memory management facilities available within the ASN1C runtime library. This allows all list memory to be freed with one call after encoding is complete.

In the case of C, the *rtxMemAlloc* or *rtxMemAllocType* function would first be used to allocate a record of the element type. This element would then be initialized and populated with data. The *rtxDListAppend* function would then be called to append it to the given list.

For C++, the compiler generates the helper methods *NewElement* and *Append* in the generated control class for a SEQUENCE OF type. An instance of this class can be created using the list element within a generated structure as a parameter. The helper methods can then be used to allocate and initialize an element and then append it to the list after it is populated.

See the `cpp/sample_ber/employee/writer.cpp` file for an example of how these methods are used. In this program, the following logic is used to populate one of the elements in the `children` list for encoding:

```
ASN1T_ChildInformation* pChildInfo;
ASN1C__SeqOfChildInformation listHelper (encodeBuffer, msgData.children);
...
pChildInfo = listHelper.NewElement();
fill_Name (&pChildInfo->name, "Ralph", "T", "Smith");
pChildInfo->dateOfBirth = "19571111";
listHelper.Append (pChildInfo);
```

In this example, `msgData` is an instance of the main PDU class being encoded (`PersonnelRecord`). This object contains an element called `children` which is a linked-list of `ChildInformation` records.

The code snippet illustrates how to use the generated control class for the list to allocate a record, populate it, and append it to the list.

ASN1C also generates helper methods in SEQUENCE, SET, and CHOICE control classes to assist in allocating and adding elements to inline SEQUENCE OF lists. These methods are named `new_<elem>_element` and `append_to_<elem>` where `<elem>` would be replaced with the name of the element they apply to.

Generation of Temporary Types for SEQUENCE OF Elements

As with other constructed types, the `<type>` variable can reference any ASN.1 type, including other ASN.1 constructed types. Therefore, it is possible to have a SEQUENCE OF SEQUENCE, SEQUENCE OF CHOICE, etc.

When a constructed type or type that maps to a C structured type is referenced, a temporary type is generated for use in the final production. The format of this temporary type name is as follows:

```
<prodName>_element
```

In this definition, `<prodName>` refers to the name of the production containing the SEQUENCE OF type.

For example, a simple (and very common) single level nested SEQUENCE OF construct might be as follows:

```
A ::= SEQUENCE OF SEQUENCE { a INTEGER, b BOOLEAN }
```

In this case, a temporary type is generated for the element of the SEQUENCE OF production. This results in the following two equivalent ASN.1 types:

```
A-element ::= SEQUENCE { a INTEGER, b BOOLEAN }
```

```
A ::= SEQUENCE OF A-element
```

These types are then converted into the equivalent C or C++ `typedef`s using the standard mapping that was previously described.

SEQUENCE OF Type Elements in Other Constructed Types

Frequently, a SEQUENCE OF construct is used to define an array of some common type in an element in some other constructed type (for example, a SEQUENCE). An example of this is as follows:

```
SomePDU ::= SEQUENCE {
   addresses SEQUENCE OF AliasAddress,
   ...
}
```

Normally, this would result in the `addresses` element being pulled out and used to create a temporary type with a name equal to `SomePDU-addresses` as follows:

```
SomePDU-addresses ::= SEQUENCE OF AliasAddress

SomePDU ::= SEQUENCE {
   addresses SomePDU-addresses,
   ...
}
```

However, when the SEQUENCE OF element references a simple defined type as above with no additional tagging or constraint information, an optimization is done to reduce the size of the generated code. This optimization is to generate a common name for the new temporary type that can be used for other similar references. The form of this common name is as follows:

```
_SeqOf<elementProdName>
```

So instead of this:

```
SomePDU-addresses ::= SEQUENCE OF AliasAddress
```

The following equivalent type would be generated:

```
_SeqOfAliasAddress ::= SEQUENCE OF AliasAddress
```

The advantage is that the new type can now be easily reused if SEQUENCE OF *AliasAddress* is used in any other element declarations. Note the (illegal) use of an underscore in the first position. This is to ensure that no name collisions occur with other ASN.1 productions defined within the specification.

Some SEQUENCE OF elements in constructed types are inlined. In other words, no temporary type is created; instead, either the `OSRTDList` reference (for linked list) or the array definition is inserted directly into the generated C structure. This is particularly true when XSD files are being compiled.

SET OF

The ASN.1 SET OF type is converted into a C or C++ structured type that is identical to that for SEQUENCE OF as described in the previous section.

CHOICE

The ASN.1 CHOICE type 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. A `#define` constant is generated for each of these values. The format of this constant is

`T_<name>_<element-name>` where `<name>` is the name of the ASN.1 production and `<element-name>` is the name of the CHOICE alternative. If a CHOICE alternative is not given an explicit name, then `<element-name>` is automatically generated by taking the type name and making the first letter lowercase (this is the same as was done for the ASN.1 SEQUENCE type with unnamed elements). If the generated name is not unique, a sequential number is appended to make it unique.

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. Unnamed elements names are automatically generated from the type name by making the first character of the name lowercase.

One difference between temporary types used in a SEQUENCE and in a CHOICE is that a pointer variable will be generated for use within the CHOICE union construct.

ASN.1 production:

```
<name> ::= CHOICE {
   <element1-name> <element1-type>,
   <element2-name> <element2-type>,
   ...
}
```

Generated C code:

```
#define T_<name>_<element1-name> 1
#define T_<name>_<element2-name> 2
...

typedef struct {
   int       t;
   union {
      <type1> <element1-name>;
      <type2> <element2-name>;
      ...
   } u;
} <name>;
```

- or -

```
typedef struct {
   ...
} <tempName1>;

typedef struct {
   ...
} <tempName2>;

typedef struct {
   int t;
   union {
      <tempName1>* <element1-name>;
      <tempName2>* <element2-name>;
      ...
   } u;
} <name>;
```

If the -*static* command line option or `<storage> static </storage>` configuration variable is set for the given production, then pointers will not be used for the variable declarations.

Note

This is true for the C case only; for C++, pointers must be used due to the fact that the generated code will not compile if constructors are used in a non-pointer variable within a union construct.

The C++ mapping is the same with the exception that the `ASN1T_` prefix is added to the generated type name.

`<type1>` and `<type2>` are the equivalent C types representing the ASN.1 types `<element1-type>` and `<element2-type>` respectively. `<tempName1>` and `<tempName2>` represent the names of temporary types that may have been generated as the result of using nested constructed types within the definition.

Choice alternatives may be unnamed, in which case `<element-name>` is derived from `<element-type>` by making the first letter lowercase. One needs to be careful when nesting CHOICE structures at different levels within other nested ASN.1 structures (SEQUENCEs, SETs, or other CHOICEs). A problem arises when CHOICE element names at different levels are not unique (this is likely when elements are unnamed). The problem is that generated tag constants are not guaranteed to be unique since only the production and end element names are used.

The compiler gets around this problem by checking for duplicates. If the generated name is not unique, a sequential number is appended to make it unique. The compiler outputs an informational message when it does this.

An example of this can be found in the following production:

```
C ::= CHOICE {
   [0] INTEGER,
   [1] CHOICE {
      [0] INTEGER,
      [1] BOOLEAN
   }
}
```

This will produce the following C code:

```
#define T_C_aInt         1
#define T_C_aChoice      2
#define T_C_aInt_1       1
#define T_C_aBool        2

typedef struct {
   int t;
   union {
      OSINT32 aInt;
      struct {
         int t;
         union {
            OSINT32 aInt;
            OSBOOL aBool;
         } u;
      } aChoice;
   } C;
```

Note that `_1` was appended to the second instance of `T_C_aInt`. Developers must take care to ensure they are using the correct tag constant value when this happens.

Populating Generated Choice Structures

Populating generated CHOICE structures is more complex then for other generated types due to the use of pointers within the union construct. As previously mentioned, the use of pointers with C can be prevented by using the -*static* command line option. If this is done, the elements within the union construct will be standard inline variable declarations and can be populated directly. Otherwise, the methods listed below can be used to populate the variables.

The recommended way to populate the pointer elements is to declare variables of the embedded type to be used on the stack prior to populating the CHOICE structure. The embedded variable would then be populated with the data to be encoded and then the address of this variable would be plugged into the CHOICE union pointer field.

Consider the following definitions:

```
AsciiString ::= [PRIVATE 28] OCTET STRING
EBCDICString ::= [PRIVATE 29] OCTET STRING
String ::= CHOICE { AsciiString, EBCDICString }
```

This would result in the following type definitions:

```
typedef OSDynOctStr AsciiString;
typedef OSDynOctStr EBCDICString;

typedef struct String {
   int t;
   union {
      /* t = 1 */
     AsciiString *asciiString;
     /* t = 2 */
     EBCDICString *eBCDICString;
   } u;
} String;
```

To set the *AsciiString* choice value, one would first declare an *AsciiString* variable, populate it, and then plug the address into a variable of type *String* structure as follows:

```
AsciiString asciiString;
String      string;

asciiString = "Hello!";
string.t = T_String_AsciiString;
string.u.asciiString = &asciiString;
```

It is also possible to allocate dynamic memory for the CHOICE union option variable; but one must be careful to release this memory when done with the structure. If the built in memory-management functions/macros are used (*rtxMem*), all memory used for the variables is automatically released when *rtxMemFree* is called.

Open Type

Note

The X.680 Open Type replaces the X.208 ANY or ANY DEFINED BY constructs. An ANY or ANY DEFINED BY encountered within an ASN.1 module will result in the generation of code corresponding to the Open Type described below.

An *Open Type* as defined in the X.680 standard is specified as a reference to a *Type Field* in an *Information Object Class*. The most common form of this is when the *Type* field in the built-in TYPE-IDENTIFIER class is referenced as follows:

```
TYPE-IDENTIFIER.&Type
```

See the section in this document on Information Objects for a more detailed explanation.

The *Open Type* is converted into a C or C++ structure used to model a dynamic OCTET STRING type. This structure contains a pointer and length field. The pointer is assumed to point at a string of previously encoded ASN.1 data. When a message containing an open type is decoded, the address of the open type contents field is stored in the pointer field and the length of the component is stored in the length field.

The general mapping of an Open Type to C/C++ is as follows:

ASN.1 production:

```
<name> ::= ANY
```

Generated C code:

```
typedef ASN1OpenType <name>;
```

Generated C++ code:

```
typedef ASN1TOpenType <name>;
```

The difference between the two types is the C++ version contains constructors to initialize the value to zero or to a given open type value.

If the *-tables* command line option is selected and the ASN.1 type definition references a table constraint, the code generated is different. In this case, *ASN1OpenType* above is replaced with *ASN1Object* (or *ASN1TObject* for C++). This is defined in *asn1type.h* as follows:

```
typedef struct { /* generic table constraint value holder */
   ASN1OpenType encoded;
   void*        decoded;
```

```
      OSINT32      index;     /* table index */
   } ASN1Object;
```

This allows a value of any ASN.1 type to be represented in both encoded and decoded forms. Encoded form is the open type form shown above. It is simply a pointer to a byte buffer and a count of the number of byes in the encoded message component. The decoded form is a pointer to a variable of a specific type. The pointer is void because there could be a potentially large number of different types that can be represented in the table constraint used to constrain a type field to a given set of values. The *index* member of the type is for internal use by table constraint processing functions to keep track of which row in a table is being referenced.

If the *-table-unions* command line option is used, a more specialized type of structure is generated. In this case, instead of a void pointer being used to hold an instance of a type containing data to be encoded, all entries from the referenced Information Object Set are used in a union structure in much the same way as is done in a CHOICE construct.

If code is being generated from an XML schema file and the file contains an *<xsd:any>* wildcard declaration, a special type of any structure is inserted into the generated C/C++ code. This is the type *OSXSDAny* which is defined in the *osSysTypes.h* header file. This structure contains a union which contains alternatives for data in either binary or XML text form. This makes it possible to transfer data in either binary form if working with binary encoding rules or XML form if working with XML.

Character String Types

All 8-bit character character-string types are derived from the C character pointer (`const char*`) base type. This pointer is used to hold a null-terminated C string for encoding/decoding. For encoding, the string can either be static (i.e., a string literal or address of a static buffer) or dynamic. The decoder allocates dynamic memory from within its context to hold the memory for the string. This memory is released when the `rtxMemFree` function is called.

The useful character string types in ASN.1 are as follows:

```
   UTF8String        ::=  [UNIVERSAL 12]  IMPLICIT OCTET STRING
   NumericString     ::=  [UNIVERSAL 18]  IMPLICIT IA5String
   PrintableString   ::=  [UNIVERSAL 19]  IMPLICIT IA5String
   T61String         ::=  [UNIVERSAL 20]  IMPLICIT OCTET STRING
   VideotexString    ::=  [UNIVERSAL 21]  IMPLICIT OCTET STRING
   IA5String         ::=  [UNIVERSAL 22]  IMPLICIT OCTET STRING
   UTCTime           ::=  [UNIVERSAL 23]  IMPLICIT GeneralizedTime
   GeneralizedTime   ::=  [UNIVERSAL 24]  IMPLICIT IA5String
   GraphicString     ::=  [UNIVERSAL 25]  IMPLICIT OCTET STRING
   VisibleString     ::=  [UNIVERSAL 26]  IMPLICIT OCTET STRING
   GeneralString     ::=  [UNIVERSAL 27]  IMPLICIT OCTET STRING
   UniversalString   ::=  [UNIVERSAL 28]  IMPLICIT OCTET STRING
   BMPString         ::=  [UNIVERSAL 30]  IMPLICIT OCTET STRING
   ObjectDescriptor  ::=  [UNIVERSAL 7]  IMPLICIT GraphicString
```

Of these, all are represented by `const char *` pointers except for the *BMPString*, *UniversalString*, and *UTF8String types*.

The *BMPString* type is a 16-bit character string for which the following structure is used:

```
typedef struct {
   OSUINT32 nchars;
   OSUNICHAR* data;
} Asn116BitCharString;
```

The `OSUNICHAR` type used in this definition represents a Unicode character (UTF-16) and is defined to be a C `unsigned short` type.

See the `rtBMPToCString`, `rtBMPToNewCString`, and the `rtCToBMPString` run-time function descriptions for information on utilities that can convert standard C strings to and from BMP string format.

The `UniversalString` type is a 32-bit character string for which the following structure is used:

```
typedef struct {
   OSUINT32 nchars;
   OS32BITCHAR* data;
} Asn132BitCharString;
```

The `OS32BITCHAR` type used in this definition is defined to be a C `unsigned int` type.

See the `rtUCSToCString`, `rtUCSToNewCString`, and the `rtCToUCSString` run-time function descriptions for information on utilities that can convert standard C strings to and from Universal Character Set (UCS-4) string format. See also the `rtUCSToWCSString` and `rtWCSToUCSString` for information on utilities that can convert standard wide character string to and from `UniversalString` type.

The `UTF8String` type is represented as a string of unsigned characters using the `OSUTF8CHAR` data type. This type is defined to be `unsigned char`. This makes it possible to use the characters in the upper range of the UTF-8 space as positive numbers. The contents of this string type are assumed to contain the UTF-8 encoding of a character string. For the most part, standard C character string functions such as `strcpy`, `strcat`, etc. can be used with these strings with some type casting.

Utility functions are provided for working with UTF-8 string data. The UTF-8 encoding for a standard ASCII string is simply the string itself. For Unicode strings represented in C/C++ using the wide character type (`wchar_t`), the run-time functions `rtxUTF8ToWCS` and `rtxWCSToUTF8` can be used for converting to and from UTF-8 format. The function `rtxValidateUTF8` can be used to ensure that a given UTF-8 encoding is valid. See the *C/C++ Run-Time Library Reference Manual* for a complete description of these functions.

Time String Types

The ASN.1 *GeneralizedTime* and *UTCTime* types are mapped to standard C/C++ null-terminated character string types.

The C++ version of the product contains additional control classes for parsing and formatting time string values. When C++ code generation is specified, a control class is generated for operating on the target time string. This class is derived from the *ASN1CGeneralizedTime* or *ASN1CUTCTime* class for *GeneralizedTime* or *UTCTime* respectively. These classes contain methods for formatting or parsing time components such as month, day, year, etc. from the strings.

Objects of these classes can be declared inline to make the task of formatting or parsing time strings easier. For example, in a SEQUENCE containing a time string element the generated type will contain a public member variable containing the ASN1T type that holds the message data. If one wanted to operate on the time string contained within that element, they could do so by using one of the time string classes inline as follows:

```
ASN1CGeneralizedTime gtime (msgbuf, <seqVar>.<element>);
gtime.setMonth (ASN1CTime::November);
```

In this example, `<seqVar>` would represent a generated SEQUENCE variable type and `<element>` would represent a time string element within this type.

See the `ASN1CTime`, `ASN1CGeneralizedTime`, and `ASN1CUTCTime` subsections in the *C/C++ Run-Time Library Reference Manual* for details on all of the methods available in these classes.

EXTERNAL

The ASN.1 EXTERNAL type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. This type is described using the following ASN.1 SEQUENCE:

```
EXTERNAL ::= [UNIVERSAL 8] IMPLICIT SEQUENCE {
   direct-reference OBJECT IDENTIFIER OPTIONAL,
   indirect-reference INTEGER OPTIONAL,
   data-value-descriptor ObjectDescriptor OPTIONAL,
   encoding CHOICE {
      single-ASN1-type [0] ABSTRACT-SYNTAX.&Type,
      octet-aligned [1] IMPLICIT OCTET STRING,
      arbitrary [2] IMPLICIT BIT STRING
   }
}
```

The ASN1C compiler is used to create a meta-definition for this structure. This code will always be generated in the `Asn1External.h` and `Asn1External.c/cpp` files. The code will only be generated if the given ASN.1 source specification requires this definition. The resulting C structure is populated just like any other compiler-generated structure for working with ASN.1 data.

Note

NOTE: It is recommended that if a specification contains multiple ASN.1 source files that reference EXTERNAL, all of these source files be compiled with a single ASN1C call in order to ensure that only a single copy of the Asn1External source files are generated.

EMBEDDED PDV

The ASN.1 EMBEDDED PDV type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. This type is described using the following ASN.1 SEQUENCE:

```
EmbeddedPDV ::= [UNIVERSAL 11] IMPLICIT SEQUENCE {
   identification CHOICE {
      syntaxes SEQUENCE {
         abstract OBJECT IDENTIFIER,
         transfer OBJECT IDENTIFIER
      },
      syntax OBJECT IDENTIFIER,
      presentation-context-id INTEGER,
      context-negotiation SEQUENCE {
         presentation-context-id INTEGER,
         transfer-syntax OBJECT IDENTIFIER
      },
      transfer-syntax OBJECT IDENTIFIER,
      fixed NULL
   },
   data-value-descriptor ObjectDescriptor OPTIONAL,
   data-value OCTET STRING
}( WITH COMPONENTS { ... , data-value-descriptor ABSENT})
```

The ASN1C compiler is used to create a meta-definition for this structure. This code will be always generated in the `Asn1EmbeddedPDV.h` and `Asn1EmbeddedPDV.c/cpp` files. The code will only be generated if the given ASN.1 source specification requires this definition. The resulting C structure is populated just like any other compiler-generated structure for working with ASN.1 data.

Note

NOTE: It is recommended that if a specification contains multiple ASN.1 source files that reference EMBEDDEDPDV, all of these source files be compiled with a single ASN1C call in order to ensure that only a singled copy of the Asn1EmbeddedPDV source files are generated.

Parameterized Types

The ASN1C compiler can parse parameterized type definitions and references as specified in the X.683 standard. These types allow dummy parameters to be declared that will be replaced with actual parameters when the type is referenced. This is similar to templates in C++.

A simple and common example of the use of parameterized types is for the declaration of an upper bound on a sized type as follows:

```
SizedOctetString{INTEGER:ub} ::= OCTET STRING (SIZE (1..ub))
```

In this definition, `ub` would be replaced with an actual value when the type is referenced. For example, a sized octet string with an upper bound of 32 would be declared as follows:

```
OctetString32 ::= SizedOctetString{32}
```

The compiler would handle this in the same way as if the original type was declared to be an octet string of size 1 to 32. That is, it will generate a C structure containing a static byte array of size 32 as follows:

```
typedef struct OctetString32 {
      OSUINT32 numocts;
      OSOCTET data[32];
   } OctetString32;
```

Another common example of parameterization is the substitution of a given type inside a common container type. For example, security specifications frequently contain a 'signed' parameterized type that allows a digital signature to be applied to other types. An example of this is as follows:

```
   SIGNED { ToBeSigned } ::= SEQUENCE {
      toBeSigned    ToBeSigned,
      algorithmOID  OBJECT IDENTIFIER,
      paramS        Params,
      signature     BIT STRING
   }
```

An example of a reference to this definition would be as follows:

```
   SignedName ::= SIGNED { Name }
```

where `Name` would be another type defined elsewhere within the module.

The compiler performs the substitution to create the proper C typedef for `SignedName`:

```
   typedef struct SignedName {
      Name           toBeSigned;
      ASN1OBJID      algorithmOID;
      Params         paramS;
      ASN1DynBitStr  signature;
   } SignedName;
```

When processing parameterized type definitions, the compiler will first look to see if the parameters are actually used in the final generated code. If not, they will simply be discarded and the parameterized type converted to a normal type reference. For example, when used with information objects, parameterized types are frequently used to pass information object set definitions to impose table constraints on the final type. Since table constraints do not affect the code that is generated by the compiler when table constraint code generation is not enabled, the parameterized type definition is reduced to a normal type definition and references to it are handled in the same way as defined type references. This can lead to a significant reduction in generated code in cases where a parameterized type is referenced over and over again.

For example, consider the following often-repeated pattern from the UMTS 3GPP specs:

```
   ProtocolIE-Field {RANAP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
      id             RANAP-PROTOCOL-IES.&id          ({IEsSetParam}),
      criticality    RANAP-PROTOCOL-IES.&criticality ({IEsSetParam}{@id}),
      value          RANAP-PROTOCOL-IES.&Value       ({IEsSetParam}{@id})
   }
```

In this case, `IEsSetParam` refers to an information object set specification that constrains the values that are allowed to be passed for any given instance of a type referencing a `ProtocolIE-Field`. The compiler does not add any extra code to check for these values, so the parameter can be discarded (note that this is not true if the *-tables* compiler option is specified). After processing the Information Object Class references within the construct (refer to the section on *Information Objects* for information on how this is done), the reduced definition for `ProtocolIE-Field` becomes the following:

```
ProtocolIE-Field ::= SEQUENCE {
   id ProtocolIE-ID,
   criticality Criticality,
   value ASN.1 OPEN TYPE
}
```

References to the field are simply replaced with a reference to the `ProtocolID-Field` typedef.

If -*tables* is specified, the parameters are used and a new type instance is created in accordance with the rules above.

Value Mappings

ASN1C can parse any type of ASN.1 value specification, but it will only generate code for following value specifications:

- BOOLEAN
- INTEGER
- REAL
- ENUMERATED
- Binary String
- Hexadecimal String
- Character String
- OBJECT IDENTIFER

All value types except INTEGER and REAL cause an "extern" statement to be generated in the header file and a global value assignment to be added to the C or C++ source file. INTEGER and REAL value specifications cause #define statements to be generated.

BOOLEAN Value

A BOOLEAN value causes an extern statement to be generated in the header file and a global declaration of type OSBOOL to be generated in the C or C++ source file. The mapping of ASN.1 declaration to global C or C++ value declaration is as follows:

ASN.1 production:

```
<name> BOOLEAN ::= <value>
```

Generated code:

```
OSBOOL <name> = <value>;
```

INTEGER Value

The INTEGER type causes a `#define` statement to be generated in the header file of the form `ASN1V_<valueName>` where `<valueName>` would be replaced with the name in the ASN.1 source file. The reason for doing this is the common use of INTEGER values for size and value range constraints in the ASN.1 specifications. By generating `#define` statements, the symbolic names can be included in the source code making it easier to adjust the boundary values.

This mapping is defined as follows:

ASN.1 production:

```
<name> INTEGER ::= <value>
```

Generated code:

```
#define ASN1V_<name> <value>;
```

For example, the following declaration:

```
ivalue INTEGER ::= 5
```

will cause the following statement to be added to the generated header file:

```
#define ASN1V_ivalue 5
```

The reason the `ASN1V_` prefix is added is to prevent collisions with INTEGER value declarations and other declarations such as enumeration items with the same name.

REAL Value

The REAL type causes a `#define` statement to be generated in the header file of the form `ASN1V_<valueName>` where `<valueName>` would be replaced with the name in the ASN.1 source file. By generating `#define` statements, the symbolic names can be included in the source code making it easier to adjust the boundary values.

This mapping is defined as follows:

ASN.1 production:

```
<name> REAL ::= <value>
```

Generated code:

```
#define ASN1V_<name> <value>;
```

For example, the following declaration:

```
rvalue REAL ::= 5.5
```

will cause the following statement to be added to the generated header file:

```
#define ASN1V_rvalue 5.5
```

The reason the `ASN1V_` prefix is added is to prevent collisions with other declarations such as enumeration items with the same name.

Enumerated Value Specification

The mapping of an ASN.1 enumerated value declaration to a global C or C++ value declaration is as follows:

ASN.1 production:

```
<name> <EnumType> ::= <value>
```

Generated code:

```
OSUINT32 <name> = <value>;
```

Binary and Hexadecimal String Value

Binary and hexadecimal string value specifications cause two global C variables to be generated: a `numocts` variable describing the length of the string and a `data` variable describing the string contents. The mapping for a binary string is as follows (note: BIT STRING can also be used as the type in this type of declaration):

ASN.1 production:

```
<name> OCTET STRING ::= '<bstring>'B
```

Generated code :

```
OSUINT32 <name>_numocts = <length>;
OSOCTET <name>_data[] = <data>;
```

A hexadecimal string would be the same except the ASN.1 constant would end in a 'H'.

Character String Value

A character string declaration would cause a C or C++ `const char *` declaration to be generated:

ASN.1 production:

```
<name> <string-type> ::= <value>
```

Generated code:

```
const char* <name> = <value>;
```

In this definition, <string-type> could be any of the standard 8-bit characters string types such as IA5String, PrintableString, etc.

Note

Code generation is not currently supported for value declarations of larger character string types such as BMPString.

Object Identifier Value Specification

Object identifier values result in a structure being populated in the C or C++ source file.

ASN.1 production:

```
<name> OBJECT IDENTIFIER ::= <value>
```

Generated code:

```
ASN1OBJID <name> = <value>;
```

For example, consider the following declaration:

```
oid OBJECT IDENTIFIER ::= { ccitt b(5) 10 }
```

This would result in the following definition in the C or C++ source file:

```
ASN1OBJID oid = {
   3, { 0, 5, 10 }
} ;
```

To populate a variable in a generated structure with this value, the `rtSetOID` utility function can be used (see the *C/C++ Run-Time Library Reference Manual* for a full description of this function). In addition, the C++ base type for this construct (`ASN1TObjId`) contains constructors and assignment operators that allow direct assignment of values in this form to the target variable.

Constructed Type Values

ASN1C will generate code for following remaining value definitions only when their use is required in legacy table constraint validation code:

- SEQUENCE
- SET
- SEQUENCE OF
- SET OF

- CHOICE

Note

SEQUENCE, SET , SEQUENCE OF, SET OF and CHOICE values are available only when the *-tables* option is selected.

The values are initialized in a module value initialization function. The format of this function name is as follows:

```
init_<ModuleName>Value (OSCTXT*
                           pctxt)
```

Where `<ModuleName>` would be replaced with the name of the module containing the value specifications.

The only required argument is an initialized context block structure used to hold dynamic memory allocated in the creation of the value structures.

If the value definitions are used in table constraint definitions, then the generated table constraint processing code will handle the initialization of these definitions; otherwise, the initialization function must be called explicitly.

SEQUENCE or SET Value Specification

The mapping of an ASN.1 SEQUENCE or SET value declaration to a global C or C++ value declaration is as follows:

ASN.1 production:

```
<name> <SeqType> ::= <value>
```

Generated code :

```
<SeqType> <name>;
```

The sequence value will be initialized in the value initialization function.

For example, consider the following declaration:

```
SeqType ::= SEQUENCE {
   id INTEGER ,
   name VisibleString
}

value SeqType ::= { id 12, name "abc" }
```

This would result in the following definition in the C or C++ source file:

```
SeqType value;
```

Code generated in value initialization function would be as follows:

```
value.id = 12;
value.name = "abc";
```

SEQUENCE OF/SET OF Value

The mapping of an ASN.1 SEQUENCE OF or SET OF value declaration to a global C or C++ value declaration is as follows:

ASN.1 production:

```
<name> <SeqOfType> ::= <value>
```

Generated code :

```
<SeqOfType> <name>;
```

The sequence of value will be initialized in the value initialization function.

For example, consider the following declaration:

```
SeqOfType ::= SEQUENCE OF (SIZE(2)) INTEGER

value SeqOfType ::= { 1, 2 }
```

This would result in the following definition in the C or C++ source file:

```
SeqOfType value;
```

Code generated in the value initialization function would be as follows:

```
value.n = 2;
value.element[0] = 1;
value.element[1] = 2;
```

CHOICE Value

The mapping of an ASN.1 CHOICE value declaration to a global C or C++ value declaration is as follows:

ASN.1 production:

```
<name> <ChoiceType> ::= <value>
```

Generated code :

```
<ChoiceType> <name>;
```

The choice value will be initialized in the value initialization function.

For example, consider the following declaration:

```
ChoiceType ::= CHOICE { oid OBJECT IDENTIFIER, id INTEGER }
```

```
value ChoiceType ::= id: 1
```

This would result in the following definition in the C or C++ source file:

```
ChoiceType value;
```

Code generated in the value initialization function would be as follows:

```
value.t = T_ChoiceType_id;
value.u.id = 1;
```

Table Constraint Related Structures

The following sections describe changes to generated code that occur when the *-tables* or *-table-unions* option is specified on the command-line or when *Table Constraint Options* are selected from the GUI. This option causes additional code to be generated for items required to support table constraints as specified in the X.682 standard. This includes the generation of structures and classes for Information Object Classes, Information Objects, and Information Object Sets as specified in the X.681 standard.

Most of the additional items that are generated are read-only tables for use by the run-time for data validation purposes. However, generated structures for types that use table constraints are different than when table constraint code generation is not enabled. These differences will be pointed out.

There are two models currently supported for table contraint generation: Unions and Legacy. These are documented in the following sections:

Unions Table Constraint Model

The unions table constraint model originated from common patterns used in a series of ASN.1 specifications in-use in 3rd Generation Partnership Project (3GPP) standards. These standards include Node Application Part B (NBAP), Radio Access Network Application Part (RANAP), and Radio Network Subsystem Application Part (RNSAP) in the current 3G network and in S1AP and X2AP protocols in the newer 4G network (LTE) standards. This model was later extended to generate these type of structures for other specifications that made use of table constraints including security and legacy telecom speifications.

Generated C Type Definitions for Message Types

The standard message type used by many specifications that employ table constraints is usually a SEQUENCE type with elements that use a relational table constraint that uses fixed-type and type fields. The general form is as follows:

```
<Type> ::= SEQUENCE {
   <element1> <Class>.&<fixed-type-field> ({<ObjectSet>}),
   <element2> <Class>.&<fixed-type-field> ({<ObjectSet>)){@element1}
   <element3> <Class>.&<type-field> ({<ObjectSet>)){@element1}
}
```

In this definition, `<Class>` would be replaced with a reference to an Information Object Class, `<fixed-type-field>` would be a fixed-type field wtihin that class, and `<type-field>` would be a type field within the class. `<ObjectSet>` would be a reference to an Information Object Set which would define all of the possibilities for content within the message. The first element (<element1>) would be used as the index element in the object set relation.

An example of this pattern from the S1AP LTE specification is as follows:

```
   InitiatingMessage ::= SEQUENCE {
      procedureCode   S1AP-ELEMENTARY-PROCEDURE.&procedureCode
                         ({S1AP-ELEMENTARY-PROCEDURES}),
      criticality     S1AP-ELEMENTARY-PROCEDURE.&criticality
                         ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode}),
      value           S1AP-ELEMENTARY-PROCEDURE.&InitiatingMessage
                         ({S1AP-ELEMENTARY-PROCEDURES}{@procedureCode})
   }
```

In this definition, `procedureCode` and `criticality` are defined to be a enumerated fixed types, and `value` is defined to be an open type field to hold variable content as defined in the object set definition.

In the legacy model defined below, a loose coupling would be defined for the open type field using the built-in `ASN1Object` structure. This structure uses a void pointer to hold a link to a variable of the typed data structure. This is inconvenient for the developer because he would need to consult the object set definition within the ASN.1 specification in order to determine what type of data is to be used with each procedure code. It is also error prone in that the void pointer provides for no type checking at compile time.

In the new model, the generated structure is designed to be similar as to what is used to represent a CHOICE type. That is to say, the structure is a union with a choice selector value and all possible types listed out in a union structure. This is the general form:

```
   typedef struct <Type> {
      <Element1Type> <element1>;
      <Element2Type> <element2>;

      /**
       * information object selector
       */
      <SelectorEnumType> t;

      /**
       * <ObjectSet> information objects
       */
      union {
         /**
          * <element1> : <object1-element1-value>
          * <element2> : <object1-element2-value>
          */
         <object1-element3-type>* <object1-name>;
         /**
          * <element1> : <object2-element1-value>
          * <element2> : <object2-element2-value>
          */
         <object2-element3-type>* <object2-name>;

         ...
      } u;
   } ;
```

In this definition, the first two elements of the sequence would use the equivalent C or C++ type as defined in the fixed-type field in the information object. This is the same as in the legacy model. The open type field (element3) would be expanded into the union structure as is shown. The <SelectorEnumType> would be an enumerated type that is generated to represent each of the choices in the referenced information object set. The union then contains an entry for each of the possible types as defined in the object set that can be used in the open type field. Comments are used to list the fixed-type fields corresponding to each open type field.

An example of the code that is generated from the S1AP sample ASN.1 snippet above is as follows:

```
typedef enum {
   T_S1AP_PDU_Descriptions_S1AP_ELEMENTARY_PROCEDURES_handoverPreparation,
   T_S1AP_PDU_Descriptions_S1AP_ELEMENTARY_PROCEDURES_handoverResourceAllocation,
   T_S1AP_PDU_Descriptions_S1AP_ELEMENTARY_PROCEDURES_pathSwitchRequest,
   etc..
} S1AP_ELEMENTARY_PROCEDURE_TVALUE;

typedef struct InitiatingMessage {
   ProcedureCode procedureCode;
   Criticality criticality;

   /**
    * information object selector
    */
   S1AP_ELEMENTARY_PROCEDURE_TVALUE t;

   /**
    * S1AP-ELEMENTARY-PROCEDURE information objects
    */
   union {
      /**
       * procedureCode: id-HandoverPreparation
       * criticality: reject
       */
      HandoverRequired*  handoverPreparation;

      /**
       * procedureCode: id-HandoverResourceAllocation
       * criticality: reject
       */
      HandoverRequest*  handoverResourceAllocation;

      /**
       * procedureCode: id-HandoverNotification
       * criticality: ignore
       */
      HandoverNotify*  handoverNotification;
```

```
        etc..

    } u;
  } InitiatingMessage;
```

Note that the long names generated in the S1AP_ELEMENTARY_PROCEDURE_TVALUE type can be reduced by using the <alias> configuration element.

Generated C Type Definitions for Information Element (IE) Types

In addition to message types, another common pattern in 3GPP specifications is protocol information element (IE) types. The general form of these types is a list of information elements as follows:

```
   <ProtocolIEsType> ::= <ProtocolIE-ContainerType> { <ObjectSet> }

   <ProtocolIE-ContainerType> { <Class> : <ObjectSetParam> } ::=
      SEQUENCE (SIZE (<size>)) OF <ProtocolIE-FieldType> {{ObjectSetParam}}

   <ProtocolIE-FieldType> { <Class> : <ObjectSetParam> } ::= SEQUENCE {
      <element1> <Class>.&<fixed-type-field> ({ObjectSetParam}),
      <element2> <Class>.&<fixed-type-field> ({ObjectSetParam}{@element1}),
      <element3> <Class>.&<Type-field> ({ObjectSetParam}{@element1})
   }
```

There are a few different variations of this, but the overall pattern is similar in all cases. A parameterized type is used as a shorthand notation to pass an information object set into a container type. The container type holds a list of the IE fields. The structure of an IE field type is similar to a message type: the first element is used as an index element to the remaining elements. That is followed by one or more fixed type or variable type elements. In the case defined above, only a single fixed-type and variable type element is shown, but there may be more.

An example of this pattern from the S1AP LTE specification follows:

```
   HandoverRequired ::= SEQUENCE {
      protocolIEs   ProtocolIE-Container   { { HandoverRequiredIEs} },
      ...
   }

   ProtocolIE-Container {S1AP-PROTOCOL-IES : IEsSetParam} ::=
         SEQUENCE (SIZE (0..maxProtocolIEs)) OF ProtocolIE-Field {{IEsSetParam}}

   ProtocolIE-Field {S1AP-PROTOCOL-IES : IEsSetParam} ::= SEQUENCE {
         id             S1AP-PROTOCOL-IES.&id            ({IEsSetParam}),
         criticality    S1AP-PROTOCOL-IES.&criticality   ({IEsSetParam}{@id}),
         value          S1AP-PROTOCOL-IES.&Value         ({IEsSetParam}{@id})
   }
```

In this case, standard parameterized type instantiation is used to create a type definition for the protocolIEs element. This results in a list type being generated:

```
/* List of HandoverRequired_protocolIEs_element */
```

```
typedef OSRTDList HandoverRequired_protocolIEs;
```

The type for the protocol IE list element is created in much the same way as the main message type was above:

```
   typedef struct HandoverRequired_protocolIEs_element {
      ProtocolIE_ID id;
      Criticality criticality;
      struct {
         /**
          * information object selector
          */
         HandoverRequiredIEs_TVALUE t;

         /**
          * HandoverRequiredIEs information objects
          */
         union {
            /**
             * id: id-MME-UE-S1AP-ID
             * criticality: reject
             * presence: mandatory
             */
            MME_UE_S1AP_ID  *_HandoverRequiredIEs_id_MME_UE_S1AP_ID;
            /**
             * id: id-HandoverType
             * criticality: reject
             * presence: mandatory
             */
            HandoverType  *_HandoverRequiredIEs_id_HandoverType;
            /**
             * id: id-Cause
             * criticality: ignore
             * presence: mandatory
             */
             ...
         } u;
      } value;
   } HandoverRequired_protocolIEs_element;
```

In this case, the protocol IE id field and criticality are generated as usual using the fixed-type field type definitions. The open type field once again results in the generation of a union structure of all possible type fields that can be used. Note in this case the field names are automatically generated (_HandoverRequiredIEs_id_MME_UE_S1AP_ID, etc.). The reason for this was the use of inline information object definitions in the information object set as opposed to defined object definitions. This is a sample from that set:

```
   HandoverRequiredIEs S1AP-PROTOCOL-IES ::= {
      { ID id-MME-UE-S1AP-ID         CRITICALITY reject   TYPE MME-UE-S1AP-ID   PRESENCE mandator
      { ID id-HandoverType           CRITICALITY reject   TYPE HandoverType     PRESENCE mandator
   ...
```

In this case, the name is formed by combining the information object set name with the name of each key field within the set.

Generated IE Append Function

A user would need to allocate objects of this structure, populate them, and add them to the protocol IE list. In order to make this easier, helper functions are generated assist in adding information items to the list. The general format of these append functions is as follows:

```
int asn1Append_<ProtocolIEsType>_<KeyValueName>
   (OSCTXT* pctxt, <ProtocolIEsType>* plist, <ValueType> value);
```

In this definition, `<ProtocolIEsType>` refers to the main list type (SEQUENCE OF) defining the information element list. `<KeyValueName>` is the name of the primary key field defined in the associated information object set. `<ValueType>` is the type of the value for the indexed information object set item.

An example of this type of function from the S1AP definitions is as follows:

```
/* Append IE with value type MME_UE_S1AP_ID to list */
int asn1Append_HandoverRequired_protocolIEs_id_MME_UE_S1AP_ID (OSCTXT* pctxt,
   HandoverRequired_protocolIEs* plist, MME_UE_S1AP_ID value);
```

Generated IE Get Function

In addition to the list append function, a second type of helper function is generated to make it easier to find an item in the list based on the key field. The general format for this type of function is as follows:

```
<ProtocolIE-FieldType>* asn1Get_<ProtocolIEsType>
   (<KeyFieldType> <key>, <ProtocolIEsType>* plist);
```

In this definition, `<ProtocolIEsType>` refers to the main list type (SEQUENCE OF) definiing the information element list. `<ProtocolIE-FieldType>` is the type of an element within this list and `<KeyFieldType>` is the type of index key field.

An example of this type of function from the S1AP definitions is as follows:

```
/* Get IE using id key value */
HandoverRequired_protocolIEs_element* asn1Get_HandoverRequired_protocolIEs
   (ProtocolIE_ID id, HandoverRequired_protocolIEs* plist);
```

Generated C++ Classes and Methods

This section discusses items that are generated idfferently for C++ for union table constraints.

Choice Selector TVALUE Type

For C, an enumerated type is generated for each of the options in a type field union. These correspond to each of the items in the information object set associated with the union. For example, the TVALUE type generated for S1AP_ELEMENTARY_PROCEDURES is as follows:

```
typedef enum {
   T_S1AP_PDU_Descriptions_S1AP_ELEMENTARY_PROCEDURES_UNDEF_,
   T_S1AP_PDU_Descriptions_S1AP_ELEMENTARY_PROCEDURES_handoverPreparation,
   T_S1AP_PDU_Descriptions_S1AP_ELEMENTARY_PROCEDURES_handoverResourceAllocation,
   T_S1AP_PDU_Descriptions_S1AP_ELEMENTARY_PROCEDURES_pathSwitchRequest,
   ...
} S1AP_ELEMENTARY_PROCEDURES_TVALUE;
```

The generated names include the name of the module, object set, and object in order to ensure that no name clashes occur between enumerations with common names. For C++, this type is generated as a class with TVALUE as a public member inside:

```
   class S1AP_ELEMENTARY_PROCEDURES {
   public:
      enum TVALUE {
         T_UNDEF_,
         T_handoverPreparation,
         T_handoverResourceAllocation,
         T_pathSwitchRequest,
      ...
      } ;
   } ;
```

In this case, the type module and object set names are not needed because the class name provides for unambiguous enumerated item names.

Generated Helper Methods

For C, special `asn1Append_<name>` and `asn1GetIE_<name>` functions are generated to help a user append information elements (IE's) to a list and get an indexed IE respectively. For C++, these are added as methods to the generated control class for the list type.

For example, for the `HandoverRequired_protocolIEs` type, the following methods are added to the control class:

```
   class EXTERN ASN1C_HandoverRequired_protocolIEs : public ASN1CSeqOfList
   {
      ...
      /* Append IE with value type ASN1T_MME_UE_S1AP_ID to list */
      int Append_id_MME_UE_S1AP_ID (ASN1T_MME_UE_S1AP_ID value);

      /* Append IE with value type ASN1T_HandoverType to list */
      int Appendid_eNB_UE_S1AP_ID (ASN1T_HandoverType value);
      ...
      /* Get IE using id key value */
      ASN1T_HandoverRequired_protocolIEs_element* GetIE (ASN1T_ProtocolIE_ID id);
   } ;
```

Legacy Table Constraint Model

The primary difference as to what a user sees and works with in the legacy model as opposed to the unions model lies in the representation of open type elements that contain a table constraint. The standard form of an open type element constrained with a table constraint within a SEQUENCE container is as follows:

```
   <Type> ::= SEQUENCE {
      <element> <Class>.&<type-field> ({<ObjectSet>)){@index-element}
   }
```

If *-tables* is not specified on the command line, a standard open type structure is used to hold the element value:

```
   typedef struct <Type> {
      ASN1OpenType <element>;
   }
```

The `ASN1OpenType` built-in type holds the element data in encoded form. The only validation that is done on the element is to verify that it is a well-formed tag-length-value (TLV) structure if BER encoding is used or a valid length prefixed structure for PER.

If the *-tables* command line option is selected, the code generated is different. In this case, `ASN1OpenType` above is replaced with `ASN1Object` (or `ASN1TObject` for C++). This is defined in `asn1type.h` as follows:

```
   typedef struct { /* generic table constraint value holder */
      ASN1OpenType encoded;
      void*        decoded;
      OSINT32      index;     /* table index */
   } ASN1Object;
```

This allows a value of any ASN.1 type to be represented in both encoded and decoded forms. Encoded form is the open type form shown above. It is simply a pointer to a byte buffer and a count of the number of byes in the encoded message component. The decoded form is a pointer to a variable of a specific type. The pointer is void because there could be a potentially large number of different types that can be represented in the table constraint used to constrain a type field to a given set of values. The `index` member of the type is for internal use by table constraint processing functions to keep track of which row in a table is being referenced.

In addition to this change in how open types are represented, a large amount of supporting code is generated to support the table constraint validation process. This additional code is described below. Note that it is not necessary for the average user to understand this as it is not for use by users in accomplishing encoding and decoding of messages. It is only described for completeness in order to know what that additional code is used for.

CLASS specification

All of the Class code will be generated in a module class header file with the following filename format:

```
   <ModuleName>Class.h
```

In this definition, `<ModuleName>` would be replaced with the name of the ASN.1 module name for this class definition.

C Code generation

The C structure definition generated to model an ASN.1 class contains member variables for each of the fields within the class.

For each of the following class fields, the corresponding member variable is included in the generated C structure as follows:

For a Value Field:

```
<TypeName> <FieldName>;
```

For TypeField definitions, an encode and decode function pointer and type size field is generated to hold the information of the type for the OpenType. If the *-print* option is selected, a print function pointer is also added.

```
int <FieldName>Size;
int (*encode<FieldName>) (... );
int (*decode<FieldName>) (... );
void (*print<FieldName>) (...);
```

For an Object Field:

```
<ClassName>* <FieldName>;
```

For an ObjectSetField definition, an array of class definitions is generated to hold the list of information objects.

```
<ClassName>* <FieldName>;
```

In each of these definitions:

`<FieldName>` would be replaced with the name of the field (without the leading '&').

`<TypeName>` would be replaced with the C type name for the ASN.1 Type.

`<ClassName>` would be replaced with the C type name of the class for the Information Object.

As an example, consider the following ASN.1 class definition :

```
ATTRIBUTE ::= CLASS {
   &Type,
   &id OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX { &Type ID &id }
```

This would result in the following definition in the C source file:

```
typedef struct ATTRIBUTE {
   int TypeSize;
   int (*encodeType) (OSCTXT* , void *, ASN1TagType );
   int (*decodeType) (OSCTXT* , void *, ASN1TagType, int );
   ASN1OBJID id;
}
```

C++ Code generation

The C++ abstract class generated to model an ASN.1 CLASS contains member variables for each of the fields within the class. Derived information object classes are required to populate these variables with the values defined in the ASN.1 information object specification. The C++ class also contains virtual methods representing each of the type fields within the ASN.1 class specification. If the field is not defined to be OPTIONAL in the ASN.1 specification, then it is declared to be abstract in the generated class definition. A class generated for an ASN.1 information object that references this base class is required to implement these abstract virtual methods.

For each of the following CLASS fields, a corresponding member variable is generated in the C ++ class definition as follows:

For a Value Field definition, the following member variable will be added. Also, an Equals() method will be added if required for table constraint processing.

```
<TypeName> <FieldName>;

inline OSBOOL idEquals (<TypeName>* pvalue)
```

For a Type Field definition, a virtual method is added for each encoding rules type to call the generated C encode and decode functions. If -print is specified, a print method is also generated.

```
virtual int encode<ER><FieldName>
   (OSCTXT* pctxt, ASN1TObject& object) { return 0; }

virtual int decode<ER><FieldName>
   (OSCTXT* pctxt, ASN1TObject& object) { return 0; }

virtual void print<FieldName>
   (ASN1ConstCharPtr name, ASN1TObject& object) {}
```

For an Object Field:

```
class <ClassName>* <FieldName>;
```

In each of these definitions:

`<FieldName>` would be replaced with the name of the field (without the leading '&').

`<TypeName>` would be replaced with the C type name for the ASN.1 Type.

`<ClassName>` would be replaced with the C type name of the class for the Information Object.

`<ER>` would be replaced by an encoding rules type (BER, PER, or XER).

As an example, consider the following ASN.1 class definition :

```
ATTRIBUTE ::= CLASS {
   &Type,
   &ParameterType OPTIONAL,
   &id            OBJECT IDENTIFIER UNIQUE }
WITH SYNTAX { &Type ID &id }
```

This would result in the following definition in the C++ source file:

```
class EXTERN ATTRIBUTE {
  protected:
     ASN1TObjId id;
```

```
      ATTRIBUTE ();
   public:
      virtual int encodeBERType
         (OSCTXT* pctxt, ASN1TObject& object) = 0;

      virtual int decodeBERType
         (OSCTXT* pctxt, ASN1TObject& object) = 0;

      OSBOOL isParameterTypePresent() {
         if(m.ParameterTypePresent) {return TRUE;} else {return FALSE;}
      }
      virtual int encodeBERParameterType
         (OSCTXT* pctxt, ASN1TObject& object) { return 0; }

      virtual int decodeBERParameterType
         (OSCTXT* pctxt, ASN1TObject& object) { return 0; }

      inline OSBOOL idEquals (ASN1TObjId* pvalue)
      {
         return (0 == rtCmpTCOID (&id, pvalue));
      }
} ;
```

This assumes that only BER or DER was specified as the encoding rules type.

First notice that member variables have been generated for the fixed-type fields in the definition. These include the `id` field. Information object classes derived from this definition are expected to populate these fields in their constructors.

Also, virtual methods have been generated for each of the type fields in the class. These include the `Type` fields. The method generated for `Type` is abstract and must be implemented in a derived information object class. The method generated for the `ParameterType` field has a default implementations that does nothing. That is because it is a optional field.

Also generated are `Equals` methods for the fixed-type fields. These are used by the generated code to verify that data in a generated structure to be encoded (or data that has just been decoded) matches the table constraint values. This method will be generated only if it is required to check a table constraint.

OPTIONAL keyword

Fields within a CLASS can be declared to be optional using the OPTIONAL keyword. This indicates that the field is not required in the information object. An additional construct is added to the generated code to indicate whether an optional field is present in the information object or not. This construct is a bit structure placed at the beginning of the generated structure. This structure always has variable name 'm' and contains single-bit elements of the form `<fieldname>Present` as follows:

```
struct {
   unsigned <field-name1>Present : 1,
   unsigned <field-name2>Present : 1,
   ...
} m;
```

In this case, the fields included in this construct correspond to only those fields marked as OPTIONAL within the CLASS. If a CLASS contains no optional fields, the entire construct is omitted.

For example, we will change the CLASS in the previous example to make one field optional:

```
ATTRIBUTE ::= CLASS {
   &Type OPTIONAL,
   &id OBJECT IDENTIFIER UNIQUE
}
```

In this case, the following C typedef is generated in C struct or C++ class definition:

```
struct {
   unsigned TypePresent : 1;
} m;
```

When this structure is populated for encoding, the information object processing code will set *TypePresent* flag accordingly to indicate whether the field is present or not.

In C++ code generation, an additional method is generated for an optional field as follows:

```
OSBOOL is<FieldName>Present() {
   if (m.<FieldName>Present) {return TRUE;} else {return FALSE;}
}
```

This function is used to check if the field value is present in an information object definition.

Generation of New ASN.1 Assignments from CLASS Assignments:

During CLASS definition code generation, the following new assignments are created for C or C ++ code generation:

1. A new Type Assignment is created for a TypeField's type definition, as follows:

   ```
   _<ClassName>_<FieldName> ::= <Type>
   ```

   Here `ClassName` is replaced with name of the Class Assignment and `FieldName` is replaced with name of this field. Type is the type definition in CLASS's TypeField.

   This type is used as a defined type in the information object definition for an absent value of the TypeField. It is also useful for generation of a value for a related Open Type definition in a table constraint.

2. A new Type Assignment is created for a Value Field or Value Set Field type definition as follows (if the type definition is one of the following: ConstraintedType / ENUMERATED / NamedList BIT STRING / SEQUENCE / SET / CHOICE / SEQUENCE OF / SET OF):

   ```
   _<ClassName>_<FieldName> ::= <Type>
   ```

   Here `ClassName` is replaced with the name of the CLASS assignment and `FieldName` is replaced with name of the ValueField or ValueSetField. `Type` is the type definition in the CLASS's ValueField or ValueSetField. This type will appear as a defined type in the CLASS's ValueField or ValueSetField.

This new type assignment is used for compiler internal code generation purpose. It is not required for a user to understand this logic.

3. A new Value Assignment is created for a ValueField's default value definition as follows:

   ```
   _<ClassName>_<FieldName>_default <Type> ::= <Value>
   ```

   Here `ClassName` is replaced with name of the Class Assignment and `FieldName` is replaced with name of this ValueField. `Value` is the default value in the Class's ValueField and `Type` is the type in Class's ValueField.

   This value is used as a defined value in the information object definition for an absent value of the field. This new value assignment is used for compiler internal code generation purpose. It is not required for user to understand this logic.

ABSTRACT-SYNTAX and TYPE-IDENTIFIER

The ASN.1 ABTRACT-SYNTAX and TYPE-IDENTIFIER classes are useful ASN.1 definitions. These classes are described using the following ASN.1 definitions:

```
TYPE-IDENTIFIER ::= CLASS {
   &id OBJECT IDENTIFIER UNIQUE,
   &Type
}
WITH SYNTAX { &Type IDENTIFIED BY &id }

ABSTRACT-SYNTAX ::= CLASS {
   &id OBJECT IDENTIFIER UNIQUE,
   &Type,
   &property BIT STRING { handles-invalid-encoding(0)} DEFAULT {}
}
WITH SYNTAX {
   &Type IDENTIFIED BY &id [HAS PROPERTY &property]
}
```

The ASN1C compiler generates code for these constructs when they are referenced in the ASN.1 source file that is being compiled. The generated code for these constructs is written to the `RtClass.h` and `.c/.cpp` source files.

Information Object

Information Object code will be generated in a header and source file with a C struct / C++ class to hold the values. The name of the header and source file are of the following format:

```
<ModuleName>Table.h
<ModuleName>Table.c/cpp
```

In this definition, `<ModuleName>` would be replaced with the name of the ASN.1 module in which the information object is defined.

C Code Generation

For C, a global variable is generated to hold the information object definition. This is very similar to the code generated for a value definition.

An example of an information object definition that is derived from the ASN.1 ATTRIBUTE class above is as follows:

```
name ATTRIBUTE ::= {
   WITH SYNTAX    VisibleString
   ID             { 0 1 1 } }
```

This results in the generation of the following C constant:

```
ATTRIBUTE name;
```

Code generated in information object initialization function:

```
name.TypeSize = sizeof(_name_Type);
name.encodeType = &asn1E__name_Type;
name.decodeType = &asn1D__name_Type;
name.id.numids = 3;
name.id.subid[0] = 0;
name.id.subid[1] = 1;
name.id.subid[2] = 1;
```

C++ Code Generation

The C++ classes generated for ASN.1 information objects are derived from the ASN.1 class objects. The constructors in these classes populate the fixed-type field member variables with the values specified in the information object. The classes also implement the virtual methods generated for the information object type fields. All non-optional methods are required to be implemented. The optional methods are only implemented if they are defined in the information object definition.

An example of an information object definition that is derived from the ASN.1 class above is as follows:

```
name ATTRIBUTE ::= {
WITH SYNTAX       VisibleString
ID                { 0 1 1 } }
```

This results in the generation of the following C++ class:

```
class EXTERN name : public ATTRIBUTE {
  public:
     name();

     virtual int encodeBERType
        (OSCTXT* pctxt, ASN1TObject& object);

     virtual int decodeBERType
        (OSCTXT* pctxt, ASN1TObject& object);
} ;
```

The constructor implementation for this class (not shown) sets the fixed type fields (`id`) to the assigned values (`{0 1 1}`). The class also implements the virtual methods for the type field virtual methods defined in the base class. These methods simply call the BER encode or decode method for the assigned type (this example assumes -ber was specified for code generation - other encode rules could have been used as well).

Generated Type Assignments

If the information object contains an embedded type definition, it is extracted from the definition to form a new type to be added to the generated C or C++ code. The format of the new type name is as follows:

```
_<ObjectName>_<FieldName>
```

where `<ObjectName>` is replaced with the information object name and `<FieldName>` is replaced with the name of the field from within the object.

Information Object Set

Table constraint processing code to support Information Object Sets is generated in a header and source file with a C struct / C++ class to hold the values. The name of the header and source file are of the following format:

```
<ModuleName >Table.h
<ModuleName >Table.c/cpp
```

In this definition, `<ModuleName>` would be replaced with the name of the ASN.1 module in which the information object is defined.

C Code Generation

A C global variable is generated containing a static array of values for the ASN.1 CLASS definition. Each structure in the array is the equivalent C structure representing the corresponding ASN.1 information object

An example of an Information Object Set definition that is derived from the ASN.1 ATTRIBUTE class above is as follows:

```
SupportedAttributes ATTRIBUTE ::= { name | commonName }
```

This results in the generation of the following C constant:

```
ATTRIBUTE SupportedAttributes[2];
int SupportedAttributes_Size = 2;
```

Code generated in the Information Object Set initialization function:

```
SupportedAttributes[0].TypeSize = sizeof(_name_Type);
SupportedAttributes[0].encodeType = &asn1E__name_Type;
SupportedAttributes[0].decodeType = &asn1D__name_Type;
SupportedAttributes[0].id.numids = 3;
SupportedAttributes[0].id.subid[0] = 0;
SupportedAttributes[0].id.subid[1] = 1;
SupportedAttributes[0].id.subid[2] = 1;
```

```
SupportedAttributes[1].TypeSize = sizeof(_commonName_Type);
SupportedAttributes[1].encodeType = &asn1E__commonName_Type;
SupportedAttributes[1].decodeType = &asn1D__commonName_Type;
SupportedAttributes[1].id.numids = 3;
SupportedAttributes[1].id.subid[0] = 0;
SupportedAttributes[1].id.subid[1] = 1;
SupportedAttributes[1].id.subid[2] = 1;
SupportedAttributes[1].id.subid[3] = 1;
```

C++ Code Generation

In C++, ASN.1 information object sets are mapped to C++ classes. In this case, a C++ singleton class is generated. This class contains a container to hold an instance of each of the ASN.1 information object C++ objects in a static array. The class also contains an object lookup method for each of the key fields. Key fields are identified in the class as either a) fields that are marked unique, or b) fields that are referenced in table constraints with the '@' notation.

The generated constructor initializes all required values and information objects.

An example of an information object set that uses the information object class defined above is as follows:

```
SupportedAttributes ATTRIBUTE ::= { name | commonName }
```

This results in the generation of the following C++ class:

```
class EXTERN SupportedAttributes {
  protected:
     ATTRIBUTE* mObjectSet[2];
     const size_t mNumObjects;
     static SupportedAttributes* mpInstance;
     SupportedAttributes (OSCTXT* pctxt);

  public:
     ATTRIBUTE* lookupObject (ASN1TObjId _id);
     static SupportedAttributes* instance(OSCTXT* pctxt);
} ;
```

The `mObjectSet` array is the container for the information object classes. These objects are created and this array populated in the class constructor. Note that this is a singleton class (as evidenced by the protected constructor and `instance` methods). Therefore, the object set array is only initialized once the first time the `instance` method is invoked.

The other method of interest is the `lookupObject` method. This was generated for the id field because it was identified as a key field. This determination was made because `id` was declared to be UNIQUE in the class definition above. A field can also be determined to be a key field if it is referenced via the `@` notation in a table constraint in a standard type definition. For example, in the following element assignment:

```
argument OPERATION.&Type ({SupportedAttributes}{@opcode})
```

the `opcode` element's *ATTRIBUTE* class field is identified as a key field.

XSD TO C/C++ TYPE MAPPINGS

ASN1C can accept as input XML schema definition (XSD) specifications in addition to ASN.1 specifications. If an XSD specification is compiled, the compiler does internal translations of the XSD types into equivalent ASN.1 types as specified in ITU-T standard X.694. The following sections provide information on the translations and the C/C++ type definitions generated for the different XSD types.

XSD Simple Types

The translation of XSD simple types into ASN.1 types is straightforward; in most cases, a one-to-one mapping from XSD simple type to ASN.1 primitive type exists. The following table summarizes these mappings:

XSD Simple Type	ASN.1 Type
anyURI	UTF8String
base64Binary	OCTET STRING
boolean	BOOLEAN
byte	INTEGER (-128..127)
date	UTF8String
datetime	UTF8String
decimal	UTF8String
double	REAL
duration	UTF8String
ENTITIES	SEQUENCE OF UTF8String
ENTITY	UTF8String
float	REAL
gDay	UTF8String
gMonth	UTF8String
gMonthDay	UTF8String
gYear	UTF8String
gYearMonth	UTF8String
hexBinary	OCTET STRING
ID	UTF8String
IDREF	UTF8String
IDREFS	SEQUENCE OF UTF8String
integer	INTEGER

XSD Simple Type	**ASN.1 Type**
int	INTEGER (-2147483648..2147483647)
language	UTF8String
long	INTEGER (-9223372036854775808..9223372036854775807)
Name	UTF8String
NCName	UTF8String
negativeInteger	INTEGER (MIN..-1)
NMTOKEN	UTF8String
NMTOKENS	SEQUENCE OF UTF8String
nonNegativeInteger	INTEGER (0..MAX)
nonPositiveInteger	INTEGER (MIN..0)
normalizedString	UTF8String
positiveInteger	INTEGER (1..MAX)
short	INTEGER (-32768..32767)
string	UTF8String
time	UTF8String
token	UTF8String
unsignedByte	INTEGER (0..255)
unsignedShort	INTEGER (0..65535)
unsignedInt	INTEGER (0..4294967295)
unsignedLong	INTEGER (0..18446744073709551615)

The C/C++ mappings for these types can be found in the section above on ASN.1 type mappings.

XSD Complex Types

XSD complex types and selected simple types are mapped to equivalent ASN.1 constructed types. In some cases, simplifications are done to make the generated code easier to work with. The following are mappings for specific XSD complex types.

xsd:sequence

The XSD sequence type is normally mapped to an ASN.1 SEQUENCE type. The following items describe special processing that may occur when processing a sequence definition:

- If the resulting SEQUENCE type contains only a single repeating element, it is converted into a SEQUENCE OF type. This can occur if either the sequence declaration has a maxOccurs attribute with a value greater than one or if the single element inside has a similar maxOccurs attribute.

- If the sequence contains an element that has a ‘minOccurs=“0”’ attribute declaration, the element is mapped to be an OPTIONAL element in the resulting ASN.1 SEQUENCE assignment.
- If the sequence contains a repeating element (denoted by having a ‘maxOccurs’ attribute with a value greater than one), then a SEQUENCE OF type for this element is used in the ASN.1 SEQUENCE for the element.
- If attributes are defined within the complex type container containing the sequence group, attributes are defined, these attribute declarations are added to the resulting ASN.1 as element declarations as per the X.694 standard. In XML encodings, these appear as attributes in the markup; in binary encodings, they are elements.

Example

```
<xsd:complexType name="Name">
   <xsd:sequence>
      <xsd:element name="givenName" type="xsd:string "/>
      <xsd:element name="initial" type="xsd:string" minOccurs="0"/>
      <xsd:element name="familyName" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
```

would result in the creation of the following C type definition:

```
typedef struct EXTERN Name {
   struct {
      unsigned initialPresent : 1;
   } m;
   const OSUTF8CHAR* givenName;
   const OSUTF8CHAR* initial;
   const OSUTF8CHAR* familyName;
} Name;
```

xsd:all

The xsd:all type is similar to an ASN.1 SET in that it allows for a series of elements to be specified that can be transmitted in any order. However, due to some technicalities with the type, it is modeled in X.694 to be a SEQUENCE type with a special embedded array called order. This array specifies the order in which XML elements were received if XML decoding of an XML instance was done. If this information were then retransmitted in binary using BER or PER, the order information would be encoded and transmitted followed by the SEQUENCE elements in the declared *order*. If the data were then serialized back into XML, the order information would be used to put the elements back in the same order in which they were originally received.

The mapping to C type would be the same as for *xsd:sequence* above with the addition of the special order array. An example of this is as follows:

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

would result in the creation of the following C type definition:

```
typedef struct EXTERN Name {
   struct {
      OSUINT32 n;
      OSUINT8 elem[3];
   } _order;
   const OSUTF8CHAR* givenName;
   const OSUTF8CHAR* initial;
   const OSUTF8CHAR* familyName;
} Name;
```

In this case, the *_order* element is for the order element described earlier. Normally, the user does not need to deal with this item. When the generated initialization is called for the type (or C++ constructor), the array will be set to indicate elements should be transmitted in the declared order. If XML decoding is done, the contents of the array will be adjusted to indicate the order the elements were received in.

xsd:choice and xsd:union

The *xsd:choice* type is converted to an ASN.1 CHOICE type. ASN1C generates exactly the same code. For example:

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

in this case, the generated code is the same as for ASN.1 CHOICE:

```
#define T_NamePart_givenName             1
#define T_NamePart_initial               2
#define T_NamePart_familyName            3

typedef struct EXTERN NamePart {
   int t;
   union {
      /* t = 1 */
      const OSUTF8CHAR* givenName;
      /* t = 2 */
      const OSUTF8CHAR* initial;
      /* t = 3 */
      const OSUTF8CHAR* familyName;
   } u;
} NamePart;
```

Similar to *xsd:choice* is *xsd:union*. The main difference is that *xsd:union* alternatives are unnamed. As specified in X.694, special names are generated in this case using the base name “alt”. The generated name for the first member is “alt”; names for successive members are “alt-*n*” where *n* is a sequential number starting at 1. An example of this naming is as follows:

```
<xsd:simpleType name="MyType">
   <xsd:union memberTypes="xsd:int xsd:language"/>
</xsd:simpleType>
```

This generates the following C type definition:

```
#define T_MyType_alt         1
#define T_MyType_alt_1       2

typedef struct EXTERN MyType {
   int t;
   union {
      /* t = 1 */
      OSINT32 alt;
      /* t = 2 */
      const OSUTF8CHAR* alt_1;
   } u;
} MyType;
```

Repeating Groups

Repeating groups are specified in XML schema definitions using the *minOccurs* and *maxOccurs* facets on sequence or choice definitions. These items are converted to ASN.1 SEQUENCE OF types.

An example of a repeating group is as follows:

```
<xsd:complexType name="Names">
   <xsd:sequence maxOccurs="unbounded">
      <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, ASN1C pulls the group out to form a type of form *<name>-element* where *<name>* would be replaced with the complex type name. In this case, the name would be *Names-element*. A SEQUENCE OF type is then formed based on this newly formed type (SEQUENCE OF *Names-element*). The generated C code corresponding to this is as follows:

```
typedef struct EXTERN Names_element {
   const OSUTF8CHAR* givenName;
   const OSUTF8CHAR* initial;
   const OSUTF8CHAR* familyName;
} Names_element;

/* List of Names_element */
```

```
typedef OSRTDList Names;
```

This generated code is not identical to the code generated by performing an X.694 translation to ASN.1 and compiling the resulting specification with ASN1C; it is much simpler. The generated encoder and decoder make the necessary adjustments to ensure that the encodings are the same regardless of the process used.

Repeating Elements

It is common in XSD to specify that elements within a composite group can occur a multiple number of times. For example:

```
<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" maxOccurs="2"/>
   </xsd:sequence>
</xsd:complexType>
```

In this case, the `familyName` element may occur one or two times. (If `minOccurs` is absent, its default value is 1.) X.694 specifies that a SEQUENCE OF type be formed for this element and then the element renamed to `familyName-list` to reference this element. The C code produced by this transformation is as follows:

```
typedef struct EXTERN Name {
   const OSUTF8CHAR* givenName;
   const OSUTF8CHAR* initial;
   struct {
      OSUINT32 n;
      const OSUTF8CHAR* elem[2];
   } familyName_list;
} Name;
```

In this case, an array was used to represent `familyName_list`. In others, a linked list might be used to represent the repeating item.

xsd:list

Another way to represent repeating items in XSD is by using xsd:list. This is a simple type in XSD that refers to a space-separated list of repeating items. When the list is converted to ASN.1, it is modeled as a SEQUENCE OF type.

For example:

```
<xsd:simpleType name="MyType">
   <xsd:list itemType="xsd:int"/>
</xsd:simpleType>
```

results in the generation of the following C type:

```
typedef struct EXTERN MyType {
      OSUINT32 n;
      OSINT32 *elem;
   } MyType;
```

Special code is added to the generated XML encode and decode function to ensure the data is encoded in spaceseparated list form instead of as XML elements.

xsd:any

The `xsd:any` element is a wildcard placeholder that allows an occurence of any element definition to occur at a given location. It is similar to the ASN.1 open type and can be modeled as such; however, ASN1C uses a special type for these items (`OSXSDAny`) that allows for either binary or xml textual data to be stored. This allows items to be stored in binary form if binary encoding rules are being used and XML text form if XML text encoding is used.

The definition of the `OSXSDAny` type is as follows:

```
typedef enum { OSXSDAny_binary, OSXSDAny_xmlText } OSXSDAnyAlt;
typedef struct OSXSDAny {
   OSXSDAnyAlt t;
   union {
      OSOpenType* binary;
      const OSUTF8CHAR* xmlText;
   } u;
} OSXSDAny;
```

The t value is set to either `OSXSDAny_binary` or `OSXSDAny_xmlText` to identify the content type. If binary decoding is being done (BER, DER, CER, or PER), the decoder will populate the `binary` alternative element; if XML decoding is being done, the `xmlText` field is populated. It is possible to perform XML-to-binary transcoding of a multi-part message (for example, a SOAP message) by decoding each part and then reencoding in binary form and switching the content type within this structure.

An example of a sequence with a single wildcard element is as follows:

```
<xsd:complexType name="MyType">
   <xsd:sequence>
      <xsd:element name="ElementOne" type="xsd:string"/>
      <xsd:element name="ElementTwo" type="xsd:int"/>
      <xsd:any processContents="lax"/>
   </xsd:sequence>
</xsd:complexType>
```

The generated C type definition is as follows:

```
typedef struct EXTERN MyType {
   const OSUTF8CHAR* elementOne;
   OSINT32 elementTwo;
   OSXSDAny elem;
} MyType;
```

As per the X.694 standard, the element was given the standard element name `elem`.

XML Attribute Declarations

XML attribute declarations in XSD are translated into ASN.1 elements that are added to a SEQUENCE type. In binary encodings, there is no way to tell encoded attributes apart from encoded elements. They just represent data fields in ASN.1. For XML, special logic is added to the generated XML encoders and decoders to encode and decode the items as attributes.

An example of an attribute being added to an xsd:sequence declaration is as follows:

```
<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:attribute name ="occupation" type="xsd:string"/>
</xsd:complexType>
```

This results in the following C type definition being generated:

```
typedef struct EXTERN Name {
   struct {
      unsigned occupationPresent : 1;
   } m;
   const OSUTF8CHAR* occupation;
   const OSUTF8CHAR* givenName;
   const OSUTF8CHAR* initial;
   const OSUTF8CHAR* familyName;
} Name;
```

The attribute is marked as optional (hence the `occupationPresent` flag in the bit mask) since XML attributes are optional by default. The attribute declarations also occur before the element declarations in the generated structure.

Attributes can also be added to a choice group. In this case, an ASN.1 SEQUENCE is formed consisting of the attribute elements and an embedded element, `choice`, for the choice group. An example of this is as follows:

```
<xsd:complexType name="NamePart">
   <xsd:choice>
      <xsd:element name="givenName" type="xsd:string "/>
      <xsd:element name="initial" type="xsd:string"/>
      <xsd:element name="familyName" type="xsd:string"/>
   </xsd:choice>
   <xsd:attribute name ="occupation" type="xsd:string"/>
</xsd:complexType>
```

This results in the following C type definitions being generated:

```
#define T_NamePart_choice_givenName        1
#define T_NamePart_choice_initial          2
#define T_NamePart_choice_familyName       3

typedef struct EXTERN NamePart_choice {
   int t;
```

```
      union {
         /* t = 1 */
         const OSUTF8CHAR* givenName;
         /* t = 2 */
         const OSUTF8CHAR* initial;
         /* t = 3 */
        const OSUTF8CHAR* familyName;
      } u;
   } NamePart_choice;

   typedef struct EXTERN NamePart {
      struct {
         unsigned occupationPresent : 1;
      } m;
      const OSUTF8CHAR* occupation;
      NamePart_choice choice;
   } NamePart;
```

In this case, `occupation` attribute declaration was added as before. But the `choice` group became a separate embedded element called `choice` which the ASN1C compiler pulled out to create the `NamePart_choice` temporary type. This type was then referenced by the `choice` element in the generated type definition for `NamePart`.

xsd:anyAttribute

An *xsd:anyAttribute* declaration is the attribute equivalent to the *xsd:any* wildcard element declaration described earlier. The main difference is that a single *xsd:anyAttribute* declaration indicates that any number of undeclared attributes may occur whereas *xsd:any* without a *maxOccurs* facet indicates that only a single wildcard element may occur at that position.

X.694 models *xsd:anyAttribute* as a *SEQUENCE OF UTF8String* in ASN.1. Each string in the sequence is expected to be in a `name=‘value’` format. The generated C type for this is simply a linked list of character strings. For example:

```
<xsd:complexType name="MyType">
   <xsd:anyAttribute processContents="lax"/>
</xsd:complexType>
```

results in the following C type:

```
typedef struct EXTERN MyType {
   /* List of const OSUTF8CHAR* */
   OSRTDList attr;
} MyType;
```

To populate a variable of this type for encoding, one would add `name=‘value’` strings to the list for each attribute. For example:

```
MyType myVar;
rtxDListInit (&myVar.attr);
rtxDListAppend (&ctxt, &myVar.attr, OSUTF8(“attr1=‘value1’”));
rtxDListAppend (&ctxt, &myVar.attr, OSUTF8(“attr2=‘value2’”));
```

and so on.

xsd:simpleContent

The *xsd:simpleContent* type is used to either extend or restrict an existing simple type definition. In the case of extension, the common use is to add attributes to a simple type. ASN1C will generate a C structure in this case with an element called *base* that is of the simple type being extended. An example of this is as follows:

```
<xsd:complexType name="SizeType">
   <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
         <xsd:attribute name="system" type ="xsd:token"/>
      </xsd:extension>
   </xsd:simpleContent>
</xsd:complexType>
```

this results in the following generated C type definition:

```
typedef struct EXTERN SizeType {
   struct {
      unsigned system_Present : 1;
   } m;
   const OSUTF8CHAR* system_;
   OSINT32 base;
} SizeType;
```

In this case, the attribute `system` was added first (note the name change to `system_` which was the result of `system` being determined to be a C reserved word). The `base` element is then added and is of type `OSINT32`, the default type used for `xsd:integer`.

In the case of a simple content restriction, the processing is similar. A complete new separate type is generated even if the result of the restriction leaves the original type unaltered (i.e. the restriction is handled by code within the generated encode and/or decode function). This proves to be a cleaner solution in most cases than trying to reuse the type being restricted. For example:

```
<xsd:complexType name="SmallSizeType">
   <xsd:simpleContent>
      <xsd:restriction base="SizeType">
         <xsd:minInclusive value="2"/>
         <xsd:maxInclusive value="6"/>
         <xsd:attribute name="system" type ="xsd:token" use="required"/>
      </xsd:restriction>
   </xsd:simpleContent>
</xsd:complexType>
```

This applies a restriction to the SizeType that was previously derived. In this case, the generated C type is as follows:

```
typedef struct EXTERN SmallSizeType {
   const OSUTF8CHAR* system_;
   OSINT32 base;
} SmallSizeType;
```

In this case, the type definition is almost identical to the original `SizeType`. The only exception is that the bit mask field for optional elements is removed—a consequence of the `use="required"` attribute that was added to the `system` attribute declaration. The handling of the `minInclusive` and `maxInclusive` attributes is handled inside the generated encode and decode function in the form of constraint checks.

xsd:complexContent

The *xsd:complexContent* type is used to extend or restrict complex types in different ways. It is similar to deriving types from base types in higher level programming languages such as C++ or Java. A common usage pattern in the case of extension is to add additional elements to an existing sequence or choice group. In this case, a new type is formed that contains all elements—those declared in the base definition and those in the derived type. Also generated is a new type with the name `<baseType>_derivations` which is a choice of all of the different derivations of the base type. This is used wherever the complex content base type is referenced to allow any derivation of the type to be used in a message.

An example of this is as follows:

```
<xsd:complexType name="MyType">
   <xsd:sequence>
      <xsd:element name="ElementOne" type="xsd:string"/>
      <xsd:element name="ElementTwo" type="xsd:int"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="MyExtendedType">
   <xsd:complexContent>
      <xsd:extension base="MyType">
         <xsd:sequence>
            <xsd:element name="ElementThree" type="xsd:string"/>
            <xsd:element name="ElementFour" type="xsd:int"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

The base type in this case is `MyType` and it is extended to contain two additional elements in `MyExtendedType`. The resulting C type definitions for `MyType`, `MyExtendedType`, and the special derivations type are as follows:

```
typedef struct EXTERN MyType {
   const OSUTF8CHAR* elementOne;
   OSINT32 elementTwo;
} MyType;

typedef struct EXTERN MyExtendedType {
   const OSUTF8CHAR* elementOne;
   OSINT32 elementTwo;
   const OSUTF8CHAR* elementThree;
   OSINT32 elementFour;
} MyExtendedType;

#define T_MyType_derivations_myType 1
#define T_MyType_derivations_myExtendedType 2
```

```
typedef struct EXTERN MyType_derivations {
   int t;
   union {
      /* t = 1 */
      MyType *myType;
      /* t = 2 */
      MyExtendedType *myExtendedType;
   } u;
} MyType_derivations;
```

The derivations type is a choice between the base type and all derivations of that base type. It will be used wherever the base type is referenced. This makes it possible to use an instance of the extended type in these places.

The case of restriction is handled in a similar fashion. In this case, instead of creating a new type with additional elements, a new type is created with all restrictions implemented. This type may be identical to the base type definition.

Substitution Groups

A substitution group is similar to a complex content type in that it allows derivations from a common base. In this case, however, the base is an XSD element and the substitution group allows any of a set of elements defined to be in the group to be used in the place of the base element. A simple example of this is as follows:

```
<xsd:element name="MyElement" type="MyType"/>
<xsd:complexType name="MyType">
   <xsd:sequence>
      <xsd:element ref="MyBaseElement"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:element name="MyBaseElement" type="xsd:string"/>
<xsd:element name="MyExtendedElement" type="xsd:string" substitutionGroup="MyBaseElement "/>
```

In this case, the global element `MyElement` references `MyType` which is defined as a sequence with a single element reference to `MyBaseElement`. `MyBaseElement` is the head element in a substitution group that also includes `MyExtendedElement`. This means `MyType` can either reference `MyBaseElement` or `MyExtendedElement`.

As per X.694, ASN1C generates a special type that acts as a container for all the different possible elements in the substitution group. This is a choice type with the name `<BaseElement>_group` where `<BaseElement>` would be replaced with the name of the subsitution group head element (`MyBaseElement` in this case).

The generated C type definitions for the above XSD definitions follow:

```
typedef const OSUTF8CHAR* MyBaseElement;

typedef const OSUTF8CHAR* MyExtendedElement;

#define T_MyBaseElement_group_myBaseElement 1
#define T_MyBaseElement_group_myExtendedElement 2
```

```
typedef struct EXTERN MyBaseElement_group {
   int t;
   union {
      /* t = 1 */
      MyBaseElement myBaseElement;
      /* t = 2 */
      MyExtendedElement myExtendedElement;
   } u;
} MyBaseElement_group;

typedef struct EXTERN MyType {
   MyBaseElement_group myBaseElement;
} MyType;

typedef MyType MyElement;
```

In this case, if `MyElement` or `MyType` is used, it can be populated with either base element or extended element data.

Generated C/C++ Source Code

Header (.h) File

The generated C or C++ include file contains a section for each ASN.1 production defined in the ASN.1 source file. Different items will be generated depending on whether the selected output code is C or C++. In general, C++ will add some additional items (such as a control class definition) onto what is generated for C.

The following items are generated for each ASN.1 production:

- Tag value constant
- Choice tag constants (CHOICE type only)
- Named bit number constants (BIT STRING type only)
- Enumerated type option values (ENUMERATED or INTEGER type only)
- C type definition
- Encode function prototype
- Decode function prototype
- Other function prototypes depending on selected options (for example, print)
- C++ control class definition (C++ only)

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

```
/**************************************************************/
/*                                                            */
/*  EmployeeNumber                                            */
/*                                                            */
/**************************************************************/

#define TV_EmployeeNumber(TM_APPL|TM_PRIM|2)

typedef OSINT32 EmployeeNumber;

EXTERN int asn1E_EmployeeNumber (OSCTXT* pctxt,
   EmployeeNumber *pvalue, ASN1TagType tagging);

EXTERN int asn1D_EmployeeNumber (OSCTXT* pctxt,
   EmployeeNumber *pvalue, ASN1TagType tagging, int length);
```

This corresponds to the following ASN.1 production specification:

```
EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER
```

In this definition, *TV_EmployeeNumber* is the tag constant. Doing a logical OR on the class, form, and identifier fields forms this constant. This constant can be used in a comparison operation with a tag parsed from a message.

The following line:

```
typedef OSINT32 EmployeeNumber;
```

declares EmployeeNumber to be of an integer type (note: OSINT32 and other primitive type definitions can be found in the *osSysTypes.h* header file).

asn1E_EmployeeNumber and *asn1D_EmployeeNumber* are function prototypes for the encode and decode functions respectively. These are BER function prototypes. If the -per switch is used, PER function prototypes are generated. The PER prototypes begin with the prefix *asn1PE_* and *asn1PD_* for encoder and decoder respectively. XER function prototypes begin with *asn1XE_* and *asn1XD_*.

A sample section from a C++ header file for the same production is as follows:

```
/**************************************************************/
/*                                                            */
/*  EmployeeNumber                                            */
/*                                                            */
/**************************************************************/

#define TV_EmployeeNumber(TM_APPL|TM_PRIM|2)

typedef OSINT32 ASN1T_EmployeeNumber;

class EXTERN ASN1C_EmployeeNumber :
public ASN1CType
{
  protected:
   ASN1T_EmployeeNumber& msgData;
  public:
   ASN1C_EmployeeNumber (ASN1T_EmployeeNumber& data);
   ASN1C_EmployeeNumber (
      ASN1MessageBufferIF& msgBuf, ASN1T_EmployeeNumber& data);

   // standard encode/decode methods (defined in ASN1CType base class):
   // int Encode ();
   // int Decode ();

   // stream encode/decode methods:
   int EncodeTo (ASN1MessageBufferIF& msgBuf);
   int DecodeFrom (ASN1MessageBufferIF& msgBuf);
} ;

EXTERN int asn1E_EmployeeNumber (OSCTXT* pctxt,
   ASN1T_EmployeeNumber *pvalue, ASN1TagType tagging);

EXTERN int asn1D_EmployeeNumber (OSCTXT* pctxt,
   ASN1T_EmployeeNumber *pvalue, ASN1TagType tagging, int length);
```

Note the two main differences between this and the C version:

1. The use of the *ASN1T_ prefix* on the type definition. The C++ version uses the *ASN1T_ prefix* for the typedef and the *ASN1C_ prefix* for the control class definition.

2. The inclusion of the ASN1C_EmployeeNumber control class.

As of ASN1C version 5.6, control classes are not automatically generated for all ASN.1 types. The only types they are generated for are those determined to be *Protocol Data Units* (or PDU's for short). A PDU is a top-level message type in a specification. These are the only types control classes are required for because the only purpose of a control class is to provide the user with a simplified calling interface for encoding and decoding a message. They are not used in any of the ASN1C internally generated logic (the exception to this rule is the XER / XML encoding rules where they are used internally and still must be generated for all types).

A type is determined to be a PDU in two different ways:

1. If it is explicitly declared to be PDU via the <isPDU/> configuration setting or -pdu command-line option.

2. If no explicit declarations exist, a type is determined to be a PDU if it is not referenced by any other types.

In the employee sample program, *EmployeeNumber* would not be considered to be a PDU because it is referenced as an element within the *Employee* production. For the purpose of this discussion, we will assume *EmployeeNumber* was explicitly declared to be a PDU via a configuration setting or command-line specification.

ASN1C_EmployeeNumber is the control class declaration. The purpose of the control class is to provide a linkage between the message buffer object and the ASN.1 typed object containing the message data. The class provides methods such as *EncodeTo* and *DecodeFrom* for encoding and decoding the contents to the linked objects. It also provides other utility methods to make populating the typed variable object easier.

ASN1C always adds an *ASN1C_*prefix to the production name to form the class name. Most generated classes are derived from the standard *ASN1CType* base class defined in asn1Message.h. The following ASN.1 types cause code to be generated from different base classes:

- BIT STRING – The generated control class is derived from the *ASN1CBitStr* class

- SEQUENCE OF or SET OF with linked list storage – The generated control class is derived from the *ASN1CSeqOfList* base class.

- Defined Type – The generated control class for defined types is derived from the generated base class for the reference type. For example, if we have A ::= INTEGER and B ::= A, then B is a defined type and would inherit from the base class generated for A (class ASN1C_B : public ASN1C_A { …).

These intermediate classes are also derived from the *ASN1CType* base class. Their purpose is the addition of functionality specific to the given ASN.1 type. For example, the *ASN1CBitStr* control class provides methods for setting, clearing and testing bits in the referenced bit string variable.

In the generated control class, the msgData member variable is a reference to a variable of the generated type. The constructor takes two arguments – an *Asn1MessageBufferIF* (message buffer interface) object reference and a reference to a variable of the data type to be encoded or decoded. The message buffer object is a work buffer object for encoding or decoding. The interface reference can also be used to specify a stream. Stream classes are derived from this same base class. The data type reference is a reference to the *ASN1T_* variable that was generated for the data type.

EncodeFrom and *DecodeTo* methods are declared that wrap the respective compiler generated C encode and decode stream functions. Standard *Encode* and *Decode* methods exist in the *ASN1CType* base class for direct encoding and decoding to a memory buffer. Command-line options may cause additional methods to be generated. For example, if the –print command line argument was specified; a Print method is generated to wrap the corresponding C print function.

Specification of the XML encoding rules option (*-xer*) causes a number of additional methods to be generated for constructed types. These additional methods are implementations of the standard Simple API for XML (SAX) content handling interface used to parse content from XML messages. The *startElement*, *characters*, and *endElement* methods are implemented as well as additional support methods. The control class is also defined to inherit from the *ASN1XERSAXHandler* base class as well as *ASN1CType* (or one of its descendents).

The equivalent C and C++ type definitions for each of the various ASN.1 types follow.

Generated C Source Files

By default, the ASN1C compiler generates the following set of .c source files for a given ASN.1 module (note: the name of the module would be substituted for <moduleName>):

`<moduleName>.c`	common definitions and functions (for example, asn1Free_<type>) and/or global value constant definitions.
`<moduleName>Enc.c`	encode functions (asn1E_<type>)
`<moduleName>Dec.c`	decode functions (asn1D_<type>)

If additional options are used (such as –genPrint, -genCopy, etc), additional files will be generated:

`<moduleName>Copy.c`	copy functions, generated if –genCopy is specified
`<moduleName>Print.c`	print functions, generated if –genPrint is specified
`<moduleName>Compare.c`	comparison functions, generated if –genCompare is specified

`<moduleName>PrtToStr.c`	print-to-string functions, generated if –genPrtToStr is specified
`<moduleName>PrtToStrm.c`	print-to-stream functions, generated if –genPrtToStrm is specified
`<moduleName>Table.c`	table constraint functions, generated if –genTable option is specified
`<moduleName>Test.c`	test functions, generated if –genTest is specified

If *–genCopy, -genPrint*, etc have a filename parameter then the code will be written to the given file instead of the default one. If the *–cfile <filename>* option is used and *–genCopy, -genPrint*, etc options do not have parameters then all code will be placed in one source file with name <filename>.

Maximum Lines per File

In each of the cases above, it is possible to specify an approximate maximum number of lines that each of the generated .c files may contain. This is done using the *-maxlines* option. If *-maxlines* is specified with no parameter, a default maximum number of lines (50,000) will be set; otherwise, the given value will be used.

If the given maximum lines limit is surpassed in a file, a new file will be started with an "_1" appended, for example *<moduleName>Enc_1.c*. Additional files will be numbered sequentially if necessary (_2, _3, etc.). Note that this limit is a lower threshold and not exact. A complete compilation unit (for example, a function) will not be split because of this threshold. The way it works is the threshold is checked before the output of a compilation unit. If it is found to be exceeded, a new file is started at that time. Therefore, a user should plan for a reserve to be in place above the limit to compensate for this overflow.

The reason for having this limit is because some C/C++ compilers have problems with very large .c files. For example, one product will not allow the debugger to work on lines in a file over the 64k threshold.

Use of the -maxcfiles Option

The *-maxcfiles* option allows generation of more compact code by putting each encode, decode, copy, compare, etc function into a separate file. This allows the linker to link in only the required functions as opposed to all functions in a compiled object module. This option might be useful for applications that have minimal space requirements (for example, embedded systems).

Note

Some sophisticated linkers have the capability to pull individual functions out of an object module directly for final inclusion in the target executable or shared object file. In this case, the -maxcfiles option does not provide any advantage in reducing the size of the application program.

To achieve the best results it is necessary to put all compiled object files into an object library (.a or .lib file) and include this library in the link command. The *–genMake* option when used in conjunction with *–maxcfiles* will generate a makefile that will compile each of the generated files and add them to a library with a name based on the name of the ASN.1 module being compiled (*<moduleName>.lib* for Windows or *lib<moduleName>.a* for *NIX).

The format of each generated .c file name is as follows:

```
asn1<suffix>_<prodname>.c
```

where `<suffix>` depends on encoding rules and the function type (encode, decode, free, etc.) and `<prodname>` is the ASN.1 production name.

For example, consider one type definition within the `employee.asn` ASN.1 specification:

```
Employee DEFINITIONS ::= BEGIN

[...]

Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
   givenName IA5String,
   initial IA5String,
   familyName IA5String
}

[...]

END
```

By default, the following .c files would be generated (note: this assumes no additional code generation options were selected):

```
Employee.c
EmployeeEnc.c
EmployeeDec.c
```

If -*maxcfiles* was selected as in the following command line:

```
asn1c employee.asn -c -ber -trace –maxcfiles
```

Running ASN1C with the -*maxcfiles* option, the following .c files for this type would be generated for the Name type:

```
asn1D_Name.c
asn1E_Name.c
```

These contain the functions to decode Name and encode Name respectively. Similar files would be generated for the other productions in the module as well.

Generated C++ files

In general, the generation logic for C++ is similar to the logic for C. Instead of the .c file extension, .cpp is used:

`<moduleName>.cpp`	Common definitions and functions (for example, asn1Free_<type>) and/or global value constant definitions. This file also contains constructors, destructors and all methods for ASN1C_<Type> and ASN1T_<Type> control classes.
`<moduleName>Enc.cpp`	C encode functions and C++ encode methods.
`<moduleName>Dec.cpp`	C decode functions and C++ decode methods.

If additional options are used (such as –genPrint, -genCopy, etc), additional files will be generated:

Filename	**Description**
`<moduleName>Copy.cpp`	copy functions, generated if –genCopy is specified
`<moduleName>Print.cpp`	print functions, generated if –genPrint is specified
`<moduleName>Compare.cpp`	comparison functions, generated if –genCompare is specified
`<moduleName>PrtToStr.cpp`	print-to-string functions, generated if –genPrtToStr is specified
`<moduleName>PrtToStrm.cpp`	print-to-stream functions, generated if –genPrtToStrm is specified
`<moduleName>Table.cpp`	table constraint functions, generated if –genTable option is specified
`<moduleName>Test.cpp`	test functions, generated if –genTest is specified

The -*maxcfiles* option for C++ works very similar to how it works for C. The only differences are a few additional files are generated and the .cpp extension is used instead of .c. Additional files are generated to hold *ASN1C_<Type>* and *ASN1T_<Type>* control classes. The format of the filenames of these files is as follows:

```
asn1<suffix>_<prodname>.cpp
ASN1C_<prodname>.cpp
ASN1T_<prodname>.cpp
```

where `<suffix>` depends on the encoding rules and function type selected (encode, decode, free, etc.) and `<prodname>` is the ASN.1 production name.

For the example presented previously in the C Files section, the following files would be generated for the Name production in the `employee.asn` file:

```
asn1D_Name.cpp
asn1E_Name.cpp
ASN1T_Name.cpp
ASN1C_Name.cpp
```

These contain the functions to decode *Name* and encode *Name* respectively. The *ASN1T_Name.cpp* file contains the type class methods, and the *ASN1C_Name.cpp* files contains the control class methods. Note that not all productions have a control class (only PDU types do for BER or PER) therefore the *ASN1C_<type>.cpp* file may not be generated.

Similar files would be generated for the other productions in the module as well.

Note that for C++, the code reduction effect is less than that for pure C. This is because most of the linkers cannot omit virtual methods even if they are not being used by the application. These virtual methods refer to separate C functions and these functions are being linked into the application even if they are not actually used. But, still, the size of the final application created with *–maxcfiles* option should be less than the size of the application created without this option.

Generated C/C++ files and the -compat Option

ASN1C 5.6 and below did not generate separate files for common definitions, encode and decode functions (*<moduleName>.c/.cpp, <moduleName>Enc.c/.cpp, <moduleName>Dec.c/.cpp*). All code was generated in a single file with the name *<moduleName>.c/.cpp*. If it is necessary to maintain this behavior then use the *–compat 5.6* option.

Also, the behavior of the *-cfile* option is slightly changed in ASN1C 5.7 and above. In 5.6 and below, the *–cfile* option did not have any effect for files containing copy, print, compare, etc functions. For ASN1C 5.7 and above, *–cfile* causes everything to be output to one file unless specific filename parameters are specified with *–genPrint*, *-genCopy*, etc. Once again, to maintain the previous behavior the –compat 5.6 option can be used.

Generated C++ files and the -symbian Option

ASN1C version 6.1 introduced the -symbian option to generate code that targets the Symbian platform. While an exhaustive discussion of the differences between Symbian C++ and standard C++ is impractical for this User's Guide, the differences in generated code are relatively minimal. Two principle areas of concern are writable static data (WSD) and extern linkage.

Writable Static Data

Writable static data are per-process data that exist throughout the lifetime of the process. The use of WSD complicates memory management in many cases, especially in shared libraries. A minimum of four kilobytes is allocated for WSD every single time a DLL is loaded, even if less space is required. If 50 bytes were needed, for example, 4046 bytes would be wasted every time the DLL was loaded. For this reason, the use of WSD is highly discouraged.

In practice, WSD are globally scoped: variables declared outside of a function, struct, or class, and static variables declared in functions. WSD may be eliminated by modifying primitive types with *const*. Complex types (i.e., classes or structs) with non-trivial constructors will be marked as WSD whether marked *const* or not.

It is common in generated code to use lookup tables for some types (e.g., ENUMERATED). These tables are composed of simple types and marked as *const* to avoid being marked as WSD by Symbian compilers.

Extern Linkage

Most common compilers support applying external linkage to an entire class, but Symbian's does not. Symbian also requires that both prototype *and* implementation be marked with the appropriate linkage. When the -*symbian* option is specified, generated code is modified to accommodate these requirements.

The following specification will demonstrate the differences between code generated with Symbian and without:

```
Test DEFINITIONS ::= BEGIN

A ::= NULL

END
```

The usual class definition for this specification looks like this:

```
class EXTERN ASN1C_A :
public ASN1CType
{
   protected:
   public:
   ASN1C_A ();
   ASN1C_A (OSRTMessageBufferIF& msgBuf);
   ASN1C_A (OSRTContext &context);
   // standard encode/decode methods (defined in ASN1CType base class):
   // int Encode ();
   // int Decode ();

   // stream encode/decode methods:
   int EncodeTo (OSRTMessageBufferIF& msgBuf);
   int DecodeFrom (OSRTMessageBufferIF& msgBuf);
} ;
```

It is very similar to the Symbian class definition:

```
class ASN1C_A :
public ASN1CType
{
protected:
public:
   EXTERN ASN1C_A ();
   EXTERN ASN1C_A (OSRTMessageBufferIF& msgBuf);
   EXTERN ASN1C_A (OSRTContext &context);
```

```
      // standard encode/decode methods (defined in ASN1CType base class):
      // int Encode ();
      // int Decode ();

      // stream encode/decode methods:
      EXTERN int EncodeTo (OSRTMessageBufferIF& msgBuf);
      EXTERN int DecodeFrom (OSRTMessageBufferIF& msgBuf);

   } ;
```

Note the use of `EXTERN` in the generated code: it prefixes the constructors and the encoding and decoding functions, but not the class declaration. These prefixes are repeated in the implementation:

```
   EXTERN ASN1C_A::ASN1C_A () : ASN1CType()
   {
   }
```

Users should not have to modify generated code for use on the Symbian platform, but should be aware of these particular differences when writing Symbian applications.

Generated Build Files

Generated Makefile

The *-genmake* option causes a portable makefile to be generated to assist in the C or C++ compilation of all of the generated C or C++ source files. This makefile contains a rule to invoke ASN1C to regenerate the .c and .h files if any of the dependent ASN.1 source files are modified. It also contains rules to compile all of the C or C++ source files. Header file dependencies are generated for all the C or 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.

When a makefile is generated, it is assumed that the ASN1C project exists within the ASN1C installation directory tree. The generation logic tries to determine the root directory of the installation by traversing upward from the project directory in an attempt to locate the *rtsrc* subdirectory which is assumed to be the installation root directory. The makefile variable *OSROOTDIR* is then set to this value. A similar traversal is done to locate the *platform.mk* and *xmlparser.mk* files. These paths are then set in the makefile. If the project directory is located outside of the ASN1C directory tree, the user must set the *OSROOTDIR* environment variable to point at the ASN1C root directory in order for the makefile generation to be successful. If this is done, it is assumed that the *platform.mk* and *xmlparser.mk* files are located in this directory as well. If the compiler is unable to determine the root directory using any of the methods described above, an error will be generated and the user will need to manually edit the makefile to set the required root directory parameters and makefile include file paths.

Generated VC++ Project Files

The *-vcproj* option causes Microsoft Visual Studio project and workspace files to be generated that can be used to build the generated code. The files are compatible with Visual Studio version 6.0; but higher versions of Visula Studio can convert these files to the newer formats. This option can be used with the *-dll* option that will generate project files to compile all generated code into a DLL and *-mt* that will add multi-threaded compilation options to generated projects.

Because there are several different versions of Visual Studio, the *-vcproj* option takes an optional argument: the release year of the version of Visual Studio used. This modifies the resulting project to link against the appropriate set of libraries distributed with ASN1C. If no year is specified, the project will link against the usual `c` and `cpp` directories. If 2003 is specified, the project will us the `c_vs2003` and `cpp_vs2003` directories. If 2005 is specified, `c_vs2005` and `cpp_vs2005` will be used. Likewise, if 2008 is specified, `c_vs2008` and `cpp_vs2008` will be used.

Generated Encode/Decode Function and Methods

Encode/Decode Function Prototypes

If BER or DER encoding is specified, a BER encode and decode function prototype is generated for each production (DER uses the same form – there are only minor differences between the two types of generated functions). These prototypes are of the following general form:

```
int asn1E_<ProdName> (OSCTXT* pctxt,
   <ProdName>* pvalue, ASN1TagType tagging);

int asn1D_<ProdName> (OSCTXT* pctxt,
   <ProdName>* pvalue, ASN1TagType tagging, int length);
```

The prototype with the *asn1E_* prefix is for encoding and the one with *asn1D_* is for decoding. The first parameter is a context variable used for reentrancy. This allows the encoder/decoder to keep track of what it is doing between function invocations.

The second parameter is for passing the actual data variable to be encoded or decoded. This is a pointer to a variable of the generated type.

The third parameter specifies whether implicit or explicit tagging should be used. In practically all cases, users of the generated function should set this parameter to *ASN1EXPL* (explicit). This tells the encoder to include an explicit tag around the encoded result. The only time this would not be used is when the encoder or decoder is making internal calls to handle implicit tagging of elements.

The final parameter (decode case only) is length. This is ignored when tagging is set to *ASN1EXPL* (explicit), so users can ignore it for the most part and set it to zero. In the implicit case, this specifies the number of octets to be extracted from the byte stream. This is necessary because implicit indicates no tag/length pair precedes the data; therefore it is up to the user to indicate how many bytes of data are present.

If PER encoding is specified, the format of the generated prototypes is different. The PER prototypes are of the following general form:

```
int asn1PE_<ProdName> (OSCTXT* pctxt, <ProdName>[*] value);

int asn1PD_<ProdName> (OSCTXT* pctxt, <ProdName>* pvalue);
```

In these prototypes, the prefixes are different (a ‘P’ character is added to indicate they are PER encoders/decoders), and the tagging argument variables are omitted. In the encode case, the value of the production to be encoded may be passed by value if it is a simple type (for example, BOOLEAN or INTEGER). Structured values will still be passed using a pointer argument.

If XER encoding is specified, function prototypes are generated with the following format:

```
int asn1XE_<ProdName> (OSCTXT* pctxt, <ProdName>[*] value,
                       const char* elemName,
                       const char* attributes);

int asn1XD_<ProdName> (OSCTXT* pctxt, <ProdName>* pvalue);
```

The encode function signature includes arguments for the context and value as in the other cases. It also has an element name argument (*elemName*) that contains the name of the element to be encoded and an attributes argument (*attributes*) that can be used to encode an attributes string. The decode function is generated for PDU-types only - decoding of internally referenced types is accomplished through generated SAX handler callback functions which are invoked by an XML parser.

If XML functions are generated using the -xml switch, the function prototypes are as follows:

```
int XmlEnc_<ProdName> (OSCTXT* pctxt, <ProdName> value,
   const OSUTF8CHAR* elemName, const OSUTF8CHAR* nsPrefix);

int XmlDec_<ProdName> (OSCTXT* pctxt, <ProdName>* pvalue);
```

In this case, the encode function contains an argument for XML element name (*elemName*) and also namespace prefix (*nsPrefix*).

Generated C++ Control Class Definition

A control class definition is generated for each defined production in the ASN.1 source file that is determined to be a *Protocol Data Unit* (PDU). By default, any type defined in an ASN.1 source file that is not referenced by any other type is a PDU. This default behavior can be overridden by using a configuration file setting (*<isPDU/>*) or a command-line option (*-pdu*) to explicitly declare that certain types are PDU's.

The generated control class is derived from the *ASN1CType* base class. This class provides a set of common attributes and methods for encoding/decoding ASN.1 messages. It hides most of the complexity of calling the encode/decode functions directly.

BER/DER or PER Class Definition

The general form of the class definition for BER, DER, or PER encoding rules is as follows:

```
class ASN1C_<name> : public ASN1CType {
  protected:
     ASN1T_<name>& msgData;
  public:
     ASN1C_<name> (ASN1T_<name>& data);
     ASN1C_<name> (
        ASN1MessageBufferIF& msgBuf, ASN1T_<name>& data);

     // standard encode/decode methods (defined in ASN1CType base class):
     // int Encode ();
```

```
      // int Decode ();

      // stream encode/decode methods:
      int EncodeTo (ASN1MessageBufferIF& msgBuf);
      int DecodeFrom (ASN1MessageBufferIF& msgBuf);
} ;
```

The name of the generated class is *ASN1C_<name>* where *<name>* is the name of the production. The only defined attribute is a protected variable reference named *msgData* of the generated type.

Two constructors are generated. The first is for stream operations and allows the control class to be created using only a reference to a variable of the generated type.

The *EncodeTo* and *DecodeFrom* methods can then be used to encode or decode directly to and from a stream. The << and >> stream operators can be used as well.

The second constructor is the legacy form that allows a message buffer to be associated with a data variable at the time of creation. The *Encode* and *Decode* methods defined in the ASN1CType base class can be used with this construction form to encode and decode to the associated buffer.

The constructor arguments are a reference to an *ASN1MessageBufferIF* (message buffer interface) type and a reference to an *ASN1T_<name>* type. The message buffer interface argument is a reference to an abstract message buffer or stream class. Implementations of the interface class are available for BER/DER, PER, or XER encode or decode message buffers or for a BER or XER encode or decode stream.

The *ASN1T_<name>* argument is used to specify the data variable containing data to be encoded or to receive data on a decode call. The procedure for encoding is to declare a variable of this type, populate it with data, and then instantiate the *ASN1C_<name>* object to associate a message buffer object with the data to be encoded. The *Encode* or *Encode To* method can then be called to encode the data. On the decode side, a variable must be declared and passed to the constructor to receive the decoded data.

Note that the *ASN1C_* class declarations are only required in the application code as an entry point for encoding or decoding a top-level message (or Protocol Data Unit – PDU). As of ASN1C version 5.6, control classes are only generated for ASN.1 types that are determined to be PDU's. A type is determined to be a PDU if it is referenced by no other types. This differs from previous versions of ASN1C where control classes were generated for all types. This default behavior can be overridden by using a configuration file entry or the -pdu command-line switch to explicitly declare the PDU types. The <*isPDU/*> flag is used to declare a type to be a PDU in a configuration file. An example of this is as follows:

```
<asn1config>
   <module>
      <name>H323-MESSAGES</name>
      <production>
         <name>H323-UserInformation</name>
         <isPDU/>
      </production>
   </module>
</asn1config>
```

This will cause only a single *ASN1C_* control class definition to be added to the generated code for the *H323- UserInformation* production.

If the module contains no PDUs (i.e,. contains support types only), the <noPDU/> empty element can be specified at the module level to indicate that no control classes should be generated for the module.

XER Class Definition

For the XML encoding rules (XER), the generated class definition is as follows:

```
   class ASN1C_<name> :
     public ASN1CType, ASN1XERSAXHandler
   {

     protected:
        ASN1T_<name>& msgData;
        ... additional control variables
     public:
        ASN1C_<name> (ASN1T_<name>& data);
        ASN1C_<name> (
           ASN1MessageBufferIF& msgBuf, ASN1T_<name>& data);

        // standard encode/decode methods (defined in ASN1CType base class):
        // int Encode ();
        // int Decode ();

        // stream encode/decode methods:
        int EncodeTo (ASN1MessageBufferIF& msgBuf);
        int DecodeFrom (ASN1MessageBufferIF& msgBuf);

        // SAX Content Handler Interface

        virtual void startElement
           (const XMLCh* const uri,
            const XMLCh* const localname,
            const XMLCh* const qname,
            const Attributes& attrs);

        virtual void characters
           (const XMLCh* const chars, const unsigned int length);

        virtual void endElement
           (const XMLCh* const uri,
            const XMLCh* const localname,
            const XMLCh* const qname);

   } ;
```

The main differences between the BER/DER/PER control class definition and this are:

1. The class generated for XER inherits from the *ASN1XERSAXHandler* base class, and

2. The class implements the standard SAX content handler methods.

This allows an object of this class to be registered as a SAX content handler with any SAX-compliant XML parser. The parser would be used to read and parse XML documents. The methods generated by ASN1C would then receive the parsed data via the SAX interface and use the results to populate the data variables with the decoded data.

Note that for XML code generation (-*xml* command-line option), the SAX handler interface is not generated. That is because XML decoders use a pull-parser instead of SAX code to parse the XML input stream.

Generated Methods

For each production, an *EncodeFrom* and *DecodeTo* method is generated within the generated class structure. These are standard methods that initialize context information and then call the generated C-like encode or decode function. If the generation of print functions was specified (by including *–print* on the compiler command line), a *Print* method is also generated that calls the C print function.

For XER, additional methods are generated to implement a SAX content handler interface to an XML parser. This includes a *startElement, characters,* and *endElement* method. An *init* and *finalize* method may also be generated to initialize a variable prior to parsing and to complete population of a variable with decoded data.

Generated Information Object Table Structures

Information Objects and Classes are used to define multi-layer protocols in which “holes” are defined within ASN.1 types for passing message components to different layers for processing. These items are also used to define the contents of various messages that are allowed in a particular exchange of messages. The ASN1C compiler extracts the types involved in these message exchanges and generates encoders/decoders for them. The “holes” in the types are accounted for by adding open type holders to the generated structures. These open type holders consist of a byte count and pointer for storing information on an encoded message fragment for processing at the next level.

The ASN1C compiler is capable of generating code in one of two forms for information in an object specification:

1. Simple form: in this form, references to variable type fields within standard types are simply treated as open types and an open type placeholder is inserted.

2. Table unions form: in this form, all of the classes, objects, and object sets within a specification result in the generation of code for parsing and formatting the information field references within

standard type structures. Open types with relational constraints result in the generation of C union structures that enumerate all of allowed fields as defined by the constraint. This form is selected by using the *-table-unions* command-line option.

3. Legacy table form: this is similar to 2 in that all information object related items result in the generation of additional code. In this case, however, instead of a union structure being generated for open types with relational constraints, a void pointer is used to hold an object in decoded form. This form is selected using the *-tables* command-line option.

To better understand the support in this area, the individual components of Information Object specifications are examined. We begin with the “CLASS” specification that provides a schema for Information Object definitions. A sample class specification is as follows:

```
OPERATION ::= CLASS {
   &operationCode      CHOICE { local INTEGER,
                                global OBJECT IDENTIFIER },
   &ArgumentType,
   &ResultType,
   &Errors             ERROR      OPTIONAL
}
```

Users familiar with ASN.1 will recognize this as a simplified definition of the ROSE OPERATION MACRO using the Information Object format. When a class specification such as this is parsed, information on its fields is maintained in memory for later reference. In the simple form of code generation, the class definition itself does not result in the generation of any corresponding C or C++ code. It is only an abstract template that will be used to define new items later on in the specification. In the table form, if C++ is specified, an abstract base class is generated off of which other classes are derived for information object specifications.

Fields from within the class can be referenced in standard ASN.1 types. It is these types of references that the compiler is mainly concerned with. These are typically “header” types that are used to add a common header to a variety of other message body types. An example would be the following ASN.1 type definition for a ROSE invoke message header:

```
Invoke ::= SEQUENCE {
   invokeID INTEGER,
   opcode OPERATION.&operationCode,
   argument OPERATION.&ArgumentType
}
```

This is a very simple case that purposely omits a lot of additional information such as Information Object Set constraints that are typically a part of definitions such as this. The reason this information is not present is because we are just interested in showing the items that the compiler is concerned with. We will use this type to demonstrate the simple form of code generation. We will then add table constraints and discuss what changes when the –tables command line options is used.

The opcode field within this definition is an example of a **fixed type** field reference. It is known as this because if you go back to the original class specification, you will see that *operationCode* is defined to be of a specific type (namely a choice between a local and global value). The generated typedef for this field will contain a reference to the type from the class definition.

The argument field is an example of a *variable type* field.. In this case, if you refer back to the class definition, you will see that no type is provided. This means that this field can contain an instance of any encoded type (note: in practice, table constraints can be used with Information Object Sets to limit the message types that can be placed in this field). The generated typedef for this field contains an "open type" (*ASN1OpenType*) reference to hold a previously encoded component to be specified in the final message.

Simple Form Code Generation

In the simple form of information object code generation, the *Invoke* type above would result in the following C or C++ typedefs being generated:

```
typedef struct Invoke ::= SEQUENCE {
   OSINT32 invokeID;
   OPERATION_operationCode opcode;
   ASN1OpenType argument;
}
```

The following would be the procedure to add the Invoke header type to an ASN.1 message body:

1. Encode the body type
2. Get the message pointer and length of the encoded body
3. Plug the pointer and length into the *numocts* and *data* items of the argument open type field in the Invoke type variable.
4. Populate the remaining Invoke type fields.
5. Encode the Invoke type to produce the final message.

In this case, the amount of code generated to support the information object references is minimal. The amount of coding required by a user to encode or decode the variable type field elements, however, can be rather large. This is a tradeoff that exists between using the compiler generated table constraints solution (as we will see below) and using the simple form.

Unions Table Form Code Generation

If we now add table constraints to our original type definition, it might look as follows:

```
Invoke ::= SEQUENCE {
   invokeID INTEGER,
   opcode OPERATION.&operationCode ({My-ops}),
   argument OPERATION.&ArgumentType ({My-ops}{@opcode})
}
```

The "{My-ops}" constraint on the opcode element specifies an information object set that constrains the element value to one of the values in the object set. The {My-ops}{@opcode} constraint on the argument element goes a step further – it ties the type of the field to the type specified in the row that matches the given opcode value.

An example of the information object set and corresponding information objects would be as follows:

```
My-ops OPERATION ::= { makeCall | fwdCall, ... }

makeCall OPERATION ::= {
   &ArgumentType MakeCallArgument,
   &operationCode local : 10
}

fwdCall OPERATION ::= {
   &ArgumentType FwdCallArgument,
   &operationCode local : 11
}
```

The C or C++ type generated for the SEQUENCE above when *–table-unions* is specified would be as follows:

```
typedef struct EXTERN Invoke {
   OSINT32 invokeID;
   _OPERATION_operationCode opcode;
   struct {
      /**
       * information object selector
       */
      My_ops_TVALUE t;

      /**
       * My_ops information objects
       */
      union {
         /**
          * operationCode: local : 10
          */
         MakeCallArgument *makeCall;
         /**
          * operationCode: local : 11
          */
         FwdCallArgument *fwdCall;

         ASN1OpenType* extElem1;
      } u;
   } argument;
} Invoke;
```

Each of the options from the information object set are enumerated in the union structure. All a user needs to do to encode a variable of this type is to set the "t" value in the structure to the selected information object field and then populate the type field. This is very similar to populating a CHOICE construct. The comments in the elements show what the value of the key element(s) must be if that alternative is selected. The open type field at the end (extElem1) is added because the object set is extensible and it therefore may contain a value that is currently not included in the set.

Legacy Table Form Code Generation

In the legacy form of table constraint code generation, the following structure would be generated for the Invoke type above:

```
typedef struct EXTERN Invoke {
   OSINT32 invokeID;
   _OPERATION_operationCode opcode;
   ASN1Object argument;
}
```

This is almost identical to the type generated in the simple case. The difference is the *ASN1Object* type (or *ASN1TObject* for C++) that is used instead of *ASN1OpenType*. This type is defined in the *asn1type.h* run-time header file as follows:

```
   typedef struct ASN1Object {
      ASN1OpenType encoded;
      void* decoded;
      OSINT32 index;
   }
```

This holds the value to be encoded or decoded in both encoded or decoded form. The way a user uses this to encode a value of this type is as follows:

1. Populate a variable of the type to be used as the argument to the invoke type.

2. Plug the address of this variable into the *decoded* void pointer in the structure above.

3. Populate the remaining Invoke type fields.

4. Encode the Invoke type to produce the final message.

Note that in this case, the intermediate type does not need to be manually encoded by the user. The generated encoder has logic built-in to encode the complete message using the information in the generated tables.

Additional Code Generated with the -tables option

When the *–tables* command line option is used, additional code is generated to support the additional processing required to verify table constraints. This code varies depending on whether C or C++ code generation is selected. The C++ code is designed to take advantage of the object-oriented capabilities of C++. These capabilities are well suited for modeling the behavior of information objects in practice. The following subsections describe the code generated for each of these languages.

The code generated to support these constraints is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

C Code Generation

For C, code is generated for the Information Object Sets defined within a specification in the form of a global array of structures. Each structure in the array is an equivalent C structure representing the corresponding ASN.1 information object.

Additional encode and decode functions are also generated for each type that contains table constraints. These functions have the following prototypes :

BER/DER

```
int asn1ETC_<ProdName> (OSCTXT* pctxt, <ProdName>* pvalue);
```

```
int asn1DTC_<ProdName> (OSCTXT* pctxt, <ProdName>* pvalue);
```

PER

```
int asn1PETC_<ProdName> (OSCTXT* pctxt, <ProdName>* pvalue);
```

```
int asn1PDTC_<ProdName> (OSCTXT* pctxt, <ProdName>* pvalue);
```

The purpose of these functions is to verify the fixed values within the table constraints are what they should be and to encode or decode the open type fields using the encoder or decoder assigned to the given table row. Calls to these functions are automatically built into the standard encode or decode functions for the given type. They should be considered hidden functions not for use within an application that uses the API.

C++ Code Generation

For C++, code is generated for ASN.1 classes, information objects, and information object sets. This code is then referenced when table constraint processing must be performed.

Each of the generated C++ classes builds on each other. First, the classes generated that correspond to ASN.1 CLASS definitions form the base class foundation. Then C++ classes derived from these base classes corresponding to the information objects are generated. Finally, C++ singleton classes corresponding to the information object sets are generated. Each of these classes provides a container for a collection of C++ objects that make up the object set.

Additional encode and decode functions are also generated as they were in the C code generation case for interfacing with the object definitions above. These functions have the following prototypes:

BER/DER

```
int asn1ETC_<ProdName> (OSCTXT* pctxt,
                        <ProdName>* pvalue,
                        <ClassName>* pobject);

int asn1DTC_<ProdName> (OSCTXT* pctxt,
                        <ProdName>* pvalue,
                        <ClassName>* pobject);
```

PER

```
int asn1PETC_<ProdName> (OSCTXT* pctxt,
                         <ProdName>* pvalue,
                         <ClassName>* pobject);

int asn1PDTC_<ProdName> (OSCTXT* pctxt,
                         <ProdName>* pvalue,
                         <ClassName>* pobject);
```

These prototypes are identical to the prototypes generated in C code generation case except for the addition of the *pobject* argument. This argument is for a pointer to the information object that matches the key field value for a given encoding. These functions have different logic for processing Relative and Simple table constraints. The logic associated with each case is as follows:

On the encode side:

Relative Table Constraint:

1. The *lookupObject* method is invoked on the object set instance to find the class object for the data in the populated type variable to be encoded.

2. If a match is found, the table constraint encode function as defined above is invoked. This function will verify all fixed type values match what is defined in the information object definition and will encode all type fields and store the resulting encoded data in the *ASN1TObject.encoded* fields.

3. If a match is not found and the information object set is not extensible, then a table constraint error status will be returned. If the information object set is extensible, a normal status is returned.

Simple Table Constraint:

1. This function will verify all the fixed type values match what is defined in the table constraint information object set. If an element value does not exist in the table (i.e. the information object set) and the object set is not extensible, then a table constraint violation exception will be thrown.

The normal encode logic is then performed to encode all of the standard and open type fields in the message.

On the decode side, the logic is reversed:

The normal decode logic is performed to populate the standard and open type fields in the generated structure.

Relative Table Constraint:

1. The *lookupObject* method is invoked on the decoded key field value to find an object match.

2. If a match is found, the table constraint decode function as defined above is invoked. This function will verify all fixed type values match what is defined in the information object definition and will fully decode all type fields and store pointers to the decoded type variables in the *ASN1TObject.decoded* fields.

3. If a match is not found and the information object set is not extensible, then a table constraint error status will be returned. If the information object set is extensible, a normal status is returned.

Simple Table Constraint:

1. This function will verify all the fixed type values match what is defined in the table constraint object set. If an element value does not exist in the table (i.e. the information object set) and the object set is not extensible, then a table constraint violation exception will be thrown.

General Procedure for Table Constraint Encoding

The general procedure to encode an ASN.1 message with table constraints is the same as without table constraints. The only difference is in the open type data population procedure.

The *-table-unions* option will cause union structure to be generated for open type field containing relationsal table constraints. These are populated for encoding in much the same way CHOICE onstructs are handled.

The *-tables* option will cause *ASN1TObject* fields to be inserted in the generated code instead of *Asn1OpenType* declarations.

The procedure to populate the value for an *ASN1TObject* item is as follows:

1. Check the ASN.1 specification or generated C code for the type of the type field value in the information object set that corresponds to the selected key field value.

2. Create a variable of that type and assign a pointer to it to the *Asn1Object.decoded* member variable as void*.

3. Follow the common BER/PER/DER encode procedure.

A complete example showing how to assign an open type value in the legacy tables case is as follows:

```
Test DEFINITIONS ::= BEGIN
```

```
    ATTRIBUTE ::= CLASS {
       &Type,
       &id                OBJECT IDENTIFIER UNIQUE }
    WITH SYNTAX {
       WITH SYNTAX &Type ID &id }

    name ATTRIBUTE ::= {
       WITH SYNTAX     VisibleString
       ID              { 0 1 1 } }

    name ATTRIBUTE ::= {
       WITH SYNTAX     INTEGER
       ID              { 0 1 2 } }

    SupportedAttributes ATTRIBUTE ::= { name | commonName }

    Invoke ::= SEQUENCE {
       opcode ATTRIBUTE.&id     ({SupportedAttributes}),
       argument ATTRIBUTE.&Type ({SupportedAttributes}{@opcode})
    }

 END
```

In the above example, the *Invoke* type contains a table constraint. Its element *opcode* refers to the ATTRIBUTE *id* field and *argument* element refers to the ATTRIBUTE *Type* field. The *opcode* element is an index element for the *Invoke* type's table constraint. The *argument* element is an open type whose type is determined by the *opcode* value. In this example, *opcode* is the key field.

The *opcode* element can have only two possible values: { 0 1 1 } or { 0 1 2 }. If the *opcode* value is { 0 1 1} then *argument* will have a *VisibleString* value and if the *opcode* value is { 0 1 2 } then *argument* will have an INTEGER value. Any other value of the *opcode* element will be violation of the Table Constraint.

If the *SupportedAttributes* information object set was extensible (indicated by a ",..." at the end of the definition), then *the argument* element may have a value of a type that is not in the defined set. In this case, if the index element value is outside the information object set, then the *argument* element will be assumed to be an *Asn1OpenType*. The *Invoke* type encode function call will use the value from *argument.encoded.data* field (i.e. it will have to be pre-encoded because the encode function will not be able to determine from the table constraint how to encode it).

A **C++ program** fragment that could be used to encode an instance of the *Invoke* type is as follows:

```
#include TestTable.h // include file generated by ASN1C

main ()
{
   const OSOCTET* msgptr;
   OSOCTET msgbuf[1024];
   int msglen;

   // step 1: construct ASN1C C++ generated class.
   // this specifies a static encode message buffer

   ASN1BEREncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));
```

```
      // step 2: populate msgData structure with data to be encoded

      ASN1T_Invoke msgData;
      ASN1C_Invoke invoke (encodeBuffer, msgData);

      msgData.opcode.numids = 3;
      msgData.opcode.subid[0] = 0;
      msgData.opcode.subid[1] = 1;
      msgData.opcode.subid[2] = 1;
      ASN1VisibleString argument = “objsys”;
      msgData.argument.decoded = (void*) &argument;
      // note: opcode value is {0 1 1 }, so argument must be
      // ASN1VisibleString type

      // step 3: invoke Encode method

      if ((msglen = invoke.Encode ()) > 0) {
         // encoding successful, get pointer to start of message
         msgptr = encodeBuffer.getMsgPtr();
      }
      else
         error processing...
   }
```

The encoding procedure for C requires one extra step. This is a call to the module initialization functions after context initialization is complete. All module initialization functions for all modules in the project must be invoked. The module initialization function definitions can be found in the *<ModuleName>Table.h* file.

The format of each module initialization function name is as follows:

```
   void <ModuleName>_init (OSCTXT* pctxt)
```

Here ModuleName would be replaced with name of the module.

A **C program** fragment that could be used to encode the *Invoke* record defined above is as follows:

```
   #include TestTable.h            /* include file generated by ASN1C */

   int main ()
   {
      OSOCTET msgbuf[1024], *msgptr;
      int     msglen;
      OSCTXT  ctxt;
      Invoke  invoke; /* typedef generated by ASN1C */

      /* Step 1: Initialize the context and set the buffer pointer */

      if (rtInitContext (&ctxt) != 0) {
         /* initialization failed, could be a license problem */
         printf (“context initialization failed (check license)\n”);
         return –1;
      }

      xe_setp (&ctxt, msgbuf, sizeof(msgbuf));
      /* step 2: call module initialization functions */

      Test_init (&ctxt);

      /* Step 3: Populate the structure to be encoded */

      msgData.opcode.numids = 3;
      msgData.opcode.subid[0] = 0;
      msgData.opcode.subid[1] = 1;
      msgData.opcode.subid[2] = 1;
      //note: opcode value is {0 1 1 }, so argument must be
      //ASN1VisibleString type
      ASN1VisibleString argument = “objsys”;
      msgData.argument.decoded = (void*) &argument;
      ...

      /* Step 4: Call the generated encode function */

      msglen = asn1E_Invoke (&ctxt, &invoke, ASN1EXPL);

      /* Step 5: Check the return status (note: the test is */
      /* > 0 because the returned value is the length of the */
      /* encoded message component)..*/

      if (msglen > 0) {

         /* Step 6: If encoding is successful, call xe_getp to */
         /* fetch a pointer to the start of the encoded message.*/
         msgptr = xe_getp (&ctxt);
         ...
      }
      else
         error processing...
   }
```

General Procedure for Table Constraint Decoding

The general procedure to decode an ASN.1 message with table constraints is the same as without table constraints. The only difference will exist in the decoded data for open type fields within the message. In this case, the *Asn1Object / Asn1TObject’s decoded* member variable will contain the original decoded type and *the encoded* member variable will contain the original data in encoded form.

Refer to the BER/DER/PER decoding procedure for further information.

The procedure to retrieve the value for open type fields is as follow:

1. Check the possible Type in the Information Object Set from index element value.

2. Assign or cast the *Asn1Object.decoded* member variable (void*) to the result type.

3. The *Asn1Object.encoded* field will hold the data in encoded form.

For the above complete example, the *Invoke* type's *argument* element will be decoded as one of the types in the *SupportedAttributes* information object set (i.e. either as a *VisibleString* or *INTEGER* type). If the *SupportedAttributes* information object set is extensible, then the *argument* element may be of a type not defined in the set. In this case, the decoder will set the *Asn1Object.encoded* field as before but the Asn1Object.decoded field will be NULL indicating the value is of an unknown type.

A **C++ program** fragment that could be used to decode the *Invoke* example is as follows:

```
#include Test.h //           include file generated by ASN1C

main ()
{
   OSOCTET msgbuf[1024];
   ASN1TAG msgtag;
   int msglen, status;

   /* step 1: logic to read message into msgbuf */
   ...

   /* step 2: create decode buffer and msg data type */

   ASN1BERDecodeBuffer decodeBuffer (msgbuf, len);

   ASN1T_Invoke msgData;
   ASN1C_Invoke invoke (decodeBuffer, msgData);

   /* step 3: call decode function */

   if ((status = invoke.Decode ()) == 0)
   {
      // decoding successful, data in msgData
      // use key field value to set type of message data
      ASN1OBJID oid1[] = { 3, { 0, 1, 1 }};
      ASN1OBJID oid2[] = { 3, { 0, 1, 2 }};
      if (msgData.opcode == oid1) {
         // argument is a VisibleString
         ASN1VisibleString* pArg =
            (ASN1VisibleString*) msgData.argument.decoded;
         ...
      }
      else if (msgData.opcode == oid2) {
         // argument is an INTEGER
         OSINT32 arg = (OSINT32) *msgData.argument.decoded;
         ...
      }
    }
   else {
        // error processing
   }
```

In this case, the type of the decoded argument can be determined by testing the key field value. In the example as shown, the *SupportedAttributes* information object set is not extensible, therefore, the type of the argument must be one of the two shown. If the set were extensible (indicated by a ",..." in the definition), then it is possible that an unknown *opcode* could be received which would mean the type can not be determined. In this case, the original encoded message data would be present in *msgData.argument.encoded* field and it would be up to the user to determine how to process it.

The decoding procedure for C requires one additional step. This is a call to the module initialization functions after context initialization is complete. All module initialization functions for all modules in the project must be invoked. The module initialization function definitions can be found in the *<ModuleName>Table.h* file.

A **C program** fragment that could be used to decode the *Invoke* example is as follows:

```
#include TestTable.h             // include file generated by ASN1C

main ()
{
   OSOCTET msgbuf[1024];
   ASN1TAG   msgtag;
   int       msglen;
   OSCTXT    ctxt;
   Invoke    invoke;
   ASN1OBJID oid1[] = { 3, { 0, 1, 1 }};
   ASN1OBJID oid2[] = { 3, { 0, 1, 2 }};

   .. logic to read message into msgbuf ..

   /* Step 1: Initialize a context variable for decoding */

   if (rtInitContext (&ctxt) != 0) {
      /* initialization failed, could be a license problem */
      printf (“context initialization failed (check license)\n”);
      return –1;
   }

   xd_setp (&ctxt, msgbuf, 0, &msgtag, &msglen);

   /* step 2: call module initialization functions */

   Test_init (&ctxt);

   /* Step 3: Call decode function */

   status = asn1D_Invoke (&ctxt, &invoke, ASN1EXPL, 0);

   /* Step 4: Check return status */

   if (status == 0)
   {
      /* process received data in ‘invoke’ variable */
      if (rtCmpTCOID (&invoke.opcode, &oid1) == 0) {
         /* argument is a VisibleString */
         ASN1VisibleString* pArg =
            (ASN1VisibleString*) msgData.argument.decoded;
         ...
      }
      else if (rtCmpTCOID (&invoke.opcode, &oid2) == 0) {
         /* argument is an INTEGER */
         OSINT32 arg = (OSINT32) *msgData.argument.decoded;
         ...
      }
      /* Remember to release dynamic memory when done! */
      ASN1MEMFREE (&ctxt);
   }
   else
      error processing...
   }
   }
```

General Procedures for Encoding and Decoding

Encoding functions and methods generated by the ASN1C compiler are designed to be similar in use across the different encoding rule types. In other words, if you have written an application to use the Basic Encoding Rules (BER) and then later decide to use the Packed Encoding Rules (PER), it should only be a simple matter of changing a few function calls to accomplish the change. Procedures for such things as populating data for encoding, accessing decoded data, and dynamic memory management are the same for all of the different encoding rules.

This section describes common procedures for encoding or decoding data that are applicable to any of the different encoding rules. Subsequent sections will then describe what will change for the different rules.

Dynamic Memory Management

The ASN1C run-time uses specialized dynamic memory functions to improve the performance of the encoder/decoder. It is imperative to understand how these functions work in order to avoid memory problems in compiled applications. ASN1C also provides the capability to plug-in a different memory management scheme at two different levels: the high level API called by the generated code and the low level API that provides the core memory managment functionality.

The ASN1C Default Memory Manager

The default ASN1C run-time memory manager uses an algorithm called the nibble-allocation algorithm. 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. These functions are very expensive in terms of performance.

The large blocks of memory are tracked through the ASN.1 context block (OSCTXT) structure. For C, this means that an initialized context block is required for all memory allocations and deallocations. All allocations are done using this block as an argument to routines such as *rtxMemAlloc*. All memory can be released at once when a user is done with a structure containing dynamic memory items by calling *rtxMemFree*. Other functions are available for doing other dynamic memory operations as well. See the *C/C++ Run-time Reference Manual* for details on these.

High Level Memory Management API

The high-level memory management API consists of C macros and functions called in gemerated code and/or in application programs to allocate and free memory within the ASN1C run-time.

At the top level are a set of macro definitions that begin with the prefix *rtxMem*. These are mapped to a set of similar functions that begin with the prefix *rtxMemHeap*. A table showing this basic mapping is as follows:

Macro	**Function**	**Description**
`rtxMemAlloc`	`rtxMemHeapAlloc`	Allocate memory
`rtxMemAllocZ`	`rtxMemHeapAllocZ`	Allocate and zero memory
`rtxMemRealloc`	`rtxMemHeapRealloc`	Reallocate memory
`rtxMemFree`	`rtxMemHeapFreeAll`	Free all memory in context
`rtxMemFreePtr`	`rtxMemHeapFreePtr`	Free a specific memory block

See the *ASN1C C/C++ Common Runtime Reference Manual* for further details on these functions and macros.

It is possible to replace the high-level memory allocation functions with functions that implement a custom memory management scheme. This is done by implementing some (or all) of the C *rtxMemHeap* functions defined in the following interface (note: a default implementation is shown that replaces the ASN1C memory manager with direct calls to the standard C run-time memory management functions):

```
#include <stdlib.h>
#include "rtxMemory.h"

/* Create a memory heap */
int rtxMemHeapCreate (void** ppvMemHeap) {
   return 0;
}

/* Allocate memory */
void* rtxMemHeapAlloc (void** ppvMemHeap, int nbytes) {
   return malloc (nbytes);
}

/* Allocate and zero memory */
void* rtxMemHeapAllocZ (void** ppvMemHeap, int nbytes) {
   void* ptr = malloc (nbytes);
   if (0 != ptr) memset (ptr, 0, nbytes);
   return ptr;
}

/* Free memory pointer */
void rtxMemHeapFreePtr (void** ppvMemHeap, void* mem_p) {
   free (mem_p);
}

/* Reallocate memory */
void* rtxMemHeapRealloc (void** ppvMemHeap, void* mem_p, int nbytes_) {
   return realloc (mem_p, nbytes_);
}

/* Clears heap memory (frees all memory, reset all heap's variables) */
void rtxMemHeapFreeAll (void** ppvMemHeap) {
   /* should remove all allocated memory. there is no analog in standard memory
      management. */
}

/* Frees all memory and heap structure as well (if was allocated) */
void rtxMemHeapRelease (void** ppvMemHeap) {
   /* should free all memory allocated + free memory heap object if exists */
}
```

In most cases it is only necessary to implement the following functions: *rtxMemHeapAlloc*, *rtxMemHeapAllocZ*, *rtxMemHeapFreePtr* and *rtxMemHeapRealloc*. Note that there is no analog in standard memory management for ASN1C's *rtxMemFree* macro (i.e. the *rtxMemHeapFreeAll* function). A user would be responsible for freeing all items in a generated ASN1C structure individually if standard memory management is used.

The *rtxMemHeapCreate* and *rtxMemHeapRelease* functions are specialized functions used when a special heap is to be used for allocation (for example, a static block within an embedded system). In this case, *rtxMemHeapCreate* must set the *ppvMemHeap* argument to point at the block of memory to be used. This will then be passed in to all of the other memory management functions for their use through the OSCTXT structure. The *rtxMemHeapRelease* function can then be used to dispose of this memory when it is no longer needed.

To add these definitions to an application program, compile the C source file (it can have any name) and link the resulting object file (.OBJ or .O) in with the application.

Built-in Compact Memory Management

A built-in version of the simple memory management API described above (i.e with direct calls to malloc, free, etc.) is available for users who have the source code version of the run-time. The only difference in this API with what is described above is that tracking of allocated memory is done through the context. This makes it possible to provide an implementation of the *rtxMemHeapFreeAll* function as described above. This memory management scheme is slower than the default manager (i.e. nibble-based), but has a smaller code footprint.

This form of memory management is enabled by defining the _MEMCOMPACT C compile time setting. This can be done by either adding -D_MEMCOMPACT to the C compiler command-line arguments, or by uncommenting this item at the beginning of the *rtxMemory.h* header file:

```
/*
 * Uncomment this definition before building the C or C++ run-time
 * libraries to enable compact memory management.  This will have a
 * smaller code footprint than the standard memory management; however,
 * the performance may not be as good.
 */
/*#define _MEMCOMPACT*/
```

Low Level Memory Management API

It is possible to replace the core memory management functions used by the ASN1C run-time memory manager. This has the advantage of preserving the existing management scheme but with the use of different core functions. Using different core functions may be necessary on some systems that do not have the standard C run-time functions *malloc*, *free*, and *realloc*, or when extra functionality is desired.

To replace the core functions, the following run-time library function would be used:

```
void rtxMemSetAllocFuncs (OSMallocFunc malloc_func,
   OSReallocFunc realloc_func, OSFreeFunc free_func);
```

The *malloc, realloc,* and *free* functions must have the same prototype as the standard C functions. Some systems do not have a realloc-like function. In this case, *realloc_func* may be set to NULL. This will cause the *malloc_func/free_func* pair to be used to do reallocations.

This function must be called before the context initialization function (*rtInitContext*) because context initialization requires low level memory management facilities be in place in order to do its work.

Note that this function makes use of static global memory to hold the function definitions. This type of memory is not available in all run-time environments (most notably Symbian). In this case, an alternative function is provided for setting the memory functions. This function is *rtxInitContextExt* which must be called in place of the standard context initialization function (*rtInitContext*). In this case, there is a bit more work required to initialize a context because the ASN.1 subcontext must be manually initialized. This is an example of the code required to do this:

```
int stat = rtxInitContextExt (pctxt, malloc_func, realloc_func, free_func);
if (0 == stat) {
   /* Add ASN.1 error codes to global table */
   rtErrASN1Init ();

   /* Init ASN.1 info block */
   stat = rtCtxtInitASN1Info (pctxt);
}
```

Memory management can also be tuned by setting the default memory heap block size. The way memory management works is that a large block of memory is allocated up front on the first memory management call. This block is then subdivided on subsequent calls until the memory is used up. A new block is then started. The default value is 4K (4096) bytes. The value can be set lower for space constrained systems and higher to improve performance in systems that have sufficient memory resources. To set the block size, the following run-time function should be used:

```
void rtxMemSetDefBlkSize (OSUINT32 blkSize);
```

This function must be called prior to context initialization.

C++ Memory Management

In the case of C++, the ownership of memory is handled by the control class and message buffer objects. These classes share a context structure and use reference counting to manage the allocation and release of the context block. When a message buffer object is created, a context block structure is created as well. When this object is then passed into a control class constructor, its reference count is incremented. Then when either the control class object or message buffer object are deleted or go out of scope, the count is decremented. When the count goes to zero (i.e. when both the message buffer object and control class object go away) the context structure is released.

What this means to the user is that a control class or message buffer object must be kept in scope when using a data structure associated with that class. A common mistake is to try and pass a data variable out of a method and use it after the control and message buffer objects go out of scope. For example, consider the following code fragment:

```
ASN1T_<type>* func2 () {
   ASN1T_<type>* p = new ASN1T_<type> ();
   ASN1BERDecodeBuffer decbuf;
   ASN1C_<type> cc (decbuf, *p);

   cc.Decode();

   // After return, cc and decbuf go out of scope; therefore
   // all memory allocated within struct p is released..

   return p;
   }

void func1 () {
   ASN1T_<type>* pType = func2 ();

   // pType is not usable at this point because dynamic memory
   // has been released..
}
```

As can be seen from this example, once func2 exits, all memory that was allocated by the decode function will be released. Therefore, any items that require dynamic memory within the data variable will be in an undefined state.

An exception to this rule occurs when the type of the message being decoded is a *Protocol Data Unit* (PDU). These are the main message types in a specification. The ASN1C compiler designates types that are not referenced by any other types as PDU types. This behavior can be overridden by using the -pdu command line argument or <isPDU> configuration file element.

The significance of PDU types is that generated classes for these types are derived from the *ASN1TPDU* base class. This class holds a reference to a context object. The context object is set by Decode and copy methods. Thus, even if control class and message buffer objects go out of scope, the memory will not be freed until the destructor of an *ASN1TPDU* inherited class is called. The example above will work correctly without any modifications in this case.

Another way to keep data is to make a copy of the decoded object before it goes out of scope. A method called *newCopy* is also generated in the control class for these types which can be used to create a copy of the decoded object. This copy of the object will persist after the control class and message buffer objects are deleted. The returned object can be deleted using the standard C++ *delete* operator when it is no longer needed.

Returning to the example above, it can be made to work if the type being decoded is a PDU type by doing the following:

```
ASN1T_<type>* func2 () {
   ASN1T_<type> msgdata;
   ASN1BERDecodeBuffer decbuf;
   ASN1C_<type> cc (decbuf, msgdata);

   cc.Decode();

   // Use newCopy to return a copy of the decoded item..

   return cc.newCopy();
}
```

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 three 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 standard *malloc* and *free* C functions or *new* and *delete* C++ operators to allocate memory to hold the data, and

3. Use the *rxtMemAlloc* and *rtxMemFree* 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 the second technique (i.e., using C *malloc* and *free*) can solve this problem. In this case, the memory for each of the elements can be safely freed after the encode function is called. But the downside is that a free call must be made for each corresponding *malloc* call. For complex structures, remembering to do this can be difficult thus leading to problems with memory leaks.

The third technique uses the compiler run-time library memory management functions to allocate and free the memory. The main advantage of this technique as opposed to using C *malloc* and *free* is that all allocated memory can be freed with a single *rtxMemFree* call. The *rtxMemAlloc* macro can be used to allocate memory in much the same way as the C malloc function with the only difference being that a pointer to an initialized OSCTXT structure is passed in addition to the number of bytes to allocate. All allocated memory is tracked within the context structure so that when the *rtxMemFree* function is called, all memory is released at once.

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 the 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. ASN.1 encoding involves the possible additions of tags and lengths 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. Even in the static buffer case, the message start address may be different then the buffer start address (see the section on encoding BER messages). For this reason, each set of encoding rules has a run-time C function for getting the message start address and length. See the *C/C++ Run-Time Library Reference Manual* for a description of these functions. The C++ message buffer classes contain the *getMsgPtr*, *getMsgCopy* , and *getMsgLength* methods for this purpose.

A **stream** message buffer can be used for BER encoding. This type of buffer is used when the -stream option was used to generate the encode functions. See the section on BER stream encoding for a complete description on how to set up an output stream to receive encoded data.

Generated BER Functions

Generated BER Encode Functions

Note

This section assumes standard memory-buffer based encoding is to be done. If stream-based encoding is to be done (specified by adding *-stream* to the ASN1C command-line), see the *Generated BER Streaming Encode Functions* section for correct procedures on using the stream-based encode functions.

For each ASN.1 production defined in the ASN.1 source file, a C encode function is generated. This function will convert a populated C variable of the given type into an encoded ASN.1 message.

If C++ code generation is specified, a control class is generated that contains an *Encode* method that wraps this function. This function is invoked through the class interface to convert a populated *msgData* attribute variable into an encoded ASN.1 message.

Generated C Function Format and Calling Parameters

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

```
asn1E_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
len = asn1E_<name> (OSCTXT* pctxt,
                    <name>* pvalue,
                    ASN1TagType tagging);
```

In this definition, `<name>` denotes the prefixed production 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 program. The variable should be initialized using the *rtInitContext* run-time library function (see the *C/C++ Common Run-Time Library Reference Manual* for a complete description of this function).

The `pvalue` argument holds a pointer to the data to be encoded and is of the type generated from the ASN.1 production.

The `tagging` argument is for internal use when calls to encode functions are nested to accomplish encoding of complex variables. It indicates whether the tag associated with the production should be applied or not (implicit versus explicit tagging). At the top level, the tag should always be applied so this parameter should always be set to the constant ASN1EXPL (for EXPLICIT).

The function result variable `len` returns the length of the data actually encoded or an error status code if encoding fails. Error status codes are negative to tell them apart from length values. Return status values are defined in the `asn1type.h` include file.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling a C BER or DER encode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done. Note that the procedures described here cannot be used if stream-based encoding is to be done (specified by the use of the *-stream* ASN1C command-line option). In this case, the procedures described in the *Generated BER Streaming Encode Functions* section.

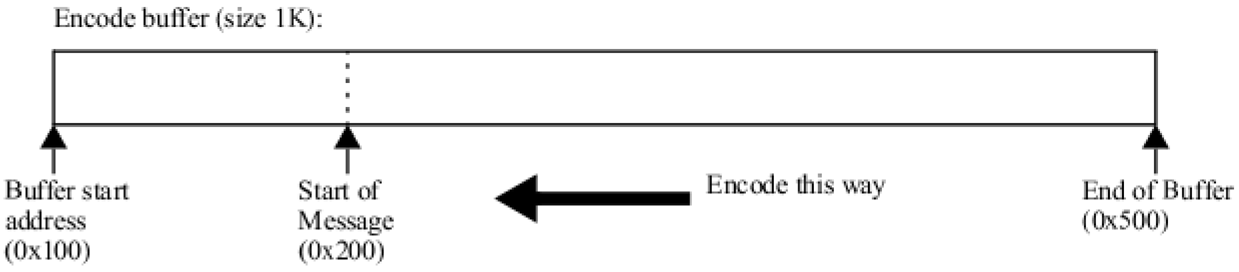
Before any encode function can be called; the user must first initialize an encoding context. This is a variable of type OSCTXT. This variable holds all of the working data used during the encoding of a message. The context variable is declared as a normal automatic variable within the top-level calling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

```
OSCTXT ctxt;

if (rtInitContext (&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 into which the message will be encoded. This is accomplished by calling the *xe_setp* run-time function. The user can 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 (positive length value), the run-time library function `xe_getp` can be called to obtain the start address of the encoded message. Note that the returned address is not the start address of the target buffer. BER encoded messages are constructed from back to front (i.e., starting at the end of the buffer and working backwards) so the start point will fall somewhere in the middle of the buffer after encoding is complete. This is illustrated in the following diagram:



In this example, a 1K encode buffer is declared which happens to start at address 0x100. When the context is initialized with a pointer to this buffer and size equal to 1K, it positions the internal encode pointer to the end of the buffer (address 0x500). Encoding then proceeds from back-to-front until encoding of the message is complete. In this case, the encoded message turned out to be 0x300 (768) bytes in length and the start address fell at 0x200. This is the value that would be returned by the `xe_getp` function.

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

```
#include employee.h             /* include file generated by ASN1C */

int main ()
{
    OSOCTET   msgbuf[1024], *msgptr;
    int       msglen;
    OSCTXT    ctxt;
    Employee  employee; /* typedef generated by ASN1C */

    /* Step 1: Initialize the context and set the buffer pointer */

    if (rtInitContext (&ctxt) != 0) {
       /* initialization failed, could be a license problem */
       printf (“context initialization failed (check license)\n”);
       return –1;
    }

    xe_setp (&ctxt, msgbuf, sizeof(msgbuf));

    /* Step 2: Populate the structure to be encoded */

    employee.name.givenName = "SMITH";
    ...

    /* Step 3: Call the generated encode function */

    msglen = asn1E_Employee (&ctxt, &employee, ASN1EXPL);

    /* Step 4: Check the return status (note: the test is
     * > 0 because the returned value is the length of the
     * encoded message component).. */

    if (msglen > 0) {
       /* Step 5: If encoding is successful, call xe_getp to
        * fetch a pointer to the start of the encoded message. */
       msgptr = xe_getp (&ctxt);
       ...
    }
    else {
       rtxErrPrint (&ctxt);
       return msglen;
    }
}
```

In general, static buffers should be used for encoding messages where possible as they offer a substantial performance benefit over dynamic buffer allocation. The problem with static buffers, however, is that you are required to estimate in advance the approximate size of the messages you will be encoding. There is no built-in formula to do this; the size of an ASN.1 message can vary widely based on data types and the number of tags required.

If performance is not a significant issue, then dynamic buffer allocation is a good alternative. Setting the buffer pointer argument to NULL in the call to *xe_setp* specifies dynamic allocation. This tells the encoding functions to allocate a buffer dynamically. The address of the start of the message is obtained as before by calling *xe_getp*. Note that this is not the start of the allocated memory; that is maintained within the context structure. To free the memory, either the *rtxMemFree* function may be used to free all memory held by the context or the *xe_free* function used to free the encode buffer only.

The following code fragment illustrates encoding using a dynamic buffer:

```
   #include employee.h            /* include file generated by ASN1C */

   main ()
   {
      OSOCTET*  msgptr;
      int       msglen;
      OSCTXT    ctxt;
      Employee  employee;      /* typedef generated by ASN1C */

      if (rtInitContext (&ctxt) != 0) {
         /* initialization failed, could be a license problem */
         printf (“context initialization failed (check license)\n”);
         return –1;
      }

      xe_setp (&ctxt, NULL, 0);

      employee.name.givenName = "SMITH";
      ...

      msglen = asn1E_Employee (&ctxt, &employee, ASN1EXPL);

      if (msglen > 0) {
         msgptr = xe_getp (&ctxt);
         ...

         rtxMemFree (&ctxt); /* don’t call free (msgptr); !!! */
      }
      else
         error processing...
   }
```

Encoding a Series of Messages Using the C Encode Functions

A common application of BER encoding is the repetitive encoding of a series of the same type of message over and over again. For example, a TAP3 batch application might read billing data out of a database table and encode each of the records for a batch transmission.

If a user was to repeatedly allocate/free memory and reinitialize the C objects involved in the encoding of a message, performance would suffer. This is not necessary however, because the C objects and memory heap can be reused to allow multiple messages to be encoded. As example showing how to do this is as follows:

```
   #include employee.h           /* include file generated by ASN1C */

   main ()
   {
      const OSOCTET* msgptr;
      OSOCTET     msgbuf[1024];
      int         msglen;
      OSCTXT      ctxt;
      PersonnelRecord data;

      /* Init context structure */

      if ((stat = rtInitContext (&ctxt)) != 0) {
         printf ("rtInitContext failed; stat = %d\n", stat);
         return -1;
      }

      /* Encode loop starts here, this will repeatedly use the
       * objects declared above to encode the messages */

      for (;;) {

         xe_setp (&ctxt, msgbuf, sizeof(msgbuf));

         /* logic here to read record from some source (database,
          * flat file, socket, etc.).. */

         /* populate structure with data to be encoded */

         data.name = “SMITH”;
         ...

         /* call encode function */

         if ((msglen = asn1E_PersonnelRecord (&ctxt, &data, ASN1EXPL)) > 0) {

            /* encoding successful, get pointer to start of message */

            msgptr = xe_getp (&ctxt);
            /* do something with the encoded message */

            ...
         }
         else
            error processing...

         /* Call rtxMemReset to reset the memory heap for the next
          * iteration. Note, all data allocated by rtxMemAlloc will
          * become invalid after this call. */

         rtxMemReset (&ctxt);
      }

      rtFreeContext (&ctxt);
   }
```

The *rtxMemReset* call does not free memory; instead, it marks it as empty so that it may be reused in the next iteration. Thus, all memory allocated by *rtxMemAlloc* will be overwritten and data will be lost.

Generated C++ Encode Method Format and Calling Parameters

When C++ code generation is specified, the ASN1C compiler generates an *Encode* method in the generated control class that wraps the C function call. This method provides a more simplified calling interface because it hides things such as the context structure and the tag type parameters.

The calling sequence for the generated C++ class method is as follows:

```
   len = <object>.Encode ();
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production. The function result variable len returns the length of the data actually encoded or an error status code if encoding fails. Error status codes are negative to tell them apart from length values. Return status values are defined in the *asn1type.h* include file.

Procedure for Using the C++ Control Class Encode Method

The procedure to encode a message using the C++ class interface is as follows:

1. Create a variable of the ASN1T_<name> type and populate it with the data to be encoded.

2. Create an *ASN1BEREncodeBuffer* object.

3. Create a variable of the generated ASN1C_<name> class specifying the items created in 1 and 2 as arguments to the constructor.

4. Invoke the Encode method.

The constructor of the ASN1C_<type> class takes a message buffer object argument. This makes it possible to specify a static encode message buffer when the class variable is declared. A static buffer can improve encoding performance greatly as it relieves the internal software from having to repeatedly resize the buffer to hold the encoded message. If you know the general size of the messages you will be sending, or have a fixed size maximum message length, then a static buffer should be used. The message buffer argument can also be used to specify the start address and length of a received message to be decoded.

After the data to be encoded is set, the Encode method is called. This method returns the length of the encoded message or a negative value indicating that an error occurred. The error codes can be found in the asn1type.h run-time header file or in Appendix A of the *C/C++ Common Functions Reference Manual*.

If encoding is successful, a pointer to the encoded message can be obtained by using the *getMsgPtr* or *getMsgCopy* methods available in the *ASN1BEREncodeBuffer* class. The *getMsgPtr* method is faster as it simply returns a pointer to the actual start-of-message that is maintained within the message buffer object. The *getMsgCopy* method will return a copy of the message. Memory for this copy will be allocated using the standard new operator, so it is up to the user to free this memory using delete when finished with the copy.

A program fragment that could be used to encode an employee record is as follows. This example uses a static encode buffer:

```
#include employee.h             // include file generated by ASN1C

main ()
{
   const OSOCTET* msgptr;
   OSOCTET msgbuf[1024];
   int msglen;

   // step 1: construct ASN1C C++ generated class.
   // this specifies a static encode message buffer

   ASN1BEREncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));
   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // step 2: populate msgData structure with data to be encoded

   msgData.name = “SMITH”;
   ...

   // step 3: invoke Encode method

   if ((msglen = employee.Encode ()) > 0) {
      // encoding successful, get pointer to start of message
      msgptr = encodeBuffer.getMsgPtr();
   }
   else
      error processing...
}
```

The following code fragment illustrates encoding using a dynamic buffer. This also illustrates using the *getMsgCopy* method to fetch a copy of the encoded message:

```
#include employee.h             // include file generated by ASN1C

main ()
{
   OSOCTET*  msgptr;
   int       msglen;

   // construct encodeBuffer class with no arguments

   ASN1BEREncodeBuffer encodeBuffer;
   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // populate msgData structure

   msgData.name = "SMITH";
   ...

   // call Encode method

   if ((msglen = employee.Encode ()) > 0) {
      // encoding successful, get copy of message
      msgptr = encodeBuffer.getMsgCopy();
      ...

      delete [] msgptr; // free the dynamic memory!
   }
   else
      error processing...
}
```

Encoding a Series of Messages Using the C++ Control Class Interface

A common application of BER encoding is the repetitive encoding of a series of the same type of message over and over again. For example, a TAP3 batch application might read billing data out of a database table and encode each of the records for a batch transmission.

If a user was to repeatedly instantiate and destroy the C++ objects involved in the encoding of a message, performance would suffer. This is not necessary however, because the C++ objects can be reused to allow multiple messages to be encoded. As example showing how to do this is as follows:

```
#include employee.h           // include file generated by ASN1C

main ()
{
   const OSOCTET* msgptr;
   OSOCTET msgbuf[1024];
   int       msglen;

   ASN1BEREncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));
```

```
      ASN1T_PersonnelRecord msgData;
      ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

      // Encode loop starts here, this will repeatedly use the
      // objects declared above to encode the messages

      for (;;) {

      // logic here to read record from some source (database,
      // flat file, socket, etc.)..

      // populate structure with data to sbe encoded

      msgData.name = "SMITH";
      ...

      // invoke Encode method

      if ((msglen = employee.Encode ()) > 0) {

         // encoding successful, get pointer to start of message

         smsgptr = encodeBuffer.getMsgPtr();

         // do something with the encoded message

         ...
      }
      else
         error processing...

      // Call the init method on the encodeBuffer object to
      // prepare the buffer for encoding another message..

      encodeBuffer.init();
   }
}
```

Generated BER Streaming Encode Functions

BER messages can be encoded directly to an output stream such as a file, network or memory stream. The ASN1C compiler has the -*stream* option to generate encode functions of this type. For each ASN.1 production defined in the ASN.1 source file, a C stream encode function is generated. This function will encode a populated C variable of the given type into an encoded ASN.1 message and write it to a stream.

If the return status indicates success (0), the message will have been encoded to the given stream. Streaming BER encoding starts from the beginning of the message until the message is complete. This is sometimes referred to as "forward encoding". This differs from regular BER where encoding is done from back-to-front. Indefinite lengths are used for all constructed elements in the message. Also, there is no permanent buffer for streaming encoding, all octets are written to the stream. The buffer in the context structure is used only as a cache.

If C++ code generation is specified, a control class is generated that contains an *EncodeTo* method that wraps the stream encode C function. This function is invoked through the class interface to convert a populated *msgData* attribute variable into an encoded ASN.1 message.

Generated Streaming C Function Format and Calling Parameters

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

```
asn1BSE_[<prefix>]<prodName>
```

where <`prodName`> is the name of the ASN.1 production for which the function is being generated and <`prefix`> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
stat = asn1BSE_<name> (OSCTXT* pctxt,
                       <name>* pvalue,
                       ASN1TagType tagging);
```

In this definition, `<name>` denotes the prefixed production 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 program. This variable must be initialized using both the *rtInitContext* and *rtStreamBufInit* run-time library functions (see the *C/C++ Common Run-Time Library Reference Manual* for a description of these functions).

The `pvalue` argument holds a pointer to the data to be encoded and is of the type generated from the ASN.1 production.

The `tagging` argument is for internal use when calls to encode functions are nested to accomplish encoding of complex variables. It indicates whether the tag associated with the production should be applied or not (implicit versus explicit tagging). At the top level, the tag should always be applied so this parameter should always be set to the constant ASN1EXPL (for EXPLICIT).

The function result variable `stat` returns the completion status of the operation. 0 (0) means the success.

Procedure for Calling Streaming C Encode Functions

This section describes the step-by-step procedure for calling a streaming C BER encode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Before any encode function can be called; the user must first initialize an encoding context. 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 within the top-level calling function. **It must be initialized before use**. This can be accomplished by using the *berStrmInitContext* function:

```
OSCTXT ctxt;

if (berStrmInitContext (&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 output device to which the data is to be encoded and is initialized by calling one of the following functions:

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

The *flags* parameter of these functions should be set to the OSRTSTRMF_OUTPUT constant value to indicate an output stream is being created (see the *C/C++ Common Run-Time Library Reference Manual* for a full description of these functions).

It is also possible to use a simplified form of these function calls to create a writer interface to a file, memory, or socket stream:

- *rtxStreamFileCreateWriter*
- *rtxStreamMemoryCreateWriter*
- *rtxStreamSocketCreateWriter*

After initializing the context and populating a variable of the structure to be encoded, an encode function can be called to encode the message to the stream. The stream must then be closed by calling the *rtxStreamClose* function.

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

```
#include employee.h            /* include file generated by ASN1C */

int main ()
{
   int       stat;
   OSCTXT    ctxt;
   Employee  employee;/* typedef generated by ASN1C */
```

```
      const char* filename = “message.dat”;

      /* Step 1: Initialize the context and stream */

      if (berStrmInitContext (&ctxt) != 0) {
         /* initialization failed, could be a license problem */
         printf (“context initialization failed (check license)\n”);
         return –1;
      }

      /* Step 2: create a file stream object within the context */

      stat = rtxStreamFileCreateWriter (&ctxt, filename);
      if (stat != 0) {
         rtxErrPrint (&ctxt);
         return stat;
      }

      /* Step 3: Populate the structure to be encoded */

      employee.name = "SMITH";
      ...

      /* Step 4: Call the generated encode function */

      stat = asn1BSE_Employee (&ctxt, &employee, ASN1EXPL);

      /* Step 5: Check the return status and close the stream */

      if (stat != 0) {

         ...error processing...
      }

      rtxStreamClose (&ctxt);
   }
```

In general, streaming encoding is slower than memory buffer based encoding. However, in the case of streaming encoding, it is not necessary to implement code to write or send the encoded data to an output device. The streaming functions also use less memory because there is no need for a large destination memory buffer. For this reason, the final performance of the streaming functions may be the same or better than buffer-oriented functions.

Encoding a Series of Messages Using the Streaming C Encode Functions

A common application of BER encoding is the repetitive encoding of a series of the same type of message over and over again. For example, a TAP3 batch application might read billing data out of a database table and encode each of the records for a batch transmission.

Encoding a series of messages using the streaming C encode functions is very similar to encoding of one message. All that is necessary is to set up a loop in which the *asn1BSE_<name>* functions will be called. It is also possible to call different *asn1BSE_<name>* functions one after another. An example showing how to do this is as follows:

```
#include employee.h          // include file generated by ASN1C

int main ()
{
   int       stat;
   OSCTXT    ctxt;
   Employee  employee;/* typedef generated by ASN1C */
   const char* filename = “message.dat”;

   /* Step 1: Initialize the context and stream */

   if (berStrmInitContext (&ctxt) != 0) {
      /* initialization failed, could be a license problem */
      printf (“context initialization failed (check license)\n”);
      return –1;
   }

   stat = rtxStreamFileCreateWriter (&ctxt, filename);
   if (stat != 0) {
      rtxErrPrint (&ctxt);
      return stat;
   }

   for (;;) {
      /* Step 2: Populate the structure to be encoded */

      employee.name = "SMITH";
      ...

      /* Step 3: Call the generated encode function */

      stat = asn1BSE_Employee (&ctxt, &employee, ASN1EXPL);

      /* Step 4: Check the return status and break the loop
         if error occurs */

      if (stat != 0) {

         ...error processing...

         break;
      }
   }

   /* Step 5: Close the stream */

   rtxStreamClose (&ctxt);
}
```

Generated Streaming C++ Encode Method Format and Calling Parameters

C++ code generation of stream-based encoders is selected by using the **–c**++ and **–stream** compiler command line options. In this case, ASN1C generates an EncodeTo method that wraps the C function call. This method provides a more simplified calling interface because it hides things such as the context structure and tag type parameters.

The calling sequence for the generated C++ class method is as follows:

```
stat = <object>.EncodeTo (<outputStream>);
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production.

The <outputStream> placeholder represents an output stream object type. This is an object derived from an *ASN1EncodeStream* class.

The function result variable `stat` returns the completion status. Error status codes are negative. Return status values are defined in the *rtxErrCodes.h* include file.

Another way to encode a message using the C++ classes is to use the << streaming operator:

```
<outputStream> << <object>;
```

Exceptions are not used in ASN1C C++, therefore, the user must fetch the status value following a call such as this in order to determine if it was successful. The *getStatus* method in the *ASN1EncodeStream* class is used for this purpose.

Also, the method *Encode* without parameters is supported for backward compatibility. In this case it is necessary to create control class (i.e., ASN1C_<prodName>) using an output stream reference as the first parameter and *msgdata* reference as the second parameter of the constructor.

Procedure for Using the Streaming C++ Control Class Encode Method

The procedure to encode a message directly to an output stream using the C++ class interface is as follows:

1. Create an *OSRTOutputStream* object for the type of output stream. Choices are *OSRTFileOutputStream* for a file, *OSRTMemoryOutputStream* for a memory buffer, or *OSRTSocketOutputStream* for an IP socket connection.

2. Create an *ASN1BEREncodeStream* object using the stream object created in 1) as an argument.

3. Create a variable of the *ASN1T_<name>* type and populate it with the data to be encoded.

4. Create a variable of the generated *ASN1C_<name>* class specifying the item created in 2 as an argument to the constructor.

5. Invoke the *EncodeTo* method or << operator.

A program fragment that could be used to encode an employee record is as follows. This example uses a file output stream:

```
#include employee.h           // include file generated by ASN1C
#include "rtbersrc/ASN1BEREncodeStream.h"
#include "rtxsrc/OSRTFileOutputStream.h"
```

```
main ()
{
   int msglen;
   const char* filename = “message.dat”

   // step 1: construct output stream object.

   ASN1BEREncodeStream out (new OSRTFileOutputStream (filename));
   if (out.getStatus () != 0) {
      out.printErrorInfo ();
      return -1;
   }

   // step 2: construct ASN1C C++ generated class.

   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (msgData);

   // step 3: populate msgData structure with data to be
   // encoded. (note: this uses the generated assignment
   // operator to assign a string).

   msgData.name = “SMITH”;
   ...
   // step 4: invoke << operator or EncodeTo method

   out << employee;
   // or employee.EncodeTo (out); can be used here.

   // step 5: check status of the operation

   if (out.getStatus () != 0) {
      printf ("Encoding failed. Status = %i\n", out.getStatus());
      out.printErrorInfo ();
      return -1;
   }

   if (trace) {
      printf ("Encoding was successful\n");
   }
}
```

Encoding a Series of Messages Using the Streaming C++ Control Class Interface

Encoding a series of messages using the streaming C++ control class is similar to the C method of encoding. All that is necessary is to create a loop in which *EncodeTo* or Encode methods will be called (or the overloaded << streaming operator). It is also possible to call different *EncodeTo* methods (or *Encode* or operator <<) one after another. An example showing how to do this is as follows:

```
#include employee.h             // include file generated by ASN1C
#include "rtbersrc/ASN1BEREncodeStream.h"
#include "rtxsrc/OSRTFileOutputStream.h"
```

```
int main ()
{
   const   OSOCTET* msgptr;
   OSOCTET msgbuf[1024];
   int     msglen;
   const   char* filename = “message.dat”

   // step 1: construct stream object.

   ASN1BEREncodeStream out (new OSRTFileOutputStream (filename));
   if (out.getStatus () != 0) {
      out.printErrorInfo ();
      return -1;
   }

   // step 2: construct ASN1C C++ generated class.

   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (msgData);

   for (;;) {
      // step 3: populate msgData structure with data to be
      // encoded. (note: this uses the generated assignment
      // operator to assign a string).

      msgData.name = “SMITH”;
      ...

      // step 4: invoke << operator or EncodeTo method

      out << employee;
      // or employee.EncodeTo (out); can be used here.

      // step 5: fetch and check status

      if (out.getStatus () != 0) {
         printf ("Encoding failed. Status = %i\n", out.getStatus());
         out.printErrorInfo ();
         return -1;
      }

      if (trace) {
         printf ("Encoding was successful\n");
      }
   }
}
```

Generated BER Decode Functions

NOTE: This section assumes standard memory-buffer based decoding is to be done. If stream-based decoding is to be done (specified by adding -*stream* to the ASN1C command-line), see the **Generated BER Streaming Decode Functions** section for correct procedures on using the stream-based functions.

For each ASN.1 production defined in an ASN.1 source file, a C decode function is generated. This function will decode an ASN.1 message into a C variable of the given type.

If C++ code generation is specified, a control class is generated that contains a *Decode* method that wraps this function. This function is invoked through the class interface to decode an ASN.1 message into the variable referenced in the *msgData* component of the class.

Generated C Function Format and Calling Parameters

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

```
asn1D_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1D_<name> (OSCTXT* pctxt,
                       <name> *pvalue,
                       ASN1TagType tagging,
                       int length);
```

In this definition, <name> denotes the prefixed production name defined above.

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 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 *rtInitContext* run-time function before use.

The `pvalue` argument is a pointer to a variable of the generated type that will receive the decoded data.

The `tagging` and `length` arguments are for internal use when calls to decode functions are nested to accomplish decoding of complex variables. At the top level, these parameters should always be set to the constants ASN1EXPL and zero respectively.

The function result variable `status` returns the status of the decode operation. The return status will be zero if decoding is successful or negative if an error occurs. Return status values are defined in the "asn1type.h" include file.

Procedure for Calling C Decode Functions

This section describes the step-by-step procedure for calling a C BER or DER decode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Before any decode function can be called; the user must first 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 as follows:

```
OSCTXT ctxt;

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

The next step is the specification of a buffer containing a message to be decoded. This is accomplished by calling the *xd_setp* run-time library function. This function takes as an argument the start address of the message to be decoded. The function returns the starting tag value and overall length of the message. This makes it possible to identify the type of message received and apply the appropriate decode function to decode it.

A decode function can then be called to decode the message. If the return status indicates success, the C variable that was passed as an argument will contain the decoded message contents. Note that the decoder may have allocated dynamic memory and stored pointers to objects in the C structure. After processing on the C structure is complete, the run-time library function rtxMemFree should be called to free the allocated memory.

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

```
#include employee.h          /* include file generated by ASN1C */

main ()
{
   OSOCTET   msgbuf[1024];
   ASN1TAG   msgtag;
   int       msglen;
   OSCTXT    ctxt;
   PersonnelRecord employee;

   .. logic to read message into msgbuf ..

   /* Step 1: Initialize a context variable for decoding */

   if (rtInitContext (&ctxt) != 0) {
      /* initialization failed, could be a license problem */
      printf (“context initialization failed (check license)\n”);
      return –1;
   }

   xd_setp (&ctxt, msgbuf, 0, &msgtag, &msglen);

   /* Step 2: Test message tag for type of message received */
   /* (note: this is optional, the decode function can be */
   /* called directly if the type of message is known).. */

   if (msgtag == TV_PersonnelRecord)
   {
      /* Step 3: Call decode function (note: last two args */
      /* should always be ASN1EXPL and 0).. */

      status = asn1D_PersonnelRecord (&ctxt,
                                      &employee,
                                      ASN1EXPL, 0);

      /* Step 4: Check return status */

      if (status == 0)
      {
         process received data in ‘employee’ variable..

         /* Remember to release dynamic memory when done! */

         rtxMemFree (&ctxt);
      }
      else
         error processing...
   }
   else
      check for other known message types..
}
```

Decoding a Series of Messages Using the C Decode Functions

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? It will be necessary to put the decoding logic into a loop:

```
main ()
{
   OSOCTET   msgbuf[1024];
   ASN1TAG   msgtag;
   int       msglen;
   OSCTXT    ctxt;
   PersonnelRecord employee;

   /* Step 1: Initialize a context variable for decoding */

   if (rtInitContext (&ctxt) != 0) {
      /* initialization failed, could be a license problem */
      printf (“context initialization failed (check license)\n”);
      return –1;
   }

   for (;;) {

      .. logic to read message into msgbuf ..

      xd_setp (&ctxt, msgbuf, 0, &msgtag, &msglen);

      /* Step 2: Test message tag for type of message received */
      /* (note: this is optional, the decode function can be */
      /* called directly if the type of message is known).. */
```

```
         /* Now switch on initial tag value to determine what type of
            message was received.. */

         switch (msgtag)
         {
            case TV_PersonnelRecord: /* compiler generated constant */
            {
               status = asn1D_PersonnelRecord (&ctxt,
                                                &employee,
                                                ASN1EXPL, 0);
               if (status == 0)
               {

               /* decoding successful, data in employee */

               process received data..
            }
            else
               error processing...
         }
         break;

         default:
            /* handle unknown message type here */
      } /* switch */

      /* Need to reinitialize objects for next iteration */

      rtxMemReset (&ctxt);
   }
}
```

The only changes were the addition of the for (;;) loop and the call to *rtxMemReset* that was added at the bottom of the loop. This function resets the memory tracking parameters within the context to allow previously allocated memory to be reused for the next decode operation. Optionally, *rtxMemFree* can be called to release all memory. This will allow the loop to start again with no outstanding memory allocations for the next pass.

The example above assumes that logic existed that would read each message to be processed into the same buffer for every message processed inside the loop (i.e the buffer is reused each time). In the case in which the buffer already contains multiple messages, encoded back-to-back, it is necessary to advance the buffer pointer in each iteration:

```
   main ()
   {
      OSOCTET   msgbuf[1024];
      ASN1TAG   msgtag;
      int       offset = 0, msglen, len;
      OSCTXT    ctxt;
      PersonnelRecord employee;
      FILE*     fp;

      /* Step 1: Initialize a context variable for decoding */

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

      if (fp = fopen (filename, "rb")) {
         msglen = fread (msgbuf, 1, sizeof(msgbuf), fp);
      }
      else {
         ... handle error ...
      }

      for (; offset < msglen; ) {
         xd_setp (&ctxt, msgbuf + offset, msglen - offset, &msgtag, &len);

         /* Decode */

         if (tag == TV_PersonnelRecord) {

            /* Call compiler generated decode function */

            stat = asn1D_PersonnelRecord (&ctxt, &employee, ASN1EXPL, 0);
            if (stat == 0) {

               /* decoding successful, data in employee */

            }
            else {
               /* error handling */
               return -1;
            }
         }
         else {
            printf ("unexpected tag %hx received\n", tag);
         }
         offset += ctxt.buffer.byteIndex;
         rtxMemReset (&ctxt);
      }
   }
```

Generated C++ Decode Method Format and Calling Parameters

Generated decode functions are invoked through the class interface by calling the base class *Decode* method. The calling sequence for this method is as follows:

```
   status = <object>.Decode ();
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production

An *ASN1BERDecodeBuffer* object reference is a required argument to the <object> constructor. This is where the message start address and length are specified

The message length argument is used to specify the size of the message, if it is known. In ASN.1 BER or DER encoded messages, the overall length of the message is embedded in the first few bytes of the message, so this variable is not required. It is used as a test mechanism to determine if a corrupt or partial message was received. If the parsed message length is greater than this value, an error is returned. If the value is specified to be zero (the default), then this test is bypassed.

The function result variable `status` returns the status of the decode operation. The return status will be zero if decoding is successful or a negative value if an error occurs. Return status values are defined in Appendix A of the *C/ C++ Common Functions Reference Manual* and online in the *asn1type.h* include file.

Procedure for Using the C++ Control Class Decode Method

Normally when a message is received and read into a buffer, it can be one of several different message types. So the first job a programmer has before calling a decode function is determining which function to call. The *ASN1BERDecodeBuffer* class has a standard method for parsing the initial tag/length from a message to determine the type of message received. This call is used in conjunction with a switch statement on generated tag constants for the known message set in order to pick a decoder to call.

Once it is known which type of message has been received, an instance of a generated message class can be instantiated and the decode function called. The start of message pointer and message length (if known) must be specified either in the constructor call or in the call to the decode function itself.

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

```
#include employee.h             // include file generated by ASN1C

main ()
{
   OSOCTET   msgbuf[1024];
   ASN1TAG   msgtag;
   int       msglen, status;

   .. logic to read message into msgbuf ..

   // Use the ASN1BERDecodeBuffer class to parse the initial
   // tag/length from the message..

   ASN1BERDecodeBuffer decodeBuffer (msgbuf, len);

   status = decodeBuffer.ParseTagLen (msgtag, msglen);

   if (status != 0) {
      // handle error
      ...
   }

   // Now switch on initial tag value to determine what type of
   // message was received..

   switch (msgtag)
```

```
      {
         case TV_PersonnelRecord: // compiler generated constant
         {
            ASN1T_PersonnelRecord msgData;
            ASN1C_PersonnelRecord employee (decodeBuffer, msgData);

            if ((status = employee.Decode ()) == 0)
            {
               // decoding successful, data in msgData

               process received data..
            }
            else
               error processing...
         }

         case TV_ ...// handle other known messages
```

Note that the call to free memory is not required to release dynamic memory when using the C++ interface. This is because the control class hides all of the details of managing the context and releasing dynamic memory. The memory is automatically released when both the message buffer object (*ASN1BERMessageBuffer*) and the control class object (*ASN1C_<ProdName>*) are deleted or go out of scope. Reference counting of a context variable shared by both interfaces is used to accomplish this.

Decoding a Series of Messages Using the C++ Control Class Interface

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? The logic shown above would not be optimal from a performance standpoint because of the constant creation and destruction of the message processing objects. It would be much better to create all of the required objects outside of the loop and then reuse them to decode and process each message.

A code fragment showing a way to do this is as follows:

```
   #include employee.h             // include file generated by ASN1C

   main ()
   {
      OSOCTET  msgbuf[1024];
      ASN1TAG  msgtag;
      int      msglen, status;

      // Create message buffer, ASN1T, and ASN1C objects

      ASN1BERDecodeBuffer decodeBuffer (msgbuf, len);
      ASN1T_PersonnelRecord employeeData;
      ASN1C_PersonnelRecord employee (decodeBuffer, employeeData);

      for (;;) {

         .. logic to read message into msgbuf ..
         status = decodeBuffer.ParseTagLen (msgtag, msglen);

         if (status != 0) {
            // handle error
            ...
         }

      // Now switch on initial tag value to determine what type of
      // message was received..

      switch (msgtag)
      {
         case TV_PersonnelRecord: // compiler generated constant
         {
            if ((status = employee.Decode ()) == 0)
            {
               // decoding successful, data in employeeData

               process received data..
            }
            else
               error processing...
         }
         break;

         default:
            // handle unknown message type here

            } // switch

            // Need to reinitialize objects for next iteration

            if (!isLastIteration) employee.memFreeAll ();

         } // end of loop
```

This is quite similar to the first example. Note that we have pulled the *ASN1T_Employee* and *ASN1C_Employee* object creation logic out of the switch statement and moved it above the loop. These objects can now be reused to process each received message.

The only other change was the call to *employee.memFreeAll* that was added at the bottom of the loop. Since we can't count on the objects being deleted to automatically release allocated memory, we need to do it manually. This call will free all memory held within the decoding context. This will allow the loop to start again with no outstanding memory allocations for the next pass.

If the buffer already contains multiple BER messages encoded back-to-back then it is necessary to modify the buffer pointer in each iteration. The *getByteIndex* method should be used at the end of loop to get the current offset in the buffer. This offset should be used with the decode buffer object's *setBuffer* method call at the beginning of the loop to determine the correct buffer pointer and length:

```
   OSUINT32 offset = 0;
   for ( ; offset < msglen;) {

      // set buffer pointer and its length to decode
      decodeBuffer.setBuffer (&msgbuf[offset], msglen - offset);
      int curlen = (int)(msglen - offset);

      status = decodeBuffer.ParseTagLen (msgtag, curlen);

      if (status != 0) {
         // handle error
         ...
      }

      // Now switch on initial tag value to determine what type of
      // message was received..

      switch (msgtag)
      {
         case TV_PersonnelRecord: // compiler generated constant
         {
            if ((status = employee.Decode ()) == 0)
            {
               // decoding successful, data in employeeData

               process received data..
            }
            else
               error processing...
         }
         break;

         default:
         // handle unknown message type here

      } // switch

      // get new offset
      offset += decodeBuffer.getByteIndex ();

      // Need to reinitialize objects for next iteration (if it is not
      // last iteration)

      if (offset < msglen) employee.memFreeAll ();
   } // end of loop
```

BER Decode Performance Enhancement Techniques

There are a number of different things that can be done in application code to improve BER decode performance. These include adjusting memory allocation parameters, using compact code generation, using decode fast copy, and using initialization functions.

Dynamic Memory Management

By far, the biggest performance bottleneck when decoding ASN.1 messages is the allocation of memory from the heap. Each call to **new** or **malloc** is very expensive.

The decoding functions must allocate memory because the sizes of many of the variables that make up a message are not known at compile time. For example, an OCTET STRING that does not contain a size constraint can be an indeterminate number of bytes in length.

ASN1C does two things by default to reduce dynamic memory allocations and improve decoding performance:

1. Uses static variables wherever it can. Any BIT STRING, OCTET STRING, character string, or SEQUENCE OF or SET OF construct that contains a size constraint will result in the generation of a static array of elements sized to the max constraint bound.

2. Uses a special nibble-allocation algorithm for allocating dynamic memory. This algorithm allocates memory in large blocks and then splits up these blocks on subsequent memory allocation requests. This results in fewer calls to the kernel to get memory. The downside is that one request for a few bytes of memory can result in a large block being allocated.

Common run-time functions are available for controlling the memory allocation process. First, the default size of a memory block as allocated by the nibble-allocation algorithm can be changed. By default, this value is set to 4K bytes. The run-time function *rtMemSetDefBlkSize* can be called to change this size. This takes a single argument - the value to which the size should be changed.

It is also possible to change the underlying functions called from within the memory management abstraction layer to obtain or free heap memory. By default, the standard C *malloc*, *realloc*, and *free* functions are used. These can be changed by calling the *rtMemSetAllocFuncs* function. This function takes as arguments function pointers to the allocate, reallocate, and free functions to be used in place of the standard C functions.

Another run-time memory management function that can improve performance is *rtMemReset*. This function is useful when decoding messages in a loop. It is used instead of *rtMemFree* at the bottom of the loop to make dynamic memory available for decoding the next message. The difference is that *rtMemReset* does not actually free the dynamic memory. It instead just resets the internal memory management parameters so that memory already allocated can be reused. Therefore, all the memory required to handle message decoding is normally allocated within the first few passes of the loop. From that point on, that memory is reused thereby making dynamic memory allocation a negligent issue in the overall performance of the decoder.

A more detailed explanation of these functions and other memory management functions can be found in the *C/ C++Common Run-Time Library Reference Manual*.

Compact Code Generation

Using the compact code generation option (*-compact*) and lax validation option (*-lax*) can also improve decoding performance.

The *-compact* option causes code to be generated that contains no diagnostic or error trace messages. In addition, some status checks and other non-critical code are removed providing a slightly less robust but faster code base.

The *–lax* option causes all constraint checks to be removed from the generated code.

Performance intensive applications should also be sure to link with the compact version of the base run-time libraries. These libraries can be found in the *lib_opt* (for optimized) subdirectory. These run-time libraries also have all diagnostics and error trace messages removed as well as some non-critical status checks.

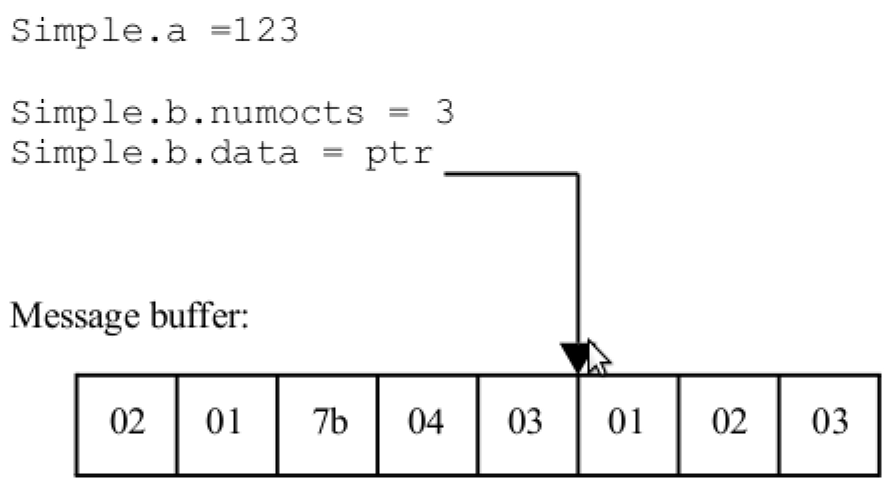
Decode Fast Copy

“Fast Copy” is a special run-time flag that can be set for the decoder that can substantially reduce the number of copy operations that need to be done to decode a message. The copy operations are reduced by taking advantage of the fact that the data contents of some ASN.1 types already exist in decoded form in the message buffer. Therefore, there is no need to allocate memory for the data and then copy the data from the buffer into the allocated memory structure.

As an example of what fast copy does, consider a simple ASN.1 SEQUENCE consisting of an element a, an INTEGER and b, an OCTET STRING:

```
Simple ::= SEQUENCE {
   a INTEGER,
   b OCTET STRING
}
```

Assume an encoded value of this type contains a value of a = 123 (hex 7B) and b contains the hex octets 0x01 0x02 0x03. The generated variable for the OCTET STRING will contain a data pointer. So rather than allocate memory for this string and copy the data to it, fast copy will simply store a pointer directly to the data in the buffer:



The pointer stored in the data structure points directly at data in the message buffer. No memory allocation or copy is done.

The user must keep in mind that if this technique is used, the message buffer containing the decoded message must be available as long as the type variable containing the decoded data is in use. This will not work in a producer-consumer threading model where one thread is decoding messages and the next thread is processing the contents. The producer thread will overwrite the buffer contents and therefore data referenced in the decoded message type variable that the consumer is processing.

This will also not work if the message buffer is an automatic variable in a function and the decoded result type is being passed out. The result type variable will point at data in the buffer variable that has gone out of scope.

To set fast copy, the *rtSetFastCopy* function must be invoked with the initialized context variable that will be used to decode a message. This should be done once prior to entering the loop that will be used to decode a stream of messages.

Using Initialization Functions

Initialization functions are generated by the ASN1C compiler when the *-genInit* option is added to the ASN1C command-line. These functions can be used as an alternative to memset'ing a variable to zero to prepare it to receive decoded data. The advantage is that the initialization functions are smarter and know exactly what within the structures needs to be zeroed as opposed to blindly clearing everything. So, for example, large byte arrays used to hold OCTET STRING data will not be zeroed. This can add up to significant performance improvements in the long run, particular in complex, deeply-nested ASN.1 types.

If initialization functions are generated, the generated decode logic will use them wherever it can in place of calls to zero memory.

BER/DER Deferred Decoding

Another way to improve decoding performance of large messages is through the use of *deferred decoding*. This allows for the selective decoding of only parts of a message in a single decode function call. When combined with the fast copy procedure defined above, this can significantly reduce decoding time because large parts of messages can be skipped.

Deferred decoding can be done on elements defined within a SEQUENCE, SET or CHOICE construct. It is done by designating an element to be an open type by using the *<isOpenType/>* configuration setting. This setting causes the ASN1C compiler to insert an *Asn1OpenType* placeholder in place of the type that would have normally been used for the element. The data in its original encoded form will be stored in the open type container when the message is decoded. If fast copy is used, only a pointer to the data in the message buffer is stored so large copies of data are avoided.

The data within the deferred decoding open type container can be fully decoded later by using a special decode function generated by the ASN1C compiler for this purpose. The format of this function is as follows:

```
asn1D_<ProdName>_<ElementName>_OpenType (OSCTXT* pctxt, <ElementType>* pvalue)
```

Here <*ProdName*> is replaced with name of the type assignment and <*ElementName* > is replaced with name of the element. In this function, the argument *pctxt* is used to pass the a pointer to a context variable initialized with the open type data and the *pvalue* argument will hold the final decoded data value.

In following example, decoding of the element *id* is deferred:

```
Identifier ::= SEQUENCE {
   id INTEGER,
   oid OBJECT IDENTIFIER
}
```

The following configuration file is required to indicate the element id is to be processed as an open type (i.e. that it will be decoded later):

```
<asn1config>
   <module>
      <name>modulename</name>
      <production>
         <name>Identifier</name>
         <element>
            <name>id</name>
            <isOpenType/>
         <element/>
      <production/>
   <module/>
<asn1config/>
```

In the generated code, the element *id* type will be replaced with an open type (*Asn1OpenType* for C or *Asn1TOpenType* for C++) and the following additional function is generated:

```
EXTERN int asn1D_Identifier_id_OpenType (OSCTXT* pctxt, OSINT32* pvalue);
```

In the *Identifier* decode function, element *id* is decoded as an open type.

Generated BER Streaming Decode Functions

BER messages can be directly read and decoded from an input stream such as a file, network or memory stream using BER streaming decode functions. The ASN1C compiler -*stream* option is used to generate decoders of this type. For each ASN.1 production defined in an ASN.1 source file, a C streaming decode function is generated. This function will decode an ASN.1 message into a C variable of the given type.

If C++ code generation is specified, a control class is generated that contains a *DecodeFrom* method that wraps this function. This function is invoked through the class interface to decode an ASN.1 message into the variable referenced in the *msgData* component of the class.

In this version, there are three types of streams: file, socket and memory. The most useful are file and socket streams. It is possible to decode data directly from a file or socket without intermediate copying into memory. If the full amount of data is not available for reading then the behavior of these streams will be different: the file and memory input streams will report an error, the socket input stream will block until data is available or an I/O error occurs (for example, the remote side closes the connection).

Generated Streaming C Function Format and Calling Parameters

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

```
asn1BSD_[<prefix>]<prodName>
```

where <`prodName`> is the name of the ASN.1 production for which the function is being generated and <`prefix`> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1BSD_<name> (OSCTXT* pctxt,
                         <name> *pvalue,
                         ASN1TagType tagging,
                         int length);
```

In this definition, <name> denotes the prefixed production name defined above.

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 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 *berStrmInitContext* run-time function before use.

The `pvalue` argument is a pointer to a variable of the generated type that will receive the decoded data.

The `tagging` and `length` arguments are for internal use when calls to decode functions are nested to accomplish decoding of complex variables. At the top level, these parameters should always be set to the constants ASN1EXPL and zero respectively.

The function result variable status returns the status of the decode operation. The return status will be zero if decoding is successful or negative if an error occurs. Return status values are defined in the *rtxErrCodes.h* include file.

Procedure for Calling Streaming C Decode Functions

This section describes the step-by-step procedure for calling a streaming C BER decode function. This procedure must be followed if C code generation was done. This procedure can also be used as an alternative to using the control class interface if C++ code generation was done.

Before any decode function can be called; the user must first 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 *berStrmInitContext* function.

```
OSCTXT ctxt; // context variable

if (berStrmInitContext (&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 output device to which the data is to be encoded and is initialized by calling one of the following functions:

- *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 (see the *C/C++ Common Run-Time Library Reference Manual* for a full description of these functions).

A simplified version of the *Open* functions are the *CreateReader* functions:

- *rtxStreamFileCreateReader*
- *rtxStreamMemoryCreateReader*
- *rtxStreamSocketCreateReader*

After initializing the context and populating a variable of the structure to be encoded, a decode function can be called to decode a message from the stream. If the return status indicates success, the C variable that was passed as an argument will contain the decoded message contents. Note that the decoder may have allocated dynamic memory and stored pointers to objects in the C structure. After processing on the C structure is complete, the run-time library function *rtxMemFree* should be called to free the allocated memory.

After stream processing is complete, the stream is closed by invoking the *rtxStreamClose* function.

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

```
#include “employee.h”         /* include file generated by ASN1C */
```

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

main ()
{
   ASN1TAG msgtag;
   int     msglen;
   OSCTXT  ctxt;
   PersonnelRecord employee;
   const char* filename = “message.dat”

   /* Step 1: Initialize a context variable for decoding */

   if (berStrmInitContext (&ctxt) != 0) {
      /* initialization failed, could be a license problem */
      printf (“context initialization failed (check license)\n”);
      return –1;
   }

   /* Step 2: Open the input stream to read data */

   stat = rtxStreamFileCreateReader (&ctxt, filename);
   if (stat != 0) {
      rtxErrPrint (&ctxt);
      return stat;
   }

   /* Step 3: Test message tag for type of message received */
   /* (note: this is optional, the decode function can be */
   /* called directly if the type of message is known).. */

   if ((stat = berDecStrmPeekTagAndLen (&ctxt, &tag, &len)) != 0) {
      rtxErrPrint (&ctxt);
      return stat;
   }

   if (msgtag == TV_PersonnelRecord)
   {
      /* Step 4: Call decode function (note: last two args */
      /* should always be ASN1EXPL and 0).. */

      status = asn1BSD_PersonnelRecord (&ctxt,
                                         &employee,
                                         ASN1EXPL, 0);

      /* Step 5: Check return status */

      if (status == 0)
      {
         process received data in ‘employee’ variable..
      }

      else
         error processing...
   }
   else
      check for other known message types..

   /* Step 6: Close the stream */

   rtxStreamClose (&ctxt);
```

```
      /* Remember to release dynamic memory when done! */

      rtFreeContext (&ctxt);
      }
```

Decoding a Series of Messages Using the Streaming C Decode Functions

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? It will be necessary to put the decoding logic into a loop.

A code fragment showing a way to do this is as follows:

```
   #include “employee.h”            /* include file generated by ASN1C */
   #include "rtxsrc/rtxStreamFile.h"

   main ()
   {
      ASN1TAG msgtag;
      int msglen, stat;
      OSCTXT ctxt;
      PersonnelRecord employee;
      const char* filename = “message.dat”

      /* Step 1: Initialize a context variable for decoding */

      if (berStrmInitContext (&ctxt) != 0) {
         /* initialization failed, could be a license problem */
         printf (“context initialization failed (check license)\n”);
         return –1;
      }

      /* Step 2: Open the input stream to read data */

      stat = rtxStreamFileCreateReader (&ctxt, filename);
      if (stat != 0) {
         rtxErrPrint (&ctxt);
         return stat;
      }

      for (;;) {
         /* Step 3: Test message tag for type of message received */
         /* (note: this is optional, the decode function can be */
         /* called directly if the type of message is known).. */

         if ((stat = berDecStrmPeekTagAndLen (&ctxt, &tag, &len)) != 0) {
            rtxErrPrint (&ctxt);
            return stat;
         }

         if (msgtag == TV_PersonnelRecord)
         {
            /* Step 4: Call decode function (note: last two args */
            /* should always be ASN1EXPL and 0).. */
            stat = asn1BSD_PersonnelRecord (&ctxt,
                                            &employee,
                                            ASN1EXPL, 0);

            /* Step 5: Check return status */

            if (stat == 0)
            {
               process received data in ‘employee’ variable..

            }
            else
               error processing...
         }
         else
         check for other known message types..

         /* Need to reset all memory for next iteration */

         rtxMemReset (&ctxt);

      } /* end of loop */

      /* Step 6: Close the stream */

      rtxStreamClose (&ctxt);

      /* Remember to release dynamic memory when done! */

      rtFreeContext (&ctxt);
   }
```

The only changes were the addition of the for (;;) loop and the call to *rtxMemReset* that was added at the bottom of the loop. This function resets the memory tracking parameters within the context to allow previously allocated memory to be reused for the next decode operation. Optionally, *rtxMemFree* can be called to release all memory. This will allow the loop to start again with no outstanding memory allocations for the next pass.

Generated Streaming C++ Decode Method Format and Calling Parameters

Generated C streaming decode functions are invoked through the C++ class interface by calling the generated DecodeFrom method. The calling sequence for this method is as follows:

```
   status = <object>.DecodeFrom (<inputStream>);
```

In this definition, <object> is an instance of the control class (i.e., ASN1C_<prodName>) generated for the given production.

The <inputStream> placeholder represents an input stream object type. This is an object derived from an *ASN1DecodeStream* class.

The function result variable `stat` returns the completion status. Error status codes are negative. Return status values are defined in the *rtxErrCodes.h* include file.

Another way to decode message using the C++ class interface is to use the >> stream operator:

```
<inputStream> >> <object>;
```

Exceptions are not used in ASN1C C++, therefore, the user must fetch the status value following a call such as this in order to determine if it was successful. The *getStatus* method in the *ASN1DecodeStream* class is used for this purpose.

Also, the method *Decode* without parameters is supported for backward compatibility. In this case it is necessary to create a control class object (i.e., ASN1C_<prodName>) using an input stream reference as the first parameter and a reference to a variable of the generated type as the second parameter of the constructor.

Procedure for Using the Streaming C++ Control Class Decode Method

Normally the receiving message can be one of several different message types. It is therefore necessary to determine the type of message that was received so that the appropriate decode function can be called to decode it. The *ASN1BERDecodeStream* class has standard methods for parsing the initial tag/length from a message to determine the type of message received. These calls are used in conjunction with a switch statement on generated tag constants for the known message set. Each switch case statement contains logic to create an object instance of a specific ASN1C generated control class and to invoke and then to invoke that object's decode method.

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

```
#include "Employee.h"         // include file generated by ASN1C
#include "rtbersrc/ASN1BERDecodeStream.h"
#include "rtxsrc/OSRTFileInputStream.h"

main ()
{
   ASN1TAG tag;
   int i, len;
   const char* filename = "message.dat";
   OSBOOL trace = TRUE;

   // Decode

   ASN1BERDecodeStream in (new OSRTFileInputStream (filename));
   if (in.getStatus () != 0) {
      in.printErrorInfo ();
      return -1;
   }

   if (in.peekTagAndLen (tag, len) != 0) {
      printf ("peekTagAndLen failed\n");
      in.printErrorInfo ();
      return -1;
   }

   // Now switch on initial tag value to determine what
```

```
      // type of message was received..

      switch (msgtag)
      {
         case TV_PersonnelRecord: // compiler generated
                                  // constant
         {
            ASN1T_PersonnelRecord msgData;
            ASN1C_PersonnelRecord employee (msgData);

            in >> employee;
            if (in.getStatus () != 0) {
               printf ("decode of PersonnelRecord failed\n");
               in.printErrorInfo ();
               return -1;
            }

            // or employee.DecodeFrom (in);
            break;
         }

         case TV_ ...// handle other known messages
            ...
         }
      }

      return 0;
   }
```

Note that the call to free memory and the stream close method are not required to release dynamic memory when using the C++ interface. This is because the control class hides all of the details of managing the context and releasing dynamic memory. The memory is automatically released when both the input stream object (*ASN1BERDecodeStream* and derived classes) and the control class object (*ASN1C_<ProdName>*) are deleted or go out of scope. Reference counting of a context variable shared by both interfaces is used to accomplish this.

Decoding a Series of Messages Using the C++ Control Class Interface

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? The logic shown above would not be optimal from a performance standpoint because of the constant creation and destruction of the message processing objects. It would be much better to create all of the required objects outside of the loop and then reuse them to decode and process each message.

A code fragment showing a way to do this is as follows:

```
   #include "Employee.h"      // include file generated by ASN1C
   #include "rtbersrc/ASN1BERDecodeStream.h"
   #include "rtxsrc/OSRTFileInputStream.h"

   int main ()
   {
      ASN1TAG tag;
      int i, len;
      const char* filename = "message.dat";
      OSBOOL trace = TRUE;

      // Decode

      ASN1BERDecodeStream in (new OSRTFileInputStream (filename));
      if (in.getStatus () != 0) {
         in.printErrorInfo ();
         return -1;
      }

      ASN1T_PersonnelRecord msgData;
      ASN1C_PersonnelRecord employee (msgData);

      for (;;) {
         if (in.peekTagAndLen (tag, len) != 0) {
            printf ("peekTagAndLen failed\n");
            in.printErrorInfo ();
            return -1;
         }

         // Now switch on initial tag value to determine what
         // type of message was received..

         switch (msgtag)
         {
            case TV_PersonnelRecord: // compiler generated
                                     // constant
         {
            in >> employee;

            if (in.getStatus () != 0) {
               printf ("decode of PersonnelRecord failed\n");
               in.printErrorInfo ();
               return -1;
            }

               // or employee.DecodeFrom (in);

            }

            case TV_ ...// handle other known messages
               ...
         }

         // Need to reinitialize objects for next iteration

         employee.memFreeAll ();

      } // end of loop

      return 0;
   }
```

This is quite similar to the first example. Note that we have pulled the *ASN1T_Employee* and *ASN1C_Employee* object creation logic out of the switch statement and moved it above the loop. These objects can now be reused to process each received message.

The only other change was the call to *employee.memFreeAll* that was added at the bottom of the loop. Since the objects are not deleted to automatically release allocated memory, we need to do it manually. This call will free all memory held within the decoding context. This will allow the loop to start again with no outstanding memory allocations for the next pass.

Generated PER Functions

Generated PER Encode Functions

PER encode/decode functions are generated when the *-per* switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C PER encode function is generated. This function will convert a populated C variable of the given type into a PER encoded ASN.1 message.

If C++ code generation is specified, a control class is generated that contains an *Encode* method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the msgData component of the class.

Generated C Function Format and Calling Parameters

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

```
asn1PE_[<prefix>]<prodName>
```

where <`prodName`> is the name of the ASN.1 production for which the function is being generated and <`prefix`> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
status = asn1PE_<name> (OSCTXT* pctxt, <name>[*] value);
```

In this definition, <name> denotes the prefixed production 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 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 ASN.1 production. The object is passed by value if it is a primitive ASN.1 data type such as BOOLEAN, INTEGER, ENUMERATED, etc.. It is passed using a pointer reference if it is a structured ASN.1 type value. Check the generated function prototype in the header file to determine how the value argument is to be passed for a given function.

The function result variable `stat` returns the status of the encode operation. Status code 0 (0) indicates the function was successful. Note that this return value differs from that of BER encode functions in that the encoded length of the message component is not returned – only an OK status indicating encoding was successful. A negative value indicates encoding failed. Return status values are defined in the "asn1type.h" include file. The error text and a stack trace can be displayed using the *rtErrPrint* function.

Generated C++ Encode Method Format and Calling Parameters

Generated encode functions are invoked through the class interface by calling the base class Encode method. The calling sequence for this method is as follows:

```
   stat = <object>.Encode ();
```

In this definition, <object> is an object of the class generated for the given production. The function result variable `stat` returns the status value from the PER encode function. This status value will be 0 (0) if encoding was successful or a negative error status value if encoding fails. Return status values are defined in the "asn1type.h" include file.

The user must call the encode buffer class methods *getMsgPtr* and *getMsgLen* to obtain the starting address and length of the encoded message component.

Populating Generated Structure Variables for Encoding

See the section *Populating Generated Structure Variables* for Encoding for a discussion on how to populate variables for encoding. There is no difference in how it is done for BER versus how it is done for PER.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling a C PER encode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Before a PER encode function can be called, the user must first initialize an encoding context block structure. The context block is initialized by calling the *rtInitContext* to initialize the block. The user then must call *pu_setBuffer* to specify a message buffer to receive the encoded message. Specification of a dynamic message buffer is possible by setting the buffer address argument to null and the buffer size argument to zero. The *pu_setBuffer* function also allows for the specification of aligned or unaligned encoding.

An encode function can then be called to encode the message. If the return status indicates success (0), then the message will have been encoded in the given buffer. PER encoding starts from the beginning of the buffer and proceeds from low memory to high memory until the message is complete. This differs from BER where encoding was done from back-to-front. Therefore, the buffer start address is where the encoded PER message begins. The length of the encoded message can be obtained by calling the *pe_GetMsgLen* run-time function. If dynamic encoding was specified (i.e., a buffer start address and length were not given), the run-time routine *pe_GetMsgPtr* can be used to obtain the start address of the message. This routine will also return the length of the encoded message.

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

```
#include employee.h            /* include file generated by ASN1C */

main ()
{
   OSOCTET msgbuf[1024];
   int msglen, stat;
   OSCTXT ctxt;
   OSBOOL aligned = TRUE;
   Employee employee; /* typedef generated by ASN1C */

   /* Populate employee C structure */

   employee.name.givenName = "SMITH";
   ...

   /* Allocate and initialize a new context pointer */

   stat = rtInitContext (&ctxt);

   if (stat != 0) {
      printf (“rtInitContext failed (check license)\n“);
      rtxErrPrint (&ctxt);
      return stat;
   }

   pu_setBuffer (&ctxt, msgbuf, msglen, aligned);

   if ((stat = asn1PE_Employee (&ctxt, &employee)) == 0) {
      msglen = pe_GetMsgLen (&ctxt);
      ...
   }
   else
      error processing...
}
```

In general, static buffers should be used for encoding messages where possible as they offer a substantial performance benefit over dynamic buffer allocation. The problem with static buffers, however, is that you are required to estimate in advance the approximate size of the messages you will be encoding. There is no built-in formula to do this, the size of an ASN.1 message can vary widely based on data types and other factors.

If performance is not a significant issue, then dynamic buffer allocation is a good alternative. Setting the buffer pointer argument to NULL in the call to *pu_setBuffer* specifies dynamic allocation. This tells the encoding functions to allocate a buffer dynamically. The address of the start of the message is obtained after encoding by calling the run-time function *pe_GetMsgPtr* .

The following code fragment illustrates PER encoding using a dynamic buffer:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
   OSOCTET *msgptr;
   int msglen, stat;
   OSCTXT ctxt;
   OSBOOL aligned = TRUE;
   Employee employee;/* typedef generated by ASN1C */

   employee.name.givenName = "SMITH";
   ...

   stat = rtInitContext (&ctxt);

   if (stat != 0) {
      printf (“rtInitContext failed (check license)\n“);
      rtxErrPrint (&ctxt);
      return stat;
   }

   pu_setBuffer (&ctxt, 0, 0, aligned);

   if ((stat = asn1PE_Employee (&ctxt, &employee)) == 0) {
      msgptr = pe_GetMsgPtr (&ctxt, &msglen);
      ...

   }
   else
      error processing...
}
```

It is also possible to encode directly to a stream interface. To do this, the call to *pu_setBuffer* above would be replaced with a call to create a stream writer within the context such as *rtxStreamFile-CreateWriter*. The call to the generated PER encode function would not change - it will automatically know to use the stream interface instead of a memory buffer.

Procedure for Using the C++ Control Class Encode Method

The procedure to encode a message using the C++ class interface is as follows:

1. Instantiate an ASN.1 PER encode buffer object (ASN1PEREncodeBuffer) to describe the buffer into which the message will be encoded. Two overloaded constructors are available. The first form takes as arguments a static encode buffer and size and a Boolean value indicating whether aligned encoding is to be done. The second form only takes the Boolean aligned argument. This form is used to specify dynamic encoding.

2. Instantiate an ASN1T_<ProdName> object and populate it with data to be encoded.

3. Instantiate an ASN1C_<ProdName> object to associate the message buffer with the data to be encoded.

4. Invoke the ASN1C_<ProdName> object Encode method.

5. Check the return status. The return value is a status value indicating whether encoding was successful or not. Zero indicates success. If encoding failed, the status value will be a negative number. The encode buffer method *printErrorInfo* can be invoked to get a textual explanation and stack trace of where the error occurred.

6. If encoding was successful, get the start-of-message pointer and message length. The start-of-message pointer is obtained by calling the *getMsgPtr* method of the encode buffer object. If static encoding was specified (i.e., a message buffer address and size were specified to the PER Encode Buffer class constructor), the start-of-message pointer is the buffer start address. The message length is obtained by calling the *getMsgLen* method of the encode buffer object.

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

```
#include employee.h       // include file generated by ASN1C

main ()
{
   const OSOCTET* msgptr;
   OSOCTET msgbuf[1024];
   int msglen, stat;
   OSBOOL aligned = TRUE;

   // step 1: instantiate an instance of the PER encode
   // buffer class. This example specifies a static
   // message buffer..

   ASN1PEREncodeBuffer encodeBuffer (msgbuf,
                                     sizeof(msgbuf),
                                     aligned);

   // step 2: populate msgData with data to be encoded

   ASN1T_PersonnelRecord msgData;
   msgData.name.givenName = "SMITH";
   ...

   // step 3: instantiate an instance of the ASN1C_<ProdName>
   // class to associate the encode buffer and message data..

   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // steps 4 and 5: encode and check return status

   if ((stat = employee.Encode ()) == 0)
   {
      printf ("Encoding was successful\n");
      printf ("Hex dump of encoded record:\n");
      encodeBuffer.hexDump ();
      printf ("Binary dump:\n");
      encodeBuffer.binDump ("employee");

      // step 6: get start-of-message pointer and message length.
      // start-of-message pointer is start of msgbuf
      // call getMsgLen to get message length..
      msgptr = encodeBuffer.getMsgPtr (); // will return &msgbuf
      len = encodeBuffer.getMsgLen ();
   }
   else
   {
      printf ("Encoding failed\n");
      encodeBuffer.printErrorInfo ();
      exit (0);
   }

   // msgptr and len now describe fully encoded message

   ...
```

In general, static buffers should be used for encoding messages where possible as they offer a substantial performance benefit over dynamic buffer allocation. The problem with static buffers, however, is that you are required to estimate in advance the approximate size of the messages you will be encoding. There is no built-in formula to do this, the size of an ASN.1 message can vary widely based on data types and other factors.

If performance is not a significant issue, then dynamic buffer allocation is a good alternative. Using the form of the *ASN1PEREncodeBuffer* constructor that does not include buffer address and size arguments specifies dynamic buffer allocation. This constructor only requires a Boolean value to specify whether aligned or unaligned encoding should be performed (aligned is true).

The following code fragment illustrates PER encoding using a dynamic buffer:

```
#include employee.h          // include file generated by ASN1C

main ()
{
   OSOCTET* msgptr;
   int msglen, stat;
   OSBOOL aligned = TRUE;

   // Create an instance of the compiler generated class.
   // This example does dynamic encoding (no message buffer
   // is specified)..

   ASN1PEREncodeBuffer encodeBuffer (aligned);
   ASN1T_PersonnelRecord msgData;
   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // Populate msgData within the class variable

   msgData.name.givenName = "SMITH";
   ...

   // Encode

   if ((stat = employee.Encode ()) == 0)
   {
      printf ("Encoding was successful\n");
      printf ("Hex dump of encoded record:\n");
      encodeBuffer.hexDump ();
      printf ("Binary dump:\n");
```

```
         encodeBuffer.binDump ("employee");

         // Get start-of-message pointer and length

         msgptr = encodeBuffer.getMsgPtr ();
         len = encodeBuffer.getMsgLen ();
      }
      else
      {
         printf ("Encoding failed\n");
         encodeBuffer.printErrorInfo ();
         exit (0);

      }
      return 0;
   }
```

It is also possible to encode a PER message to a stream rather than a memory buffer. To do this, you would first declare a variable of one of the OSRT output stream classes. This would then be associated with the encode buffer through the ASN1PERencode buffer declaration. Everything after that would be similar to the memory buffer based program. The preceding program fragment rewritten to do streaming output to a file would look like this:

```
   #include employee.h          // include file generated by ASN1C

   main ()
   {
      int stat;
      OSBOOL aligned = TRUE;
      const char* filename = "message_out.per";

      // Create an instance of the compiler generated class.
      // This example write output to a file stream..

      OSRTFileOutputStream fostrm (filename);
      ASN1PEREncodeBuffer encodeBuffer (fostrm, aligned);
      ASN1T_PersonnelRecord msgData;
      ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

      // Populate msgData within the class variable

      msgData.name.givenName = "SMITH";
      ...

      // Encode

      if ((stat = employee.Encode ()) == 0)
      {
         printf ("Encoding was successful\n");
         printf ("Hex dump of encoded record:\n");
         encodeBuffer.hexDump ();
         printf ("Binary dump:\n");
         encodeBuffer.binDump ("employee");
      }
      else
      {
         printf ("Encoding failed\n");
         encodeBuffer.printErrorInfo ();
```

```
        exit (0);

     }
     return 0;
  }
```

Note that the encodeBuffer.hexDump and encodeBuffer.binDump commands work despite the fact that the output has been written to a stream. This is because a capture buffer is used when tracing is enabled to record all of the encoded information. If memory is tight, a user should ensure that trace output is turned off when using the stream.

Encoding a Series of PER Messages using the C++ Interface

When encoding a series of PER messages using the C++ interface, performance can be improved by reusing the message processing objects to encode each message rather than creating and destroying the objects each time. A detailed example of how to do this was given in the section on BER message encoding. The PER case would be similar with the PER function calls substituted for the BER calls. As was the case for BER, the encode message buffer object *init* method can be used to reinitialize the encode buffer between invocations of the encode functions.

Generated PER Decode Functions

PER encode/decode functions are generated when the *-per* switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C PER decode function is generated. This function will parse the data contents from a PER-encoded ASN.1 message and populate a variable of the corresponding type with the data.

If C++ code generation is specified, a control class is generated that contains a *Decode* method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the msgData component of the class.

Generated C Function Format and Calling Parameters

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

```
  asn1PD_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1PD_<name> (OSCTXT* pctxt, <name>* pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

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 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 `pvalue` argument is a pointer to a variable to hold the decoded result. This variable is of the type generated from the ASN.1 production. 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 `stat` returns the status of the decode operation. Status code 0 (0) indicates the function was successful. A negative value indicates decoding failed. Return status values are defined in the "asn1type.h" include file. The reason text and a stack trace can be displayed using the rtErrPrint function described later in this document.

Generated C++ Decode Method Format and Calling Parameters

Generated decode functions are invoked through the class interface by calling the base class *Decode* method. The calling sequence for this method is as follows:

```
status = <object>.Decode ();
```

In this definition, <object> is an object of the class generated for the given production.

An *ASN1PERDecodeBuffer* object must be passed to the <object> constructor prior to decoding. This is where the message start address and length are specified. A Boolean argument is also passed indicating whether the message to be decoded was encoded using aligned or unaligned PER

The function result variable `status` returns the status of the decode operation. The return status will be zero (0) if decoding is successful or a negative value if an error occurs. Return status values are documented in the *C/C++ Common Functions Reference Manual* and in the *rtxErrCodes.h* include file.

Procedure for Calling C Decode Functions

This section describes the step-by-step procedure for calling a C PER decode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Unlike BER, the user must know the ASN.1 type of a PER message before it can be decoded. This is because the type cannot be determined at run-time. There are no embedded tag values to reference to determine the type of message received.

The following are the basic steps in calling a compiler-generated decode function:

1. Prepare a context variable for decoding
2. Initialize the data structure to receive the decoded data
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 PER decode function can be called, the user must first initialize a context block structure. The context block is initialized by either calling the *rtNewContext* function (to allocate a dynamic context block), or by calling *rtInitContext* to initialize a static block. The *pu_setBuffer* function must then be called to specify a message buffer that contains the PER-encoded message to be decoded. This function also allows for the specification of aligned or unaligned decoding.

The variable that is to receive the decoded data must then be initialized. This can be done by either initializing the variable to zero using memset, or by calling the ASN1C generated initialization function.

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 ASN.1 type variable. The decode function may automatically allocate dynamic memory to hold variable length variables 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 free the context block. This must be done regardless of whether the block is static (declared on the stack and initialized using *rtInitContext*), or dynamic (created using *rtNewContext*). The function to free the context is *rtFreeContext*.

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

```
#include employee.h           /* include file generated by ASN1C */

main ()
{
   OSOCTET msgbuf[1024];
   ASN1TAG msgtag;
   int msglen, stat;
   OSCTXT ctxt;
   OSBOOL aligned = TRUE;
   PersonnelRecord employee;

   .. logic to read message into msgbuf ..

   /* This example uses a static context block */

   /* step 1: prepare the context block */

   stat = rtInitContext (&ctxt);

   if (stat != 0) {
      printf (“rtInitContext failed (check license)\n“);
      rtErrPrint (&ctxt);
      return stat;
   }

   pu_setBuffer (&ctxt, msgbuf, msglen, aligned);

   /* step 2: initialize the data variable */

   asn1Init_PersonnelRecord (&employee);

   /* step 3: call the decode function */

   stat = asn1PD_PersonnelRecord (&ctxt, &employee);

   if (stat == 0)
   {
      process received data..
   }
   else {
      /* error processing... */
      rtxErrPrint (&ctxt);
   }

   /* step 4: free the context */

   rtFreeContext (&ctxt);
}
```

An input stream can be used instead of a memory buffer as the data source by replacing the *pu_setBuffer* call above with one of the *rtxStream*CreateReader* functions to set up a file, memory, or socket stream as input.

Procedure for Using the C++ Control Class Decode Method

The following are the steps are involved in decoding a PER message using the generated C++ class:

1. Instantiate an ASN.1 PER decode buffer object (*ASN1PERDecodeBuffer*) to describe the message to be decoded. The constructor takes as arguments a pointer to the message to be decoded, the length of the message, and a flag indicating whether aligned encoding was used or not.

2. Instantiate an ASN1T_<ProdName> object to hold the decoded message data.

3. Instantiate an ASN1C_<ProdName> object to decode the message. This class associates the message buffer object with the object that is to receive the decoded data. The results of the decode operation will be placed in the variable declared in step 2.

4. Invoke the ASN1C_<ProdName> object *Decode* method.

5. Check the return status. The return value is a status value indicating whether decoding was successful or not. Zero (0) indicates success. If decoding failed, the status value will be a negative

number. The decode buffer method *PrintErrorInfo* can be invoked to get a textual explanation and stack trace of where the error occurred.

6. Release dynamic memory that was allocated by the decoder. All memory associated with the decode context is released when both the *ASN1PERDecodeBuffer* and *ASN1C_<ProdName>* objects go out of scope.

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

```
#include employee.h           // include file generated by ASN1C

main ()
{
   OSOCTET msgbuf[1024];
   int     msglen, stat;
   OSBOOL  aligned = TRUE;

   .. logic to read message into msgbuf ..

   // step 1: instantiate a PER decode buffer object

   ASN1PERDecodeBuffer decodeBuffer (msgbuf, msglen, aligned);

   // step 2: instantiate an ASN1T_<ProdName> object

   ASN1T_PersonnelRecord msgData;

   // step 3: instantiate an ASN1C_<ProdName> object

   ASN1C_PersonnelRecord employee (decodeBuffer, msgData);

   // step 4: decode the record

   stat = employee.Decode ();

   // step 5: check the return status

   if (stat == 0)
   {
      process received data..
   }
   else {
      // error processing..
      decodeBuffer.PrintErrorInfo ();
   }

   // step 6: free dynamic memory (will be done automatically
   // when both the decodeBuffer and employee objects go out
   // of scope)..
}
```

A stream can be used as the data input source instead of a memory buffer by creating an OSRT input stream object in step1 and associating it with the decodeBuffer object. For example, to read from a file input stream, the decodeBuffer declaration in step 1 would be replaced with the following two statements:

```
OSRTFileInputStream istrm (filename);
```

```
ASN1PERDecodeBuffer decodeBuffer (istrm, aligned);
```

Decoding a Series of Messages Using the C++ Control Class Interface

The above example is fine as a sample for decoding a single message, but what happens in the more typical scenario of having a long-running loop that continuously decodes messages? The logic shown above would not be optimal from a performance standpoint because of the constant creation and destruction of the message processing objects. It would be much better to create all of the required objects outside of the loop and then reuse them to decode and process each message.

A code fragment showing a way to do this is as follows:

```
#include employee.h          // include file generated by ASN1C

main ()
{
   OSOCTET msgbuf[1024];
   int     msglen, stat;
   OSBOOL  aligned = TRUE;

   // step 1: instantiate a PER decode buffer object

   ASN1PERDecodeBuffer decodeBuffer (msgbuf, msglen, aligned);

   // step 2: instantiate an ASN1T_<ProdName> object

   ASN1T_PersonnelRecord msgData;

   // step 3: instantiate an ASN1C_<ProdName> object

   ASN1C_PersonnelRecord employee (decodeBuffer, msgData);

   // loop to continuously decode records

   for (;;) {
      .. logic to read message into msgbuf ..

      stat = employee.Decode ();
      // step 5: check the return status

      if (stat == 0)
      {
         process received data..
      }
      else {
         // error processing..
         decodeBuffer.PrintErrorInfo ();
      }

      // step 6: free dynamic memory

      employee.memFreeAll ();

      // If reading unaligned data, it is necessary to do a byte align operation to move
```

```
         // to the next octet boundary before decoding the next message..

         decodeBuffer.byteAlign();
      }
   }
```

The only difference between this and the previous example is the addition of the decoding loop and the modification of step 6 in the procedure. The decoding loop is an infinite loop to continuously read and decode messages from some interface such as a network socket. The decode calls are the same, but before in step 6, we were counting on the message buffer and control objects to go out of scope to cause the memory to be released. Since the objects are now being reused, this will not happen. So the call to the *memFreeAll* method that is defined in the *ASN1C_Type* base class will force all memory held at that point to be released.

Performance Considerations: Dynamic Memory Management

Please refer to *Performance Considerations: Dynamic Memory Management* in the *BER Decode Functions* section for a discussion of memory management performance issues. All of the issues that apply to BER and DER also apply to PER as well.

Generated Octet Encoding Rules (OER) Functions

Generated OER Encode Functions

OER encode/decode functions are generated when the *-oer* switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C OER encode function is generated. This function will convert a populated C variable of the given type into an OER-encoded ASN.1 message.

C++ is not directly supported for OER; however, it is possible to call the generated C functions from a C++ program.

Generated C Function Format and Calling Parameters

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

```
OEREnc_[<prefix>]<prodName>
```

where <`prodName`> is the name of the ASN.1 production for which the function is being generated and <`prefix`> is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
status = OEREnc_<name> (OSCTXT* pctxt, <name>[*] value);
```

In this definition, <name> denotes the prefixed production 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 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 ASN.1 production. The object is passed by value if it is a primitive ASN.1 data type such as BOOLEAN, INTEGER, ENUMERATED, etc.. It is passed using a pointer reference if it is a structured ASN.1 type value. Check the generated function prototype in the header file to determine how the value argument is to be passed for a given function.

The function result variable `stat` returns the status of the encode operation. Status code 0 (0) indicates the function was successful. Note that this return value differs from that of BER encode functions in that the encoded length of the message component is not returned – only an OK status indicating encoding was successful. A negative value indicates encoding failed. Return status values are defined in the "asn1type.h" include file. The error text and a stack trace can be displayed using the *rtxErrPrint* function.

Populating Generated Structure Variables for Encoding

See the section *Populating Generated Structure Variables* for Encoding for a discussion on how to populate variables for encoding. There is no difference in how it is done for BER versus how it is done for OER.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling a C OER encode function. This method must be used if C code generation was done.

Before an OER encode function can be called, the user must first initialize an encoding context block structure. The context block is initialized by calling the *rtInitContext* function.

The user then has the option to do stream-based or memory-based encoding. If stream-based is to be done, the user must call a *rtxStreamCreateWriter* function for the type of stream to which data will be written. For example, if the user wishes to write data to a file, the *rtxStreamFileCreateWriter* function would be called.

To do memory-based encoding, the *rtxInitContextBuffer* function would be called. This can be used to specify use of a static or dynamic memory buffer. Specification of a dynamic buffer is possible by setting the buffer address argument to null and the buffer size argument to zero.

An encode function can then be called to encode the message. If the return status indicates success (0), then the message will have been encoded in the given buffer or written to the given stream. OER encoding starts from the beginning of the buffer and proceeds from low memory to high memory until the message is complete. This differs from definite-length BER where encoding was done from back-to-front. Therefore, the buffer start address is where the encoded OER message begins. The length of the encoded message can be obtained by calling the *rtxCtxtGetMsgLen* run-time function. If dynamic encoding was specified (i.e., a buffer start address and length were not given), the *rtxCtxtGetMsgPtr* run-time function can be used to obtain the start address of the message. This routine will also return the length of the encoded message. If a memory stream was used, the message start address and length can be obtained by calling the *rtxStreamMemoryGetBuffer* function.

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

```
#include employee.h            /* include file generated by ASN1C */
```

```
main ()
{
   PersonnelRecord employee;
   OSCTXT     ctxt;
   OSOCTET*   msgptr;
   int        i, len, stat;
   const char* filename = "message.dat";

   /* Populate employee C structure */

   asn1Init_PersonnelRecord (&employee);
   employee.name.givenName = "SMITH";
   ...

   /* Initialize context */

   stat = rtInitContext (&ctxt);

   if (stat != 0) {
      printf (“rtInitContext failed (check license)\n“);
      rtxErrPrint (&ctxt);
      return stat;
   }

   /* Create memory output stream */

   stat = rtxStreamMemoryCreateWriter (&ctxt, 0, 0);

   if (stat < 0) {
      printf ("Create memory output stream failed\n");
      rtxErrPrint (&ctxt);
      rtFreeContext (&ctxt);
      return stat;
   }

   /* Encode */

   stat = OEREnc_PersonnelRecord (&ctxt, &employee);

   msgptr = rtxStreamMemoryGetBuffer (&ctxt, &len);

   if (trace) {
      printf ("Hex dump of encoded record:\n");
      rtxHexDump (msgptr, len);
   }
   ...
}
```

In general, static buffers should be used for encoding messages where possible as they offer a substantial performance benefit over dynamic buffer allocation. The problem with static buffers, however, is that you are required to estimate in advance the approximate size of the messages you will be encoding. There is no built-in formula to do this, the size of an ASN.1 message can vary widely based on data types and other factors.

The use of streams is a good alternative for large messages as the entire encoded message does not need to fit into memory.

Generated OER Decode Functions

OER encode/decode functions are generated when the *-oer* switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C OER decode function is generated. This function will parse the data contents from an OER-encoded ASN.1 message and populate a variable of the corresponding type with the data.

Generated C Function Format and Calling Parameters

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

```
OERDec_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = OERDec_<name> (OSCTXT* pctxt, <name>* pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

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 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 `pvalue` argument is a pointer to a variable to hold the decoded result. This variable is of the type generated from the ASN.1 production. 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 `stat` returns the status of the decode operation. Status code 0 (0) indicates the function was successful. A negative value indicates decoding failed. Return status values are defined in the "asn1ErrCodes.h" and "rtxErrCodes.h" header files. The reason text and a stack trace can be displayed using the rtxErrPrint function described later in this document.

Procedure for Calling C Decode Functions

This section describes the step-by-step procedure for calling a C OER decode function.

Unlike BER, the user must know the ASN.1 type of an OER message before it can be decoded. This is because the type cannot be determined at run-time. There are no embedded tag values to reference to determine the type of message received.

The following are the basic steps in calling a compiler-generated decode function:

1. Prepare a context variable for decoding

2. Initialize the data structure to receive the decoded data

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 an OER decode function can be called, the user must first initialize a context block structure. The context block is initialized by calling the *rtInitContext* function.

The user then has the option to do stream-based or memory-based decoding. If stream-based is to be done, the user must call a *rtxStreamCreateReader* function for the type of stream from which data will be read. For example, if the user wishes to read data from a file, the *rtxStreamFileCreateReader* function would be called.

To do memory-based decoding, the *rtxInitContextBuffer* function would be called. The message to be decoded must reside in memory. The arguments to this function would then specify the message buffer in which the data to be decoded exists.

The variable that is to receive the decoded data must then be initialized. This can be done by either initializing the variable to zero using memset, or by calling the ASN1C generated initialization function.

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 ASN.1 type variable. The decode function may automatically allocate dynamic memory to hold variable length variables 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 either the context is freed or explicitly when the *rtxMemFree* or *rtxMemReset* function is called.

The final step of the procedure is to free the context block. This must be done regardless of whether the block is static (declared on the stack and initialized using *rtInitContext*), or dynamic (created using *rtNewContext*). The function to free the context is *rtFreeContext*.

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

```
#include employee.h           /* include file generated by ASN1C */

main ()
{
   OSOCTET* pMsgBuf;
   int len, stat;
```

```
      OSCTXT ctxt;
      PersonnelRecord employee;
      const char* filename = "message.dat";

      /* step 1: initialize context */

      stat = rtInitContext (&ctxt);

      if (stat != 0) {
         printf (“rtInitContext failed (check license)\n“);
         rtErrPrint (&ctxt);
         return stat;
      }

      /* step 2: read input file into a memory buffer */

      stat = rtxFileReadBinary (&ctxt, filename, &pMsgBuf, &len);
      if (0 == stat) {
         stat = rtxInitContextBuffer (&ctxt, pMsgBuf, len);
      }
      if (0 != stat) {
         rtxErrPrint (&ctxt);
         rtFreeContext (&ctxt);
         return stat;
      }

      /* step 3: initialize the data variable */

      asn1Init_PersonnelRecord (&employee);

      /* step 4: call the decode function */

      stat = OERDec_PersonnelRecord (&ctxt, &employee);

      if (stat == 0)
      {
         process received data..
      }
      else {
         /* error processing... */
         rtxErrPrint (&ctxt);
      }

      /* step 4: free the context */

      rtFreeContext (&ctxt);
   }
```

An input stream can be used instead of a memory buffer as the data source by replacing the *rtxFileReadBinary* code block above with one of the *rtxStreamCreateReader* functions to set up a file, memory, or socket stream as input.

Generated Medical Device Encoding Rules (MDER) Functions

Note

MDER is available only as a professional compiler option. The encoding and decoding functions are built on top of ASN1C's streaming functions and will not work with the typical buffer-based implementations seen in BER or PER. When introduced in version 6.4, no C ++ implementation for MDER was available.

The Medical Device Encoding Rules (MDER) are described in IEEE standard 11073-20601-2008, Annex F. This standard describes a simplified encoding to be used across medical devices. ASN1C can generate encoders and decoders for specifications based on the IEEE standard, which uses a strict subset of ASN.1.

To generate encoding and decoding functions, use the `-mder` switch on the command-line or select the appropriate option in the GUI. The following sections describe the generated encoding and decoding functions. Descriptions of the MDER run time functions may be found in our *C MDER Runtime Library Reference Manual*.

Generated MDER Encode Functions

Generated C Function Format and Calling Parameters

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

```
mderEnc_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function follows:

```
stat = mderEnc_<name> (OSCTXT* pctxt,
                       <name>* pvalue);
```

In this definition, `<name>` denotes the prefixed production 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 program. The variable should be initialized using the `mderInitContext` run-time library function (see the *C MDER Runtime Library Reference Manual* for a complete description of this function).

The `pvalue` argument holds a pointer to the data to be encoded and is of the type generated from the ASN.1 production.

The function result variable `stat` returns an error status code if encoding fails. Return status values are defined in the `asn1type.h` include file.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling a C MDER encode function.

Before any encode function can be called; the user must first initialize an encoding context. 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 `mderInitContext` function as follows:

```
OSCTXT ctxt;

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

After initializing the context and populating a variable of the structure to be encoded, an encode function can be called to encode the message.

A program fragment that could be used to encode a simple release request PDU is as follows:

```
#include "IEEE-11073-20601-ASN1.h"            /* include file generated by ASN1C */

int main (void) {
   ApduType  data;           /* typedef generated by ASN1C */
   OSCTXT    ctxt;
   OSOCTET*  msgptr;
   int       stat;

   /* Step 1: Initialize the context and set the buffer pointer */

   stat = mderInitContext (&ctxt);
   if (stat != 0) {
      /* initialization failed, could be a license problem */
      rtxErrPrint (&ctxt);
      return stat;
   }

   /* Step 2: Populate the structure to be encoded */

   asn1Init_ApduType (&data);
   data.t = T_ApduType_rlrq;
   data.u.rlrq = rtxMemAllocTypeZ (&ctxt, RlrqApdu);
```

```
      data.u.rlrq->reason = normal;


      /* Step 3: Call the generated encode function */

      stat = mderEnc_ApduType (&ctxt, &data);

      /* Step 4: Check the return status. */

      if (stat < 0) {
         rtxErrPrint (&ctxt);
         return stat;
      }

      stat = rtFreeContext (&ctxt);
      if (stat != 0) {
         printf ("Error freeing context!\n");
         return stat;
      }

      return 0;
   }
```

Encoding a Series of Messages Using the C Encode Functions

Encoding a series of messages in MDER is very similar to encoding a series of messages in any other set of encoding rules. Performance can be improved by calling `rtxMemReset` to avoid allocating new memory for dynamic message structures, as in the code below:

```
   /* initialize context, et c. */

   for ( ; ; ) {
      /* initialize / populate message structure to be encoded */

      /* call mderEnc_<messageType> (...); */

      /* call rtxMemReset when finished encoding: */
      rtxMemReset (pctxt);
   }
```

More details may be found in the sample programs included in the ASN1C software development kit.

Generated MDER Decode Functions

Generated C Function Format and Calling Parameters

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

```
   mderDec_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
   status = mderDec_<name> (OSCTXT* pctxt, <name> *pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

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 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 `mderInitContext` run-time function before use.

The `pvalue` argument is a pointer to a variable of the generated type that will receive the decoded data.

The function result variable `status` returns the status of the decode operation. The return status will be greater than or equal to zero if decoding is successful or negative if an error occurs. Return status values are defined in the "asn1type.h" include file.

Procedure for Calling C Decode Functions

This section describes the step-by-step procedure for calling a C MDER decode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

Before any decode function can be called; the user must first 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 as follows:

```
   OSCTXT ctxt;
   int stat;

   stat = mderInitContext (&ctxt);
   if (stat != 0) {
      rtxErrPrint (&ctxt);
      rtFreeContext (&ctxt);
      return stat;
   }
```

The next step is to create a stream reader that will read from the given source. In our example, we read from a file, but it is also possible to read data from a socket or other source as well.

A decode function can then be called to decode the message. If the return status indicates success, the C variable that was passed as an argument will contain the decoded message contents. Note that the decoder may have allocated dynamic memory and stored pointers to objects in the C structure. After processing on the C structure is complete, the run-time library function `rtxMemFree` should be called to free the allocated memory.

A program fragment that could be used to decode a simple PDU type follows:

```
#include ISO-11073-20601-ASN1.h          /* include file generated by ASN1C */

int main (void)
{
   ApduType data;
   OSCTXT      ctxt;
   const char* filename = "message.dat";
   int         stat;

   /* Step 1: Initialize a context variable for decoding */

   stat = mderInitContext (&ctxt);
   if (stat != 0) {
      /* initialization failed, could be a license problem */
      rtxErrPrint (&ctxt);
      rtFreeContext (&ctxt);
      return stat;
   }

   /* Step 2: Initialize a stream reader */

   stat = rtxStreamFileCreateReader (&ctxt, filename);
   if (0 != stat) {
      rtxErrPrint (&ctxt);
      rtFreeContext (&ctxt);
      return stat;
   }

   /* Step 3: decode */
   stat = MDERDec_ApduType (&ctxt, &data);
   if (stat != 0) {
      printf ("decode of data failed\n");
      rtxErrPrint (&ctxt);
      rtFreeContext (&ctxt);
      return -1;
   }
}
```

Decoding a Series of Messages Using the C Decode Functions

Decoding a series of messages is very similar to the method used for other encoding rules. MDER, however, is simpler. Users need not concern themselves with idiosyncrasies like byte alignment or length markers.

Short pseudo-code is shown below. As in the encoding example, `rtxMemReset` is used at the end of the loop to avoid allocating new memory for dynamic data structures. This helps to improve performance.

```
/* initialize context, et c. */
for ( ; ; ) {
   /* initialize data structure */

   /* call mderDec_<name> function */

   /* perform operations on decoded structure */

   /* reset memory: */
   stat = rtxMemReset (&ctxt);
}
```

More details may be found in the sample programs included in the ASN1C software development kit.

Two-Phase Messaging

MDER is specified using ITU-T X.208, which uses a two-phase method for encoding and decoding `ANY DEFINED BY` types. These types are used to allow flexibility in message transmission by specifying a "hole" in a message to be filled in by a type specified by an object identifier.

Let us assume for sake of example that we wish to transmit a message that has a generic header and a payload defined by some object identifier. We must first encode the payload and attach it to the message. Then we can encode the whole message and transmit it. Because this takes two steps, we call this two-phase encoding.

The inverse holds for decoding. The whole message is first decoded. The payload may then be identified and decoded according to its specific type.

This section describes how to perform two-phase encoding and decoding using the MDER Run Time Library. Examples are also provided in the distribution.

Two-phase Encoding

We demonstrate an example of two-phase encoding using communication with an electrocardiogram. This code is based on the ASN.1 specification provided in IEEE 11073-20601-2008 and submitted drafts for an application of this type.

Two-phase encoding requires the use of multiple encoding contexts to encode both the header and payload. They must therefore be declared with the rest of the pertinent variables:

```
ApduType   apdu;
AarqApdu   aarq;
DataProto* pDataProto;
PhdAssociationInformation phdAssocInfo;

OSCTXT     ctxt, ctxt2;
OSOCTET*   msgptr;
int        len;
const char* filename = "message.dat";
```

We first populate and encode the payload; specific details will vary depending on the application:

```
/* Populate and encode PhdAssociationInformation */
OSCRTLMEMSET (&phdAssocInfo, 0, sizeof(phdAssocInfo));

phdAssocInfo.protocol_version.numbits = 32;
phdAssocInfo.protocol_version.data[0] = 0x40;

phdAssocInfo.encoding_rules.numbits = 16;
rtxSetBit (phdAssocInfo.encoding_rules.data, 16, EncodingRules_mder);

phdAssocInfo.nomenclature_version.numbits = 32;
rtxSetBit (phdAssocInfo.nomenclature_version.data, 32,
           NomenclatureVersion_nom_version1);

phdAssocInfo.functional_units.numbits = 32;

phdAssocInfo.system_type.numbits = 32;
rtxSetBit (phdAssocInfo.system_type.data, 32, SystemType_sys_type_agent);
{
static const OSOCTET sysId[] = {
   0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38
} ;
phdAssocInfo.system_id.numocts = (OSUINT32) 8;
phdAssocInfo.system_id.data = (OSOCTET*) sysId;
}
phdAssocInfo.dev_config_id = extended_config_start;

phdAssocInfo.data_req_mode_capab.data_req_mode_flags.numbits = 16;
phdAssocInfo.data_req_mode_capab.data_req_init_agent_count = 1;
phdAssocInfo.data_req_mode_capab.data_req_init_manager_count = 0;

/* Create memory output stream */
stat = rtxStreamMemoryCreateWriter (&ctxt, 0, 0);
if (stat < 0) {
   printf ("Create memory output stream failed\n");
   rtxErrPrint (&ctxt);
   rtFreeContext (&ctxt);
   return stat;
}

/* Encode */
stat = MDEREnc_PhdAssociationInformation (&ctxt, &phdAssocInfo);

msgptr = rtxStreamMemoryGetBuffer (&ctxt, &len);
```

In brief, the data structures used for the payload are initialized to zero using the `OSCRTLMEMSET` macro, which here acts just like the C runtime library `memset` function. The data used for populating this example are taken from a draft specification.

After filling in the necessary fields, the `rtxStreamMemoryCreateWriter` function is used to create a memory stream for encoding the payload. More information on this function can be found in the *C/C++ Common Run Time Library* manual. In case of failure, errors are trapped and reported.

Finally, the proper `MDEREnc` function is called to encode the data. A pointer to the message content is retrieved using the `rtxStreamMemoryGetBuffer` function. This pointer is used later to fill in the contents of the payload.

After encoding the payload, the rest of the message content must be populated and encoded:

```
/* Initialize 2nd context structure */
stat = rtInitContext (&ctxt2);

/* Populate apdu with test data */
OSCRTLMEMSET (&aarq, 0, sizeof(AarqApdu));
aarq.assoc_version.numbits = 32;
rtxSetBit (aarq.assoc_version.data, 32, AssociationVersion_assoc_version1);

pDataProto = rtxMemAllocType (&ctxt2, DataProto);
pDataProto->data_proto_id = data_proto_id_20601;
pDataProto->data_proto_info.numocts = len;
pDataProto->data_proto_info.data = msgptr;

rtxDListAppend (&ctxt2, &aarq.data_proto_list, pDataProto);
```

The `msgptr` variable is used here to fill in the contents of the `data_proto_info` structure. The rest of the contents are initialized and then encoded:

```
apdu.t = T_ApduType_aarq;
apdu.u.aarq = &aarq;

/* Create memory output stream */
stat = rtxStreamMemoryCreateWriter (&ctxt2, 0, 0);
if (stat < 0) {
   printf ("Create memory output stream failed\n");
   rtxErrPrint (&ctxt);
   rtFreeContext (&ctxt);
}

/* Encode */
stat = MDEREnc_ApduType (&ctxt2, &apdu);
```

Again, a memory stream writer is used here for encoding, but other options exist to write to a file or a socket.

Two-phase Decoding

Two-phase decoding is the reverse operation of two-phase encoding. In this scenario, a message is received and decoded. The header and payload are contained in the message, and the payload type and content must be decoded after the message is received.

This example shows how to decode the message encoded in the previous section. As before, some setup is required to perform the decode:

```
ApduType data;
OSCTXT      ctxt;
OSBOOL      trace = TRUE, verbose = FALSE;
const char* filename = "message.dat";
int         i, stat;

/* Initialize context structure */
stat = mderInitContext (&ctxt);
```

```
if (stat != 0) {
   rtxErrPrint (&ctxt);
   return stat;
}
rtxSetDiag (&ctxt, verbose);
```

In this case, the content is read from an input file, so a file stream is created using `rtxStreamFile-CreateReader`. Thereafter, the PDU data type is initialized using its initialization function and the message is decoded with the generated `MDERDec` function:

```
/* Create file input stream */
stat = rtxStreamFileCreateReader (&ctxt, filename);
if (0 != stat) {
   rtxErrPrint (&ctxt);
   rtFreeContext (&ctxt);
   return stat;
}

asn1Init_ApduType (&data);

/* Call compiler generated decode function */
stat = MDERDec_ApduType (&ctxt, &data);
if (stat != 0) {
   printf ("decode of ApduType failed\n");
   rtxErrPrint (&ctxt);
   rtFreeContext (&ctxt);
   return -1;
}

   rtxStreamClose (&ctxt);
```

The second phase of the decode can now proceed. Because the open type data can appear in a list, a `while` loop is used to cycle through the data:

```
/* Decode APDU open type data */
if (data.t == T_ApduType_aarq) {
   OSRTDListNode* pnode = data.u.aarq->data_proto_list.head;
   while (0 != pnode) {
      PhdAssociationInformation phdAssocInfo;
      DataProto* pDataProto = (DataProto*) pnode->data;

      /* Create memory input stream */
      stat = rtxStreamMemoryCreateReader
         (&ctxt, (OSOCTET*)pDataProto->data_proto_info.data,
          pDataProto->data_proto_info.numocts);
```

Note here that the `rtxStreamMemoryCreateReader` function is used to stream data from the previously decoded message. It points to the octets held inside of the open type. After initializing the stream reader, the data can be decoded into the appropriate structure using the corresponding `MDERDec` function:

```
      /* Decode PhdAssociationInformation */
      asn1Init_PhdAssociationInformation (&phdAssocInfo);

      stat = MDERDec_PhdAssociationInformation (&ctxt, &phdAssocInfo);
      if (stat != 0) {
         printf ("decode of ApduType failed\n");
         rtxErrPrint (&ctxt);
         rtFreeContext (&ctxt);
         return -1;
      }

      rtxStreamClose (&ctxt);

      pnode = pnode->next;
   }
}
```

Generated XML Functions

Generated XER Encode Functions

XER stands for “XML Encoding Rules”, a form of XML specified in the X.693 standard for use with ASN.1.

NOTE: XER is maintained as a legacy XML format for ASN.1. New applications should consider using XML as described in the next section instead of XER. XML is more closely aligned with W3C standard XML and XML schema.

XER C encode functions are generated when the *-xer* switch is specified on the command line. For each ASN.1 prouction defined in the ASN.1 source file, a C XER encode function is generated. This function will convert a populated C variable of the given type into an XER encoded ASN.1 message (i.e. an XML document).

If C++ code generation is specified, a control class is generated that contains an *Encode* method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the *msgData* component of the class.

Generated C Function Format and Calling Parameters

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

```
asn1XE_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each encode function is as follows:

```
status = asn1XE_<name> (OSCTXT* pctxt, <name>[*] value,
                        const char* elemName,
                        const char* attributes);
```

In this definition, <name> denotes the prefixed production 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 ASN.1 production. The object is passed by value if it is a primitive ASN.1 data type such as BOOLEAN, INTEGER, ENUMERATED, etc.. It is passed using a pointer reference if it is a structured ASN.1 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* and *attributes* arguments are used to pass the XML element name and attributes respectively. The *elemName* argument is the name that will be included in the <name> </name> brackets used to delimit an XML item. There are three distinct ways this argument can be specified:

1. If it contains standard text, this text will be used as the element name.
2. If it is null, a default element name will be applied. Default names for all of the built-in ASN.1 types are defined in the 2002 X.680 standard. For example, <BOOLEAN> is the default element name for the BOOLEAN built-in type.
3. If the name is empty (i.e. equal to “”, a zero-length string – not to be confused with null), then no element name is applied to the encoded data.

The function result variable `stat` returns the status of the encode operation. Status code 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 Method Format and Calling Parameters

Generated encode functions are invoked through the class interface by calling the base class Encode method. The calling sequence for this method is as follows:

```
   stat = <object>.Encode ();
```

In this definition, <object> is an object of the class generated for the given production. The function result variable stat returns the status value from the XER encode function. This status value will be zero if encoding was successful or a negative error status value if encoding fails. Return status values are defined in the *rtxErrCodes.h* include file.

The user must call the encode buffer class methods *getMsgPtr* and *getMsgLen* to obtain the starting address and length of the encoded message component.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling C XER encode functions. This procedure is similar to that for the other encoding methods except that some of the functions used are specific to XER.

Before an XER encode function can be called, the user must first initialize an encoding context block structure. The context block is initialized by calling *rtInitContext* to initialize a context block structure. The user then must call the `xerSetEncBufPtr` function to specify a message buffer to receive the encoded message. Specification of a dynamic message buffer is possible by setting the buffer address argument to null and the buffer size argument to zero. This function also also allows specification of whether standard XER or canonical XER encoding should be done.

An encode function can then be called to encode the message. If the return status indicates success (0), then the message will have been encoded in the given buffer. XER encoding starts from the beginning of the buffer and proceeds from low memory to high memory until the message is complete. This differs from BER where encoding was done from back-to-front. Therefore, the buffer start address is where the encoded XER message begins. The length of the encoded message can be obtained by calling the *xerGetMsgLen* run-time function. If dynamic encoding was specified (i.e., a buffer start address and length were not given), the run-time routine *xerGetMsgPtr* can be used to obtain the start address of the message. This routine will also return the length of the encoded message.

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

```
#include employee.h              /* include file generated by ASN1C */

main ()
{
   OSOCTET msgbuf[4096];
   int     msglen, stat;
   OSCTXT  ctxt;
   OSBOOL  cxer = FALSE; /* canonical XER flag */
   OSBOOL  aligned = TRUE;
   Employee employee; /* typedef generated by ASN1C */

   /* Initialize context and set encode buffer pointer */

   if (rtInitContext (&ctxt) != 0) {
      rtxErrPrint (&ctxt);
      return -1;
   }
   xerSetEncBufPtr (&ctxt, msgbuf, sizeof(msgbuf), cxer);

   /* Populate variable with data to be encoded */
   employee.name.givenName = “John”;
   ...

   /* Encode data */
   stat = asn1XE_Employee (&ctxt, &employee, 0, 0);

   if (stat) == 0) {
      msglen = xerGetMsgLen (&ctxt);
      ...
   }
   else
      error processing...
}

rtFreeContext (&ctxt); /* release the context pointer */
```

After encoding is complete, msgbuf contains the XML textual representation of the data. By default, a UTF-8 encoding is used. For the ASCII character set, this results in a buffer containing normal textual data. Therefore, the contents of the buffer are represented as a normal text string and can be displayed using the C *printf* run-time function or any other function capable of displaying text.

Procedure for Using the C++ Control Class Encode Method

The procedure to encode a message using the C++ class interface is as follows:

1. Instantiate an ASN.1 XER encode buffer object (*ASN1XEREncodeBuffer*) to describe the buffer into which the message will be encoded. Constructors are available that allow a static message buffer to be specified and/or canonical encoding to be turned on (canonical encoding removes all encoding options from the final message to produce a single encoded representation of the data). The default constructor specifies use of a dynamic encode buffer and canonical encoding set to off.

2. Instantiate an ASN1T_<type> object and populate it with data to be encoded.

3. Instantiate an ASN1C_<type> object to associate the message buffer with the data to be encoded.

4. Invoke the ASN1C_<type> object Encode method.

5. Check the return status. The return value is a status value indicating whether encoding was successful or not. Zero indicates success. If encoding failed, the status value will be a negative number. The encode buffer method *printErrorInfo* can be invoked to get a textual explanation and stack trace of where the error occurred.

6. If encoding was successful, get the start-of-message pointer and message length. The start-of-message pointer is obtained by calling the *getMsgPtr* method of the encode buffer object. If static encoding was specified (i.e., a message buffer address and size were specified to the XER Encode Buffer class constructor), the start-of-message pointer is the buffer start address. The message length is obtained by calling the *getMsgLen* method of the encode buffer object.

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

```
#include employee.h             // include file generated by ASN1C

main ()
{
   const OSOCTET* msgptr;
   OSOCTET msgbuf[1024];
   int     msglen, stat;
   OSBOOL  canonical = FALSE;

   // step 1: instantiate an instance of the XER encode
   // buffer class. This example specifies a static
   // message buffer..

   ASN1XEREncodeBuffer encodeBuffer (msgbuf,
                                  sizeof(msgbuf),
                                     canonical);

   // step 2: populate msgData with data to be encoded

   ASN1T_PersonnelRecord msgData;
   msgData.name.givenName = "SMITH";
   ...

   // step 3: instantiate an instance of the ASN1C_<ProdName>
   // class to associate the encode buffer and message data..

   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // steps 4 and 5: encode and check return status

   if ((stat = employee.Encode ()) == 0)
   {
      printf ("encoded XML message:\n");
      printf ((const char*)msgbuf);
      printf ("\n");

      // step 6: get start-of-message pointer and message length.
      // start-of-message pointer is start of msgbuf
      // call getMsgLen to get message length..

      msgptr = encodeBuffer.getMsgPtr (); // will return &msgbuf
      len = encodeBuffer.getMsgLen ();
   }
   else
   {
      printf ("Encoding failed\n");
      encodeBuffer.printErrorInfo ();
      exit (0);
   }

   // msgptr and len now describe fully encoded message

   ...
```

Generated XER Decode Functions

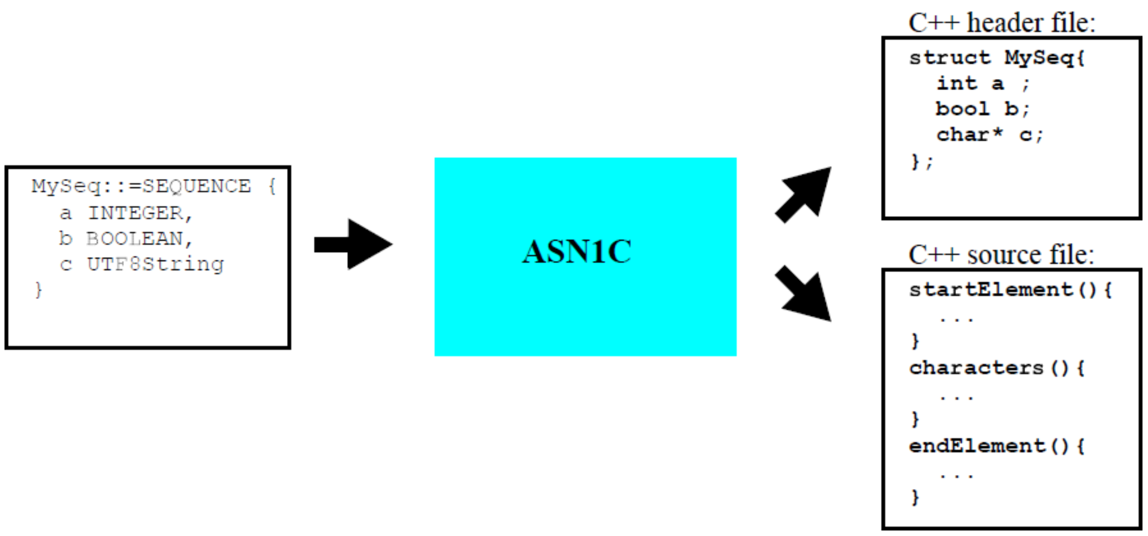
NOTE: XER is maintained as a legacy XML format for ASN.1. New applications should consider using XML as described in the next section instead of XER. XML is more closely aligned with W3C standard XML and XML schema.

The code generated to decode XML messages is different than that of the other encoding rules. This is because off-theshelf XML parser software is used 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. ASN1C 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://expat.sourceforge.net). This is a lightweight, open-source parser that was implemented in C. The C++ SAX interface was added by adapting the headers of the Apache XERCES C++ XML Parser (http://xml.apache.org) to work with the underlying C code. These headers were used to build a common C++ SAX interface across different vendor's SAX interfaces (unlike Java, these interfaces are not all the same). The ASN1C XER SAX C and C++ libraries come with the EXPAT parser as the default parser and also include plug-in interfaces that allow the code to work with the Microsoft XML parser (MSXML), The GNOME libxml2 parser, and the XERCES XML parser. Interfacing to other parsers only requires building an abstraction layer to map the common interface to the vendor's interface.

A diagram showing the components used in the XML decode process is as follows:

Step 1: Generate code

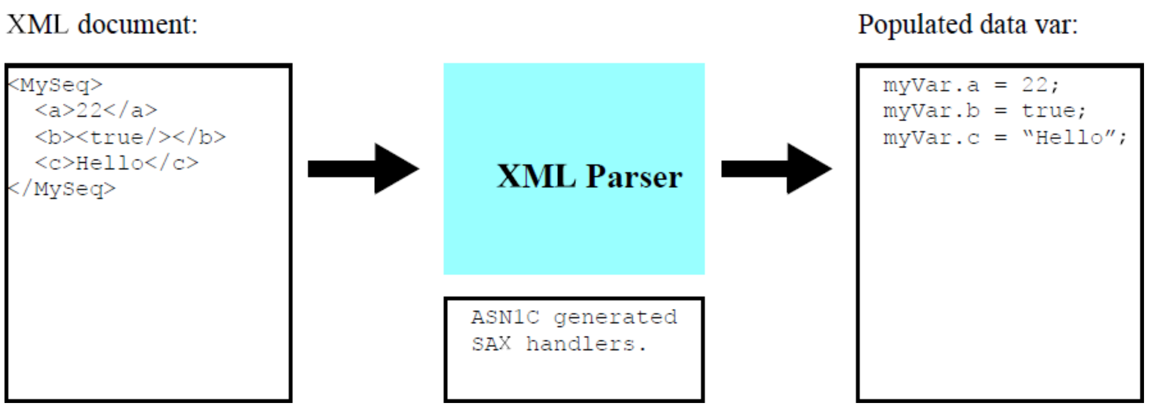


```
MySeq::=SEQUENCE {
  a INTEGER,
  b BOOLEAN,
  c UTF8String
}
```

```
struct MySeq{
  int a ;
  bool b;
  char* c;
};
```

```
startElement(){
  ...
}
characters(){
  ...
}
endElement(){
  ...
}
```

Step 2: Build Application



```
<MySeq>
  <a>22</a>
  <b><true/></b>
  <c>Hello</c>
</MySeq>
```

```
myVar.a = 22;
myVar.b = true;
myVar.c = "Hello";
```

ASN1C generates code to implement the following methods 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 XER encoded data.

Procedure for Using the C Interface

The ASN1C compiler generates XER decode functions for C for constructed types in a specification. These can be invoked in the same manner as other decode functions. In this case, they install the generated SAX content handler functions and invoke the XML parser's *parse* function to parse a document. The procedure to call these generated functions is described below.

Generated C Function Format and Calling Parameters

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

```
asn1XD_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each decode function is as follows:

```
status = asn1XD_<name> (OSCTXT* pctxt, <name>* pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

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 *rtInitContext* run-time function before use.

C XER decoding is stream-oriented. To perform streaming operations, the context pointer pctxt must also be initialized as a stream by using the *rtxStreamInit* run-time library function (see the *C/C++ Common Run-Time Library Reference Manual* for a description of the run-time stream C functions).

The `pvalue` argument is a pointer to a variable to hold the decoded result. This variable is of the type generated from the ASN.1 production. 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 `stat` 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

This section describes the step-by-step procedure for calling a C XER decode function. This method must be used if C code generation was done. This method cannot be used as an alternative to using the control class interface if C++ code generation was done. Use the C++ procedure instead.

There are four steps to calling a compiler-generated 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 XER 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 *rtInitContext* function. Also, the context must be initialized for streaming operations by calling the *rtxStreamInit* function:

```
OSCTXT ctxt; // context variable

if (rtInitContext (&ctxt) != 0) {
   /* initialization failed, could be a license problem */
   printf (“context initialization failed (check license)\n”);
   return –1;
}

rtxStreamInit (&ctxt)); // Initialize stream
```

The next step is to create a stream object within the context. This object is an abstraction of the output device to which the data is to be encoded and is initialized by calling one of the following functions:

- *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 (see the *C/C++ Common Run-Time Library Reference Manual* for a full description of these functions).

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 ASN.1 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 free the context is *rtFreeContext*.

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

```
#include employee.h             /* include file generated by ASN1C */

main ()
{
   int     stat;
   OSCTXT  ctxt;
   PersonnelRecord employee;
   ASN1ConstCharPtr filename = "message.xml";

   /* Step 1: Init context structure */

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

   rtxStreamInit (&ctxt);

   /* Step 2: Open a stream */

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

   /* Step 3: decode the record */

   stat = asn1XD_PersonnelRecord (&ctxt, &employee);
   if (stat == 0) {
      if (trace) {
         printf ("Decode of PersonnelRecord was successful\n");
         printf ("Decoded record:\n");
         asn1Print_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);
      rtFreeContext (&ctxt);

      return 0;
   }
```

Procedure for Using the C++ Interface

SAX handler methods are added to the C++ control class generated for each ASN.1 production.

The procedure to invoke the generated decode method is similar to that for the other encoding rules. It is as follows:

1. Instantiate an ASN.1 XER decode buffer object (*ASN1XERDecodeBuffer*) to describe the message to be decoded. Constructors exist that allow an XML file or memory buffer to be specified as an input source.

2. Instantiate an ASN1T_*TypeName* object to hold the decoded message data.

3. Instantiate an ASN1C_*TypeName* object to decode the message. This class associates the message buffer object with the object that is to receive the decoded data. The results of the decode operation will be placed in the variable declared in step 2.

4. Invoke the ASN1C_*TypeName* object *Decode* method. This method initiates and invokes the XML parser's parse method to parse the document. This, in turn, invokes the generated SAX handler methods.

5. Release dynamic memory that was allocated by the decoder. All memory associated with the decode context is released when both the ASN1XERDecodeBuffer and ASN1C_*TypeName* objects go out of scope.

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

```
   int main (int argc, char* argv[])
   {
      const char* filename = "employee.xml";
      int stat;

      // steps 1, 2, and 3: instantiate an instance of the XER
      // decoding classes. This example specifies an XML file
      // as the message input source..

      ASN1T_PersonnelRecord employee;
      ASN1XERDecodeBuffer decodeBuffer (filename);
      ASN1C_PersonnelRecord employeeC (decodeBuffer, employee);

      // step 4: invoke the decode method

      stat = employeeC.Decode ();
      if (0 == stat) {
         employeeC.Print ("employee");
      }
      else
         decodeBuffer.printErrorInfo ();

      // step 5: dynamic memory is released when employeeC and
      // decode buffer objects go out of scope.

      return (stat);

   }
```

Procedure for Interfacing with Other C and C++ X ML Parser Libraries

As mentioned previously, the Expat XML Parser library is the default XML parser library implementation used for decoding XER messages. It is also possible to use the C++ SAX handlers generated by ASN1C with other XML parser library implementations. The XER Run-Time Library (ASN1XER) provides a common interface to other parsers via a common adapter interface layer. There is a special XML interface object file for each of the following supported XML parsers:

- rtXmlLibxml2IF - interface to LIBXML2;
- rtXmlXercesIF - interface to XERCES;
- rtXmlExpatIF - interface to EXPAT.
- rtXmlMicroIF - interface to custom, small footprint, microparser

The XER Run-Time Library is completely independent from the XML readers because the adapter layer within these libraries defines a common SAX API.

If an application is linked statically then the static variant of one of these interface objects (their names have suffix “_a”) should be linked cooperatively with the XML parser, ASN1XER and ASN1RT libraries.

If the application is linked dynamically (using dynamically-linked libraries (DLL) in Windows or shared objects (SO or SL in UNIX/Linux) then it is necessary to link the application with the dynamic variant of the interfaces (without suffix “_a”), dynamic version of the XML parser, ASN1XER and ASN1RT dynamic libraries.

Generated XML Encode Functions

XML C encode and decode functions are generated when the *-xml* switch is specified on the command line. These are similar to the XER encode functions described earlier. Like XER, this function allows data in a populated variable to be formatted into an XML document. Unlike the XER variant, this function will produce XML that adheres more closely to the Worldwide Web Consortium (W3C) XML conventions. In particular, the following differences exist:

- Lists of numbers, enumerated tokens, and named bits are expressed in space-separated list form instead of as individually wrapped elements or value lists.

  For example, the ASN.1 specification “A ::= SEQUENCE OF INTEGER” with value “{ 1 2 3 }” would produce the following encoding in XER:

  <A><INTEGER>1</INTEGER><INTEGER>2</INTEGER><INTEGER>3</
  INTEGER></A>

  in XML, it would be the following:

  <A>1 2 3</A>

- The values of the BOOLEAN data type are expressed as the lower case words “true” or “false” with no delimiters. In XER, the values are <TRUE/> and <FALSE/>.

- Enumerated token values are expressed as the identifiers themselves instead of as empty XML elements (i.e. elements wrapped in ‘< />’). For example, a value of the ASN.1 type “Colors ::= ENUMERATED { red, blue, green }” equal to “red” would simply be “<color>red</color>” instead of “<color><red/></color>”.

- The special REAL values <PLUS-INFINITY/> and <MINUS-INFINITY/> are represented as INF and -INF respectively.

- GeneralizedTime and UTCTime values are transformed into the XSD representation for dateTime (YYYY-MMDDTHH: MM:SS[.SSSS][(Z|(+|-)HH:MM)]) when encoded to XML. When an XML document is decoded, the time format is transformed into the ASN.1 format.

Also, if code is generated by compiling XML schema specifications, the generated XML will contain features defined in the schema which cannot be specified using plain ASN.1 such as attributes and namespaces. Note that it is possible to support these items if ASN.1 with Extended XER notation (E-XER) is used, but this is not supported by ASN1C. Its method of supporting these constructs is the direct compilation of XML schema files.

It is also important to note that the *-xsd* switch is complementary to the *-xml* switch when generating XML encoders and decoders. This is because the XML schema produced from the ASN.1 specification using the -xsd switch can be used to validate the XML messages generated using the XML encode functions. Similarly, an XML instance can be validated using the generated XML schema prior to decoding.

XML C encode functions are generated when the *-xml* switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C XML encode function is generated. In the case of XML schema, a C encode function is generated for each type and global element declaration. This function will convert a populated C variable of the given type into an XML encoded message (i.e. an XML document).

If C++ code generation is specified, a control class is generated that contains an *Encode* method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the *msgData* component of the class.

Generated C Function Format and Calling Parameters

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

```
[<namespace>]XmlEnc_[<prefix>]<prodName>
```

where `<namespace>` is an optional C namespace prefix, `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production. <namespace> is set using the ASN1C *-namespace* command-line argument. Note that this should not be confused with the notion of an XML namespace.

The calling sequence for each encode function is as follows:

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

In this definition, <ns> is short for <namespace> and <name> denotes the prefixed production 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 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 ASN.1 production. The object is passed by value if it is a primitive ASN.1 data type such as BOOLEAN, INTEGER, ENUMERATED, etc.. It is passed using a pointer reference if it is a structured ASN.1 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* and *nsPrefix* arguments are used to pass the XML element name and namespace prefix respectively. The two arguments are combined to form a qualified name (QName) of the form <nsPrefix:elemName>. If *elemName* is null or empty, then no element tag is added to the encoded content. If *nsPrefix* is null or empty, the element name is applied as a local name only without a prefix.

The function result variable `stat` returns the status of the encode operation. Status code 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.

In addition to the XML encode function generated for types, a different type of encode function is generated for *Protocol Data Units (PDU's).* These are types in an ASN.1 specification that are not referenced by any other types. In an XML schema specification, these are global elements that are not reference within any other types or global elements.

The format of the a PDU encode function is the same name format as above with the suffix *_PDU*. This function does not contain the *elemName* and *nsPrefix* arguments - these are built into the function as defined in the schema. For this reason, calling PDU functions is usually more convenient than calling the equivalent function for the referenced type.

Procedure for Calling C Encode Functions

This section describes the step-by-step procedure for calling C XML encode functions. This procedure is similar to that for the other encoding methods except that some of the functions used are specific to XML.

Before an XML encode function can be called, the user must first initialize an encoding context block structure. The context block is initialized by calling *rtXmlInitContext* to initialize a context block structure. The user then must call the *rtXmlSetEncBufPtr* function to specify a message buffer to receive the encoded message. Specification of a dynamic message buffer is possible by setting the buffer address argument to null and the buffer size argument to zero.

An encode function can then be called to encode the message. If the return status indicates success, then the message will have been encoded in the given buffer. XML encoding starts from the beginning of the buffer and proceeds from low memory to high memory until the message is complete. This differs from BER where encoding was done from back-to-front. Therefore, the buffer start address is where the encoded XML message begins. If a dynamic message buffer was used, the start address of the encoded message can be obtained by calling the *rtXmlEncGetMsgPtr* function. Since the encoded XML message is nothing more than a null-terminated string in a memory buffer, the standard C library function *strlen* can be used to obtain the length.

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

```
#include employee.h              /* include file generated by ASN1C */

int main (int argc, char** argv)
{
   PersonnelRecord employee;
   OSCTXT       ctxt;
   OSOCTET      msgbuf[4096];
   int          stat;

   /* Initialize context and set encode buffer pointer */
   stat = rtXmlInitContext (&ctxt);
   if (0 != stat) {
      printf ("context initialization failed\n");
```

```
         rtxErrPrint (&ctxt);
         return stat;
      }

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

      /* Populate variable with data to be encoded */
      employee.name.givenName = “John”;
      ...

      /* Encode data */
      stat = XmlEnc_PersonnelRecord_PDU (&ctxt, &employee);

      if (stat) == 0) {
         /* Note: message can be treated as a null-terminated string
            in memory */
         printf ("encoded XML message:\n");
         puts ((char*)msgbuf);
         printf ("\n");
         ...
      }
      else
         error processing...
   }

   rtFreeContext (&ctxt); /* release the context pointer */
```

Generated C++ Encode Method Format and Calling Parameters

When C++ code generation is specified using the -xml switch, the generated *EncodeTo* and *DecodeFrom* methods in the PDU control class are set up to encode complete XML documents including the start document header as well as namespace attributes in the main element tag.

Generated encode functions are invoked through the class interface by calling the base class *Encode* method. The calling sequence for this method is as follows:

```
stat = <object>.Encode ();
```

In this definition, <object> is an object of the class generated for the given production. The function result variable `stat` returns the status value from the XML encode function. This status value will be zero if encoding was successful or a negative error status value if encoding fails. Return status values are defined in the *rtxErrCodes.h* include file.

The user must call the encode buffer class methods *getMsgPtr* and *getMsgLen* to obtain the starting address and length of the encoded message component.

Procedure for Using the C++ Control Class Encode Method

The procedure to encode a message using the C++ class interface is as follows:

1. Instantiate an XML encode buffer object (*OSXMLEncodeBuffer*) to describe the buffer into which the message will be encoded. Constructors are available that allow a static message buffer to be specified. The default constructor specifies use of a dynamic encode buffer.

2. Instantiate an ASN1T_<type> object and populate it with data to be encoded.

3. Instantiate an ASN1C_<type> object to associate the message buffer with the data to be encoded.

4. Invoke the ASN1C_<type> object *Encode* or *EncodeTo* method.

5. Check the return status. The return value is a status value indicating whether encoding was successful or not. Zero indicates success. If encoding failed, the status value will be a negative number. The encode buffer method *printErrorInfo* can be invoked to get a textual explanation and stack trace of where the error occurred.

6. If encoding was successful, get the start-of-message pointer and message length. The start-of-message pointer is obtained by calling the *getMsgPtr* method of the encode buffer object. If static encoding was specified (i.e., a message buffer address and size were specified to the XML Encode Buffer class constructor), the start-of-message pointer is the buffer start address. The message length is obtained by calling the *getMsgLen* method of the encode buffer object.

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

```
#include employee.h            // include file generated by ASN1C

main ()
{
   OSOCTET msgbuf[4096];
   int     msglen, stat;

   // step 1: instantiate an instance of the XML encode
   // buffer class. This example specifies a static
   // message buffer..

   OSXMLEncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));

   // step 2: populate msgData with data to be encoded

   ASN1T_PersonnelRecord msgData;
   msgData.name.givenName = "SMITH";
   ...

   // step 3: instantiate an instance of the ASN1C_<ProdName>
   // class to associate the encode buffer and message data..

   ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

   // steps 4 and 5: encode and check return status

   if ((stat = employee.Encode ()) == 0)
   {
      printf ("encoded XML message:\n");
      printf ((const char*)msgbuf);
      printf (“\n”);

      // step 6: get start-of-message pointer and message length.
      // start-of-message pointer is start of msgbuf
      // call getMsgLen to get message length..

      msgptr = encodeBuffer.getMsgPtr (); // will return &msgbuf
      len = encodeBuffer.getMsgLen ();
   }
   else
   {
      printf ("Encoding failed\n");
      encodeBuffer.printErrorInfo ();
      exit (0);
   }

   // msgptr and len now describe fully encoded message

   ...
```

Generated XML Decode Functions

A major difference between generated XER decode functions and generated XML decode functions in ASN1C version 6.0 and later is that the XML functions no longer use the Simple API for XML (SAX) interface. Instead, the XML runtime now uses an XML pull-parser developed in-house for improved efficiency. The pull-parser also provides a similar interface to that of binary encoding rules such as BER or PER meaning easier integration with existing encoding rules. Finally, the pull-parser interface does not require integration with any 3rd-party XML parser software.

XML decode functions are generated when the *-xml* switch is specified on the command line. For each ASN.1 production defined in the ASN.1 source file, a C XML decode function is generated. This function will parse the data contents from an XML message of the corresponding ASN.1 or XML schema type and populate a variable of the C type with the data.

If C++ code generation is specified, a control class is generated that contains a *DecodeFrom* method that wraps this function. This function is invoked through the class interface to encode an ASN.1 message into the variable referenced in the *msgData* component of the class.

Generated C Function Format and Calling Parameters

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

```
[<namespace>]XmlDec_[<prefix>]<prodName>
```

where `<namespace>` is an optional C namespace prefix, `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production. <namespace> is set using the ASN1C *-namespace* command-line argument. Note that this should not be confused with the notion of an XML namespace.

The calling sequence for each decode function is as follows:

```
   status = <ns>XmlDec_<name> (OSCTXT* pctxt, <name>* pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

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 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 `pvalue` argument is a pointer to a variable to hold the decoded result. This variable is of the type generated from the ASN.1 production. 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 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

This section describes the step-by-step procedure for calling a C XML decode function. This method must be used if C code generation was done. This method can also be used as an alternative to using the control class interface if C++ code generation was done.

These are the steps involved calling a compiler-generated decode function:

1. Prepare a context variable for decoding
2. Open an input stream for the XML document to be decoded
3. Decode the initial tag value to figure out what type of message was received (optional).
4. Call the appropriate compiler-generated decode function to decode the message
5. 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 first initialize a context block structure. The context block structure is initialized by calling the *rtXmlInitContext* function.

An input stream is then opened using one of the *rtxStream* functions. If the data is to be read from a file, the *rtxStreamFileCreateReader* function can use used. Similar functions exist for opening a memory or socket-based stream.

If the user knows the type of XML message that is to be processed, he can directly call the decode function at this point. If not, the user may call the *rtXmlpMatchStartTag* method to match the initial tag in the message with a known start tag. The user can continue to do this until a match is found with one of the expected message types. Note that the *rtXmlpMarkLastEvent* function must be called if the tag is to be reparsed to attempt another match operation.

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 ASN.1 type variable. The decode function may automatically allocate dynamic memory to hold variable length variables 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 free the context block. The function to free the context is *rtFreeContext*.

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

```
#include employee.h            /* include file generated by ASN1C */

main ()
{
   PersonnelRecord data;
   OSCTXT ctxt;
   OSBOOL trace = TRUE, verbose = FALSE;
   const char* filename = "message.xml";
   int i, stat;

   .. logic to read message into msgbuf ..

   /* This example uses a static context block */

   /* step 1: initialize the context block */

   stat = rtXmlInitContext (&ctxt);

   if (stat != 0) {
      rtxErrPrint (&ctxt);
      return stat;
   }

   /* step 2: open an input stream */

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

   /* step 3: attempt to match the start tag to a known value */

   if (0 == rtXmlpMatchStartTag (&ctxt, OSUTF8(“Employee”)) {
      /* Note that it is necessary to mark the last event active in
         the pull-parser to that it can be parsed again in the PDU
         decode function. */
      rtXmlpMarkLastEventActive (&ctxt);
```

```
        /* step 4: call the decode function */

        stat = XmlDec_PersonnelRecord_PDU (&ctxt, &data);

        if (stat == 0)
        {
           process received data..
        }
        else {
           /* error processing... */
           rtxErrPrint (&ctxt);
        }
     }
     /* can check for other possible tag matches here.. */

     /* step 5: free the context */

     rtFreeContext (&ctxt);
  }
```

Generated C++ Decode Method Format and Calling Parameters

Generated decode functions are invoked through the class interface by calling the base class *Decode* or *DecodeFrom* methods. The calling sequence for this method is as follows:

```
status = <object>.Decode ();
```

In this definition, <object> is an object of the class generated for the given production.

An *OSXMLDecodeBuffer* object must be passed to the <object> constructor prior to decoding. This is where the message stream containing the XML document to be decoded is specified. Several constructors are available allowing the specification of XML input from a file, memory buffer, or another stream.

The function result variable `status` returns the status of the decode operation. The return status will be zero if decoding is successful or a negative value if an error occurs. Return status values are documented in the *C/C++ Common Functions Reference Manual* and in the *rtxErrCodes.h* include file.

Procedure for Using the C++ Control Class Decode Method

The following are the steps are involved in decoding an XML message using the generated C++ class:

1. Instantiate an XML decode buffer object (*OSXMLDecodeBuffer*) to describe the message to be decoded. There are several choices of constructors that can be used including one that takes

the name of a file which contains the XML message, one the allows a memory buffer to be specified, and one that allows an existing stream object to be used.

2. Instantiate an ASN1T_<ProdName> object to hold the decoded message data.

3. Instantiate an ASN1C_<ProdName> object to decode the message. This class associates the message buffer object with the object that is to receive the decoded data. The results of the decode operation will be placed in the variable declared in step 2.

4. Invoke the ASN1C_<ProdName> object *Decode* or *DecodeFrom* method.

5. Check the return status. The return value is a status value indicating whether decoding was successful or not. Zero indicates success. If decoding failed, the status value will be a negative number. The decode buffer method *printErrorInfo* can be invoked to get a textual explanation and stack trace of where the error occurred.

6. Release dynamic memory that was allocated by the decoder. All memory associated with the decode context is released when both the *OSXMLDecodeBuffer* and *ASN1C_<ProdName>* objects go out of scope.

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

```
#include employee.h              // include file generated by ASN1C

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

   .. logic to read message into msgbuf ..

   // step 1: instantiate an XML decode buffer object

   OSXMLDecodeBuffer decodeBuffer (filename);

   // step 2: instantiate an ASN1T_<ProdName> object

   ASN1T_PersonnelRecord msgData;

   // step 3: instantiate an ASN1C_<ProdName> object

   ASN1C_PersonnelRecord employee (decodeBuffer, msgData);

   // step 4: decode the record

   stat = employee.Decode ();

   // step 5: check the return status

   if (stat == 0)
   {
      process received data..
   }
```

```
      else {
         // error processing..
         decodeBuffer.PrintErrorInfo ();
      }

      // step 6: free dynamic memory (will be done automatically
      // when both the decodeBuffer and employee objects go out
      // of scope)..

   }
```

Additional Generated Functions

Generated Initialization Functions

As of ASN1C version 6.0, initialization functions are automatically generated (in previous versions, it was necessary to use the *-genInit* option to force this action). If for some reason, a user does want initialization functions to be generated, the *-noInit* switch can be used to turn initialization function generation off.

The use of initialization functions are optional - a variable can be initialized by simply setting its contents to zero (for example, by using the C run-time *memset* function). The advantage of initialization function is that they provide smarter initialization which can lead to improved application performance. For example, it is not necessary to set a large byte array to zero prior to its receiving a populated value. The use of *memset* in this situation can result in degraded performance.

Generated initialization functions are written to the main *<module>.c* file. This file contains common constants, global variables, and functions that are generic to all type of encode/decode functions. If the *-cfile* command-line option is used, the functions are written to the specified .c or .cpp file along with all other generated functions. If *-maxcfiles* is specified, each generated initialization function is written to a separate .c file.

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

```
asn1Init_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

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

```
asn1Init_<name> (<name>* pvalue)
```

In this definition, <name> denotes the prefixed production name defined above.

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

Generated Memory Free Functions

The *-genFree* option causes functions to be generated that free dynamic memory allocated using the ASN1C run-time memory management functions and macros (*rtxMem*). By default, all memory held within a context is freed using the *rtxMemFree* run-time function. It is also possible to free an individual memory item using the *rtMemFreePtr* function. But it is not possible to free all memory held within a specific generated type container. For example, a SEQUENCE type could contain elements that require dynamic memory. These elements in turn can reference other types that require dynamic memory. The generated memory free functions make it possible to release all memory held within a variable of the type with a single call.

Generated memory free functions are written to the main *<module>.c* file. This file contains common constants, global variables, and functions that are generic to all type of encode/decode functions. If the *-cfile* command-line option is used, the functions are written to the specified .c or .cpp file along with all other generated functions. If *-maxcfiles* is specified, each generated function is written to a separate .c file.

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

```
asn1Free_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

The calling sequence for each generated memory free function is as follows:

```
asn1Free_<name> (OSCTXT* pctxt, <name>* pvalue)
```

In this definition, <name> denotes the prefixed production name defined above.

The *pctxt* argument is used to hold the context pointer that the memory to be freed was allocated with. 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 *pvalue* argument is used to pass a pointer to a variable of the item that contains the dynamic memory to be freed.

Generated Print Functions

The following options are available for generating code to print the contents of variables of generated types:

-print - This is the standard print option that causes print functions to be generated that output data to the standard output device (*stdout*).

-genPrtToStr - This option causes print functions to be generated that write their output to a memory buffer.

-genPrtToStrm - This option causes print functions to be generated that write their output to an output stream via a user-defined print callback function.

Print to Standard Output

The *-print* option causes functions to be generated that print the contents of variables of generated types to the standard output device. It is possible to specify the name of a .c or .cpp file as an argument to this option to specify the name of the file to which these functions will be written. This is an optional argument. If not specified, the functions are written to separate files for each module in the source file. The format of the name of each file is *<module>Print.c*. If an output filename is specified after the *–print* qualifier, all functions are written to that file.

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

```
asn1Print_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

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

```
asn1Print_<name> (const char* name, <name>* pvalue)
```

In this definition, <name> denotes the prefixed production 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 *asn1Print_Employee* (*"employee", &employee*) might be used).

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

If C++ code generation is specified, a *Print* method is added to the ASN1C control class for the type. This method takes only a name argument; the *pvalue* argument is obtained from the msgData reference contained within the class.

Print to String

The *-genPrtToStr* option causes functions to be generated that print the contents of variables of generated types to a given text buffer. This buffer can then be used to output the information to other mediums such as a file or window display.

It is possible to specify the name of a .c or .cpp file as an argument to this option to specify the name of the file to which these functions will be written. This is an optional argument. If not specified, the functions are written to separate files for each module in the source file. The format of the name of each file is *<module>PrtToStr.c*. If an output filename is specified after the *–genPrtToStr* qualifier, all functions are written to that file.

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

```
   asn1PrtToStr_<name> (const char* name, <name>* pvalue,
                        char* buffer, int bufSize)
```

The *name* and *pvalue* arguments are the same as they were in the *-print* case.

The *buffer* and *bufSize* arguments are used to describe the memory buffer the text is to be written into. These arguments specify a fixed-size buffer. If the generated text is larger than the given buffer size, as much text as possible is written to the buffer and a –1 status value is returned. If the buffer is large enough to hold the text output, all text is written to the buffer and a zero status is returned. If there is text already in the buffer, the function will append to this text, rather than overwrite it, starting at the first null character. So in this case there must be enough space in the buffer starting from the first null character to hold all of the generated text; otherwise, a status of -1 is returned. For this reason initializing a newly allocated buffer with zeroes before passing it to the function is a good idea.

For C++, two *toString* methods are generated in the control class that call the generated print-to-string function. With the first signature, in addition to the name argument, the method also takes a *buffer* and *bufSize* argument to describe the buffer to which the text is to be written. The second signature does not take a name argument; instead, the name of the item that the control class instance describes is defaulted.

Print to Stream

The *-genPrtToStrm* option causes functions to be generated that print the contents of variables of generated types to an output stream via a user-defined callback function. The advantage of these functions is that a user can register a callback function, and then the print stream is automatically directed to the callback function. This makes it easier to support print-to-file or print-to-window type of functionalities. It also make it possible to create an optimized print-to-string capability by maintaining a structure that keeps track of the current end-of-string position in the output buffer. The generated print-to-string functions always must use the strlen run-time function to find the end position, an operation that becomes compute intensive in large string buffers that are constantly appended to.

It is possible to specify the name of a .c or .cpp file as an argument to this option to specify the name of the file to which these functions will be written. This is an optional argument. If not specified, the functions are written to separate files for each module in the source file. The format of the name of each file is *<module>PrtToStrm.c*. If an output filename is specified after the *–genPrtToStrm* qualifier, all functions are written to that file.

Before calling generated print-to-stream functions, a callback function should be registered. Otherwise, a default callback function will be used that directs the print stream to the standard output device.

The callback function is declared as:

```
   void (*rtxPrintCallback)
      void* pPrntStrmInfo, const char* fmtspec, va_list arglist);
```

The first parameter is user-defined data which will be passed to each invocation of the callback function. This parameter can be used to pass print stream specific data, for example, a file pointer if the callback function is to output data to a file. The second and third parameters to the callback function constitute the data to be printed, in the form of format specification followed by list of arguments. A simple callback function for printing to file can be defined as follows:

```
void writeToFile (void* pPrntStrmInfo, const char* fmtspec, va_list arglist)
{
   FILE * fp = (FILE*) pPrntStrmInfo;
   vfprintf (fp, fmtspec, arglist);
}
```

Once the callback function is defined, it has to be registered with the runtime library. There are two types of registrations possible: 1. global, which applies to all print streams and, 2. context level, which applies to print streams associated with a particular context.

For registering a global callback use:

```
rtxSetGlobalPrintStream (rtxPrintCallback myCallback, void* pStrmInfo);
```

For registering a context level callback use:

```
rtxSetPrintStream (OSCTXT *pctxt, rtxPrintCallback myCallback, void* pStrmInfo);
```

Once the callback function is registered, the calling of each generated print-to-stream function will result in output being directed to the callback function.

The print to stream functions are declared as follows:

```
asn1PrtToStrm_<name> (OSCTXT *pctxt, const char* name, <name>* pvalue);
```

The *name* and *pvalue* arguments are the same as they were in the *-print* case.

The *pctxt* argument is used to specify an ASN1C context. If a valid context argument is passed and there is a context level callback registered, then that callback will be used. If there is no context level callback registered, or the *pctxt* argument is NULL, then the global callback will be used. If there is no global callback registered, the default callback will be used which writes the print output to stdout.

If C++ code generation is specified, *setPrintStream* and *toStream* methods are added to the ASN1C control class for the type. The *setPrintStream* method takes only *myCallback* and *pStrmInfo* arguments; the *pctxt* argument is obtained from the context pointer reference contained within the class. The *toStream* method takes only a *name* argument; the *pctxt* argument is derived from the context pointer reference within the class and the *pvalue* argument is obtained from the *msgData* reference contained within the class.

Print Format

The *-prtfmt* option can be used in conjunction with any of the *-genPrint* options documented above to alter the format of the printed data. There are two possible print formats: *details* and *bracetext*.

The *details* format prints a line-by-line display of every item in the generated structure. For example, the following is an excerpt from a details display:

```
Employee.name.givenName = 'John'
Employee.name.initial = 'P'
Employee.name.familyName = 'Smith'
Employee.number = 51
Employee.title = 'Director'
...
```

The alternative format - *bracetext* - provides a C-like structure printout. This is a more concise format that will omit optional fields that are not present in the decoded data. An example of this is as follows:

```
Employee {
   name {
      givenName = 'John'
      initial = 'P'
      familyName = 'Smith'
   }
   number = 51
   title = 'Director'
...
```

As of ASN1C version 6.0 and higher, bracetext is the default format used if -prtfmt is not specified on the commandline. Previous versions of ASN1C had details as the default setting.

Generated Compare Functions

The -*genCompare* option causes comparison functions to be generated. These functions can be used to compare the contents of two generated type variables.

If an output file is not specified with the *–genCompare* qualifier, the functions are written to separate .c files for each module in the source file. The format of the name of each file is *<module>Compare.c*. If an output filename is specified after the *–genCompare* qualifier, all functions are written to that file.

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

```
asn1Compare_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

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

```
OSBOOL asn1Compare_<name> (const char* name,
                           <name>* pvalue, <name>* pCmpValue,
                           char* errBuff, int errBufSize);
```

In this definition, <name> denotes the prefixed production name defined above.

The name argument is used to hold the top-level name of the variable being compared. It is typically set to the same name as the pvalue argument in quotes (for example, to compare employee records, a call to ‘asn1Compare_Employee (“employee”, &employee, etc.)’ might be used).

The pvalue argument is used to pass a pointer to a variable of the item to the first item to be compared. The *pCmpValue* argument is used to pass the second value. The two items are then compared field-by-field for equality.

The *errBuff* and *errBuffSize* arguments are used to describe a text buffer into which information on what fields the comparison failed on is written. These arguments specify a fixed-size buffer – if the generated text is larger than the given buffer size, the text is terminated. These arguments may be omitted by passing null (0) values if you only care to know if the structures are different and not concerned with the details.

The return value of the function is a Boolean value that is true if the variables are equal and false if they are not.

Generated Copy Functions

The -*genCopy* option causes copy functions to be generated. These functions can be used to copy the content of one generated type variable to another.

If no output file is specified with the *–genCopy* qualifier, the functions are written to separate .c/.cpp files for each module in the source file. The format of the name of each file is *<module>Copy.c/.cpp* where <*module*> would be replaced with the name of the ASN.1 module. If an output filename is specified after the *–genCopy* qualifier, all functions are written to that file.

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

```
asn1Copy_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the `<typePrefix>` element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

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

```
void asn1Copy_<name> (OSCTXT* pctxt,
                      <name>* pSrcValue,
                      <name>* pDstValue);
```

In this definition, <name> denotes the prefixed production name defined above.

The pointer to the context structure (*pctxt*) provides a storage area for the function to store all variables that have been copied

The *pSrcValue* argument is used to pass a pointer to a variable to be copied. The *pDstValue* argument is used to pass the pointer to the destination value. The source value is then copied to the destination value field-by-field. Memory will be allocated for dynamic fields using calls to the *rtxMemAlloc* function.

If C++ is used (-cpp option is specified) and PDU generation is not disabled (<noPDU> config option is not used) then the control class ASN1C_<name> additionally will contain:

- A copy constructor that can be used to create an exact copy of the class instance.

  The calling sequence is as follows:

  ```
  ASN1C_<name> (ASN1C_<name>& orginal);
  ```

  For example:

  ```
  ASN1C_PersonnelRecord (ASN1C_PersonnelRecord& original);
  ```

- A *getCopy* method that creates a copy of the *ASN1T_<name>* variable:

  ```
  ASN1T_<name>& getCopy (ASN1T_<name>* pDstData = 0);
  ```

  For example:

  ```
  ASN1T_PersonnelRecord& getCopy (ASN1T_PersonnelRecord* pDstData = 0);
  ```

  The *pDstData* argument is used to pass the pointer to a destination variable where the value will be copied. It may be null, in this case the new ASN1T_<name> variable will be allocated via a call to the *rtxMemAlloc* function.

- A *newCopy* method that will create a new, dynamically allocated copy of the referenced *ASN1T_data* member variable.

- An assignment operator. This is used to copy one instance of a control class to another one:

  ```
  inline ASN1C_<name>& operator= (ASN1C_<name>& srcData)
  {
     srcData.getCopy (&msgData);
     return *this;
  }
  ```

  For example:

  ```
  inline ASN1C_PersonnelRecord& operator=
     (ASN1C_PersonnelRecord& srcData)
  {
     srcData.getCopy (&msgData);
     return *this;
  }
  ```

Finally, the class declaration might look as follows:

```
class EXTERN ASN1C_PersonnelRecord :
```

```
  public ASN1CType
{
protected:
   ASN1T_PersonnelRecord& msgData;
public:
   ASN1C_PersonnelRecord (
      ASN1MessageBuffer& msgBuf, ASN1T_PersonnelRecord& data);

   ASN1C_PersonnelRecord (ASN1C_PersonnelRecord& original);

   ...

   ASN1T_PersonnelRecord& getCopy (ASN1T_PersonnelRecord*
                                   pDstData = 0);

   ASN1T_PersonnelRecord* newCopy ();

   inline ASN1C_PersonnelRecord&
      operator= (ASN1C_PersonnelRecord& srcData)
   {
      srcData.getCopy (&msgData);
      return *this;
   }
} ;
```

The generated *ASN1T<name>* structure will also contain an additional copy constructor if C++ is used and PDU generation is not disabled. A destructor is also generated if the type contains dynamic memory fields. This destructor will free the dynamic memory upon destruction of the type instance.

For example:

```
typedef struct EXTERN ASN1T_PersonnelRecord : public ASN1TPDU {
   ...
   ASN1T_PersonnelRecord () {
      m.uniPresent = 0;
      m.seqOfseqPresent = 0;
   }
   ASN1T_PersonnelRecord (ASN1C_PersonnelRecord& srcData);
   ~ASN1T_PersonnelRecord();
} ASN1T_PersonnelRecord;
```

This constructor is used to create an instance of the *ASN1T_<name>* type from an *ASN1C_<name>* control class object.

Memory Management of Copied Items

Prior to ASN1C version 5.6, dynamic memory of decoded or copied items would only be available as long as the control class instance and/or the message buffer object used to decode or copy the item remained in scope or was not deleted. This restriction no longer exists as long as the type being copied is a Protocol Data Unit (PDU). The copied item will now hold a reference to the context variable used to create the item and will increment the item's reference count. This reference is contained in the *ASN1TPDU* base class variable from which PDU data types are now derived. The dynamic memory within the item will persist until the item is deleted.

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. The main purpose is to provide a code template to users for writing code to populate variables. This is quite useful to users because generated data types can become very complex as the ASN.1 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 or .cpp file with a name of the following format:

```
<asn1ModuleName>Test.c
```

where *<asn1ModuleName>* is the name of the ASN.1 module that contains the type definitions. The format of the name of each generated test function is as follows:

```
asn1Test_[<prefix>]<prodName>
```

where `<prodName>` is the name of the ASN.1 production for which the function is being generated and `<prefix>` is an optional prefix that can be set via a configuration file setting. The configuration setting used to set the prefix is the <typePrefix> element. This element specifies a prefix that will be applied to all generated typedef names and function names for the production.

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 have been previously initialized using the *rtInitContext* run-time function.

The `pvalue` argument is a pointer to hold the populated data variable. This variable is of the type generated for the ASN.1 production. 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.

In the case of C++, a method is added to the generated control class for test code generation. The name of this method is **genTestInstance**. The prototype is as follows:

```
<typeName>* pvalue = <object>.genTestInstance();
```

where <typeName> is the ASN1T_<name> type name of the generated type and <object> is an instance of the ASN1C_<name> generated control class.

Generated DOM Test Functions

A new command-line option added in ASN1C version 6.0 is *-domtest*. This is similar to *-gentest* except that data for the test variables is not random, it is extracted from an XML Document Object Model (DOM) tree at run-time. In order to use this capability, it is necessary to have the *libxml2* (http://xmlsoft.org) XML parser installed on your system. Calls are then made to parse a given XML document and create a DOM tree. Data from the DOM tree will then be transfered to data varaibles of generated structures.

This has the same end result as decoding the XML documents using the XML decoder.

Generated Sample Programs

In addition to test functions, it is possible to generate writer and reader sample programs. These programs contain sample code to populate and encode an instance of ASN.1 data and then read and decode this data respectively. These programs are generated using the *-genwriter* and *-genreader* command-line switches.

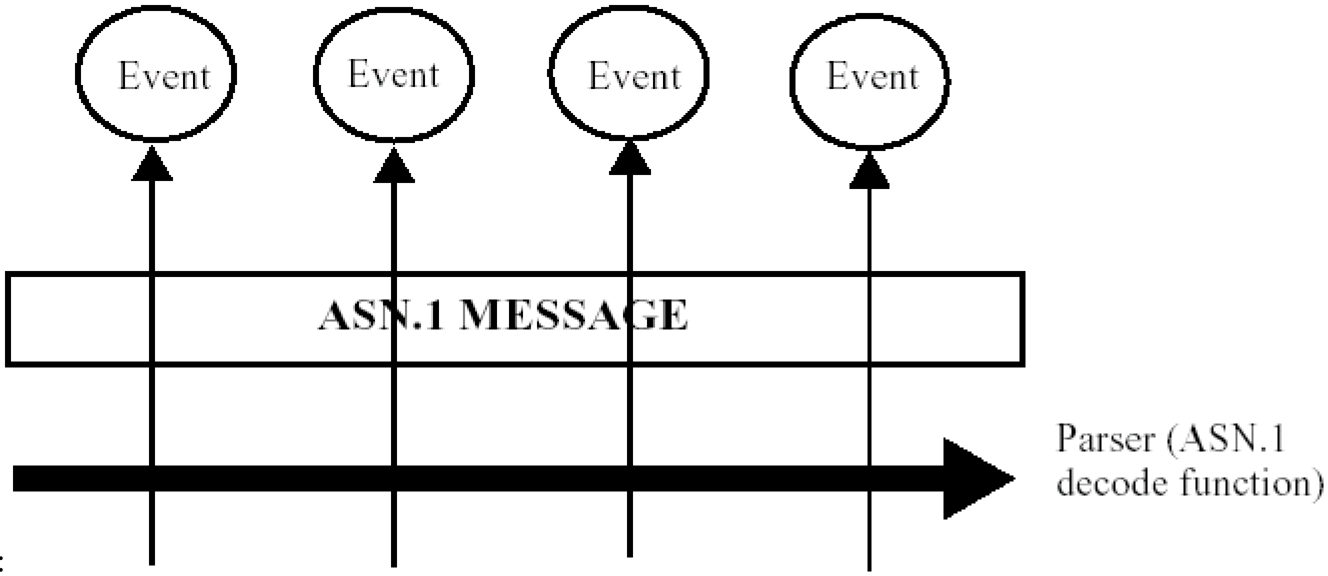
Event Handler Interface

The *–events* command line switch causes hooks for user-defined event handlers to be inserted into the generated decode functions. What these event handlers do is up to the user. They fire when key message-processing events or errors occur during the course of parsing an ASN.1 message. They are similar in functionality to the Simple API for XML (SAX) that was introduced to provide a simple interface for parsing XML messages.

The *-notypes* option can be used in conjunction with the *-events* option to generate pure parsing functions. In this case, no C types or encode or decode functions are generated for the productions within the given ASN.1 source file. Instead, only a set of parser functions are generated that invoke the event handler callback functions. This gives the user total control over what is done with the message data. Data that is not needed can be discarded and only the parts of the message needed for a given application need to be saved.

How it Works

Users of XML parsers are probably already quite familiar with the concepts of SAX. Significant events are defined that occur during the parsing of a message. As a parser works through a message, these events are 'fired' as they occur by invoking user defined callback functions. These callback functions are also known as event handler functions. A diagram illustrating this parsing process is as fol-



lows:

The events are defined to be significant actions that occur during the parsing process. We will define the following events that will be passed to the user when an ASN.1 message is parsed:

1. **startElement** – This event occurs when the parser moves into a new element. For example, if we have a SEQUENCE { a, b, c } construct (type names omitted), this event will fire when we begin parsing a, b, and c. The name of the element is passed to the event handling callback function.

2. **endElement –** This event occurs when the parser leaves a given element space. Using the example above, these would occur after the parsing of a, b, and c are complete. The name of the element is once again passed to the event handling callback function.

3. **contents methods –** A series of virtual methods are defined to pass all of the different types of primitive values that might be encountered when parsing a message (see the event handler class definition below for a complete list).

4. **error –** This event will be fired when a parsing error occurs. It will provide fault-tolerance to the parsing process as it will give the user the opportunity to fix or ignore errors on the fly to allow the parsing process to continue.

These events are defined as unimplemented virtual methods in two base classes: *Asn1NamedEventHandler* (the first 3 events) and *Asn1ErrorHandler* (the error event). These classes are defined in the *asn1CppEvtHndlr.h* header file.

The start and end element methods are invoked when an element is parsed within a constructed type. The start method is invoked as soon as the tag/length is parsed in a BER message or the preamble/length is parsed in a PER message. The end method is invoked after the contents of the field are processed. The signature of these methods is as follows:

```
virtual void startElement (const char* name, int index) = 0;
virtual void endElement (const char* name, int index) = 0;
```

The *name* argument is used pass the element name. The *index* argument is used for SEQUENCE OF/SET OF constructs only. It is used to pass the index of the item in the array. This argument is set to –1 for all other constructs.

There is one contents method for passing each of the ASN.1 data types. Some methods are used to handle several different types. For example, the *charValue* method is used for values of all of the different character string types (IA5String, NumericString, PrintableString, etc.) as well as for big integer values. Note that this method is overloaded. The second implementation is for 16-bit character strings. These strings are represented as an array of unsigned short integers in ASN1C. All of the other contents methods correspond to a single equivalent ASN.1 primitive type.

The error handler base class has a single virtual method that must be implemented. This is the error method and this has the following signature:

```
virtual int error (OSCTXT* pCtxt, ASN1CCB* pCCB, int stat) = 0;
```

In this definition, pCtxt is a pointer to the standard ASN.1 context block that should already be familiar. The pCCB structure is known as a “Context Control Block”. This can be thought of as a sub-context used to control the parsing of nested constructed types within a message. It is included as a parameter to the error method mainly to allow access to the “seqx” field. This is the sequence element index used when parsing a SEQUENCE construct. If parsing a particular element is to be retried, this item must be decremented within the error handler.

How to Use It

To define event handlers, two things must be done:

1. One or more new classes must be derived from the *Asn1NamedEventHandler* and/or the *Asn1ErrorHandler* base classes. All pure virtual methods must be implemented.

2. Objects of these classes must be created and registered prior to calling the generated decode method or function.

The best way to illustrate this procedure is through examples. We will first show a simple event handler application to provide a customized formatted printout of the fields in a PER message. Then we will show a simple error handler that will ignore unrecognized fields in a BER message.

Example 1: A Formatted Print Handler

The ASN1C evaluation and distribution kits include a sample program for doing a formatted print of parsed data. This code can be found in the *cpp/sample_per/eventHandler* directory. Parts of the code will be reproduced here for reference, but refer to this directory to see the full implementation.

The format for the printout will be simple. Each element name will be printed followed by an equal sign (=) and an open brace ({) and newline. The value will then be printed followed by another newline. Finally, a closing brace (}) followed by another newline will terminate the printing of the element. An indentation count will be maintained to allow for a properly indented printout.

A header file must first be created to hold our print handler class definition (or the definition could be added to an existing header file). This file will contain a class derived from the *Asn1NamedEventHandler* base class as follows:

```
class PrintHandler : public Asn1NamedEventHandler {
  protected:
     const char* mVarName;
     int mIndentSpaces;
  public:
     PrintHandler (const char* varName);
     ~PrintHandler ();
     void indent ();
     virtual void startElement (const char* name, int index = -1);
     virtual void endElement (const char* name, int index = -1);
     virtual void boolValue (OSBOOL value);

     ... other virtual contents method declarations
}
```

In this definition, we chose to add the *mVarName* and *mIndentSpaces* member variables to keep track of these items. The user is free to add any type of member variables he or she wants. The only firm requirement in defining this derived class is the implementation of the virtual methods.

We implement these virtual methods as follows:

In *startElement*, we print the name, equal sign, and opening brace:

```
void PrintHandler::startElement (const char* name, int index)
{
   indent();
   printf (“%s = {\n”, name);
   mIndentLevel++;
}
```

In this simplified implementation, we simply indent (this is another private method within the class) and print out the name, equal sign, and opening brace. We then increment the indent level. Note that this is a highly simplified form. We don’t even bother to check if the index argument is greater than or equal to zero. This would determine if a ‘[x]’ should be appended to the element name. In the sample program that is included with the compiler distribution, the implementation is complete.

In *endElement*, we simply terminate our brace block as follows:

```
void PrintHandler::endElement (const char* name, int index)
{
   mIndentLevel--;
   indent();
   printf (“}\n”);
}
```

Next, we need to create an object of our derived class and register it prior to invoking the decode method. In the *reader.cpp* program, the following lines do this:

```
// Create and register an event handler object

PrintHandler* pHandler = new PrintHandler ("employee");
decodeBuffer.addEventHandler (pHandler);
```

The *addEventHandler* method defined in the *Asn1MessageBuffer* base class is the mechanism used to do this. Note that event handler objects can be stacked. Several can be registered before invoking the decode function. When this is done, the entire list of event handler objects is iterated through and the appropriate event handling callback function invoked whenever a defined event is encountered.

The implementation is now complete. The program can now be compiled and run. When this is done, the resulting output is as follows:

```
employee = {
   name = {
      givenName = {
         "John"
      }
      initial = {
         "P"
      }
      familyName = {
         "Smith"
      }
   }
   ...
```

This can certainly be improved. For one thing it can be changed to print primitive values out in a “name = value” format (i.e., without the braces). But this should provide the general idea of how it is done.

Example 2: An Error Handler

Despite the addition of things like extensibility and version brackets, ASN.1 implementations get out-of-sync. For situations such as this, the user needs some way to intervene in the parsing process to set things straight. This is faulttolerance – the ability to recover from certain types of errors.

The error handler interface is provided for this purpose. The concept is simple. Instead of throwing an exception and immediately terminating the parsing process, a user defined callback function is first invoked to allow the user to check the error. If the user can fix the error, all he or she needs to do is apply the appropriate patch and return a status of 0. The parser will be none the wiser. It will continue on thinking everything is fine.

This interface is probably best suited for recovering from errors in BER or DER instead of PER. The reason is the TLV format of BER makes it relatively easy to skip an element and continue on. It is much more difficult to find these boundaries in PER.

Our example can be found in the *cpp/sample_ber/errorHandler* subdirectory. In this example, we have purposely added a bogus element to one of the constructs within an encoded employee record. The error handler will be invoked when this element is encountered. Our recovery action will simply be to print out a warning message, skip the element, and continue.

As before, the first step is to create a class derived from the *Asn1ErrorHandler* base class. This class is as follows:

```
class MyErrorHandler : public Asn1ErrorHandler {
  public:

     // The error handler callback method. This is the method
     // that the user must override to provide customized
     // error handling..

     virtual int error (OSCTXT* pCtxt, ASN1CCB* pCCB, int stat);
} ;
```

Simple enough. All we are doing is providing an implementation of the error method.

Implementing the error method requires some knowledge of the run-time internals. In most cases, it will be necessary to somehow alter the decoding buffer pointer so that the same field isn't looked at again. If this isn't done, an infinite loop can occur as the parser encounters the same error condition over and over again. The run-time functions *xd_NextElement* or *xd_OpenType* might be useful in the endeavor as they provide a way to skip the current element and move on to the next item.

Our sample handler corrects the error in which an unknown element is encountered within a SET construct. This will cause the error status ASN_E_NOTINSET to be generated. When the error handler sees this status, it prints information on the error that was encountered to the console, skips to the next element, and then returns an 0 status that allows the decoder to continue. If some other error occurred (i.e., status was not equal to ASN_E_NOTINSET), then the original status is passed out which forces the termination of the decoding process.

The full text of the handler is as follows:

```
int MyErrorHandler::error (OSCTXT* pCtxt, ASN1CCB* pCCB, int stat)
{

   // This handler is set up to look explicitly for ASN_E_NOTINSET
   // errors because we know the SET might contain some bogus elements..

   if (stat == ASN_E_NOTINSET) {
      // Print information on the error that was encountered

      printf ("decode error detected:\n");
      rtErrPrint (pCtxt);
      printf ("\n");

      // Skip element

      xd_NextElement (pCtxt);

      // Return an OK status to indicate parsing can continue

      return 0;
   }

   else return stat; // pass existing status back out
}
```

Now we need to register the handler. Unlike event handlers, only a single error handler can be registered. The method to do this in the message buffer class is *setErrorHandler*. The following two lines of code in the reader program register the handler:

```
MyErrorHandler errorHandler;

decodeBuffer.setErrorHandler (&errorHandler);
```

The error handlers can be as complicated as you need them to be. You can use them in conjunction with event handlers in order to figure out where you are within a message in order to look for a specific error at a specific place. Or you can be very generic and try to continue no matter what.

Example 3: A Pure Parser to Convert PER-encoded Data to XML

A pure-parser is created by using the *-notypes* option along with the *-events* option. In this case, no backing data types to hold decoded data are generated. Instead, parsing functions are generated that store the data internally within local variables inside the parsing functions. This data is dispatched to the callback functions and immeditely disposed of upon return from the function. It is up to the user to decide inside the callback handler what they want to keep and they must make copies at that time. The result is a very fast and low-memory consuming parser that is ideal for parsing messages in which only select parts of the messages are of interest.

Another use case for pure-parser functions is validation. These functions can be used to determine if a PER message is valid without going through the high overhead operation of decoding. They can be used on the front-end of an application to reject invalid messages before processing of the messages is done. In some cases, this can result in significantly increased performance.

An example of a pure-parser can be found in the *cpp/sample_per/per2xmlEH* directory. This program uses a pure-parser to convert PER-encoded data into XML. The steps in creating an event handler are the same as in Example 1 above. An implementation of the *Asn1NamedEventHandler* interface must be created. This is done in the *xmlHandler.h* and *xmlHandler.cpp* files. A detailed discussion of this code will not be provided here. What it does in a nutshell is adds the angle brackets (< >) around the element names in the *startElement* and *endElement* callbacks. The data callbacks simply output a textual representation of the data as they do in the print handler case.

The only difference in *reader.cpp* from the other examples is that:

1. There is no declaration of an employee variable to hold decoded data because no type for this variable was generated, and

2. The *Parse* method is invoked instead of the *Decode* method. This is the generated method definition for a pureparser. If one examines the generated class definitions, they will see that no *Encode* or *Decode* methods were generated.

Compiling and running this program will show the encoded PER message written to stdout as an XML message. The resulting message is also saved in the *message.xml* file.

IMPORT/EXPORT of Types

ASN1C allows productions to be shared between different modules through the ASN.1 IMPORT/EXPORT mechanism. The compiler parses but ignores the EXPORTS declaration within a module. As far as it is concerned, any type defined within a module is available for import by another module.

When ASN1C sees an IMPORT statement, it first checks its list of loaded modules to see if the module has already been loaded into memory. If not, it will attempt to find and parse another source file containing the module. The logic for locating the source file is as follows:

1. The configuration file (if specified) is checked for a <sourceFile> element containing the name of the source file for the module. Note that the <oid> configuration item can be used to distinguish modules that have the same names but different object identifiers.

2. If this element is not present, the compiler looks for a file with the name <ModuleName>.asn where module name is the name of the module specified in the IMPORT statement.

In both cases, the –I command line option can be used to tell the compiler where to look for the files.

The other way of specifying multiple modules is to include them all within a single ASN.1 source file. It is possible to have an ASN.1 source file containing multiple module definitions in which modules IMPORT definitions from other modules. An example of this would be the following:

```
ModuleA DEFINITIONS ::= BEGIN
   IMPORTS B From ModuleB;

    A ::= B

END

ModuleB DEFINITIONS ::= BEGIN

   B ::= INTEGER

END
```

This entire fragment of code would be present in a single ASN.1 source file.

ROSE and SNMP Macro Support

The ASN1C compiler has a special processing mode that contains extensions to handle items in the older 1990 version of ASN.1 (i.e. the now deprecated X.208 and X.209 standards). This mode is activated by using the -asnstd x208 command-line option.

Although the X.208 and X.209 standards are no longer supported by the ITU-T, they are still in use today. This version of ASN1C contains logic to parse some common MACRO definitions that are still in widespread use despite the fact that MACRO syntax was retired with this version of the standard. The types of MACRO definitions that are supported are ROSE OPERATION and ERROR and SNMP OBJECT-TYPE.

ROSE OPERATION and ERROR

ROSE stands for "Remote Operations Service Element" and defines a request/response transaction protocol in which requests to a conforming entity must be answered with the result or errors defined in operation definitions. Variations of this are used in a number of protocols in use today including CSTA and TCAP.

The definition of the ROSE OPERATION MACRO that is built into the ASN1C is as follows:

```
OPERATION MACRO ::=
BEGIN
   TYPE NOTATION        ::= Parameter Result Errors LinkedOperations
   VALUE NOTATION       ::= value (VALUE INTEGER)
   Parameter            ::= ArgKeyword NamedType | empty
   ArgKeyword           ::= "ARGUMENT" | "PARAMETER"
   Result               ::= "RESULT" ResultType | empty
   Errors               ::= "ERRORS" "{"ErrorNames"}" | empty
   LinkedOperations     ::= "LINKED" "{"LinkedOperationNames"}" | empty
   ResultType           ::= NamedType | empty
   ErrorNames           ::= ErrorList | empty
   ErrorList            ::= Error | ErrorList "," Error
   Error                ::= value(ERROR)  -- shall reference an error value
                            | type        -- shall reference an error type
                                          -- if no value is specified
   LinkedOperationNames ::= OperationList | empty
   OperationList        ::= Operation | OperationList "," Operation
   Operation            ::= value(OPERATION) -- shall reference an op value
                            | type           -- shall reference an op type
                                             -- if no value is specified
   NamedType            ::= identifier type | type

END
```

This MACRO does not need to be defined in the ASN.1 specification to be parsed. In fact, any attempt to redefine this MACRO will be ignored. Its definition is hard-coded into the compiler.

The compiler uses this definition to parse types and values out of OPERATION definitions. An example of an OPERATION definition is as follows:

```
login OPERATION
ARGUMENT SEQUENCE { username IA5String, password IA5String }
RESULT SEQUENCE { ticket OCTET STRING, welcomeMessage IA5String }
ERRORS { authenticationFailure, insufficientResources }
::= 1
```

In this case, there are two embedded types (an ARGUMENT type and a RESULT type) and an integer value (1) that identifies the OPERATION. There are also error definitions.

The ASN1C compiler generates two types of items for the OPERATION:

1. It extracts the type definitions from within the OPERATION definitions and generates equivalent C/C++ structures and encoders/decoders, and

2. It generates value constants for the value associated with the OPERATION (i.e., the value to the right of the '::=' in the definition).

The compiler does not generate any structures or code related to the OPERATION itself (for example, code to encode the body and header in a single step). The reason is because of the multi-layered nature of the protocol. It is assumed that the user of such a protocol would be most interested in doing the processing in multiple stages, hence no single function or structure is generated.

Therefore, to encode the login example the user would do the following:

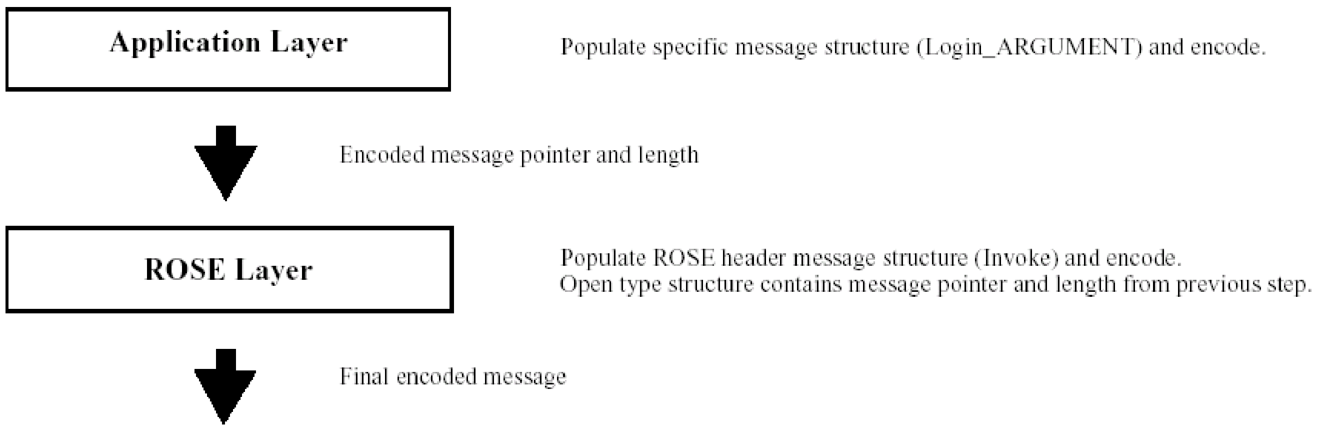
1. At the application layer, the Login_ARGUMENT structure would be populated with the username and password to be encoded.

2. The encode function for Login_ARGUMENT would be called and the resulting message pointer and length would be passed down to the next layer (the ROSE layer).

3. At the ROSE layer, the Invoke structure would be populated with the OPERATION value, invoke identifier, and other header parameters. The parameter.numocts value would be populated with the length of the message passed in from step 2. The parameter.data field would be populated with the message pointer passed in from step 2.

4. The encode function for Invoke would be called resulting in a fully encoded ROSE Invoke message ready for transfer across the communications link.

The following is a picture showing this process:


Application Layer
Populate specific message structure (Login_ARGUMENT) and encode.
Encoded message pointer and length
ROSE Layer
Populate ROSE header message structure (Invoke) and encode.
Open type structure contains message pointer and length from previous step.
Final encoded message

On the decode side, the process would be reversed with the message flowing up the stack:

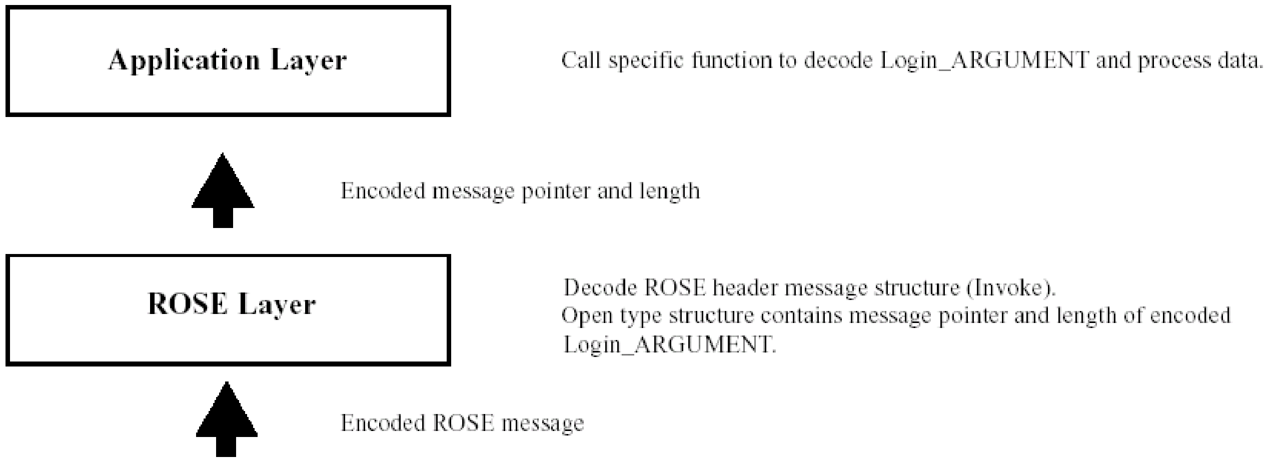
1. At the ROSE layer, the header would be decoded producing information on the OPERATION type (based on the MACRO definition) and message type (Invoke, Result, etc..). The invoke identifier would also be available for use in session management. In our example, we would know at this point that we got a login invoke request.

2. Based on the information from step 1, the ROSE layer would know that the Open Type field contains a pointer and length to an encoded Login_ARGUMENT component. It would then route this information to the appropriate processor within the Application Layer for handling this type of message.

3. The Application Layer would call the specific decoder associated with the Login_ARGUMENT. It would then have available to it the username/password the user is logging in with. It could then do whatever applicationspecific processing is required with this information (database lookup, etc.).

4. Finally, the Application Layer would begin the encoding process again in order to send back a Result or Error message to the Login Request.

A picture showing this is as follows:


Application Layer
Call specific function to decode Login_ARGUMENT and process data.
Encoded message pointer and length
ROSE Layer
Decode ROSE header message structure (Invoke).
Open type structure contains message pointer and length of encoded Login_ARGUMENT.
Encoded ROSE message

The login OPERATION also contains references to ERROR definitions. These are defined using a separate MACRO that is built into the compiler. The definition of this MACRO is as follows:

```
ERROR MACRO ::=
BEGIN
   TYPE NOTATION      ::= Parameter

   VALUE NOTATION     ::= value (VALUE INTEGER)

   Parameter          ::= "PARAMETER" NamedType | empty

   NamedType          ::= identifier type | type

END
```

In this definition, an error is assigned an identifying number as well as on optional parameter type to hold parameters associated with the error. An example of a reference to this MACRO for the `authenticationFailure` error in the login operation defined earlier would be as follows:

```
applicationError ERROR
PARAMETER SEQUENCE {
   errorText IA5String
}     }
::= 1
```

The ASN1C compiler will generate a type definition for the error parameter and a value constant for the error value. The format of the name of the type generated will be "<name>_PARAMETER" where <name> is the ERROR name (applicationError in this case) with the first letter set to uppercase. The name of the value will simply be the ERROR name.

SNMP OBJECT-TYPE

The SNMP OBJECT-TYPE MACRO is one of several MACROs used in Management Information Base (MIB) definitions. It is the only MACRO of interest to ASN1C because it is the one that specifies the object identifiers and data that are contained in the MIB.

The version of the MACRO currently supported by this version of ASN1C can be found in the SMI Version 2 RFC (RFC 2578). The compiler generates code for two of the items specified in this MACRO definition:

1. The ASN.1 type that is specified using the SYNTAX command, and

2. The assigned OBJECT IDENTIFIER value

For an example of the generated code, we can look at the following definition from the UDP MIB:

```
udpInDatagrams OBJECT-TYPE
   SYNTAX      Counter32
   MAX-ACCESS  read-only
   STATUS      current
   DESCRIPTION
      "The total number of UDP datagrams delivered to UDP users."
      ::= { udp 1 }
```

In this case, a type definition is generated for the SYNTAX element and an Object Identifier value is generated for the entire item. The name used for the type definition is "<name>_SYNTAX" where <name> would be replaced with the OBJECT-TYPE name (i.e., udpInDatagrams). The name used for the Object Identifier value constant is the OBJECTTYPE name. So for the above definitions, the following two C items would be generated:

```
typedef Counter32 udpInDatagrams_SYNTAX;

ASN1OBJID udpInDatagrams = {
   8,
   { 1, 3, 6, 1, 2, 1, 7, 1 }
} ;
```

Appendix A. Runtime Status Codes

This appendix describes status code messages returned by the ASN1C C/C++ runtime libraries. When deploying applications linked against optimized runtime libraries, ASN1C by default does not include a stacktrace; instead only an error code is provided. These codes are described more fully in this appendix.

The descriptions are derived from the contents of `rtxsrc/rtxErrCodes.h` and `rtsrc/asn1ErrCodes.h`. Users may always look at these two files or the documentation generated from them for a fully updated list of error messages and their descriptions.

ASN1C Error Messages

The following table describes error messages that ASN1C may report during the course of code generation, not during runtime. These include syntax errors, import warnings, type resolution failures, and others.

Users should note that there are several classes of status messages in this list: errors (`ASN-E` messages), warnings (`ASN-W` messages), and informational notices (`ASN-I` messages).

Error Code	**Error Description**
ASN-E-NOTYPE	No type was defined for the referenced element in a SEQUENCE or SET.
ASN-E-UNDEFTYPE	The type referenced was not defined within the context of this module.
ASN-E-NOTAG	The object must be tagged in this context. This usually occurs when context-specific tags are required to disambiguate elements in a SEQUENCE or SET.
ASN-W-DUPLICATE	The referenced type or value was previously defined.
ASN-W-DUPLTAG	The referenced tag was previously defined in a CHOICE or SET; this happens when an contextual tag is provided more than once.
ASN-E-UNRECTYP	The type described is not recognized by the compiler.
ASN-E-MULTDEF	A choice tag has multiple definitions.
ASN-E-UNDEFVAL	The referenced value is not defined or cannot be found.
ASN-E-INVTYPNAM	Invalid type name. This is a parsing failure; all type names must begin with an uppercase letter.
ASN-E-UNDEFTAG	The referenced type must be tagged in this context.
ASN-E-UNKNOWN	Undocumented error occurred in routine. A status value is provided with this error message to help locate the cause of the failure.

Error Code	**Error Description**
ASN-I-NOCASE	A case statement for the named object was not generated.
ASN-E-IMPFILOPN	The compiler was unable to open the named import file.
ASN-E-IMPFILPAR	The compiler was unable to parse the named imported module.
ASN-E-IMPNOTFOU	The named type was not found in the import module as specified in the IMPORT statement.
ASN-E-INVCNSTRNT	Invalid constraint specification.
ASN-W-INVOBJNAM	Invalid object name. The object name must begin with a lowercase letter.
ASN-E-SETTOOBIG	Set contains more than 32 elements.
ASN-E-DUPLCASE	This tag was used in a previous switch case statement.
ASN-E-AMBIGUOUS	This indicates a general ambiguity in the specification such as multiple embedded extensible elements.
ASN-E-VALTYPMIS	This indicates the value specified does not match the type it is associated with.
ASN-E-RANGERR	This indicates the value is not within defined range for its associated type.
ASN-E-VALPARSE	This indicates a general failure to parse a value definition. It would be raised, for example, if a floating point number was used as part of a SIZE constraint.
ASN-E-INVRANGE	This indicates an invalid range specification, for example when the lower bound is greater than the upper bound.
ASN-E-IMPORTMOD	This indicates that the specified import module object was not found.
ASN-E-NOTSUPP	This indicates that the requested functionality is not supported by the compiler. Most often the error is raised when generating test code for complex value definitions.
ASN-E-IDNOTFOU	This indicates the compiler was unable to look up the specified identifier.
ASN-E-NOFIELD	This indicates that the specified field could not be found in the named class.
ASN-E-DUPLNAME	This indicates the specified name is already defined.
ASN-W-UNNAMED	This warning is raised when specifications use unnamed fields. These fields not allowed in X.680, but ASN1C supports them for purposes of backwards compatibility with X.208.

Error Code	**Error Description**
ASN-E-UNDEFOBJ	This indicates that the named object is not defined within context of the requested module.
ASN-E-ABSCLSFLD	This indicates that the specified field is absent in an information object definition.
ASN-E-UNDEFCLAS	This indicates that the specified class is not defined within context of the module that uses it.
ASN-E-INVFIELD	This indicates the specified class field is not valid; it must be defined.
ASN-E-UNDEFOSET	This indicates that ASN1C was unable to find the specified object set in the context of the module in which it's used.
ASN-E-INVVALELM	An invalid value was supplied for an element in a type.
ASN-E-MISVALELM	This indicates that a non-optional element is missing a value when it should have one.
ASN-E-INVLIDENT	This indicates that an invalid identifier was specified in an enumeration.
ASN-E-FILNOTFOU	This indicates that the requested file was not found.
ASN-E-INVSIZE	This indicates that an invalid size specification for a type was provided; check size constraints for base types.
ASN-E-UNRESOBJ	This indicates that the specified information object could not be resolved within the context of the named module.
ASN-E-TOOMANY	This indicates that too many sub-elements for the specified type were provided.
ASN-E-LOOPDETECTED	This indicates a loop was detected in the course of code generation; typically this is raised during test code generation.
ASN-E-INVXMLATTR	This indicates that the specified attribute type must be a simple type.
ASN-E-INTERNAL	This indicates that internal structures used for generating code are inconsistent.
ASN-E-NOPDU	This indicates that a PDU type was not found for generating a reader or writer program.

General Status Messages

The following table contains both system and validation failures that may occur during program execution. These failures do not arise from ASN.1-specific features (such as an invalid PER encoding), but instead comprehend such failures as buffer overflows, invalid socket options, or closed streams.

Error Code	Error Name	Description
0	RT_OK	Normal completion status.
2	RT_OK_FRAG	Message fragment return status. This is returned when a part of a message is successfully decoded. The application should continue to invoke the decode function until a zero status is returned.
-1	RTERR_BUFOVFLW	Encode buffer overflow. This status code is returned when encoding into a static buffer and there is no space left for the item currently being encoded.
-2	RTERR_ENDOFBUF	Unexpected end-of-buffer. This status code is returned when decoding and the decoder expects more data to be available but instead runs into the end of the decode buffer.
-3	RTERR_IDNOTFOU	Expected identifier not found. This status is returned when the decoder is expecting a certain element to be present at the current position and instead something different is encountered. An example is decoding a sequence container type in which the declared elements are expected to be in the given order. If an element is encountered that is not the one expected, this error is raised.
-4	RTERR_INVENUM	Invalid enumerated identifier. This status is returned when an enumerated value is being encoded or decoded and the given value is not in the set of values defined in the enumeration facet.
-5	RTERR_SETDUPL	Duplicate element in set. This status code is returned when decoding an ASN.1 SET or XSD xsd:all construct. It is raised if a given element defined in the content model group occurs multiple times in the instance being decoded.
-6	RTERR_SETMISRQ	Missing required element in set. This status code is returned when decoding an ASN.1 SET or XSD xsd:all construct and all required elements in the content model group are not found to be present in the instance being decoded.
-7	RTERR_NOTINSET	Element not in set. This status code is returned when encoding or decoding an ASN.1 SET or XSD xsd:all construct. When encoding, it occurs

Error Code	Error Name	Description
		when a value in the generated _order member variable is outside the range of indexes of items in the content model group. It occurs on the decode side when an element is received that is not defined in the content model group.
-8	RTERR_SEQOVFLW	Sequence overflow. This status code is returned when decoding a repeating element (ASN.1 SEQUENCE OF or XSD element with minmaxOccurs > 1) and more instances of the element are received the content model group.
-9	RTERR_INVOPT	Invalid option in choice. This status code is returned when encoding or decoding an ASN.1 CHOICE or XSD xsd:choice construct. When encoding, it occurs when a value in the generated 't' member variable is outside the range of indexes of items in the content model group. It occurs on the decode side when an element is received that is not defined in the content model group.
-10	RTERR_NOMEM	No dynamic memory available. This status code is returned when a dynamic memory allocation request is made and an insufficient amount of memory is available to satisfy the request.
-11	RTERR_INVHEXS	Invalid hexadecimal string. This status code is returned when decoding a hexadecimal string value and a character is encountered in the string that is not in the valid hexadecimal character set ([0-9A-Fa-f] or whitespace).
-12	RTERR_INVREAL	Invalid real number value. This status code is returned when decoding a numeric floating-point value and an invalid character is received (i.e. not numeric, decimal point, plus or minus sign, or exponent character).
-13	RTERR_STROVFLW	String overflow. This status code is returned when a fixed-sized field is being decoded as specified by a size constraint and the item contains more characters or bytes then this amount. It can occur when a run-time function is called with a fixed-sixed static buffer and whatever operation is being done causes the bounds of this buffer to be exceeded.

Error Code	Error Name	Description
-14	RTERR_BADVALUE	Bad value. This status code is returned anywhere where an API is expecting a value to be within a certain range and it not within this range. An example is the encoding or decoding date values when the month or day value is not within the legal range (1-12 for month and 1 to whatever the max days is for a given month).
-15	RTERR_TOODEEP	Nesting level too deep. This status code is returned when a preconfigured maximum nesting level for elements within a content model group is exceeded.
-16	RTERR_CONSVIO	Constraint violation. This status code is returned when constraints defined the schema are violated. These include XSD facets such as minmaxOccurs, minmaxLength, patterns, etc.. Also ASN.1 value range, size, and permitted alphabet constraints.
-17	RTERR_ENDOFFILE	Unexpected end-of-file error. This status code is returned when an unexpected end-of-file condition is detected on decode. It is similar to the ENDOFBUF error code described above except that in this case, decoding is being done from a file stream instead of from a memory buffer.
-18	RTERR_INVUTF8	Invalid UTF-8 character encoding. This status code is returned by the decoder when an invalid sequence of bytes is detected in a UTF-8 character string.
-19	RTERR_OUTOFBND	Array index out-of-bounds. This status code is returned when an attempt is made to add something to an array and the given index is outside the defined bounds of the array.
-20	RTERR_INVPARAM	Invalid parameter passed to a function of method. This status code is returned by a function or method when it does an initial check on the values of parameters passed in. If a parameter is found to not have a value in the expected range, this error code is returned.
-21	RTERR_INVFORMAT	Invalid value format. This status code is returned when a value is received or passed into a function that is not in the expected format. For example, the time string parsing function expects a

Error Code	Error Name	Description
		string in the form "nn:nn:nn" where n's are numbers. If not in this format, this error code is returned.
-22	RTERR_NOTINIT	Context not initialized. This status code is returned when the run-time context structure (OSCTXT) is attempted to be used without having been initialized. This can occur if rtxInitContext is not invoked to initialize a context variable before use in any other API call. It can also occur is there is a license violation (for example, evaluation license expired).
-23	RTERR_TOOBIG	Value will not fit in target variable. This status is returned by the decoder when a target variable is not large enough to hold a a decoded value. A typical case is an integer value that is too large to fit in the standard C integer type (typically a 32-bit value) on a given platform. If this occurs, it is usually necessary to use a configuration file setting to force the compiler to use a different data type for the item. For example, for integer, the <isBigInteger> setting can be used to force use of a big integer type.
-24	RTERR_INVCHAR	Invalid character. This status code is returned when a character is encountered that is not valid for a given data type. For example, if an integer value is being decoded and a non-numeric character is encountered, this error will be raised.
-25	RTERR_XMLSTATE	XML state error. This status code is returned when the XML parser
-26	RTERR_XMLPARSE	XML parser error. This status code in returned when the underlying XML parser application (by default, this is Expat) returns an error code. The parser error code or text is returned as a parameter in is not in the correct state to do a certain operation.
-27	RTERR_SEQORDER	Sequence order error. This status code is returned when decoding an ASN.1 SEQUENCE or XSD xsd:sequence construct. It is raised if the elements were received in an order different than that specified the errInfo structure within the context structure.

Error Code	Error Name	Description
-28	RTERR_FILNOTFOU	File not found. This status code is returned if an attempt is made to open a file input stream for decoding and the given file does not exist.
-29	RTERR_READERR	Read error. This status code if returned if a read IO error is encountered when reading from an input stream associated with a physical device such as a file or socket.
-30	RTERR_WRITEERR	Write error. This status code if returned if a write IO error is encountered when attempting to output data to an output stream associated with a physical device such as a file or socket.
-31	RTERR_INVBASE64	Invalid Base64 encoding. This status code is returned when an error is detected in decoding base64 data.
-32	RTERR_INVSOCKET	Invalid socket. This status code is returned when an attempt is made to read or write from a scoket and the given socket handle is invalid. This may be the result of not having established a proper connection before trying to use the socket handle variable.
-33	RTERR_INVATTR	Invalid attribute. This status code is returned by the decoder when an attribute is encountered in an XML instance that was not defined in the XML schema.
-34	RTERR_REGEXP	Invalid regular expression. This status code is returned when a syntax error is detected in a regular expression value. Details of the syntax error can be obtained by invoking rtxErrPrint to print the details of the error contained within the context variable.
-35	RTERR_PATMATCH	Pattern match error. This status code is returned by the decoder when a value in an XML instance does not match the pattern facet defined in the XML schema. It can also be returned by numeric encode functions that cannot format a numeric value to match the pattern specified for that value.
-36	RTERR_ATTRMISRQ	Missing required attribute. This status code is returned by the decoder when an XML instance is

Error Code	Error Name	Description
		missing a required attribute value as defined in the XML schema.
-37	RTERR_HOSTNOTFOU	Host name could not be resolved. This status code is returned from run-time socket functions when they are unable to connect to a given host computer.
-38	RTERR_HTTPERR	HTTP protocol error. This status code is returned by functions doing HTTP protocol operations such as SOAP functions. It is returned when a protocol error is detected. Details on the specific error can be obtained by calling rtxErrPrint.
-39	RTERR_SOAPERR	SOAP error. This status code when an error is detected when tryingto execute a SOAP operation.
-40	RTERR_EXPIRED	Evaluation license expired. This error is returned from evaluation versions of the run-time library when the hard-coded evaluation period is expired.
-41	RTERR_UNEXPELEM	Unexpected element encountered. This status code is returned when an element is encountered in a position where something else (for example, an attribute) was expected.
-42	RTERR_INVOCCUR	Invalid number of occurrences. This status code is returned by the decoder when an XML instance contains a number of occurrences of a repeating element that is outside the bounds (minOccursmaxOccurs) defined for the element in the XML schema.
-43	RTERR_INVMSGBUF	Invalid message buffer has been passed to decode or validate method. This status code is returned by decode or validate method when the used message buffer instance has type different from OSMessageBufferIF::XMLDecode.
-44	RTERR_DECELEMFAIL	Element decode failed. This status code and parameters are added to the failure status by the decoder to allow the specific element on which a decode error was detected to be identified.
-45	RTERR_DECATTRFAIL	Attribute decode failed. This status code and parameters are added to the failure status by the

Error Code	Error Name	Description
		decoder to allow the specific attribute on which a decode error was detected to be identified.
-46	RTERR_STRMINUSE	Stream in-use. This status code is returned by stream functions when an attempt is made to initialize a stream or create a reader or writer when an existing stream is open in the context. The existing stream must first be closed before initializing a stream for a new operation.
-47	RTERR_NULLPTR	Null pointer. This status code is returned when a null pointer is encountered in a place where it is expected that the pointer value is to be set.
-48	RTERR_FAILED	General failure. Low level call returned error.
-49	RTERR_ATTRFIXEDVAL	Attribute fixed value mismatch. The attribute contained a value that was different than the fixed value defined in the schema for the attribute.
-50	RTERR_MULTIPLE	Multiple errors occurred during an encode or decode operation. See the error list within the context structure for a full list of all errors.
-51	RTERR_NOTYPEINFO	This error is returned when decoding a derived type definition and no information exists as to what type of data is in the element content. When decoding XML, this normally means that an xsi:type attribute was not found identifying the type of content.
-52	RTERR_ADDRINUSE	Address already in use. This status code is returned when an attempt is made to bind a socket to an address that is already in use.
-53	RTERR_CONNRESET	Remote connection was reset. This status code is returned when the connection is reset by the remote host (via explicit command or a crash).
-54	RTERR_UNREACHABLE	Network failure. This status code is returned when the network or host is down or otherwise unreachable.
-55	RTERR_NOCONN	Not connected. This status code is returned when an operation is issued on an unconnected socket.
-56	RTERR_CONNREFUSED	Connection refused. This status code is returned when an attempt to communicate on an open socket is refused by the host.

Error Code	Error Name	Description
-57	RTERR_INVSOCKOPT	Invalid option. This status code is returned when an invalid option is passed to socket.
-58	RTERR_SOAPFAULT	This error is returned when the decoded SOAP envelope is a fault message.
-59	RTERR_MARKNOTSUP	This error is returned when an attempt is made to mark a stream position on a stream type that does not support it.
-60	RTERR_NOTSUPP	Feature is not supported. This status code is returned when a feature that is currently not supported is encountered.
-61	RTERR_CODESETCONVFAIL	This status code is returned when transcoding from one character set to another one (for example, from UTF-8 to UTF-16) and a conversion error occurs.

ASN.1-specific Status Messages

The following table describes status messages that may arise during the course of encoding or decoding an ASN.1 message. The errors below indicate that while the system was able to read the data successfully, it was unable to decode it properly.

Error Code	Error Name	Description
2	ASN_OK_FRAG	Fragment decode success status. This is returned when decoding is successful but only a fragment of the item was decoded. User should repeat the decode operation in order to fully decode message.
-100	ASN_E_BASE	Error base. ASN.1 specific errors start at this base number to distinguish them from common and other error types.
(ASN_E_BASE)	ASN_E_INVOBJID	Invalid object identifier. This error code is returned when an object identifier is encountered that is not valid. Possible reasons for being invalid include invalid first and second arc identifiers (first must be 0, 1, or 2; second must be less than 40), not enough subidentifier values (must be 2 or more), or too many arc values (maximum number is 128).
(ASN_E_BASE-1)	ASN_E_INVLEN	Invalid length. This error code is returned when a length value is parsed that is not consistent with

Error Code	Error Name	Description
		other lengths in a BER or DER message. This typically happens when an inner length within a constructed type is larger than the outer length value.
(ASN_E_BASE-2)	ASN_E_BADTAG	Bad tag value. This error code is returned when a tag value is parsed with an identifier code that is too large to fit in a 32-bit integer variable.
(ASN_E_BASE-3)	ASN_E_INVBINS	Invalid binary string. This error code is returned when decoding XER data and a bit string value is received that contains something other than '1' or '0' characters.
(ASN_E_BASE-4)	ASN_E_INVINDEX	Invalid table constraint index. This error code is returned when a value is provided to index into a table and the value does not match any of the defined indexes.
(ASN_E_BASE-5)	ASN_E_INVTCVAL	Invalid table constraint value. This error code is returned when a the value for an element in a table-constrained message instance does not match the value for the element defined in the table.
(ASN_E_BASE-6)	ASN_E_CONCMODF	Concurrent list modification error. This error is returned from within a list iterator when it is detected that the list was modified outside the control of the iterator.
(ASN_E_BASE-7)	ASN_E_ILLSTATE	Illegal state for operation. This error is returned in places where an operation is attempted but the object is not in a state that would allow the operation to be completed. One example is in a list iterator class when an attempt is made to remove a node but the node does not exist.
(ASN_E_BASE-8)	ASN_E_NOTPDU	This error is returned when a control class Encode or Decode method is called on a non-PDU. Only PDUs have implementations of these methods.
(ASN_E_BASE-9)	ASN_E_UNDEFTYP	Element type could not be resolved at run-time. This error is returned when the run-time parser module is used (Asn1RTProd) to decode a type at run-time and the type of the element could not be resolved.

Error Code	**Error Name**	**Description**
(ASN_E_BASE-10)	ASN_E_INVPERENC	Invalid PER encoding. This occurs when a given element within an ASN.1 specification is configured to have an expected PER encoding and the decoded value does not match this encoding.