ASN1C

ASN.1 Compiler
Version 5.3
C/C++ User's Manual

Author's Contact Information:

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

CHANGE HISTORY

Date	Author	Version	Description
11/06/2001	ED	5.3	Initial version
05/01/2002	ED	5.32	Updated sections on calling C BER/DER encode/decode functions to specify use of rtInitContext.

TABLE OF CONTENTS

Overview of ASN1C

The ASN1C code generation tool translates an Abstract Syntax Notation 1 (ASN.1) 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 three different languages: 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.

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 C/C++ source (.c or .cpp) file containing C/C++ encode and decode functions. One encode and decode function is generated for each ASN.1 production.

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 the ITU standard X.680. It generates code for encoding/decoding data as specified in the Basic Encoding Rules (BER) published in the ITU X.690 standard and the Packed Encoding Rules (PER) published in the ITU X.691 standard. The compiler is capable of parsing all ASN.1 syntax as defined in the standards. Its mission is to get to the base types that are the basis for encoding and decoding the messages that the specification defines. It will skip over all other definitions and related 'fluff'.

This release of the compiler contains a special executable (asn1c90.exe) that backward compatible with 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 ROSE.

< this page intentionally left blank >

Using the Compiler

Running the Compiler

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 5.3x
Copyright (c) 1997-2002 Objective Systems, Inc. All Rights Reserved.

Usage: asn1c <filename> options

    <filename>         ASN.1 source file name

  options:
    -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)
    -print <filename>  Generate print routines and write
                         to filename
    -ber               generate BER encode/decode functions
    -der               generate DER encode/decode functions
    -per               generate PER encode/decode functions
    -trace             add trace diag msgs to generated code
    -c                 generate C code
    -c++               generate C++ code
    -java              generate Java code
    -events            generate code to invoke event handlers
    -config <file>     specify configuration file
    -nodecode          do not generate decode functions
    -noencode          do not generate encode functions
    -noIndefLen        do not generate indefinite length tests
    -compact           generate compact code
    -warnings          Output compiler warning messages
    -o <directory>     Output file directory
    -I <directory>     Import file directory
    -pkgpfx <text>     Java package prefix
    -pkgname <text>    Java package name
    -list              generate listing
    -compat <version>  generate code compatible with previous
                       compiler version. <version> format is
                       x.x (for example, 5.2)
```

This indicates that to use the compiler, at a minimum, an ASN.1 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.

The source file 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 X.680.. The asn1c90 executable 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:

Option	Argument	Description
-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.
-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 <modulename>.c where <modulename> is the name of the module from the ASN.1 source file.
-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. They are optional: if –print is not specified, no print functions will be generated. The <filename> argument to this option is also 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.
-ber	None	This option instructs the compiler to generate functions that implement the Basic Encoding Rules (BER) Rulesas specified in the ASN.1 standards.
-der	None	This option instructs the compiler to generate functions that implement the Distinguished Encoding Rules (DER) as specified in the ASN.1 standards.
-per	None	This option instructs the compiler to generate functions that implement the Packed Encoding Rules (PER) as specified in the ASN.1 standards.
-trace	None	This option is used to tell the compiler to add trace diagnostic messages to the generated code. These messages cause printf statements to be added to the generated code to print entry and exit information into the generated functions. This is a debugging option that allows encode/decode problems to be isolated to a given production processing function. Once the code is debugged, this option should not be used as it adversely affects performance.
-java	None	Generate Java source code.
-c	None	Generate C source code.
-c++	None	Generate C++ source code.
-events	None	Generate extra code to invoke user defined event and error handler callback methods (see the *Event Handlers* section).
-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 a later section.
-noencode	None	This option suppresses the generation of encode functions. .
-nodecode	None	This option suppresses the generation of decode 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.
-compact	None	This option instructs the compiler to generate more compact code at the expense of some constraint and error checking. This is an optimization option that should be used after an application is thoroughly tested.
-warnings	None	Output information on compiler generated warnings. .
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-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.
-pkgpfx	<prefixName>	This is a Java option for adding a prefix in front of the assigned Java package name. By default, the Java package name is set to the module name. If the package is embedded within a hierarchy, this option can be used to set the other directory names that must be added to allow Java to find the .class files.
-pkgname	<packageName>	This is a Java option that allows the entire Java package name to be changed. Instead of the module name, the full name specified using this option will be used. This option cannot be used in conjunction with –pkgpfx option.
-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.
-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)

Several options from the 5.0x release have been decommissioned and replaced with entries in the configuration file. These options include –dynamic, -pdu, and –enum_prefix. See the section on the configuration file to find the equivalent configuration settings for these options.

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 library. On Windows systems, the name of this file is either asn1ber.lib or asn1per.lib depending on whether BER or PER source-code generation was specified; on UNIX, the library names are libasn1ber.a and libasn1per.a respectively. The library file can be found in the lib subdirectory. For UNIX, the –L switch should be used to point to the subdirectory path and –lasn1ber or –lasn1per used to link with the library. For Windows, the –LIBPATH switch should be used to specify the library path.

Windows systems also include dynamic-link library (dll) versions of the library. These are located in the dll subdirectory. To use them, link with the version of the asn1ber.lib or asn1per.lib file that is contained in the dll subdirectory.

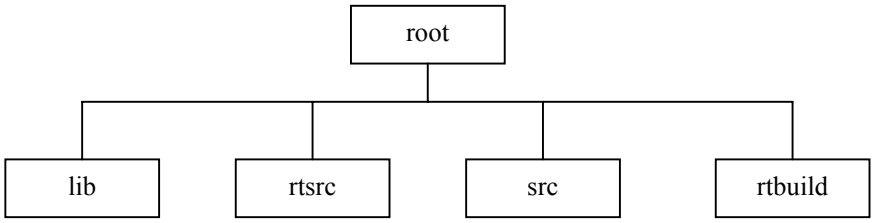
See the makefile in any of the sample subdirectories of the distribution for an example of what must be included to build a program using generated source code.

Porting Run-time Code to Other Platforms

The standard 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, rtsrc, and rtbuild subdirectories. 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 going 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 rtsrc subdirectories from the installation to the src and rtsrc directories on the target platform (note: if going 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 rtbuild subdirectory of the installation to the rtbuild subdirectory on the target platform (note: if going 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:

   CC C compiler executable name
   CCC C++ compiler executable name
   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 rtbuild 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 the Extended Markup Language (XML) is used to format items in the file. This language 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 to 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. 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. 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 the '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
<storage></storage>	dynamic, static, or list 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 optional elements within SEQUENCE or SET constructs. If static (the default), it indicates static type 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.

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. It is required.
<include 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 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 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).
<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, or list 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. This replaces the **module.txt** file used in previous versions of the compiler to accomplish this function.
<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).
<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.
<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.

Production Level

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

Name	Values	Description
<name> </name>	module name	This attribute identifies the module to which this section applies. It is required.
<storage> </storage>	dynamic, static, or list 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.
<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).
<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.

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. ASN1CPP 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. C programmers tend to like to use the underscore character (_) in identifiers. This is not allowed in ASN.1. Conversely, C does not allow hyphens in identifiers. To get around this problem, ASN1CPP 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.

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 index and mask constants (BIT STRING type only)
- Enumerated type option values (ENUMERATED or INTEGER type only)
- C type definition
- Encode function prototype
- Decode function prototype
- C++ class definition which 'wraps' an instance of the production type variable and associated encode/decode functions. In some cases, the compiler may generate additional methods specific to a particular production type. (C++ only)

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

```
/*************************************************************/
/*                                                           */
/*  EmployeeNumber                                           */
/*                                                           */
/*************************************************************/

#define TV_EmployeeNumber     TM_APPL|TM_PRIM|2

typedef ASN1INT  EmployeeNumber;

int asn1E_EmployeeNumber (ASN1CTXT* ctxt_p,
   ASN1T_EmployeeNumber *object_p, ASN1TagType tagging);

int asn1D_EmployeeNumber (ASN1CTXT* ctxt_p,
   ASN1T_EmployeeNumber *object_p, 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 'typdef ASN1INT EmployeeNumber' declares EmployeeNumber to be of an integer type (note: ASN1INT and other primitive type definitions can be found in the asn1type.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.

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 ASN1INT  ASN1T_EmployeeNumber;

class ASN1C_EmployeeNumber : public ASN1CType {
public:
   ASN1T_EmployeeNumber& msgData;
   ASN1C_EmployeeNumber (ASN1MessageBuffer& msgBuf,
                         ASN1T_EmployeeNumber& data);
   int Encode ();
   int Decode ();
} ;

int asn1E_EmployeeNumber (ASN1CTXT* ctxt_p,
   ASN1T_EmployeeNumber *object_p, ASN1TagType tagging);

int asn1D_EmployeeNumber (ASN1CTXT* ctxt_p,
   ASN1T_EmployeeNumber *object_p, 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.

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 *Encocde* and *Decode* 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.

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, a public msgData variable reference of the generated type is declared. The constructor takes two arguments – an Asn1MessageBuffer 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 data type reference is a reference to the 'ASN1T_' variable that was generated for the data type.

Encode and Decode methods are declared that wrap the respective compiler generated C encode and decode functions. If the –print function command line argument was used, a Print method is also generated to wrap the corresponding C print function.

The equivalent C and C++ type definitions for each of the various ASN.1 types follow.

BOOLEAN

The ASN.1 BOOLEAN type is converted into a C type named "ASN1BOOL". In the global include file "asn1type.h", ASN1BOOL is defined to be an "unsigned character".

ASN.1 production:	`<name> ::= BOOLEAN`
Generated C code:	`typedef ASN1BOOL <name>;`
Generated C++ code:	`typedef ASN1BOOL ASN1T_<name>;`

For example, if "`B ::= [PRIVATE 10] BOOLEAN`" was defined as an ASN.1 production, the generated C type definition would be "`typedef ASN1BOOL B`". Note that the tag information is not represented in the type definition, this is handled within the generated encode/decode functions.

Note that 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 a C type named either "ASN1INT" or "ASN1UINT". In the global include file "asn1type.h", ASN1INT is defined to be an "int", ASN1UINT is defined to be an "unsigned int".

ASN.1 production:	`<name> ::= INTEGER`
Generated C code:	`typedef ASN1INT <name>;`
Generated C++ code:	`typedef ASN1INT ASN1T_<name>;`

The ASN1INT type represents a signed integer number, ASN1UINT represents an unsigned integer number. ASN1UINT is used if a value range constraint on a production specification exceeds the maximum value that can be stored in a signed integer. An example of this would be the Counter production in the SNMP SMI specification:

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

This would cause the following typedef to be generated:

```
typedef ASN1UINT Counter;
```

Large Integer Support

In C and C++, the maximum size for an integer type is normally 32 bits (or 64 bits on some newer, 64-bit machines). 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. Only hexadecimal string representations of the integer value are supported in this release.

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 ASN1ConstCharPtr SecurityKeyType
```

The ASN1ConstCharPtr type is declared to be a 'char*' type for C and a 'const char*' type for C++ in the asn1type.h header file. 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.

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 make setting the value a bit easier.

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 {
   ASN1UINT numbits;
   ASN1OCTET* data;
} ASN1TDynBitStr;
```

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

```
typedef struct ASN1TDynBitStr {
   ASN1UINT numbits;
   ASN1OCTET* data;
   // ctors
   ASN1TDynBitStr () : numbits(0) {}
   ASN1TDynBitStr (ASN1UINT _numbits, ASN1OCTET* _data);
} ASN1TDynBitStr;
```

Static (sized) BIT STRING

ASN.1 production:

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

Generated C code:

```
typedef struct {
   int        numbits;
   ASN1OCTET  data[<adjusted_len>*];
} <name>;
```

Generated C++ code:

```
typedef struct {
   int        numbits;
   ASN1OCTET  data[<adjusted_len>*];
   // ctors
   ASN1T_<name> ();
   ASN1T_<name> (ASN1UINT _numbits,
                 ASN1OCTET* _data);
} ASN1T_<name>;
```

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

For example, the following ASN.1 production:

```
BS ::= [PRIVATE 220] BIT STRING (SIZE (18))
```

Would translate to the following C typedef:

```
typedef struct ASN1T_BS {
   ASN1UINT numbits;
   ASN1OCTET data[3];
} ASN1T_BS;
```

In this case, three octets would be required to hold the 18 bits: eight in the first two bytes, and two in the third.

Note that for C++, the compiler 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 of construct by generating symbolic constants

that can be used to set, clear, or test these named bits. These symbolic constants are in the form of a byte index and a bit mask. 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.

Bits are defined in order from left to right in a bit string. The starting bit number is zero. Therefore, a bit string containing one set bit would result in a single octet value of 0x80 (left most bit set). If this bit were named, the compiler would generate a byte index constant of 0, and a bit mask constant of 0x80. The byte index would be used to access the specific octet in the octet array. The bit mask would then be used to access the bit using a logical bit operator.

For example, the following ASN.1 production:

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

Would translate to:

```
/* Named bit constants */

#define BitMbitOne      0x40
#define BytXbitOne      0
#define BitMbitTen      0x20
#define BytXbitTen      1

/* Type definitions */

typedef struct ASN1T_NamedBS {
   ASN1UINT numbits;
   ASN1OCTET data[2];
} NamedBS;
```

The named bit constants would be used to access the data array within the ASN1T_NamedBS type. The named bit 'bitOne' could be set with the following code:

```
NamedBS bs;
memset (&bs, 0, sizeof(bs));
bs.data[BytXbitOne] |= BitMbitOne;
```

The statement to clear the bit would be as follows:

```
bs.data[BytXbitOne] &= ~BitMbitOne;
```

Finally, the bit could be tested using the following statement:

```
bs.data[BytXbitOne] & BitMbitOne
```

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.

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 *ASN1CBitStr* in the C++ Run-Time Classes section 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 and assignment operators that make setting the value a bit easier.

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 {
   ASN1UINT numbits;
   ASN1OCTET* data;
} ASN1TDynBitStr;
```

The ASN1TDynOctStr type is defined in the asn1CppTypes.h header file and has the following definition:

```
typedef struct ASN1TDynOctStr {
   ASN1UINT numocts;
   ASN1OCTET* data;
   // ctors
   ASN1TDynOctStr ();
   ASN1TDynOctStr (ASN1UINT _numocts, ASN1OCTET* _data);
   ASN1TDynOctStr (char* cstring);
   // assignment operators
   ASN1TDynOctStr& operator= (char* cstring)
} ASN1TDynOctStr;
```

Static (sized) OCTET STRING

ASN.1 production:

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

Generated C code:

```
typedef struct {
   ASN1UINT   numocts;
   ASN1OCTET  data[<len>];
} <name>;
```

Generated C++ code:

```
typedef struct {
   ASN1UINT   numocts;
   ASN1OCTET  data[<len>];

   // ctors
   ASN1T_<name> ();
   ASN1T_<name> (ASN1UINT _numocts,
                 ASN1OCTET* _data);
   ASN1T_<name> (char* cstring);

   // assignment operators
   ASN1T_<name>& operator= (char* cstring);

} ASN1T_<name>;
```

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>
```

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.

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>
```

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 <name>::Root 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. In this case, an ASN1UINT is used in the typedef; otherwise, the Root section of the enumeration is used to define the type.

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 {
   ASN1UINT numids;             /* number of subidentifiers */
   ASN1UINT 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 value of this constant can be changed to a lower number for applications with restricted memory requirements.

The ASN1TObjId type used in the C++ mapping is defined in Asn1CppTypes.h as follows:

```
struct EXTERN ASN1TObjId {
   ASN1UINT numids;
   ASN1UINT subid[ASN_K_MAXSUBIDS];

   ASN1TObjId () : numids(0) {}
   ASN1TObjId (ASN1OCTET _numids, const ASN1USINT* _subids);
   ASN1TObjId (const ASN1OBJID& oid);
   ASN1TObjId (const ASN1TObjId& oid);
   void operator= (const ASN1OBJID& rhs);
   void operator= (const ASN1TObjId& rhs);
} ;
```

The definition is the same as the C type with the addition of the constructors and assignment operators. Note that a constructor and assignment operator are overloaded to use the C ASN1OBJID type. That is because value assignments are generated using the ASN1OBJID type so these methods allow direct assignment of these generated values to an object of this type.

REAL

The ASN.1 REAL type is mapped to the C type "ASN1REAL". In the global include file "asn1type.h", ASN1REAL is defined to be a "double".

ASN.1 production:	`<name> ::= REAL`
Generated C code:	`typedef ASN1REAL <name>;`
Generated C++ code:	`typedef ASN1REAL ASN1T_<name>;`

SEQUENCE

The mapping for the ASN.1 SEQUENCE type for C and C++ is identical with the exception of:

1. The C++ type having the 'ASN1T_' prefix, and

2. The C++ type may have a constructor to initialize an optional bit mask (see the subsection on optional elements).

This section shows the C mapping. The C++ mapping is the same with the addition of the 'ASN1T_' prefix on each of the type names.

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 temporary types. The name of the temporary types are of the form <name>_<element-name1>_<element-name2>_ … <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 {
   ASN1INT   a1;
   ASN1BOOL  a2;
} A_x;

typedef struct A_y {
   ASN1UINT numocts;
   ASN1OCTET 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). This is a change from v5.1x and lower where only that production name and last element name would be used (i.e., X-aa). The change was made to ensure uniqueness of the generated names when multiple nesting levels are used.

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 ASN1INT AnInt;

typedef struct Aseq {
   ASN1INT        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, we will change the production in the previous example 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;
   ASN1INT        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 C++ version of the compiler will generate a constructor for the structured type for a SEQUENCE 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, so it 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 ASN1OpenType and have the name extElem1. This field will contain the complete encoding 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.

If the SEQUENCE type contains an extension marker and extension elements, then the open extension type field will not be added. Instead, the actual extension elements will be present. 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).

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 a C or C++ structured type containing an integer to hold the number of occurrences of the referenced data element and an array or pointer to the referenced type to hold the actual data values. An option is also available to use a doubly-linked structure as the generated type.

The allocation for the contents field 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 default behavior of allocating a static array for a sized SEQUENCE OF construct can be modified by the use of a configuration item. The <storage> qualifier with the 'dynamic' 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>dynamic</storage>
```

The 'list' keyword can also be used in a similar fashion to specify the use of a doubly-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 {
   int        n;
   <type>*    elem;
```

```
} <name>;
```

Generated C++ code:

```
typedef struct {
   int        n;
   <type>*    elem;
} 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]...).

Static (sized) SEQUENCE OF Type

ASN.1 production:

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

Generated C code:

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

Generated C++ code:

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

List-based SEQUENCE OF Type

A doubly-linked list header type (Asn1RTDList) is used for the typedef 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 Asn1RTDList <name>;
```

Generated C++ code:

```
typedef Asn1RTDList ASN1T_<name>;
```

The type definition of the Asn1RTDList structure can be found in the asn1type.h header file. The common run-time utility functions *rtDListInit* and *rtDListAppend* are available for initializing and adding elements to the list. See the Common Run-time Functions section for a full description of these functions.

In addition to the *Asn1RTDList* C structure and C functions, a C++ class if provided for linked list support. This is the *ASN1CSeqOfList* class. This class provides methods for adding and deleting elements to and from lists and an iterator interface for traversing lists. See the *ASN1CSeqOfList* section in the C++ Run-Time Classes area for details on all of the methods available in this class.

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 { INTEGER a, BOOLEAN b }
```

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

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

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

These types are then converted into the equivalent C or C++ typedefs 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 cut down on 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.

An example of the savings of this optimization can be found in H.225. The above element reference is repeated 25 different times in different places. The result is the generation of one new temporary type that is referenced in 25 different places. Without this optimization, 25 unique types with the same definition would have been generated.

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 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>;
```

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 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 {
      ASN1INT aInt;
      struct {
         int t;
         union {
            ASN1INT aInt;
            ASN1BOOL aBool;
         } u;
      } aChoice;
   } C;
```

Note that an '_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. The recommended way to do it 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 ASN1DynOctStr AsciiString;
typedef ASN1DynOctStr 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 the 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.

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.

The ASN.1 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.

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.

The ASN.1 "ANY DEFINED BY Type" construct is treated the same as an ANY. No attempt is made to verify the identified Type.

Character String Types

As of version 5.0 and above, character string types are now built into the compiler. Previous versions used compiled definitions based on the OCTET STRING base type to model these types. All 8-bit character character-string types now are derived from the C character pointer (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 rtMemFree 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 char* pointers except for the BMPString and UniversalString types. The BMPString is a 16-bit character string for which the following structure is used:

```
typedef struct {
   ASN1UINT       nchars;
   ASN116BITCHAR* data;
} Asn116BitCharString;
```

The ASN116BITCHAR type used in this definition is defined to be an “unsigned short”.

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 is a 32-bit character string for which the following structure is used:

```
typedef struct {
   ASN1UINT       nchars;
   ASN132BITCHAR* data;
} Asn132BitCharString;
```

The ASN132BITCHAR type used in this definition is defined to be an “unsigned int”.

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.

Utilities are also provided for working with UTF-8 string dataThe contents for this string type are assumed to contain the UTF-8 encoding of a character string. 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 *rtUTF8ToWCS* and *rtWCSToUTF8* can be used for converting to and from Unicode. The function *rtValidateUTF8* can be used to ensure that a given UTF-8 encoding is valid. See the *Run-Time Common Library* section 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++ Run-Time Classes section for details on all of the methods available in these classes.

External Type

The ASN.1 EXTERNAL type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. The 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] ANY,
      octet-aligned     [1] IMPLICIT OCTET STRING,
      arbitrary         [2] IMPLICIT BIT STRING
   }
}
```

The ASN.1 compiler is used to create a meta-definition for this structure. The definition is stored in the file asn1External.h. The resulting C structure is populated just like any other compiler-generated structure for working with ASN.1 data.

Parameterized Types

The 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 {
   ASN1UINT       numocts;
   ASN1OCTET      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 would be 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, 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. 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.

Information Objects

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.

ASN1C compiler support for these types of specifications is limited to the correct application of reference types in places where Information Object Class references are embedded in standard ASN.1 types. Other applications of these constructs are parsed but do not result in the generation of any application code.

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. 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.

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 which 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.

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.

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.

Other constructs can be built using class definitions such as Information Object instances and Information Object Sets. This document will not get into the definition and uses for these items other than to say that the ASN1C compiler will parse and silently ignore them. They provide additional information on how to put messages together, but are not part of the actual types themselves. For this reason, the compiler does not generate any additional code for their use.

Value Specifications

The compiler can parse any type of ASN.1 value specification, but will only generate code for certain types. In this release of the compiler, the following types of value specifications will result in generated code:

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

All value types except INTEGER 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 value specifications cause #define statements to be generated.

INTEGER Value Specification

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 on the fly.

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.

BOOLEAN Value Specification

A BOOLEAN value causes an "extern" statement to be generated in the header file and a global declaration of type ASN1BOOL 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:
```
ASN1BOOL <name> = <value>;
```

Binary and Hexadecimal String Value Specification

These 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:
```
ASN1UINT  <name>_numocts = <length>;
ASN1OCTET <name>_data[]  = <data>;
```

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

Character String Value Specification

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

ASN.1 production: `<name> <string-type> ::= <value>`

Generated code:
```
ASN1ConstCharPtr <name> = <value>;
```

In this definition, <string-type> could be any of the standard 8-bit characters string types such as IA5String, PrintableString, etc. (note: this version of the compiler does not contain support for value declarations of larger character string type such as BMPString). The ASN1ConstCharPtr type used in the generated code is a type defined in asn1type.h designed to be a char* type for C or const char* type for C++.

Object Identifier Value Specification

Object identifier values are somewhat different in that they 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 section in the run-time API guide 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 from to the target variable.

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> (ASN1CTXT* ctxt_p,
   <ProdName>* data_p, ASN1TagType tagging);

int asn1D_<ProdName> (ASN1CTXT* ctxt_p,
   <ProdName>* data_p, 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> (ASN1CTXT* ctxt_p, <ProdName>[*] value);

int asn1PD_<ProdName> (ASN1CTXT* ctxt_p, <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.

Generated Class Definition

A class definition is generated for each defined production in the ASN.1 source file. This 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.

The general form of the class definition is as follows:

```
class ASN1C_<name> : public ASN1CType {
public:
   ASN1T_<name>& msgData;
   ASN1C_<name> (ASN1MessageBuffer& msgBuf, ASN1T_<name>& data);
   int Encode ();
   int Decode ();
} ;
```

The name of the generated class is 'ASN1C_<name>' where '<name>' is the name of the production. The only defined attribute is a public variable reference named 'msgData' of the generated type.

The constructor arguments are a reference to an 'ASN1MessageBuffer' type and a reference to an 'ASN1T_<name>' type. The message buffer argument is a class defined in either the Asn1BerCppTypes.h or Asn1PerCppTypes.h. There are special subclasses for encoding (ASN1BEREncodeBuffer or ASN1PEREncodeBuffer) and decoding (ASN1BERDecodeBuffer and ASN1PERDecodeBuffer). Variables of either of these subclasses can be passed to the constructor depending on whether encoding or decoding is to be performed. The purpose of the buffer objects is to wrap all of the internal values required to manage encode or decode buffers. Examples of using this object can be found in the section on Encoding and Decoding messages.

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 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). Identifying these PDUs and declaring them in a configuration file using the <isPDU/> empty element can attain large savings in the amount of code generated for a particular application. For example, in some H.323 applications, the main PDU structure used is H323-UserInformation. The following configuration file entry could be used to only generate the ASN1C_ control class for this PDU:

```
<asn1config>
  <module>
    <name>H323-MESSAGES</name>
    <production>
      <name>H323-UserInformation</name>
      <isPDU/>
    </production>
  </module>
</asn1config>
```

This will cause only a single ASN1C_ class definition to be added to the generated code – that for the H323-UserInformation production. If this information was not included an ASN1C_ class would be generated for all productions and the vast majority of them would never be used.

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.

Generated Methods

For each production, an Encode and Decode 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.

Generated BER 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 filled-in 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 which 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> (ASN1CTXT* ctxt_p,
                    <name>* object,
                    ASN1TagType tagging);
```

In this definition, <name> denotes the prefixed production name defined above.

The ctxt_p 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 either the *rtInitContext* or *rtNewContext* run-time library functions (see the Run-Time Library API section for a description of these functions).

The object 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.

Generated C++ Encode Method Format and Calling Parameters

The C++ version of the compiler generates an Encode 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 the tag type parameters.

The calling sequence for the generated C++ class method is as follows:

```
len = class_var.Encode ();
```

In this definition, class_var is a variable 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.

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 techniques 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 to allocate memory to hold the data, and
3. Use the **rtMemAlloc** and **rtMemFree** run-time library functions

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 rtMemFree call. The ASN1MALLOC 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 ASN1CTXT 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 rtMemFree is called, all memory can be released at once.

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.

Before any encode function can be called; the user must first initialize an encoding context. This is a variable of type ASN1CTXT. This variable holds all of the working data used during the encoding of a message. The context variable can be initialized in one of two ways:

1. Allocating a context dynamically using the *rtNewContext* function or,

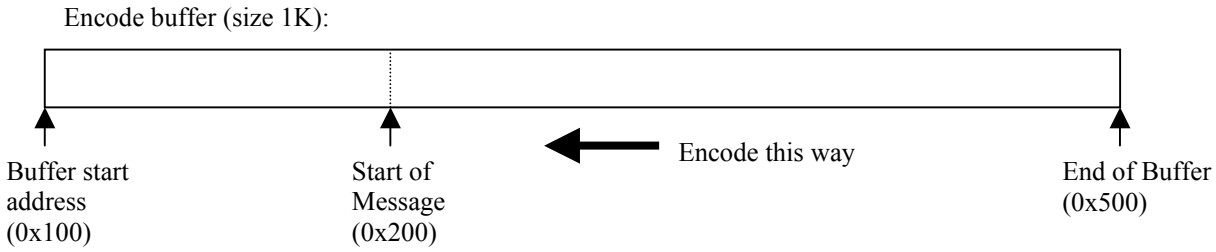
2. Initializing a static variable using the *rtInitContext* function.

An example of initializing a static variable is as follows:

```
ASN1CTXT ctxt;          // context variable
rtInitContext (&ctxt);  // INITIALIZE BEFORE USE!
```

The next step is to specify an encode buffer into which the message will be encoded. This is done by calling *xe_setp* run-time function. An encode buffer must be specified when initializing the context. 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 parameter to zero and let the encode function manage the buffer memory allocation (referred to as a dynamic buffer). Better performance can 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 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 xe_getp.

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

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
    ASN1OCTET msgbuf[1024], *msgptr;
    int       msglen;
    ASN1CTXT  ctxt;
    Employee  employee;   /* typedef generated by ASN1C */

    /* Step 1: Initialize the context and set the buffer pointer */

    rtInitContext (&ctxt);
    xe_setp (&ctxt, msgbuf, sizeof(msgbuf));

    /* Step 2: Populate the structure to be encoded */

    employee.name.numocts = 5;
    employee.name.data = "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
      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 the number of tags required.

If performance is not a significant an 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, the *xe_free* run-time library function must be called.

The following code fragment illustrates encoding using a dynamic buffer:

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
    ASN1OCTET *msgptr;
    int       msglen;
    ASN1CTXT  ctxt;
    Employee  employee;   /* typedef generated by ASN1C */

    rtInitContext (&ctxt);
    xe_setp (&ctxt, NULL, 0);

    employee.name.numocts = 5;
    employee.name.data = "SMITH";
    ...

    msglen = asn1E_Employee (&ctxt, &employee, ASN1EXPL);

    if (msglen > 0) {
      msgptr = xe_getp (&ctxt);
      ...

      xe_free (&ctxt);   /* don't call free (msgptr); !!! */
    }
    else
      error processing...
}
```

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 Asn1BerEncodeMessageBuffer 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 this document.

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 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 ASN1OCTET* msgptr;
    ASN1OCTET 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
    // (note: this uses the generated assignment operator to assign
    // a string)..

    employee.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 ASN1CPP

main ()
{
```

```
    ASN1OCTET* msgptr;
    int        msglen;

    // construct encodeBuffer class with no arguments

    ASN1BEREncodeBuffer encodeBuffer;
    ASN1T_PersonnelRecord msgData;
    ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

    // populate msgData structure

    employee.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 ASN1OCTET* msgptr;
    ASN1OCTET msgbuf[1024];
    int       msglen;

    ASN1BEREncodeBuffer encodeBuffer (msgbuf, sizeof(msgbuf));
    ASN1T_PersonnelRecord msgData;
    ASN1C_PersonnelRecord employee (encodeBuffer, msgData);

    // Encode loop start 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 be encoded
```

```
        employee.msgData.name = "SMITH";
        ...

        // invoke Encode method

        if ((msglen = employee.Encode ()) > 0) {

            // encoding successful, get pointer to start of message

            msgptr = 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 Decode Functions

For each ASN.1 production defined in the 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 that 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> (ASN1CTXT* ctxt_p,
                       <name> *object,
                       ASN1TagType tagging,
                       int length);
```

In this definition, <name> denotes the prefixed production name defined above.

The ctxt_p 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 *xd_setp* run-time function before use.

The object 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 0 respectively.

The function result variable status returns the status of the decode operation. The return status will be zero (ASN_OK) if decoding is successful or negative if an error occurs. Return status values are defined in the "asn1type.h" include file.

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 = class_var.Decode ();
```

In this definition, class_var is a variable of the class generated for the given production.

An ASN1BERDecodeBuffer object must be passed to the class_var constructor prior to decoding. This is where the start address of the message to be decoded and message length are specified.

The message length argument is used to specify the size of the message, if it is known. In ASN.1 messages, the overall length of the message is embedded in the first few bytes of the message, so this variable is really not needed. It is used as 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. As was the case for message pointer above, this parameter can be specified in the constructor if only a single message is being decoded using the class.

The function result variable `status` returns the status of the decode operation. The return status will be zero (ASN_OK) if decoding is successful or a negative value if an error occurs. Return status values are defined in Appendix A of this document and online 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 ASN1CTXT. This variable holds all of the working data used during the encoding of a message. The context variable can be initialized in one of two ways:

3. Allocating a context dynamically using the *rtNewContext* function or,

4. Initializing a static variable using the *rtInitContext* function.

An example of initializing a static variable is as follows:

```
ASN1CTXT ctxt;          // context variable
rtInitContext (&ctxt);  // INITIALIZE BEFORE USE!
```

The next step is the specification of a buffer containing a message to be decoded. This is done 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 to be decoded. 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 "xu_freeall" 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 ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen;
    ASN1CTXT  ctxt;
    PersonnelRecord employee;

    .. logic to read message into msgbuf ..

    /* Step 1: Initialize a context variable for decoding */
```

```
    rtInitContext (&ctxt);

    status = xd_setp (&ctxt, msgbuf, 0, &msgtag, &msglen);

    if (status != ASN_OK) {
       error processing..
    }

    /* 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 == ASN_OK)
        {
            process received data in 'employee' variable..

            /* Remember to release dynamic memory when done! */

            xu_freeall (&ctxt);
        }
        else
            error processing...
    }
    else
       check for other known message types..
}
```

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 Asn1Message 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 ()
{
    ASN1OCTET 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 != ASN_OK) {
   // 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 ()) == ASN_OK)
      {
         // decoding successful, data in employee.msgData

         process received data..
      }
      else
         error processing...
   }

   case TV_ ...       // handle other known messages
```

Note that the call to 'xu_freeall' 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 ()
{
    ASN1OCTET 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 != ASN_OK) {
        // 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 ()) == ASN_OK)
          {
             // 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

      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.

Performance Considerations: 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 a lot 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 relieve the burden of allocating dynamic memory:

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 them 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.

The user has some control over the memory allocation process provided that they have purchased the standard version of the product that contains run-time source code. First, the default size of a memory block as allocated by the nibble-allocation algorithm can be changed. By default, this is set to 4K bytes by the following constant in the asn1type.h header file:

```
#define XM_K_MEMBLKSIZ ((4*1024) - (sizeof(long) + sizeof(void*)))
```

The number (4*1024) can be modified to change this size (the rest of the expression is an adjustment for the size of a header that is automatically added). After modification, the run-time source code must be recompiled for this change to take effect.

The other thing that can be done is changing the algorithm all together. All memory allocation and free requests are routed through two functions: *rtMemAlloc* and *rtMemFree*. The bodies of these functions can be changed to implement whatever type of memory allocation scheme is desired. For example, embedded applications may use an operating system that does not contain malloc and free calls. So whatever is available can be included here. Another example is an extremely high-performance decoder. In this case, the nibble-allocation algorithm can be replaced with a fixed-size static block algorithm. The allocate function will split up the block, the free function will simply reset all pointers and/or indexes to make the entire block available again.

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 filled-in 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 which 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> (ASN1CTXT* ctxt_p, <name>[*] value);
```

In this definition, <name> denotes the prefixed production name defined above.

The ctxt_p 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 object 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 object argument is to be passed for a given function.

The function result variable stat returns the status of the encode operation. Status code 0 (ASN_OK) indicates the functions was successful. A negative value indicates encoding 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++ 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 = class_var.Encode ();
```

In this definition, class_var is a variable 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 ASN_OK (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' in 'Generated BER Encode Functions' 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 either calling the *pu_newContext* function (to allocate a dynamic context block), or by calling *pu_initContext* to initialize a static block. Both of these routines allow a message buffer to be specified to receive the encoded message. Specification of a message buffer is optional; if not specified, the encoder will allocate memory automatically for the encoded message. These routines also allow 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 (ASN_OK), then the message will have been encoded in the given buffer. Unlike BER, PER encoding starts from the beginning of the buffer and proceeds from left to right. 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 ()
{
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1CTXT* pCtxt;
    ASN1BOOL  aligned = TRUE;
    Employee  employee;   /* typedef generated by ASN1C */

    /* Populate employee C structure */

    employee.name.givenName = "SMITH";
    ...

    /* Allocate and initialize a new context pointer */

    pCtxt = pu_newContext (msgbuf, sizeof(msgbuf), aligned);

    if ((stat = asn1PE_Employee (pCtxt, &employee)) == ASN_OK) {
      msglen = pe_GetMsgLen (pCtxt);
      ...
    }
    else
      error processing...
```

```
    }

    free (pCtxt);  /* release the context pointer */
```

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_newContext* or *pu_initContext* 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 ()
    {
        ASN1OCTET *msgptr;
        int       msglen, stat;
        ASN1CTXT* pCtxt;
        ASN1BOOL  aligned = TRUE;
        Employee  employee;   /* typedef generated by ASN1C */

        employee.name.givenName = "SMITH";
        ...

        pCtxt = pu_newContext (0, 0, aligned);

        if ((stat = asn1PE_Employee (pCtxt, &employee)) == ASN_OK) {
          msgptr = pe_GetMsgPtr (pCtxt, &msglen);
          ...

        }
        else
          error processing...
}
```

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 (ASN_OK) 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 ASN1CPP

main ()
{
    const ASN1OCTET* msgptr;
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1BOOL  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 ()) == ASN_OK)
    {
       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. Dynamic buffer allocation is specified by using the form of the ASN1PEREncodeBuffer constructor that does not take a buffer address and size as an argument. This constructor only requires the aligned 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 ()
{
    ASN1OCTET *msgptr;
    int       msglen, stat;
    ASN1BOOL  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

    employee.msgData.name.givenName = "SMITH";
    ...

    // Encode

    if ((stat = employee.Encode ()) == ASN_OK)
    {
       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;
      }
```

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> (ASN1CTXT* ctxt_p, <name>* pvalue);
```

In this definition, <name> denotes the prefixed production name defined above.

The ctxt_p 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 (ASN_OK) 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 = class_var.Decode ();
```

In this definition, class_var is a variable of the class generated for the given production.

An ASN1PERDecodeBuffer object must be passed to the class_var constructor prior to decoding. This is where the start address of the message to be decoded and message 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 (ASN_OK) if decoding is successful or a negative value if an error occurs. Return status values are defined in Appendix A of this document and online in the "asn1type.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.

There are three steps to calling a compiler-generated decode function:

1. Prepare a context variable for decoding
2. Call the appropriate compiler-generated decode function to decode the message
3. 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 *pu_newContext* function (to allocate a dynamic context block), or by calling *pu_initContext* to initialize a static block. Both of these routines allow a message buffer that contains a PER-encoded message to be specified. These routines also allow for the specification of aligned or unaligned decoding.

A decode function can then be called to decode the message. If the return status indicates success (ASN_OK), 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 *pu_initContext*), or dynamic (created using *pu_newContext*). The function to free the context is *pu_freeContext*.

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

```
#include employee.h         /* include file generated by ASN1C */

main ()
{
    ASN1OCTET msgbuf[1024];
    ASN1TAG   msgtag;
    int       msglen, stat;
    ASN1CTXT  ctxt;
    ASN1BOOL  aligned = TRUE;
    PersonnelRecord employee;

    .. logic to read message into msgbuf ..

    /* This example uses a static context block */

    /* step 1: prepare the context block */

    pu_initContext (&ctxt, msgbuf, msglen, aligned);
```

```
        /* step 2: decode the record */

        stat = asn1PD_PersonnelRecord (&ctxt, &employee);

        if (stat == ASN_OK)
        {
            process received data..
        }
        else {
            /* error processing... */
            rtErrPrint (&ctxt);
        }

        /* step 3: free the context */

        pu_freeContext (&ctxt);
}
```

Procedure for Using the C++ Control Class Encode 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 (ASN_OK) 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 ASN1CPP

main ()
{
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1BOOL  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 == ASN_OK)
    {
        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)..

}
```

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 ()
{
    ASN1OCTET msgbuf[1024];
    int       msglen, stat;
    ASN1BOOL  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 == ASN_OK)
            {
                process received data..
            }
            else {
                // error processing..
                decodeBuffer.PrintErrorInfo ();
            }

            // step 6: free dynamic memory

            employee.memFreeAll ();
        }

    }
```

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 the section of the same name in the BER Decode Functions section for a discussion of memory management performance issues. All of those issues that apply to BER and DER also apply to PER as well.

Generated Print Functions

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

If no output file is specified with the –print 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>Print.c. If an output filename is specified after the –print qualifier, all functions are written to this 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 function is as follows:

```
asn1Print_<name> (ASN1ConstCharPtr 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.

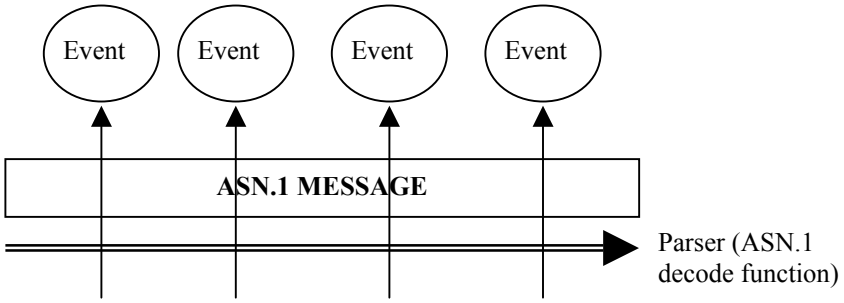
If C++ code generation is specified, a Print method is added to the ASN1C control class for the type. This method takes only a name argument; the pvalue argument is obtained from the msgData reference contained within the class.

Event Handler Interface

The –events command line switch causes hooks for user-defined event handlers to be inserted into the generated decoded functions. This feature is only available when C++ code generation is being done. 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.

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 follows:



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 (ASN1CTXT* 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 (ASN1BOOL 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");
}
```

All that each of the various value methods have to do is print a stringified representation of the value out to stdout. For example, the *intValue* callback would just print an integer value:

```
void PrintHandler::intValue (int value)
{
   indent();
   printf ("%d\n", value);
}
```

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 fault-tolerance – 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 ASN_OK. 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 (ASN1CTXT* 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 encounter 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 ASN_OK 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 (ASN1CTXT* 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");
      xu_perror (pCtxt);
      printf ("\n");

      // Skip element

      xd_NextElement (pCtxt);

      // Return an OK status to indicate parsing can continue

      return ASN_OK;
   }

   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.

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.

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.

ASN1C90

The ASN1C90 version of the compiler is a separate executable that contains extensions to handle the older 1990 version of ASN.1. Although this version is no longer supported by the ITU-T, it is still in use today. This version of the compiler also 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 ASN1C90 version of the compiler 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 error value is specified
  LinkedOperationNames    ::= OperationList | empty
  OperationList           ::= Operation | OperationList "," Operation
  Operation               ::= value(OPERATION)    -- shall reference an operation value
                              | type              -- shall reference an operation type
                                                  -- if no operation 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.

What the compiler does with this definition is uses it 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 ASN1C90 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:

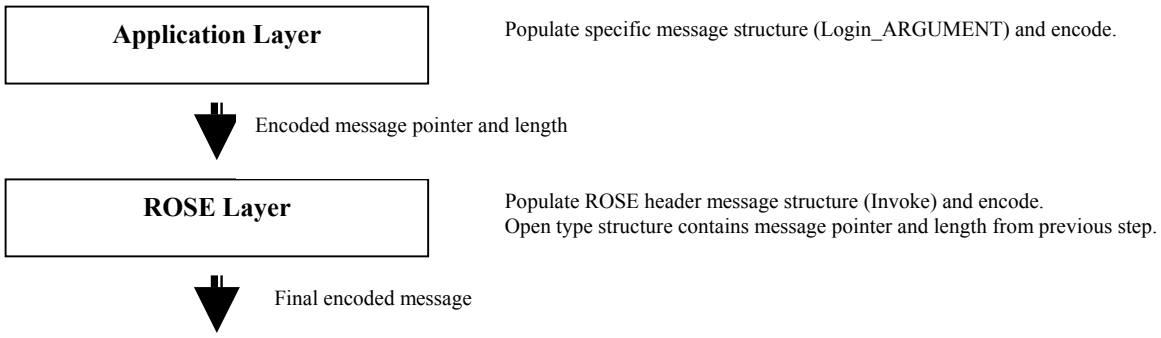
1. At the application layer, the Login_ARGUMENT structure would be populated with the username and password to be encoded.

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:



On the decode side, the process would be reversed with the message flowing up the stack:

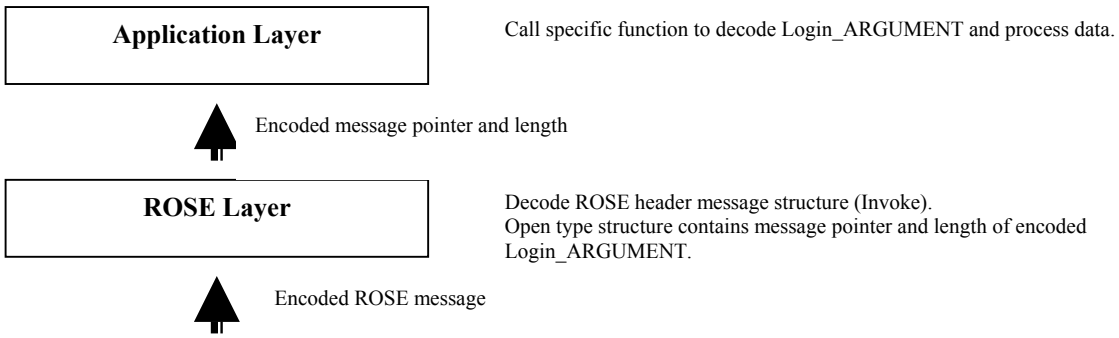
1. At the ROSE layer, the header would be decoded producing information on the OPERATION type (based on the MACRO definition) and message type (Invoke, Result, etc..). The invoke identifier would also be available for use in session management. In our example, we would know at this point that we got 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 application-specific 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:



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 ASN1C90 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 OBJECT-TYPE 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 }
} ;
```

ASN.1 C++ Run-time Classes

The ASN.1 C++ run-time classes are wrapper classes that provide an object-oriented interface to the ASN.1 C run-time library functions. The following base classes form the foundation on which a set of derived classes are built:

- The 'ASN1Context' class wraps the C ASN1CTXT structure that encapsulates all global data used in the encode/decode process.
- The 'ASN1MessageBuffer' class is the base class for encapsulating message buffers. From this, BER/DER and PER encode and decode message buffer classes are derived.
- The 'ASN1CType' class is the base class from which all compiler-generated ASN.1 production classes are derived.
- The 'Asn1NamedEventHandler' class is the base class from which custom event handler classes are derived.
- The 'Asn1ErrorHandler' class is the base class from which custom error handler class are derived.

ASN1Context

This class wraps an ASN.1 context variable. It is implemented to be a reference counted class to allow the ASN1MessageBuffer and ASN1CType classes to share a single ASN1Context instance. Its purpose is to maintain context information on an encode/decode operation across different function invocations.

In general, a user will have no need for direct use of this class. Objects are constructed from it and used internally inside the message buffer and type base classes.

ASN1Context::ASN1Context

The constructor initializes the encapsulated ASN1CTXT member variable.

Input Parameters:

None

Output Parameters:

None

ASN1Context::~ASN1Context

The destructor frees all dynamic memory associated with the given context.

ASN1Context::GetPtr

This method returns a pointer to the encapsulated ASN1CTXT member variable. It can be used if direct access to the encapsulated context variable is required (for example, to make a direct call to a C run-time library function).

Calling Sequence:

```
ptr = context.GetPtr ();
```

where ‘context’ is an ASN1Context object.

Return Value:

Name	Type	Description
ptr	ASN1CTXT*	Pointer to encapsulated context structure.

Input Parameters:

None

Output Parameters:

None

ASN1Context::PrintErrorInfo

This method prints information from the error structure within the encapsulated context to the standard output (stdout).

Calling Sequence:

```
context.PrintErrorInfo ();
```

where ‘context’ is an ASN1Context object.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1MessageBuffer

This in an abstract base class from which the ASN1BEREncodeBuffer, ASN1BERDecodeBuffer, ASN1PEREncodeBuffer, and ASN1PERDecodeBuffer classes are derived. This class allows for the management of buffer pointers and lengths used in the encoding/decoding of ASN.1 messages. A user must declare a variable of one of these derived classes prior to using a compiler generated encode/decode class. This is because a reference to an ASN1MessageBuffer object is a required argument to the constructor of the ASN1C_<ProdName> generated class.

This base class defines the following public methods:

ASN1MessageBuffer::addEventHandler

This method is used to register an event handler object. This is an object derived from the *Asn1NamedEventHandler* base class that contains custom event handler callback methods. See the section on Event Handlers for further details. Each time this method is invoked, the specified event handler object is added to the list of registered handlers.

Calling Sequence:

```
messageBuffer.addEventHandler (pEventHandler);
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

None

Input Parameters:

Name	Type	Description
pEvent Handler	Asn1Named EventHandler*	Pointer to an object of a class derived from the Asn1NamedEventHandler base class.

Output Parameters:

None

ASN1MessageBuffer::CStringToBMPString

This method converts a standard 8-bit null-terminated C string into a 16-bit character string.

Calling Sequence:

```
bmpString = messageBuffer.CStringToBMPString (cstring,
                                               pBMPString,
                                               pCharSet);
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

Name	Type	Description
bmpString	ASN1BMP String*	Pointer to the structure containing the converted string value. This pointer is equal to the *pBMPString* output parameter. This parameter specifies the buffer into which

		the converted string is to be stored.

Input Parameters:

Name	Type	Description
cstring	ASN1Const CharPtr	Pointer to C string to be converted. The ASN1ConstCharPtr type maps to a char* for C or a const char* for C++.

Output Parameters:

Name	Type	Description
pBMP String	ASN1BMP String*	Pointer to BMP string structure to receive converted string.
pCharSet	Asn116Bit CharSet*	An optional character set to filter the conversion through. Any characters not in the defined set will be discarded. By default, this argument is set to NULL (i.e., no filtering will be done).

ASN1MessageBuffer::getByteIndex

This method returns the current byte index into the encode or decode buffer.

Calling Sequence:

```
index = messageBuffer.getByteIndex ();
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

Name	Type	Description
index	int	Index to current position in the encode or decode buffer.

Input Parameters:

None

Output Parameters:

None

ASN1MessageBuffer::getContext

This method returns a pointer to the underlying ASN1Context object.

Calling Sequence:

```
ptr = messageBuffer.getContext ();
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Note that the pointer returned is to an ASN1Context object as defined above – not the ASN1CTXT structure used in calls to BER or PER C run-time library routines. The complete calling sequence to get the underlying ASN1CTXT structure is as follows:

```
ptr = messageBuffer.getContext()->GetPtr();
```

Return Value:

Name	Type	Description
ptr	OSRefCntPtr <ASN1Context>	A reference-counted pointer to an ASN1Context object. The ASN1Context object will not be released until all referenced-counted pointer variables go out of scope. This allows safe sharing of the context between the ASN1MessageBuffer and ASN1Ctype classes.

Input Parameters:

None

Output Parameters:

None

ASN1MessageBuffer::GetMsgCopy

This is a virtual method that can be overridden by derived classes to return a deep-copy of the encoded message encapsulated within the message buffer object. The base class variant returns a null pointer.

Calling Sequence:

```
ptr = messageBuffer.GetMsgCopy ();
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

Name	Type	Description
ptr	ASN1OCTET*	A pointer to a copy of the message encapsulated within the message buffer object. The base class version of this method returns a null pointer.

Input Parameters:

None

Output Parameters:

None

ASN1MessageBuffer::GetMsgPtr

This is a virtual method that can be overridden by derived classes to return a pointer to the encoded message encapsulated within the message buffer object. The base class variant returns a null pointer.

Calling Sequence:

```
ptr = messageBuffer.GetMsgPtr ();
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

Name	Type	Description
ptr	const ASN1OCTET*	A pointer to the message encapsulated within the message buffer object. The base class version of this method returns a null pointer.

Input Parameters:

None

Output Parameters:

None

ASN1MessageBuffer::Init

This is a virtual method that can be overridden by derived classes to reinitialize the underlying encode or decode buffer. The base class variant does nothing. This method is normally overridden by derived encode buffer classes to allow multiple messages to be encoded using the same message buffer object.

Calling Sequence:

```
messageBuffer.Init ();
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1MessageBuffer::isA

This is a virtual method that must be overridden by derived classes to allow identification of the class. The base class variant is abstract. This method matches an enumerated identifier defined in the base class. One identifier is declared for each of the derived classes.

Calling Sequence:

```
bool = messageBuffer.isA (ident);
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

Name	Type	Description
bool	boolean	Boolean result of the match operation. True if this is the class corresponding to the passed in identifier.

Input Parameters:

Name	Type	Description
ident	enum	Enumerated identifier specifying a derived class. This type is defined as a public access type in the ASN1MessageBuffer base class. Possible values include BEREncode, BERDecode, PEREncode, and PERDecode.

Output Parameters:

None

ASN1MessageBuffer::PrintErrorInfo

This method is used to print information on the last encode or decode error associated with the message buffer object to standard output (stdout).

Calling Sequence:

```
messageBuffer.PrintErrorInfo ();
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1MessageBuffer::setErrorHandler

This method is used to register an error handler object. This is an object derived from the *Asn1ErrorHandler* base class that contains custom error handler callback methods. See the section on Event Handlers for further details. This method sets the single allowed error handler for a given decoder. If this method is invoked multiple times, only the last error handling object specified will be registered.

Calling Sequence:

```
messageBuffer.setErrorHandler (pErrorHandler);
```

where messageBuffer is one of the ASN1Message buffer derived class objects.

Return Value:

None

Input Parameters:

Name	Type	Description
pError Handler	Asn1 ErrorHandler*	Pointer to an object of a class derived from the Asn1ErrorHandler base class.

Output Parameters:

None

ASN1BERMessageBuffer

```
ASN1MessageBuffer
  |
  +- ASN1BERMessageBuffer
```

The ASN1BERMessageBuffer class is derived from the ASN1MessageBuffer base class. It is the base class for the ASN1BEREncodeBuffer and ASN1BERDecodeBuffer derived classes. It contains variables and methods specific to encoding or decoding ASN.1 messages using the Basic Encoding Rules (BER). It is used to manage the buffer into which an ASN.1 message is to be encoded or decoded.

ASN1BERMessageBuffer::CalcIndefLen

This method calculates the actual length of an indefinite length message component.

Calling Sequence:

```
len = messageBuffer.CalcIndefLen (buf_p)
```

where messageBuffer is an ASN1BERMessageBuffer derived class object.

Return Value:

Name	Type	Description
len	int	Length (in octets) of message component.

Input Parameters:

Name	Type	Description
buf_p	ASN1OCTET*	A pointer to a message component encoded using indefinite length encoding.

Output Parameters:

None

ASN1BERMessageBuffer::BinDump

This method outputs a formatted binary dump of the current buffer contents to stdout.

Calling Sequence:

```
messageBuffer.BinDump ();
```

where messageBuffer is an ASN1BERMessageBuffer derived class object.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1BERMessageBuffer::HexDump

This method outputs a hexadecimal dump of the current buffer contents to stdout.

Calling Sequence:

```
messageBuffer.HexDump ();
```

where messageBuffer is an ASN1BERMessageBuffer derived class object.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1BEREncodeBuffer

```
ASN1MessageBuffer
  |
  +- ASN1BERMessageBuffer
       |
       +- ASN1BEREncodeBuffer
```

The ASN1BEREncodeBuffer class is derived from the ASN1BERMessageBuffer base class. It contains variables and methods specific to encoding ASN.1 messages using the Basic Encoding Rules (BER). It is used to manage the buffer into which an ASN.1 message is to be encoded.

ASN1BEREncodeBuffer::ASN1BEREncodeBuffer

The ASN1BEREncodeBuffer class has two overloaded constructors:

1. A version that takes no arguments (dynamic encoding version), and
2. A version that takes a message buffer and size argument (static encoding version)

Input Parameters:

Name	Type	Description
pMsgBuf	ASN1OCTET*	A pointer to a fixed-size message buffer to receive the encoded message.
msgBufLen	int	Size of the fixed-size message buffer.

Output Parameters:

None

ASN1BEREncodeBuffer::GetMsgCopy

This method returns a copy of the current encoded message. Memory is allocated for the message using the 'new' operation. It is the user's responsibility to free the memory using 'delete'.

Calling Sequence:

```
ptr = encodeBuffer.GetMsgCopy ();
```

where encodeBuffer is an ASN1BEREncodeBuffer object.

Return Value:

Name	Type	Description
ptr	ASN1OCTET*	Pointer to copy of encoded message. It is the user's responsibility to release the memory using the 'delete' operator (i.e., delete [] ptr;)

Input Parameters:

None

Output Parameters:

None

ASN1BEREncodeBuffer::GetMsgPtr

This method returns the internal pointer to the current encoded message.

Calling Sequence:

```
ptr = encodeBuffer.GetMsgPtr ();
```

where encodeBuffer is an ASN1BEREncodeBuffer object.

Return Value:

Name	Type	Description
ptr	const ASN1OCTET*	Pointer to encoded message.

Input Parameters:

None

Output Parameters:

None

ASN1BEREncodeBuffer::Init

This method reinitializes the encode buffer pointer to allow a new message to be encoded. This makes it possible to reuse one message buffer object in a loop to encode multiple messages. After this method is called, any previously encoded message in the buffer will be overwritten on the next encode call.

Calling Sequence:

```
encodeBuffer.Init ();
```

where encodeBuffer is an ASN1BEREncodeBuffer object.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1BERDecodeBuffer

```
ASN1MessageBuffer
  |
  +- ASN1BERMessageBuffer
       |
       +- ASN1BERDecodeBuffer
```

ASN1BERDecodeBuffer derived class. This class is derived from the ASN1BERMessageBuffer base class. It contains variables and methods specific to decoding ASN.1 messages. It is used to manage the input buffer containing the ASN.1 message to be decoded.

ASN1BERDecodeBuffer::ASN1BERDecodeBuffer

The ASN1BERDecodeBuffer constructor constructs a buffer describing an encoded ASN.1 message. Parameters describing the message to be decoded are passed as arguments.

Input Parameters:

Name	Type	Description
pMsgBuf	ASN1OCTET*	A pointer to buffer containing an encoded ASN.1 message.
msgBufLen	int	Size of the message buffer. This does not have to be equal to the length of the message. The message length can be determined from the outer tag-length-value in the message. This parameter is used to determine if the length of the message is valid; therefore it must be greater than or equal to the actual length. Typically, the size of the buffer the message was read into is passed.

Output Parameters:

None

ASN1BERDecodeBuffer::FindElement

This method finds a tagged element within a message.

Calling Sequence:

```
ptr = decodeBuffer.FindElement (tag, elemLen, firstFlag);
```

where decodeBuffer is an ASN1BERDecodeBuffer object.

Return Value:

Name	Type	Description
ptr	ASN1OCTET*	Pointer to tagged component in message or NULL if component not found.

Input Parameters:

Name	Type	Description
tag	ASN1TAG	ASN.1 tag value to search for.
firstFlag	int	Flag indicating if this the first time this search is being done. If true, internal pointers will be set to start the search from the beginning of the message. If false, the search will be resumed from the point at which the last matching tag was found.

		This makes it possible to find all instances of a particular tagged element within a message.

Output Parameters:

Name	Type	Description
len	int&	Reference to an integer value to receive the length of the found element.

ASN1BERDecodeBuffer::ParseTagLen

This method will parse the initial tag-length pair from the message.

Calling Sequence:

```
stat = decodeBuffer.ParseTagLen (tag, msglen);
```

where decodeBuffer is an ASN1BERDecodeBuffer object.

Return Value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input Parameters:

None

Output Parameters:

Name	Type	Description
tag	ASN1TAG&	Reference to a tag structure to receive the outer level tag value parsed from the message.
msglen	int&	Length of the message. This is the total length of the message obtained by adding the number of bytes in initial tag-length to the parsed length value.

ASN1PERMessageBuffer

```
ASN1MessageBuffer
  |
  +- ASN1PERMessageBuffer
```

The ASN1PERMessageBuffer class is derived from the ASN1MessageBuffer base class. It is the base class for the ASN1PEREncodeBuffer and ASN1PERDecodeBuffer derived classes. It contains variables and methods specific to encoding or decoding ASN.1 messages using the Packed Encoding Rules (PER). It is used to manage the buffer into which an ASN.1 message is to be encoded or decoded.

ASN1PERMessageBuffer::BinDump

This method outputs a binary dump of the current buffer contents to stdout.

Calling Sequence:

```
messageBuffer.BinDump ();
```

where messageBuffer is an ASN1PERMessageBuffer derived class object.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1PERMessageBuffer::HexDump

This method outputs a hexadecimal dump of the current buffer contents to stdout.

Calling Sequence:

```
messageBuffer.HexDump ();
```

where messageBuffer is an ASN1PERMessageBuffer derived class object.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1PERMessageBuffer::GetMsgLen

This method returns the length of a previously encoded PER message.

Calling Sequence:

```
len = messageBuffer.GetMsgLen ();
```

where messageBuffer is an ASN1PERMessageBuffer derived class object.

Return Value:

Name	Type	Description
len	int	Length of the PER message encapsulated within this buffer object.

Input Parameters:

None

Output Parameters:

None

ASN1PERMessageBuffer::SetTrace

This method turns PER diagnostic tracing on or off. This enables the collection of the bit statistics inside the PER library functions that can be displayed using the *BinDump* method.

Calling Sequence:

```
len = messageBuffer.SetTrace (enabled);
```

where messageBuffer is an ASN1PERMessageBuffer derived class object.

Return Value:

None

Input Parameters:

Name	Type	Description
enabled	ASN1BOOL	Boolean value indicating whether tracing should be turned on (true) or off (false).

Output Parameters:

None

ASN1PEREncodeBuffer

```
ASN1MessageBuffer
  |
  +- ASN1PERMessageBuffer
       |
       +- ASN1PEREncodeBuffer
```

The ASN1PEREncodeBuffer class is derived from the ASN1PERMessageBuffer base class. It contains variables and methods specific to encoding ASN.1 messages. It is used to manage the buffer into which an ASN.1 PER message is to be encoded.

ASN1PEREncodeBuffer::ASN1PEREncodeBuffer

The ASN1PEREncodeBuffer class has two overloaded constructors:

1. A version that takes one argument, aligned flag (dynamic encoding version), and
2. A version that takes a message buffer and size argument and an aligned flag argument (static encoding version)

Input Parameters:

Name	Type	Description
pMsgBuf	ASN1OCTET*	A pointer to a fixed-size message buffer to receive the encoded message.
msgBufLen	int	Size of the fixed-size message buffer.
aligned	ASN1BOOL	Flag indicating if aligned (TRUE) or unaligned (FALSE) encoding should be done.

Output Parameters:

None

ASN1PEREncodeBuffer::GetMsgBitCnt

This method returns the length (in bits) of the encoded message.

Calling Sequence:

```
len = encodeBuffer.GetMsgBitCnt ();
```

where encodeBuffer is an ASN1PEREncodeBuffer object.

Return Value:

Name	Type	Description
len	int	Length (in bits) of encoded message.

Input Parameters:

None

Output Parameters:

None

ASN1PEREncodeBuffer::GetMsgCopy

This method returns a copy of the current encoded message. Memory is allocated for the message using the 'new' operation. It is the user's responsibility to free the memory using 'delete'.

Calling Sequence:

```
ptr = encodeBuffer.GetMsgCopy ();
```

where encodeBuffer is an ASN1PEREncodeBuffer object.

Return Value:

Name	Type	Description
ptr	ASN1OCTET*	Pointer to copy of encoded message. It is the user's responsibility to release the memory using the 'delete' operator (i.e., delete [] ptr;)

Input Parameters:

None

Output Parameters:

None

ASN1PEREncodeBuffer::GetMsgPtr

This method returns the internal pointer to the current encoded message.

Calling Sequence:

```
ptr = encodeBuffer.GetMsgPtr ();
```

where encodeBuffer is an ASN1PEREncodeBuffer object.

Return Value:

Name	Type	Description
ptr	ASN1OCTET*	Pointer to encoded message.

Input Parameters:

None

Output Parameters:

None

ASN1PEREncodeBuffer::Init

This method reinitializes the encode buffer pointer to allow a new message to be encoded. This makes it possible to reuse one message buffer object in a loop to encode multiple messages. After this method is called, any previously encoded message in the buffer will be overwritten on the next encode call.

Calling Sequence:

```
encodeBuffer.Init ();
```

where encodeBuffer is an ASN1PEREncodeBuffer object.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1PERDecodeBuffer

```
ASN1MessageBuffer
  |
  +- ASN1PERMessageBuffer
       |
       +- ASN1PERDecodeBuffer
```

The ASN1PERDecodeBuffer class is derived from the ASN1PERMessageBuffer base class. It contains variables and methods specific to decoding ASN.1 messages. It is used to manage the input buffer containing the ASN.1 message to be decoded.

The only method associated with this class is the following constructor:

ASN1PERDecodeBuffer::ASN1PERDecodeBuffer

This constructor is used to describe the message to be decoded.

Input Parameters:

Name	Type	Description
pMsgBuf	ASN1OCTET*	Pointer to the message to be decoded.
msgBufLen	int	Length of the message buffer.
aligned	ASN1BOOL	Flag indicating if message was encoded using aligned (TRUE) or unaligned (FALSE) encoding.

Output Parameters:

None

ASN1CType

The ASN1CType base class. This is the class from which all class definitions generated by the ASN.1 compiler are (eventually) derived. In some cases, the generated type may be derived from an intermediate class which in turn is derived from the *ASN1CType* class. This class contains a single constructor that allows a message buffer object to be specified. It also contains abstract virtual prototypes for Encode and Decode methods. These functions are implemented in the derived classes generated by the compiler.

ASN1CType::ASN1CType

This constructor is used to set up a message buffer object to either receive the data of a message being encoded or to specify a message to be decoded.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).

Output Parameters:

None

ASN1CType::Encode

This virtual method encodes a message of the given type.

Calling Sequence:

```
stat = asn1TypeVar.Encode ();
```

where asn1TypeVar is an object of a compiler-generated class ASN.1 production class.

Return Value:

Name	Type	Description
stat	int	Status of the encode operation. For BER, a positive value indicates success (it is also the length of the encoded message). For PER, ASN_OK is returned if encoding is successful. In either case, if encoding fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CType::Decode

This virtual method decodes a message of the given type.

Calling Sequence:

```
stat = asn1TypeVar.Decode ();
```

where asn1TypeVar is an object of a compiler-generated class ASN.1 production class.

Return Value:

Name	Type	Description
stat	int	Status of the decode operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CType::memAlloc

This method allocates memory using the underlying rtMemAlloc function. This function uses the compiler's nibble memory allocation scheme to improve performance.

The allocated memory is owned by the enveloping context object. This object is shared between the message buffer and type objects using reference counting. This means the allocated memory will automatically be released when both the message buffer and type objects are destroyed or go out of scope.

Calling Sequence:

```
ptr = asn1TypeVar.memAlloc (numocts);
```

where asn1TypeVar is an object of a compiler-generated class ASN.1 production class.

Return Value:

Name	Type	Description
ptr	void*	Pointer to allocated memory block

Input Parameters:

Name	Type	Description
numocts	int	Number of octets (bytes) to allocate.

Output Parameters:

None

ASN1CType::memFreeAll

This method frees all memory allocated within the encapsulated context. This includes all memory allocated by the decoder as well as memory allocated by the user using the *xu_malloc, rtMemAlloc,* or *ASN1CType::memAlloc* functions.

Normally, this memory is released automatically when the message buffer and ASN1C control objects are deleted or go out of scope. However, there are times when memory must be manually released. An example is when decoder objects are reused in a decoding loop. After decoding and processing on a given message is complete, this method should be called to free all memory that was used.

Calling Sequence:

```
asn1TypeVar.memFreeAll ();
```

where asn1TypeVar is an object of a compiler-generated class ASN.1 production class.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1CBitStr

```
ASN1Type
  |
  +- ASN1CBitStr
```

The ASN1CBitStr class is derived from the ASN1CType base class. It is used as the base class for generated control classes for the ASN.1 BIT STRING type. This class provides utility methods for operating on the bit string referenced by the generated class. This class can also be used inline to operate on the bits within generated BIT STRING elements in a SEQUENCE, SET, or CHOICE construct.

ASN1CBitStr::ASN1CBitStr

There are a number of different constructors available for this object. The different types are as follows:

```
ASN1CBitStr (ASN1MessageBuffer& msgBuf, ASN1UINT nbits);
```

This constructor creates an empty bit string. If number of bits equals zero then the bit string is dynamic; otherwise the capacity will be fixed to the given number of bits.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
nbits	ASN1UINT	Bit string capacity. If zero, string is dynamic as opposed to fixed-size.

Output Parameters:

None

```
ASN1CBitStr (ASN1MessageBuffer& msgBuf, ASN1OCTET* bitStr,
             ASN1UINT& numbits, ASN1UINT maxNumbits);
```

This constructor creates a bit string from an array of bits. It does not deep-copy bytes, it just assigns the passed array to an internal reference variable. This from of the constructor is normally used with static bit strings (i.e. those containing fixed-size arrays as a result of a size constraint being placed on the string).

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
bitStr	ASN1OCTET*	Pointer to static byte array.
numbits	ASN1UINT	Reference to length of bit string (in bits).
maxNumbits	ASN1UINT	Maximum length of string in bits.

Output Parameters:

None

```
ASN1CBitStr (ASN1MessageBuffer& msgBuf, ASN1TDynBitStr& bitStr);
```

This constructor creates a bit string using the *ASN1TDynBitStr* argument. The constructor does not deep-copy the variable, it assigned a reference to it to an internal variable. The object will then directly operate on the given data variable. This for of the constructor is used with a compiler-generated dynamic bit string variable (i.e. one that is not sized).

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
bitStr	ASN1TDynBitStr&	Reference to a dynamic bit string structure.

Output Parameters:

None

```
ASN1CBitStr (const ASN1CBitStr& bitStr);
```

This is the copy constructor. This will create a deep-copy of the given variable.

```
ASN1CBitStr (const ASN1CBitStr& bitStr, ASN1BOOL extendable);
```

A second form of the copy constructor. This form can be used to mark the copied string as 'extendable' meaning it can grow dynamically if additional bits are added.

ASN1CBitStr::change

```
inline int change (ASN1UINT bitIndex, ASN1BOOL value);
```

This method changes the value of the bit at the given index to the given value.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitIndex	ASN1UINT	Relative index of bit to set in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).
value	ASN1BOOL	Boolean value to which the bit is to be set.

Output Parameters:

None

ASN1CBitStr::clear

There are a number of different overloaded versions of the bit string clear method for clearing bits in the target bit string variable. They are as follows:

```
int clear (ASN1UINT bitIndex);
```

This version of the clear method sets the given bit in the target string to zero.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitIndex	ASN1UINT	Relative index of bit in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).

Output Parameters:

None

```
int clear (ASN1UINT fromIndex, ASN1UINT toIndex);
```

This version of the clear method sets the bits from the specified fromIndex (inclusive) to the specified toIndex (exclusive) to zero.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
fromIndex	ASN1UINT	Relative start index (inclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).
toIndex	ASN1UINT	Relative end index (exclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress

		from left to right (MS to LS bits).

Output Parameters:

None

```
int clear ();
```

This version of the clear method sets all bits in the bit string to zero.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CBitStr::set

There are a number of different overloaded versions of the bit string set method for setting bits in the target bit string variable. They are as follows:

```
int set (ASN1UINT bitIndex);
```

This version of the set method sets the given bit in the target string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitIndex	ASN1UINT	Relative index of bit to set in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).

Output Parameters:

None

```
int set (ASN1UINT fromIndex, ASN1UINT toIndex);
```

This version of the set method sets the bits from the specified fromIndex (inclusive) to the specified toIndex (exclusive) to one.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
fromIndex	ASN1UINT	Relative start index (inclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).
toIndex	ASN1UINT	Relative end index (exclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).

Output Parameters:

None

ASN1CBitStr::invert

There are a number of different overloaded versions of the bit string invert method for inverting bits in the target bit string variable. All zero bits in the bit string will be set to '1', all '1' bits will be set to '0'. The overloaded methods are as follows:

```
int invert (ASN1UINT bitIndex);
```

This version of the invert method inverts the given bit in the target string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitIndex	ASN1UINT	Relative index of bit to set in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).

Output Parameters:

None

```
int invert (ASN1UINT fromIndex, ASN1UINT toIndex);
```

This version inverts the bits from the specified fromIndex (inclusive) to the specified toIndex (exclusive) to one.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
fromIndex	ASN1UINT	Relative start index (inclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).
toIndex	ASN1UINT	Relative end index (exclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).

Output Parameters:

None

ASN1CBitStr::get

There are a number of different overloaded versions of the bit string get method for getting bits from the target bit string variable. They are as follows:

```
ASN1BOOL get (ASN1UINT bitIndex);
```

This method returns the value of the bit with the specified index.

Return Value:

Name	Type	Description
bit	ASN1BOOL	TRUE, if bit at specified index is set to '1', FALSE else.

Input Parameters:

Name	Type	Description
bitIndex	ASN1UINT	Relative index of bit to set in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).

Output Parameters:

None

```
int get (ASN1UINT fromIndex, ASN1UINT toIndex,
         ASN1OCTET* pBuf, int bufSz);
```

This version of the get method copies the bit string composed of bits from this bit string from the specified fromIndex (inclusive) to the specified toIndex (exclusive) into given buffer.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
fromIndex	ASN1UINT	Relative start index (inclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).
toIndex	ASN1UINT	Relative end index (exclusive) of bits in string. Bit index 0 refers to the MS bit (bit 8) in the first octet. The index values then progress from left to right (MS to LS bits).
bufSz	int	Size of given destination buffer. If size of buffer is not enough to receive whole bit string negative status will be returned.

Output Parameters:

Name	Type	Description
pBuf	ASN1OCTET*	Pointer to destination buffer, where bytes will be copied.

ASN1CBitStr::isSet

```
inline ASN1BOOL isSet (ASN1UINT bitIndex);
```

This method is the same as ASN1CBitStr::get.

ASN1CBitStr::isEmpty

```
ASN1BOOL isEmpty ();
```

This method returns TRUE if this bit string contains no bits that are set to '1'.

Return Value:

Name	Type	Description
empty	ASN1BOOL	TRUE, if this bit string contains no bits that are set to '1'.

Input Parameters:

None

Output Parameters:

None

ASN1CBitStr::size

```
int size () const;
```

This method returns the number of bytes of space actually in use by this bit string to represent bit values.

Return Value:

Name	Type	Description
size	int	Number of bytes of space actually in use by this bit string to represent bit values.

Input Parameters:

None

Output Parameters:

None

ASN1CBitStr::length

```
ASN1UINT length () const;
```

This method calculates the "logical size" of this bit string: the index of the highest set bit in the bit string plus one. Returns zero if the bit string contains no set bits. Highest bit in the bit string is the LS bit in the last octet set to ‘1’.

Return Value:

Name	Type	Description
len	ASN1UINT	Returns the "logical size" of this bit string.

Input Parameters:

None

Output Parameters:

None

ASN1CBitStr::cardinality

```
int cardinality () const;
```

This method calculates the cardinality of target bit string. Cardinality of the bit string is the number of bits set to '1'.

Return Value:

Name	Type	Description
num	Int	Number of bytes of space actually in use by this bit string to represent bit values.

Input Parameters:

None

Output Parameters:

None

ASN1CBitStr::getBytes

```
int getBytes (ASN1OCTET* pBuf, int bufSz);
```

This method copies the bit string to the given buffer.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bufSz	int	Size of given destination buffer. If size of buffer is not enough to receive whole bit string negative status will be returned.

Output Parameters:

Name	Type	Description
pBuf	ASN1OCTET*	Pointer to destination buffer, where bytes will be copied.

ASN1CBitStr::doAnd

There are a number of different overloaded versions of the bit string doAnd method for performing a logical AND of this target bit string with the argument bit string. They are as follows:

```
int doAnd (const ASN1OCTET* pOctstr, ASN1UINT octsNumbits);
```

This method performs a logical AND of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
pOctstr	ASN1OCTET*	A pointer to octets of another bit string for performing logical operation.
octsNumbits	ASN1UINT	A number of bits in argument bit string.

Output Parameters:

None

```
inline int doAnd (const ASN1TDynBitStr& bitStr);
```

This method performs a logical AND of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitStr	ASN1TDynBitStr&	A reference to another bit string represented by ASN1TDynBitStr type for performing logical operation.

Output Parameters:

None

```
inline int doAnd (const ASN1CBitStr& bitStr);
```

This method performs a logical AND of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description

bitStr	ASN1CBitStr&	A reference to another bit string represented by ASN1CBitStr for performing logical operation.

Output Parameters:

None

ASN1CBitStr::doOr

There are a number of different overloaded versions of the bit string doOr method for performing a logical OR of this target bit string with the argument bit string. They are as follows:

```
int doOr (const ASN1OCTET* pOctstr, ASN1UINT octsNumbits);
```

This method performs a logical OR of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
pOctstr	ASN1OCTET*	A pointer to octets of another bit string for performing logical operation.
octsNumbits	ASN1UINT	A number of bits in argument bit string.

Output Parameters:

None

```
inline int doOr (const ASN1TDynBitStr& bitStr);
```

This method performs a logical OR of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitStr	ASN1TDynBitStr&	A reference to another bit string represented by ASN1TDynBitStr type for performing logical operation.

Output Parameters:

None

```
inline int doOr (const ASN1CBitStr& bitStr);
```

This method performs a logical OR of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitStr	ASN1CBitStr&	A reference to another bit string represented by ASN1CBitStr for performing logical operation.

Output Parameters:

None

ASN1CBitStr::doXor

There are a number of different overloaded versions of the bit string doXor method for performing a logical XOR of this target bit string with the argument bit string. They are as follows:

```
int doXor (const ASN1OCTET* pOctstr, ASN1UINT octsNumbits);
```

This method performs a logical XOR of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
pOctstr	ASN1OCTET*	A pointer to octets of another bit string for performing logical operation.
octsNumbits	ASN1UINT	A number of bits in argument bit string.

Output Parameters:

None

```
inline int doXor (const ASN1TDynBitStr& bitStr);
```

This method performs a logical XOR of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitStr	ASN1TdynBitStr&	A reference to another bit string represented by ASN1TDynBitStr type for performing logical operation.

Output Parameters:

None

```
inline int doXor (const ASN1CBitStr& bitStr);
```

This method performs a logical XOR of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitStr	ASN1CBitStr&	A reference to another bit string represented by ASN1CBitStr for performing logical operation.

Output Parameters:

None

ASN1CBitStr::doAndNot

There are a number of different overloaded versions of the bit string doAndNot method for performing a logical ANDNOT of this target bit string with the argument bit string. Logical ANDNOT clears all of the bits in this bit string whose corresponding bit is set in the specified bit string. These methods are as follows:

```
int doAndNot (const ASN1OCTET* pOctstr, ASN1UINT octsNumbits);
```

This method performs a logical ANDNOT of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
pOctstr	ASN1OCTET*	A pointer to octets of another bit string for performing logical operation.
octsNumbits	ASN1UINT	A number of bits in argument bit string.

Output Parameters:

None

```
inline int doAndNot (const ASN1TDynBitStr& bitStr);
```

This method performs a logical ANDNOT of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	Int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
bitStr	ASN1TDynBitStr&	A reference to another bit string represented by ASN1TDynBitStr type for performing logical operation.

Output Parameters:

None

```
inline int doAndNot (const ASN1CBitStr& bitStr);
```

This method performs a logical ANDNOT of the target bit string with the argument bit string.

Return Value:

Name	Type	Description
stat	Int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description

bitStr	ASN1CBitStr&	A reference to another bit string represented by ASN1CBitStr for performing logical operation.

Output Parameters:

None

ASN1CBitStr::shiftLeft

```
int shiftLeft (ASN1UINT shift);
```

This method shifts all bits to the left by the number of bits specified in the 'shift' operand. If bit string can dynamically grow, then the length of bit string will be decreased by ‘shift’ bits. Otherwise, shifted in bits are filled by zeros from the right. Most left bits are lost.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
shift	int	Number of bits to be shifted.

Output Parameters:

None

ASN1CBitStr::shiftRight

```
int shiftRight (ASN1UINT shift);
```

This method shifts all bits to the right by the number of bits specified in the 'shift' operand. If the bit string can dynamically grow, then the length of the bit string will be increased by ‘shift’ bits. Otherwise, shifted in bits are lost. The leftmost bits are filled by zeros.

Return Value:

Name	Type	Description
stat	int	Status of the operation. ASN_OK is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

Name	Type	Description
shift	int	Number of bits to be shifted.

Output Parameters:

None

ASN1CBitStr::unusedBitsInLastUnit

```
int unusedBitsInLastUnit ();
```

This method returns the number of unused bits in the last octet.

Return Value:

Name	Type	Description
num	int	Number of bits in the last octet. It equals to length() % 8.

Input Parameters:

None

Output Parameters:

None

ASN1CBitStr::operator ASN1TDynBitStr

There are a number of different overloaded versions of the cast operator for performing a casting of the target bit string to an ASN1TDynBitStr variable. These operators are as follows:

```
operator ASN1TDynBitStr();
```

This method returns a filled ASN1TDynBitStr. Memory is not allocated, only a pointer is assigned. Thus, the ASN1TDynBitStr variable is only valid while this ASN1CBitStr is in scope.

Return Value:

Name	Type	Description
bitstr	`ASN1TDynBitStr`	Filled ASN1TdynBitStr.

Input Parameters:

None

Output Parameters:

None

```
operator ASN1TdynBitStr*();
```

This method returns a pointer to the filled ASN1TDynBitStr. Memory for the ASN1TDynBitStr variable is allocated using memAlloc and bits are copied into it.

Return Value:

Name	Type	Description
bitstr	`ASN1TdynBitStr*`	Pointer to filled ASN1TdynBitStr.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList

```
ASN1Type
  |
  +- ASN1CSeqOfList
```

The ASN1CSeqOfList class is derived from the ASN1CType base class. It is used as the base class for generated control classes for the ASN.1 SEQUENCE OF or SET OF type. This class provides utility methods for operating on the linked list referenced by the generated class. This class can also be used inline to operate on the linked lists within generated SEQUENCE OF or SET OF elements in a SEQUENCE, SET, or CHOICE construct.

ASN1CSeqOfList:: ASN1CSeqOfList

There are a number of different constructors available for this object. The different types are as follows:

```
ASN1CSeqOfList (ASN1MessageBuffer& msgBuf,
                Asn1RTDList& lst,
                ASN1BOOL initBeforeUse = TRUE);
```

This constructor creates a linked list using the Asn1RTDList argument . The constructor does not deep-copy the variable, it assigned a reference to it to an internal variable. The object will then directly operate on the given list variable. This constructor is used with a compiler-generated linked list variable.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
lst	Asn1RTDList	Reference to a linked list structure.
initBeforeUsed	ASN1BOOL	Set to TRUE if the passed linked list needs to be initialized (rtDListInit to be called).

Output Parameters:

None

```
ASN1CSeqOfList (ASN1MessageBuffer& msgBuf);
```

This constructor creates an empty linked list.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).

Output Parameters:

None

ASN1CSeqOfList::append

```
void append (void* data);
```

This method appends an item to the linked list. The item is represented by a void pointer that can point to an object of any type. The rtMemAlloc function is used to allocate memory for the list node structure, therefore all internal list memory will be released whenever rtMemFree is called.

Return Value:

None

Input Parameters:

Name	Type	Description
pData	void*	Pointer to data item to be appended to the list.

Output Parameters:

None

ASN1CSeqOfList::insert

```
void insert (int index, void* pData);
```

This method inserts an item into the linked list structure. The item is represented by a void pointer that can point to an object of any type. The rtMemAlloc function is used to allocate memory for the list node structure. All internal list memory will be released when the rtMemFree function is called.

Return Value:

None

Input Parameters:

Name	Type	Description
index	int	Index at which the specified item is to be inserted.
pData	void*	Pointer to data item to be appended to the list.

Output Parameters:

None

ASN1CSeqOfList::remove

There are a number of different overloaded versions of the linked list remove method for removing nodes from the target linked list variable. They are as follows:

```
void remove (int index);
```

This method removes a node at specified index from the linked list structure. The *rtMemAlloc* function was used to allocate the memory for the list node structure, therefore, all internal list memory will be released whenever *rtMemFree* is called.

Return Value:

None

Input Parameters:

Name	Type	Description
index	int	Index of item to be removed.

Output Parameters:

None

```
void remove (void* pData);
```

This method removes the first occurrence of the node with specified data from the linked list structure. The *rtMemAlloc* function was used to allocate the memory for the list node structure, therefore, all internal list memory will be released whenever *rtMemFree* is called.

Return Value:

None

Input Parameters:

Name	Type	Description
pData	void*	Pointer to data item to be removed from the list.

Output Parameters:

None

ASN1CSeqOfList::removeFirst

```
inline void removeFirst ();
```

This method removes the first node (head) from the linked list structure.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::removeLast

```
inline void removeLast ();
```

This method removes the last node (tail) from the linked list structure.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::indexOf

```
int indexOf (void* pData);
```

This method returns the index in this list of the first occurrence of the specified item, or -1 if the list does not contain the item.

Return Value:

Name	Type	Description
index	`int`	The index in this list of the first occurrence of the specified item, or -1 if the list does not contain the item.

Input Parameters:

Name	Type	Description
pData	void*	Pointer to data item to search for.

Output Parameters:

None

ASN1CSeqOfList::contains

```
ASN1BOOL contains (void* pData);
```

This method returns TRUE if this list contains the specified item.

Return Value:

Name	Type	Description
val	`ASN1BOOL`	TRUE if this list contains the specified item.

Input Parameters:

Name	Type	Description
pData	void*	Pointer to data item whose presence in this list is to be tested.

Output Parameters:

None

ASN1CSeqOfList::getFirst

```
void* getFirst ();
```

This method returns the first item from the list or null if there are no elements in the list.

Return Value:

Name	Type	Description
pData	void*	The first item in this list.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::getLast

```
void* getLast ();
```

This method returns the last item from the list or null if there are no elements in the list.

Return Value:

Name	Type	Description
pData	void*	The last item in this list.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::get

```
void* get (int index);
```

This method returns the item at the specified position in the list.

Return Value:

Name	Type	Description

pData	void*	The item at the specified index in the list.

Input Parameters:

Name	Type	Description
index	int	Index of item to be returned.

Output Parameters:

None

ASN1CSeqOfList::operator[]

```
inline void* operator[] (int index) const;
```

This method is the overloaded operator [ ]. It returns the item at the specified position in this list. See the section on the get method for further details.

ASN1CSeqOfList::set

```
void* set (int index, void* pData);
```

This method replaces the item at the specified index in this list with the specified item.

Return Value:

Name	Type	Description
pOldData	void*	The item previously at the specified position.

Input Parameters:

Name	Type	Description
index	int	Index of item to replace.
pData	void*	The item to be stored at the specified index.

Output Parameters:

None

ASN1CSeqOfList::clear

```
void clear ();
```

This method removes all items from the list.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::isEmpty

```
ASN1BOOL isEmpty () const;
```

This method returns TRUE if the list is empty.

Return Value:

Name	Type	Description
val	`ASN1BOOL`	TRUE if this list is empty.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::size

```
int size () const;
```

This method returns the number of nodes in the list.

Return Value:

Name	Type	Description
size	`int`	The number of items in this list.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::iterator

```
ASN1CSeqOfListIterator* iterator ();
```

This method returns an iterator over the elements in this linked list in the sequence from the first to the last. See ASN1CSeqOfListIterator for more details.

Return Value:

Name	Type	Description
iterator	ASN1CSeqOf ListIterator	The iterator over this linked list.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::iteratorFromLast

```
ASN1CSeqOfListIterator* iteratorFromLast ();
```

This method returns a reverse iterator over the elements in this linked list in the sequence from the last to the first. See ASN1CSeqOfListIterator for more details.

Return Value:

Name	Type	Description
iterator	ASN1CSeqOf ListIterator	The reverse iterator over this linked list.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfList::iteratorFrom

```
ASN1CSeqOfListIterator* iteratorFrom (void* pData);
```

This method returns an iterator over the elements in this linked list starting from the specified item in the list. See ASN1CSeqOfListIterator for more details.

Return Value:

Name	Type	Description
iterator	ASN1CSeqOf ListIterator	The iterator over this linked list.

Input Parameters:

Name	Type	Description
pData	void*	The item of the list to be iterated first.

Output Parameters:

None

ASN1CSeqOfListIterator

The ASN1CSeqOfListIterator class is an iterator for linked lists (represented by ASN1CSeqOfList) that allows the programmer to traverse the list in either direction and modify the list during iteration. The iterator is *fail-fast*. This means if the list is structurally modified at any time after the ASN1CSeqOfListIterator class is created, in any way except through the iterator's own *remove* or *insert* methods, the iterator's methods *next* and *prev* will return NULL. The *remove*, *set* and *insert* methods will return the ASN_E_CONCMODF error code.

ASN1CSeqOfListIterator::hasNext

```
inline ASN1BOOL hasNext ();
```

This method returns TRUE, if this iterator has more elements when traversing the list in the forward direction. (In other words, returns TRUE, if *next* would return an element rather than returning a null value).

Return Value:

Name	Type	Description
stat	ASN1BOOL	TRUE, if *next* would return an element rather than returning a null value.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfListIterator::hasPrev

```
inline ASN1BOOL hasPrev ();
```

This method returns TRUE, if this iterator has more elements when traversing the list in the reverse direction. (In other words, returns TRUE, if *prev* would return an element rather than returning a null value).

Return Value:

Name	Type	Description
stat	ASN1BOOL	TRUE, if *prev* would return an element rather than returning a null value.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfListIterator::next

```
void* next ();
```

This method returns the next element in the list. This method may be called repeatedly to iterate through the list, or intermixed with calls to *prev* to go back and forth.

Return Value:

Name	Type	Description
item	void*	The next element in the list. A null value will be returned if iteration is not successful.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfListIterator::prev

```
void* prev ();
```

This method returns the previous element in the list. This method may be called repeatedly to iterate through the list, or intermixed with calls to *next* to go back and forth.

Return Value:

Name	Type	Description
item	void*	The previous element in the list. A null value will be returned if iteration is not successful.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfListIterator::remove

```
int remove ();
```

This method removes from the list the last element that was returned by *next* or *prev* methods. This call can only be made once per call to *next* or *prev* methods.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

None

Output Parameters:

None

ASN1CSeqOfListIterator::set

```
int set (void* pData);
```

This method replaces the last element returned by *next* or *prev* methods with the specified element. This call can be made only if neither *remove* nor *insert* methods have been called after the last call to *next* or *prev* methods.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
pData	void*	The element with which to replace the last element returned by *next* or *prev* methods.

Output Parameters:

None

ASN1CSeqOfListIterator::insert

```
int insert (void* pData);
```

This method inserts the specified element into the list. The element is inserted immediately before the next element that would be returned by *next* method, if any, and after the next element that would be returned by *prev* method, if any. (If the list contains no elements, the new element becomes the sole element on the list). The new element is inserted before the implicit cursor: a subsequent call to *next* would be unaffected, and a subsequent call to *prev* would return the new element.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description

pData	void*	The element to be inserted.

Output Parameters:

None

ASN1CTime

```
ASN1Type
  |
  +- ASN1CTime
```

The ASN1CTime class is derived from the ASN1CType base class. It is used as the abstract base class for generated control classes for the ASN.1 Generalized Time ([UNIVERSAL 24] IMPLICIT VisibleString) and Universal Time ([UNIVERSAL 23] IMPLICIT VisibleString) types. This class provides utility methods for operating on the time information referenced by the generated class. This class can also be used inline to operate on the times within generated time string elements in a SEQUENCE, SET, or CHOICE construct. The time strings are generally formatted according to ISO 8601 format with some exceptions (see X.680).

ASN1CTime::ASN1CTime

There are a number of different constructors available for this object. The different types are as follows:

```
ASN1CTime (ASN1MessageBuffer& msgBuf, char*& buf, int bufSize);
```

This constructor creates a time string from buffer. It does not deep-copy the data; it just assigns the passed array to an internal reference variable. The object will then directly operate on the given data variable.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
buf	char*	Reference to pointer to time string buffer.
bufSize	int	Size of passed buffer, in bytes.

Output Parameters:

None

```
ASN1CTime (ASN1MessageBuffer& msgBuf, ASN1VisibleString& buf);
```

This constructor creates a time string using the *ASN1VisibileString* argument. The constructor does not deep-copy the variable, it assigned a reference to it to an internal variable. The object will then directly operate on the given data variable. This form of the constructor is used with a compiler-generated time string variable.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
buf	ASN1VisibleString&	Reference to a visible string structure.

Output Parameters:

None

ASN1CTime::getYear

```
int getYear ();
```

This method returns the year component of the time value. Note that the return value may differ for different inherited ASN1Ctime classes.

Return Value:

Name	Type	Description
year	int	Year component (full 4 digits) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getMonth

```
int getMonth ();
```

This method returns the month number component of the time value. The number of January is 1, February 2, … up to December 12. You may use enumerated values for decoded months: ASN1CTime::January, ASN1CTime::February, etc. Also short aliases for months can be used: ASN1CTime::Jan, ASN1CTime::Feb, etc. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
month	int	Month component (1 – 12) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getDay

```
int getDay ();
```

This method returns the day of month number component of the time value. The number of the first day in month is 1, the number of the last day may be in interval from 28 to 31. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
day	int	Day of month component (1 – 31) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getHour

```
int getHour ();
```

This method returns the hour component of the time value. As the ISO 8601 is based on the 24-hour timekeeping system, hours are represented by two-digit values from 00 to 23. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
hour	int	Hour component (0 – 23) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getMinute

```
int getMinute ();
```

This method returns the minute component of the time value. Minutes are represented by two digits from 00 to 59. Note that the return value may be different for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
minute	int	Minute component (0 – 59) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in

		Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getSecond

```
int getSecond ();
```

This method returns the second component of the time value. Seconds are represented by two digits from 00 to 59. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
second	int	Second component (0 – 59) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getFraction

```
int getFraction ();
```

This method returns the second's decimal fraction component of the time value. Second's decimal fraction is represented by one digit from 0 to 9. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
fraction	int	Second's decimal fraction component (0 – 9) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getDiffHour

```
int getDiffHour ();
```

This method returns the hour component of the difference between the time zone of the object and Coordinated Universal Time (UTC). The UTC time is the sum of the local time and positive or negative time difference. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
dhour	int	The negative or positive hour component of the difference between the time zone of the object and UTC time (-12 – +12) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getDiffMinute

```
int getDiffMinute ();
```

This method returns the minute component of the difference between the time zone of the object and Coordinated Universal Time (UTC). The UTC time is the sum of the local time and positive or negative time difference. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
dmin	int	The negative or positive minute component of the difference between the time zone of the object and UTC time (-59 – +59) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getDiff

```
int getDiff ();
```

This method returns the difference between the time zone of the object and Coordinated Universal Time (UTC), in minutes. The UTC time is the sum of the local time and positive or negative time difference. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
diff	int	The negative or positive difference, in minutes, between the time zone of the object and UTC time (-12*60 – +12*60) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getUTC

```
ASN1BOOL getUTC ();
```

This method returns the UTC flag state. If the UTC flag is TRUE, then the time is a UTC time and symbol 'Z' is added at the end of time string. Otherwise, it is a local time.

Return Value:

Name	Type	Description
utc	`ASN1BOOL`	UTC flag state is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CTime::getTime

```
time_t getTime ();
```

This method converts the time string to a value of the built-in C type *time_t*. The value is the number of seconds from January 1, 1970. If the time is represented as UTC time plus or minus a time difference, then the resulting value will be recalculated as local time. For example, if the time string is "19991208120000+0930", then this string will be converted to "19991208213000" and then converted to a time_t value. Note that the return value may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description

time	time_t	The time value, expressed as number of seconds from January 1, 1970. If the operation fails, one of the negative status codes defined in Appendix A is returned

Input Parameters:

None

Output Parameters:

None

ASN1CTime::setYear

```
int setYear (int year);
```

This method sets the year component of the time value. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
year	int	Year component (full 4 digits)

Output Parameters:

None

ASN1CTime::setMonth

```
int setMonth (int month);
```

This method sets the month number component of the time value. The number of January is 1, February 2, …, through December (12). You may use enumerated values for months encoding: ASN1CTime::January, ASN1CTime::February, etc. Also you can use short aliases for months: ASN1CTime::Jan, ASN1CTime::Feb, etc. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
month	int	Month component (1 – 12).

Output Parameters:

None

ASN1CTime::setDay

```
int setDay (int day);
```

This method sets the day of month number component of the time value. The number of the first day in month is 1; the number of the last day may be in interval from 28 to 31. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
day	int	Day of month component (1 – 31).

Output Parameters:

None

ASN1CTime::setHour

```
int setHour (int hour);
```

This method sets the hour component of the time value. As the ISO 8601 is based on the 24-hour timekeeping system, hours are represented by two digits from 00 to 23. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
hour	int	Hour component (0 – 23).

Output Parameters:

None

ASN1CTime::setMinute

```
int setMinute (int minute);
```

This method sets the minute component of the time value. Minutes are represented by two digits from 00 to 59. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
minute	int	Minute component (0 – 59).

Output Parameters:

None

ASN1CTime::setSecond

```
int setSecond (int second);
```

This method sets the second component of the time value. Seconds are represented by two digits from 00 to 59. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
second	int	Second component (0 – 59).

Output Parameters:

None

ASN1CTime::setFraction

```
int setFraction (int fraction);
```

This method sets the second's decimal fraction component of the time value. Second's decimal fraction is represented by one digit from 0 to 9. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
fraction	int	Second's decimal fraction component (0 – 9).

Output Parameters:

None

ASN1CTime::setDiffHour

```
int setDiffHour (int dhour);
```

This method sets the hour component of the difference between the time zone of the object and Coordinated Universal Time (UTC). The UTC time is the sum of the local time and positive or negative time difference. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
dhour	int	The negative or positive hour component of the difference between the time zone of the object and UTC time (-12 – +12) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Output Parameters:

None

ASN1CTime::setDiff

```
int setDiff (int dhour, int dminute);
```

This method sets the hour and minute components of the difference between the time zone of the object and Coordinated Universal Time (UTC). The UTC time is the sum of the local time and positive or negative time difference. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
dhour	int	The negative or positive hour component of the difference between the time zone of the object and UTC time (-12 – +12).
dminute	int	The negative or positive minute component of the difference between the time zone of the object and UTC time (-59 – +59).

Output Parameters:

None

ASN1CTime::setDiff

```
int setDiff (int inMinutes);
```

This method sets the difference between the time zone of the object and Coordinated Universal Time (UTC), in minutes. The UTC time is the sum of the local time and positive or negative time difference. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
inMinutes	int	The negative or positive difference, in minutes, between the time zone of the object and UTC time (-12*60 – +12*60) is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Output Parameters:

None

ASN1CTime::setUTC

```
int setUTC (ASN1BOOL utc);
```

This method sets the UTC flag state. If the UTC flag is TRUE, then the time is a UTC time and symbol ‘Z’ is added at the end of time string. Otherwise, it is a local time.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
utc	`ASN1BOOL`	UTC flag state.

Output Parameters:

None

ASN1CTime::setTime

```
int setTime (time_t time, ASN1BOOL diffTime);
```

This method converts the value of the C built-in type *time*_t to a time string. The value is the number of seconds from January 1, 1970. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
time	time_t	The time value, expressed as number of seconds from January 1, 1970.
diffTime	ASN1BOOL	TRUE means the difference between local time and UTC time will be calculated; in other case only local time will be stored.

Output Parameters:

None

ASN1CTime::parseString

```
int parseString (ASN1ConstCharPtr string);
```

This method parses the given time string. The string is expected to be in the ASN.1 value notation format for the given ASN.1 time string type. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
string	ASN1ConstCharPtr	The time string value to be parsed.

Output Parameters:

None

ASN1CTime::clear

```
void clear ();
```

This method clears the time string. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

ASN1CTime::operator =

```
ASN1CTime& operator = (const ASN1CTime& other);
```

This overloaded assignment operator copies one ASN1CTime class instance to another. Note that the action of this method may differ for different inherited ASN1CTime classes.

Return Value:

Name	Type	Description
this	ASN1CTime&	Returns reference to this instance.

Input Parameters:

Name	Type	Description
other	ASN1CTime&	Reference to the time value to be copied.

Output Parameters:

None

ASN1CTime::operator ==
ASN1CTime::operator >
ASN1CTime::operator <
ASN1CTime::operator >=
ASN1CTime::operator <=

```
ASN1BOOL operator == (ASN1CTime& other);
ASN1BOOL operator != (ASN1CTime& other);
ASN1BOOL operator > (ASN1CTime& other);
ASN1BOOL operator < (ASN1CTime& other);
ASN1BOOL operator >= (ASN1CTime& other);
ASN1BOOL operator <= (ASN1CTime& other);
```

These are overloaded comparison operators that can be used with the time classes. They can be used to compare two time strings for the various conditions.

Return Value:

Name	Type	Description
result	ASN1BOOL	Returns result of the comparison of two time class instances.

Input Parameters:

Name	Type	Description
other	ASN1CTime&	Reference to the time value to be compared.

Output Parameters:

None

ASN1CGeneralizedTime

```
ASN1Type
  |
  +- ASN1CTime
      |
      +- ASN1CGeneralizedTime
```

The ASN1CGeneralizedTime class is derived from the ASN1CTime base class. It is used as the base class for generated control classes for the ASN.1 Generalized Time ([UNIVERSAL 24] IMPLICIT VisibleString) type. This class provides utility methods for operating on the time information referenced by the generated class. This class can also be used inline to operate on the times within generated time string elements in a SEQUENCE, SET, or CHOICE construct. Time string generally is encoding according to ISO 8601 format with some exceptions (see X.680).

ASN1CGeneralizedTime::ASN1CGeneralizedTime

There are a number of different constructors available for this object. The different types are as follows:

```
ASN1CGeneralizedTime (ASN1MessageBuffer& msgBuf,
                      char*& buf, int bufSize);
```

This constructor creates a time string from a buffer. It does not deep-copy the data, it just assigns the passed array to an internal reference variable. The object will then directly operate on the given data variable.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
buf	char*	Reference to the pointer to time string buffer.
bufSize	int	Size of passed buffer, in bytes.

Output Parameters:

None

```
ASN1CGeneralizedTime (ASN1MessageBuffer& msgBuf,
                      ASN1GeneralizedTime& buf);
```

This constructor creates a time string using the *ASN1GeneralizedTime* argument. The constructor does not deep-copy the variable, it assigns a reference to it to an internal variable. The object will then directly operate on the given data variable. This form of the constructor is used with a compiler-generated time string variable.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
buf	ASN1VisibleString&	Reference to a visible string structure.

Output Parameters:

None

ASN1CGeneralizedTime::getCentury

```
int getCentury ();
```

This method returns the century part (first two digits) of the year component of the time value.

Return Value:

Name	Type	Description
century	int	Century part (first two digits) of the year component is returned if the operation is successful. If the operation fails, one of the negative status codes defined in Appendix A is returned.

Input Parameters:

None

Output Parameters:

None

ASN1CGeneralizedTime::setCentury

```
int setCentury (int century);
```

This method sets the century part (first two digits) of the year component of the time value.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
century	int	Century part (first two digits) of the year component

Output Parameters:

None

ASN1CUTCTime

```
ASN1Type
  |
  +- ASN1CTime
      |
      +- ASN1CUTCTime
```

The ASN1CUTCTime class is derived from the ASN1CTime base class. It is used as the base class for generated control classes for the ASN.1 Universal Time ([UNIVERSAL 23] IMPLICIT VisibleString) type. This class provides utility methods for operating on the time information referenced by the generated class. This class can also be used inline to operate on the times within generated time string elements in a SEQUENCE, SET, or CHOICE construct. Time string generally is encoding according to ISO 8601 format with some exceptions (see X.680).

ASN1CUTCTime::ASN1CUTCTime

There are a number of different constructors available for this object. The different types are as follows:

```
ASN1CUTCTime (ASN1MessageBuffer& msgBuf, char*& buf, int bufSize);
```

This constructor creates a time string from a buffer. It does not deep-copy the data, it just assigns the passed array to an internal reference variable. The object will then directly operate on the given data variable.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
buf	char*	Reference to a pointer to a time string buffer.
bufSize	int	Size of passed buffer, in bytes.

Output Parameters:

None

```
ASN1CUTCTime (ASN1MessageBuffer& msgBuf, ASN1UTCTime& buf);
```

This constructor creates a time string using the *ASN1UTCTime* argument. The constructor does not deep-copy the variable, it assigns a reference to it to an internal variable. The object will then directly operate on the given data variable. This form of the constructor is used with a compiler-generated time string variable.

Input Parameters:

Name	Type	Description
msgBuf	ASN1Message Buffer&	Reference to an ASN1Message buffer derived object (for example, an ASN1BEREncodeBuffer).
buf	ASN1UTCTime&	Reference to a time string structure.

Output Parameters:

None

ASN1CUTCTime::setYear

```
int setYear (int year);
```

This method sets the year component of the time value. The 'year' parameter can be passed as either the two last digits of the year (00 – 99) or as the full 4 digits (0 – 9999). Note: the 'getYear' method returns the year in the full 4 digits format, independent of the format of the 'year' parameter used in this method.

Return Value:

Name	Type	Description
stat	int	Returns ASN_OK if operation is successful, a negative status value will be returned if not.

Input Parameters:

Name	Type	Description
year	int	Year component (full 4 digits or only two last digits).

Output Parameters:

None

Asn1NamedEventHandler

The Asn1NamedEventHandler class is an abstract base class from which user-defined event handlers are derived. This class contains pure virtual function definitions for all of the methods that must be implemented to create a customized event handler class. See the section above on Event Handlers for a discussion on how event handlers work.

Asn1NamedEventHandler::startElement

This method is invoked from within a decode function when an element of a SEQUENCE, SET, SEQUENCE OF, SET OF, or CHOICE construct is parsed.

Calling Sequence:

```
eventHandler.startElement (name, index);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
name	const char*	For SEQUENCE, SET, or CHOICE, this is the name of the element as defined in the ASN.1 definition. For SEQUENCE OF or SET OF, this is set to the name "element".
index	int	For SEQUENCE, SET, or CHOICE, this is not used and is set to the value –1. For SEQUENCE OF or SET OF, this contains the zero-based index of the element in the conceptual array associated with the construct.

Output Parameters:

None

Asn1NamedEventHandler::endElement

This method is invoked from within a decode function when parsing is complete on an element of a SEQUENCE, SET, SEQUENCE OF, SET OF, or CHOICE construct.

Calling Sequence:

```
eventHandler.endElement (name, index);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
name	const char*	For SEQUENCE, SET, or CHOICE, this is the name of the element as defined in the ASN.1 definition. For SEQUENCE OF or SET OF, this is set to the name "element".
index	int	For SEQUENCE, SET, or CHOICE, this is not used and is set to the value –1. For SEQUENCE OF or SET OF, this contains the zero-based index of the element in the conceptual array associated with the construct.

Output Parameters:

None

Asn1NamedEventHandler::boolValue

This method is invoked from within a decode function when a value of the BOOLEAN ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.boolValue (value);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
value	ASN1BOOL	Parsed value.

Output Parameters:

None

Asn1NamedEventHandler::intValue

This method is invoked from within a decode function when a value of the INTEGER ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.intValue (value);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
value	ASN1INT	Parsed value.

Output Parameters:

None

Asn1NamedEventHandler::uIntValue

This method is invoked from within a decode function when a value of the INTEGER ASN.1 type is parsed. In this case, constraints on the integer value forced the use of an unsigned integer C type to represent the value.

Calling Sequence:

```
eventHandler.uIntValue (value);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
value	ASN1UINT	Parsed value.

Output Parameters:

None

Asn1NamedEventHandler::bitStrValue

This method is invoked from within a decode function when a value of the BIT STRING ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.bitStrValue (numbits, data);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
numbits	ASN1UINT	Number of bits in the parsed value.
data	const ASN1OCTET*	Pointer to byte array containing the bit string data.

Output Parameters:

None

Asn1NamedEventHandler::octStrValue

This method is invoked from within a decode function when a value of the OCTET STRING ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.octStrValue (numocts, data);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
numocts	ASN1UINT	Number of octets in the parsed value.
data	const ASN1OCTET*	Pointer to byte array containing the octet string data.

Output Parameters:

None

Asn1NamedEventHandler::charStrValue

This method is invoked from within a decode function when a value of one of the 8-bit ASN.1 character string types is parsed.

Calling Sequence:

```
eventHandler.charStrValue (value);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
value	ASN1Const CharPtr	Null-terminated character string value.

Output Parameters:

None

Asn1NamedEventHandler::charStrValue (16-bit version)

This method is invoked from within a decode function when a value of one of the 16-bit ASN.1 character string types is parsed.

Calling Sequence:

```
eventHandler.charStrValue (nchars, data);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
nchars	ASN1UINT	Number of characters in the parsed value.
data	ASN116 BITCHAR*	Pointer to array containing 16-bit character values. These are represented using unsigned short integer values.

Output Parameters:

None

Asn1NamedEventHandler::nullValue

This method is invoked from within a decode function when a value of the NULL ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.nullValue ();
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

None

Output Parameters:

None

Asn1NamedEventHandler::oidValue

This method is invoked from within a decode function when a value the OBJECT IDENTIFIER ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.oidValue (numSubIds, pSubIds);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
numSubIds	ASN1UINT	Number of subidentifiers in the object identifier.
pSubIds	ASN1UINT*	Pointer to array containing the subidentifier values.

Output Parameters:

None

Asn1NamedEventHandler::realValue

This method is invoked from within a decode function when a value the REAL ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.realValue (value);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
value	double	Parsed value.

Output Parameters:

None

Asn1NamedEventHandler::enumValue

This method is invoked from within a decode function when a value of the ENUMERATED ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.enumValue (value);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
value	ASN1UINT	Parsed value.

Output Parameters:

None

Asn1NamedEventHandler::octStrValue

This method is invoked from within a decode function when a value of the OCTET STRING ASN.1 type is parsed.

Calling Sequence:

```
eventHandler.octStrValue (numocts, data);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
numocts	ASN1UINT	Number of octets in the parsed value.
data	const ASN1OCTET*	Pointer to byte array containing the octet string data.

Output Parameters:

None

Asn1NamedEventHandler::openTypeValue

This method is invoked from within a decode function when an ASN.1 open type value is parsed.

Calling Sequence:

```
eventHandler.openTypeValue (numocts, data);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

None

Input Parameters:

Name	Type	Description
numocts	ASN1UINT	Number of octets in the parsed value.
data	const ASN1OCTET*	Pointer to byte array containing the encoded ASN.1 value.

Output Parameters:

None

Asn1ErrorHandler

The Asn1ErrorHandler class is an abstract base class from which user-defined error handlers are derived. These user-defined handlers allow for intervention in the decoding process to allow for fault-tolerant behavior. This class contains pure virtual function definitions for the methods that must be implemented to create a customized error handler class. See the section above on Event Handlers for a discussion on how error handlers work.

Asn1ErrorHandler::error

This method is invoked from within a decode function when certain types of recoverable errors occur.

Calling Sequence:

```
ret = errorHandler.error (pCtxt, pCCB, stat);
```

where eventHandler is an object of a class derived from the Asn1NamedEventHandler base class.

Return Value:

Name	Type	Description
ret	int	Updated status value. This will normally be set to ASN_OK if the parsing process is to continue or the original status value (stat) if decoding is to be aborted.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT*	Pointer to the context structure associated with the decoder. This can be used in call to C run-time functions to manipulate the current decode buffer position.
pCCB	ASN1CCB*	Pointer to a 'context control block' structure. This is basically a loop control mechanism to keep the variable associated with parsing a nested constructed element straight. It is passed into the error handler to allow the loop control variables to be manipulated to force certain retry behavior. The item within this structure that is of greatest interest is the 'seqx' element. This is the sequence index of the current item being pared within a SEQUENCE construct. If the user would like to retry parsing of an element, this item should be decremented; if the element is to be skipped altogether, this element should be left alone.
stat	int	The original error status value.

Output Parameters:

None

BER Run-time Library Functions

The ASN.1 Basic Encoding Rules (BER) run-time library contains all of the low-level constants, types, and functions that are assembled by the compiler to encode/decode more complex structures.

This library consists of two items:

1. A global include file ("asn1type.h") that is compiled into all generated source files

2. An object library of functions that are linked in with the C functions after compilation with a C compiler.

In general, programmers will not need to be too concerned with the details of these functions. The ASN.1 compiler generates calls to them in the C or C++ source files that it creates. However, the functions in the library may also be called on their own in applications requiring their specific functionality.

asn1type.h Include File

Every C source file produced by the ASN1C compiler includes the asn1type.h include file either directly or at a nested level. This file contains function error code constants, tagging value and mask constants, sizing constants for internal arrays and buffers, and ASN.1 primitive type definitions.

Error Constants

All error code constants begin with the prefix "ASN_E_" and run from zero, which is success, into negative numbers that describe all of the various error conditions. A complete list of error codes and a description of what causes them can be found in Appendix A.

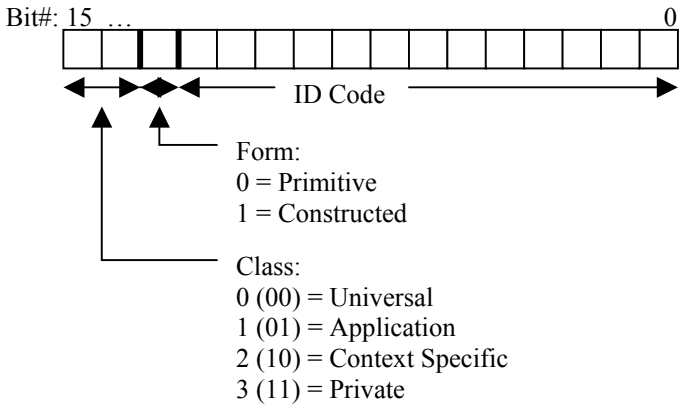
There are several error message utility functions defined within the run-time libraries that provide detailed information on error conditions. See the descriptions of the functions beginning with the prefix *rtErr* in the Common Functions section.

Tagging Value and Mask Constants

Tagging constants provide a means for setting up class and form fields within ASN.1 tag variables. To understand how these work, one must first understand the internal representation of ASN.1 tags within the run-time library.

In the ASN.1 standard, tags are represented as a class, form, and ID code. Class and form are relatively straightforward. Class is a two bit code which can take on one of four possible values (UNIVERSAL, APPLICATION, CONTEXT, or PRIVATE) and form is a single bit which can take on one of two values (PRIMITIVE or CONSTRUCTED). Together, these occupy the upper three bits of a tag value. The remaining bits are for the ID code. An ID code having a value of 30 or less can be accommodated in the remaining bits of the first byte. If the value is larger, additional bytes are added.

This is where the internal representation differs from the standard. The internal representation assumes an unsigned, 13-bit number (i.e up to 2 to the 13th power, or 8191 ID codes) can represent all possible ID codes defined in a particular environment. Using this assumption, a tag can always be represented as an unsigned, 16-bit integer value as follows:



Bits 14 and 15 hold the 2-bit class value, bit 13 holds the form, and bits 0 through 12 hold the integer value of the ID code. Run-time library functions handle conversions to and from this format and the ASN.1 standard format in messages.

The tagging constants that allow construction of internal tag values come in two forms: values and masks. Values are simply the values of the classes and forms shown above. These start with the prefix 'TV_' (for 'Tag Value'). Examples are TV_UNIV for UNIVERSAL (0) and TV_APPL for APPLICATION (1). In addition, values are defined for the ID codes of the universal ASN.1 types such as BOOLEAN and INTEGER - these are preceded by the prefix 'ASN_ID_'. Masks are hexadecimal values which can be logically OR'd together with integer values to form full internal tag specifications. These start with the prefix 'TM_' (for 'Tag Mask'). For example, to form an internal representation of a private, constructed tag with ID code 21, the following could be used:

```
TM_PRIV|TM_CONS|21.
```

Sizing Constants

Sizing constants are provided to define the maximum sizes of internal buffers and arrays used by the run-time library functions. These can be modified if the given values are not sufficient for a given application. The constants and default values can be found in asn1type.h.

ASN.1 Primitive Type Definitions

C typedef statements are used to represent several of the ASN.1 primitive types (i.e., universal, non-constructed types). These include ASN1INT, ASN1OCTET, ASN1BOOL, ASN1OBJID, and the special type ASN1TAG used to represent the internal tag discussed above. In addition, ASN1OctStr and ASN1DynOctStr provide generic representations of static and dynamic octet strings that can be used in type cast operations.

BER/DER C Encode Functions

BER/DER C encode functions handle the BER encoding of the primitive ASN.1 data types and ASN.1 length and tag fields within a message. Calls to these functions are assembled in the C source code generated by the ASN1C compiler to accomplish the encoding of complex ASN.1 structures. These functions are also directly callable from within a user's application program if the need to accomplish a low level encoding function exists.

The procedure to call the encode function that encodes a primitive type is the same as the procedure to call a compiler generated encode function described above. The xe_setp function must first be called to set a pointer to the buffer into which the variable is to be encoded. A static encode buffer is specified by specifying a pointer to a buffer and buffer size. Setting the buffer address to NULL and buffer size to 0 specifies a dynamic buffer. The primitive encode function is then invoked. Finally, xe_getp is called to retrieve a pointer to the encoded message component.

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

```
ASN1OCTET buf[10], *msg_p;
ASN1BOOL  boolValue = 1;  /* true */
ASN1CTXT  ctxt;
int msglen;

xe_setp (&ctxt, buf, sizeof(buf));
msglen = xe_boolean (&ctxt, &boolValue, ASN1EXPL);
msg_p  = xe_getp (&ctxt);
```

The msg_p variable now contains a pointer to the encoded boolean value and msglen contains the length.

xe_setp - Set Encode Buffer Pointer

The xe_setp function is used to set the internal encode buffer within the Run-time library encode module. It must be called prior to calling any other compiler generated or run-time library encode function.

Calling sequence:

```
xe_setp (ctxt_p, bufptr, buflen);
```

Return value:

None

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
bufptr	ASN1OCTET*	Pointer to a memory buffer to use to encode a message. The buffer should be declared as an array of unsigned characters (ASN1OCTETs). This parameter can be set to NULL to specify dynamic encoding (i.e., the encode functions will dynamically allocate a buffer for the encoded message).
buflen	int	Length of the memory buffer in bytes.

Output parameters:

None

xe_getp - Get Encode Buffer Pointer

The xe_getp function is used to obtain a pointer to the start of an encoded message after calls to the encode function(s) are complete. ASN.1 messages are encoded from the end of a given buffer toward the beginning, therefore, in practically all cases, the start of the message will not be at the beginning of the buffer.

Calling sequence:

```
msgptr = xe_getp (ctxt_p);
```

Return value:

Name	Type	Description
msgptr	ASN1OCTET*	Pointer to beginning of encoded message.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

None

xe_tag_len - Encode Tag and Length

The xe_tag_len function is used to encode the ASN.1 tag and length fields that preface each block of message data. The ASN1C compiler generates calls to this function to handle the encoding of user defined tags within an ASN.1 specification. This function is also called from within the RUN-TIME LIBRARY functions to handle the addition of the universal tags defined for each of the ASN.1 primitive data types.

Calling sequence:

```
msglen = xe_tag_len (ctxt_p, asntag, length);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component equal to the given length plus the additional bytes that are added for the tag and length fields. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store

		all working variables that must be maintained between function calls.
asntag	ASN1TAG	The ASN.1 tag to be encoded in the message. This parameter is passed using the internal representation discussed in Section 4.1.2. It is passed as an unsigned 16 bit integer.
length	int	The length of the contents field previously encoded. This parameter can be used to specify the actual length, or the special constant 'ASN_K_INDEFLEN' can be used to specify that an indefinite length specification should be encoded.

Output Parameters:

None

xe_boolean - Encode BOOLEAN

The xe_boolean function will encode a variable of the ASN.1 BOOLEAN type.

Calling sequence:

```
msglen = xe_boolean (ctxt_p, object, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1BOOL*	A pointer to the BOOLEAN value to be encoded (note that a pointer to the BOOLEAN is passed, not the BOOLEAN value itself. This may seem awkward, but to keep the calling sequence of all encode functions the same, pointers were used in all cases). A BOOLEAN is defined as a single OCTET whose value is 0 for FALSE and any other value for TRUE.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_integer - Encode INTEGER

The xe_integer function will encode a variable of the ASN.1 INTEGER type.

Calling sequence:

```
msglen = xe_integer (ctxt_p, object, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1INT*	A pointer to the INTEGER value to be encoded (note that a pointer to the INTEGER is passed, not the INTEGER value itself. This may seem awkward, but to keep the calling sequence of all encode functions the same, pointers were used in all cases). The ASN1INT type is set to the C type 'int' in the asn1type.h file. This is assumed to represent a 32 bit integer value.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_unsigned - Encode Unsigned INTEGER

The xe_unsigned function will encode an unsigned variable of the ASN.1 INTEGER type.

Calling sequence:

```
msglen = xe_unsigned (ctxt_p, object, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1UINT*	A pointer to the unsigned INTEGER value to be encoded (note that a pointer to the value is passed, not the value itself. This may seem awkward, but to keep the calling sequence of all encode functions the same, pointers were used in all cases). The ASN1UINT type is set to the C type 'unsigned int' in the asn1type.h file. This

		is assumed to represent a 32-bit integer value.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_bigint – Encode Big Integer

The xe_bigint function will encode a variable of the ASN.1 INTEGER type. In this case, the integer is assumed to be of a larger size than can fit in a C or C++ long type (normally 32 or 64 bits). For example, parameters used to calculate security values are typically larger than these sizes.

Items of this type are stored in character string constant variables. They can be represented as decimal strings (with no prefixes), as hexadecimal strings starting with a "0x" prefix, as octal strings starting with a "0o" prefix or as binary strings starting with a "0b" prefix. Other radixes are currently not supported.

Calling sequence**:**

```
msglen = xe_bigint (ctxt_p, object_p, tagging);
```

Return value:

Name	**Type**	**Description**
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters**:**

Name	**Type**	**Description**
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	char*	A pointer to a character string containing the value to be encoded.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_bitstr - Encode BIT STRING

The xe_bitstr function will encode a variable of the ASN.1 BIT STRING type.

Calling sequence**:**

```
msglen = xe_bitstr (ctxt_p, object, numbits, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1OCTET*	A pointer to an OCTET string containing the bit data to be encoded. This string contains bytes having the actual bit settings as they are to be encoded in the message.
numbits	int	The number of bits within the bit string to be encoded.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_octstr - Encode OCTET STRING

The xe_octstr function will encode a variable of the ASN.1 OCTET STRING type.

Calling sequence:

```
msglen = xe_octstr (ctxt_p, object, numocts, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1OCTET*	A pointer to an OCTET STRING containing the octet data to be encoded.
numocts	int	The number of octets (bytes) within the bit string to be encoded.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_charstr – Encode Character String

The xe_charstr function will encode a variable one of the ASN.1 character string types that are based on 8-bit character sets. This includes IA5String, VisibleString, PrintableString, and NumericString.

Calling sequence:

```
msglen = xe_charstr (ctxt_p, object_p, tagging, tag);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	char*	A pointer to a null-terminated C character string to be encoded.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.
tag	ASN1TAG	The Universal ASN.1 tag to be encoded in the message. This parameter is passed using the internal representation discussed in Section 4.1.2. It is passed as an unsigned 16-bit integer. The tag value must be represent one of the 8-bit character string type documented in the X.680 standard.

Output Parameters:

None

xe_16BitCharStr – Encode 16-bit Character String

The xe_16BitCharStr function will encode a variable one of the ASN.1 character string types that are based on a 16-bit character sets. This includes the BMPString type.

Calling sequence:

```
msglen = xe_16BitCharStr (ctxt_p, object_p, tagging, tag);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	Asn116Bit CharString*	A pointer to a structure representing a 16-bit character string to be encoded. This structure contains a character count element and a pointer to an array of 16-bit character elements represented as 16-bit short integers.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.
tag	ASN1TAG	The Universal ASN.1 tag to be encoded in the message. This parameter is passed using the internal representation discussed in Section 4.1.2. It is passed as an unsigned 16-bit integer. The tag value must be represent one of the 16-bit character string type documented in the X.680 standard.

xe_32BitCharStr – Encode 32-bit Character String

The xe_32BitCharStr function will encode a variable one of the ASN.1 character string types that are based on a 32-bit character sets. This includes the UniversalString type.

Calling sequence:

```
msglen = xe_32BitCharStr (ctxt_p, object_p, tagging, tag);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	Asn132Bit CharString*	A pointer to a structure representing a 32-bit character string to be encoded. This structure contains a character count element and a pointer to an array of 32-bit character elements represented as 32-bit unsigned integers.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.
tag	ASN1TAG	The Universal ASN.1 tag to be encoded in the message. This parameter is passed using the internal representation discussed in Section 4.1.2. It is passed as an

		unsigned 16-bit integer. The tag value must be represent one of the 16-bit character string type documented in the X.680 standard.

xe_enum - Encode ENUMERATED

The xe_enum function will encode a variable of the ASN.1 ENUMERATED type.

Calling sequence:

```
msglen = xe_enum (ctxt_p, object, tagging);
```

The enumerated encoding is identical to that of an integer. The compiler adds additional checks to the generated code to ensure the value is within the given set.

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1ENUM*	A pointer to an integer containing the enumerated value to be encoded (note that a pointer to the value is passed, not the value itself. This may seem awkward, but to keep the calling sequence of all encode functions the same, pointers were used in all cases).
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_null - Encode NULL

The xe_null function will encode an ASN.1 NULL placeholder.

```
msglen = xe_null (ctxt_p, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_objid - Encode OBJECT IDENTIFIER

The xe_objid function will encode a variable of the ASN.1 OBJECT IDENTIFIER type.

Calling sequence:

```
msglen = xe_objid (ctxt_p, object, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1OBJID*	A pointer to an object identifier structure. This structure contains an integer to hold the number of subidentifers in the object and an array to hold the subidentifier values.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_real – Encode Real

xe_real will encode a variable of the REAL data type. This function provides support for the plus-infinity and minus- infinity special real values.

Calling sequence:

```
msglen = xe_real (ctxt_p, object, tagging);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1REAL*	A pointer to a variable of the ASN1REAL data type. This is defined to be the C double type. Special real values plus and minus infinity are encoded by using the xu_GetPlusInfinity and xu_GetMinusInfinity functions to set the real value to be encoded.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is added or not. Users will generally always set this value to 'ASN1EXPL'.

Output Parameters:

None

xe_OpenType - Encode Open Type

The xe_OpenType function will encode a variable of the old (pre-1994) ASN.1 ANY type or other elements defined in the later standards to be Open Types (for example, a variable type declaration in a CLASS construct as defined in X.681). A variable of this is considered to be a previously encoded ASN.1 message component.

Calling sequence:

```
msglen = xe_OpenType (ctxt_p, object);
```

Note that the tagging argument present on other encode functions is not present here. This is because these variables must always be encoded explicitly.

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object	ASN1OpenType	A pointer to a buffer containing an encoded ASN.1 message component.

Output Parameters:

None

xe_free – Free Encoder Dynamic Memory

The xe_free function will free a dynamic encode buffer. This is the buffer that is allocated if dynamic encoding of a message is enabled (passing NULL as the buffer pointer argument to xe_setp enables dynamic encoding).

Note that this is different than the *xu_freeall* function associated with freeing decoder memory. This function only releases the memory associated with a dynamic encoded buffer. The *xu_freeall* function will not release this memory.

Calling sequence**:**

```
xe_free (ctxt_p);
```

Return value:

None

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls. The dynamic encode buffer pointer is contained within this structure.

Output Parameters:

None

xe_expandBuffer – Expand Dynamic Encode Buffer

The xe_expandBuffer function will expand a dynamic encode buffer. This is the buffer that is allocated if dynamic encoding of a message is enabled (passing NULL as the buffer pointer argument to xe_setp enables dynamic encoding).

The size of the new buffer is determined by the length argument. If the length is less than a configurable buffer expansion increment size (the constant ASN_K_ENCBUFSIZ), the buffer is expanded by the increment size; otherwise it is expanded by the actual length value.

Calling sequence**:**

```
status = xe_expandBuffer (ctxt_p, length);
```

Return value:

Name	Type	Description
status	int	Status of the operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if failure.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls. The dynamic encode buffer pointer is contained within this structure.
length	int	The number of bytes required. This may not be size the buffer is actually expanded by. The buffer will be expanded by a fixed-size increment defined by ASN_K_ENCBUFSIZ for small requests to limit the required number of expansions.

Output Parameters:

None

xe_memcpy – Copy Bytes to Encode Buffer

The xe_memcpy function is used to copy bytes into the encode buffer. BER and DER messages are encoded from back-to-front and this function will take this into account when copying bytes. It will also check to ensure that enough space is available in the buffer for the bytes to be copied. If not and the encode buffer is specified to be a dynamic buffer, it will automatically be expanded. If the buffer is static and enough space is not available, an error status (ASN_E_BUFOVFLW) will be returned.

Calling sequence:

```
msglen = xe_memcpy (ctxt_p, object_p, length);
```

Return value:

Name	Type	Description
msglen	int	Length of the copied message component (this is the length that was passed in if the copy was successful). A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
object_p	ASN1OCTET*	A pointer to a buffer containing the bytes to be copied.
length	ASN1UINT	Number of bytes to copy.

Output Parameters:

None

xe_len – Encode a Length Value

The xe_len function is used to encode a BER or DER length determinant value.

Calling sequence:

```
msglen = xe_len (ctxt_p, length);
```

Return value:

Name	Type	Description
msglen	int	Length of the encoded message component. A negative status value will be returned if encoding is not successful.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
length	int	The length variable to encode. A negative value is interpreted that an indefinite length identifier should be encoded.

Output Parameters:

None

xe_derCanonicalSort – DER Canonical Sort

The xe_derCanonicalSort function is added to the generated code for SEQUENCE OF/SET OF constructs to ensure the elements are in the required canonical order for DER. If the elements are not in the right order, they are sorted to be in the correct order prior to encoding.

Calling sequence:

```
status = xe_derCanonicalSort (ctxt_p, pList);
```

Return value:

Name	Type	Description
status	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative status codes defined in Appendix A if failure.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pList	Asn1RTSList	Linked list of message components to be sorted. The elements of this list are offsets to encoded components within the encode buffer.

Output Parameters:

None

xe_TagAndIndefLen – Encode Tag and Indefinite Length

The xe_TagAndIndefLen function is used to encode a tag value and an indefinite length. This can be used to manually create an indefinite length wrapper around long records.

Calling sequence:

```
status = xe_TagAndIndefLen (ctxt_p, tag, length);
```

Return value:

Name	Type	Description
status	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative status codes defined in Appendix A if failure.

Input Parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tag	ASN1TAG	ASN.1 tag value to be encoded.
length	int	Actual length of existing message components.

Output Parameters:

None

BER/DER C Decode Functions

BER/DER C decode functions handle the decoding of the primitive ASN.1 data types and ASN.1 length and tag fields within a message. Calls to these functions are assembled in the C source code generated by the ASN1C compiler to decode complex ASN.1 structures. These functions are also directly callable from within a user's application program if the need to decode a primitive data item exists.

The procedure to decode a primitive data item is as follows:

1. Call the xd_setp low-level decode function to specify the address of the buffer containing the encoded ASN.1 data to be decode, and

2. Call the specific decode function to decode the value. The tag value obtained in step 1 can be used to determine the decode function to call to decode the variable.

For example, to decode a message containing a single object identifier with no special tagging, the following code fragment could be used:

```
ASN1OBJID objId;  /* variable to receive decoded result */
ASN1CTXT  ctxt;
ASN1TAG   tag;
int       len, stat;

memset (&ctxt, 0, sizeof(ctxt));

/* assume 'buf' contains message fragment to be decoded.. */

xd_setp (&ctxt, buf, sizeof(buf), &tag, &len);

if (tag == TM_UNIV|TM_PRIM|ASN_ID_OBJID) {  /* OID tag */
   stat = xd_objid (&ctxt, &objId, ASN1EXPL, 0);
   if (stat != ASN_OK) {
      xu_perror (&ctxt);
   }
}
```

The objId variable now contains the decoded object identifier value.

xd_setp - Set Decode Buffer Pointer

The xd_setp function is used to set the internal decode buffer pointer within the run-time library decode module. It must be called prior to calling any other compiler generated or run-time library decode function.

Calling sequence**:**

```
xd_setp (ctxt_p, bufptr, msglen, asntag, length);
```

Return value:

None

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
bufptr	ASN1OCTET*	Pointer to a memory buffer containing the ASN.1 message. The pointer must point

		at the first byte in the message.
msglen	int	Length of the message that was read. This is used to set an internal message length to check for field length errors. If this length is not known, a zero value can be passed to cause these checks to be bypassed.

Output parameters:

Name	Type	Description
asntag	ASN1TAG*	Pointer to a variable to receive the ASN.1 tag value corresponding to the outer level tag on the message. This value can be tested to determine the appropriate function to call to decode the message. This is an optional parameter, if not needed, a null pointer can be passed.
length	int*	Pointer to a variable to receive the overall length of the message. Note that this is not the length contained in the length field of the outer level tag, but the overall message length taking into account the extra bytes added by the outer level tag. This is an optional parameter, if not needed, a null pointer can be passed.

xd_tag_len - Decode Tag and Length

The xd_tag_len function decodes the tag and length at the current decode pointer location and returns the results.

Calling sequence:

```
status = xd_tag_len (ctxt_p, asntag, length, flags);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A is decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
flags	u_short	Bit flags used to control function operation. The following flags can be set: XM_ADVANCE Advance decode pointer to the contents field. XM_SKIP Skip to next field. Flags are set by or'ing the above mask values together.
msglen	int	Length of the message that was read. This is used to set an internal message length to check for field length errors. If this length is not known, a zero value can be passed to cause these checks to be bypassed.

Output parameters:

Name	Type	Description
asntag	ASN1TAG*	Pointer to a variable to receive the decoded ASN.1 tag value This value is returned as an unsigned short integer in the internal format described in Section 4.1.2.
length	int*	Pointer to a variable to receive the decoded length of the tagged component. The returned value will either be the actual length or the special constant 'ASN_K_INDEFLEN' which indicates indefinite length.

xd_match - Match Tag

The xd_match function does a comparison between the given tag and the tag at the current decode pointer position to determine if they match. It then returns the result of the match operation. Alternately, the function will scan through tags in a message and compare each tag with the given tag and stop when either the tag is found or all tags in the message have been exhausted.

Calling sequence:

```
status = xd_match (ctxt_p, asntag, length, flags);
```

Return value:

Name	Type	Description
status	int	Status of the match operation. Possible values are ASN_OK if match operation was successful, ASN_E_TAGNOTFOU if matching tag not found, or one of the other negative status codes defined in Appendix A if a different error occurs.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
flags	u_short	Bit flags used to control function operation. The following flags can be set: XM_ADVANCE Advance decode pointer to the contents field. XM_SKIP Skip to next field. XM_SEEK Scan tags until match found or EOM (end-of-message). Flags are set by or'ing the above mask values together.

Output parameters:

Name	Type	Description
asntag	ASN1TAG*	Pointer to a variable to receive the decoded ASN.1 tag value. This value is returned as an unsigned short integer in the internal format described in Section 4.1.2.
length	int*	Pointer to a variable to receive the decoded length of the tagged component. The returned value will either be the actual length or the special constant 'ASN_K_INDEFLEN' which indicates indefinite length.

xd_boolean - Decode BOOLEAN

The xd_boolean function will decode a variable of the ASN.1 BOOLEAN type.

Calling sequence:

```
status = xd_boolean (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in OCTETs, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1BOOL*	Pointer to a variable to receive the decoded boolean value.

xd_integer - Decode INTEGER

The xd_integer function will decode a variable of the ASN.1 INTEGER type.

Calling sequence:

```
status = xd_integer (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in OCTETs, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1INT*	Pointer to a variable to receive the decoded integer value.

xd_unsigned - Decode Unsigned INTEGER

The xd_unsigned function will decode a variable of the unsigned variant of ASN.1 INTEGER type.

Calling sequence:

```
status = xd_unsigned (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in OCTETs, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1UINT*	Pointer to a variable to receive the decoded unsigned integer value.

xd_bigint – Decode Big Integer

The xd_bigint function will decode a variable of the ASN.1 INTEGER type. In this case, the integer is assumed to be of a larger size than can fit in a C or C++ long type (normally 32 or 64 bits). For example, parameters used to calculate security values are typically larger than these sizes.

These variables are stored in character string constant variables. They are represented as decimal strings starting with no prefix. If it is necessary to convert a decimal string to another radix then use rtSetStrToBigInt / rtBigIntToString functions.

Calling sequence:

```
status = xd_bigint (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in OCTETs, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	char**	Pointer to a character pointer variable to receive the decoded unsigned value. Dynamic memory is allocated for the variable using the rtMemAlloc function. The decoded variable is represented as a decimal string starting with no prefix.

xd_bitstr - Decode BIT STRING

The xd_bitstr function will decode a variable of the ASN.1 BIT STRING type. This function will allocate dynamic memory to store the decoded result.

Calling sequence:

```
status = xd_bitstr (ctxt_p, object_p2, numbits_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in OCTETs, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p2	ASN1OCTET**	Pointer to a pointer variable to receive the decoded bit string. Dynamic memory is allocated to hold the string.
numbits_p	int*	Pointer to an integer value to receive the decoded number of bits.

xd_bitstr_s - Decode BIT STRING (static)

The xd_bitstr_s function will decode a variable of the ASN.1 BIT STRING type into a static memory structure. This function call is generated by ASN1C to decode a sized bit string production.

Calling sequence:

```
    status = xd_bitstr_s (ctxt_p, object_p, numbits_p, tagging,
length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is

		successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
numbits_p	int*	Pointer to an integer variable containing the size (in bits) of the sized ASN.1 bit string. An error will occur if the number of bits in the decoded string is larger than this value. Note that this is also used as an output variable – the actual number of decoded bits will be returned in this variable.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in OCTETs, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1OCTET*	Pointer to a variable to receive the decoded bit string. This is assumed to be a static array large enough to hold the number of bits specified in the *numbits_p input parameter.
numbits_p	int*	Pointer to an integer value to receive the decoded number of bits.

xd_octstr - Decode OCTET STRING

The xd_octstr will decode a variable of the ASN.1 OCTET STRING type. This function will allocate dynamic memory to store the decoded result.

Calling sequence:

```
status = xd_octstr (ctxt_p, object_p2, numocts_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description

ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p2	ASN1OCTET**	Pointer to a pointer variable to receive the decoded octet string. Dynamic memory is allocated to hold the string.
numocts_p	int*	Pointer to an integer value to receive the decoded number of octets.

xd_octstr_s - Decode OCTET STRING (static)

The xd_octstr_s function will decode a variable of the ASN.1 OCTET STRING type into a static memory structure. This function call is generated by ASN1C to decode a sized octet string production.

Calling sequence:

```
    status = xd_octstr_s (ctxt_p, object_p, numocts_p, tagging,
length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
numocts_p	int*	Pointer to an integer variable containing the size (in octets) of the sized ASN.1 octet string. An error will occur if the number of octets in the decoded string is larger than this value. Note that this is also used as an output variable – the actual number of decoded octets will be returned in this variable.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has

		meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1OCTET*	Pointer to a variable to receive the decoded octet string. This is assumed to be a static array large enough to hold the number of octets specified in the *numocts_p input parameter.
numocts_p	int*	Pointer to an integer value to receive the decoded number of octets.

xd_charstr – Decode Character String

The xd_charstr function will decode a variable of one of the ASN.1 8-bit character string types. These types include IA5String, VisibleString, PrintableString, and NumericString.

Calling sequence:

```
status = xd_charstr (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	char**	Pointer to a character string pointer variable to receive the decoded string. The string as stored as a standard null-terminated C string. Memory is allocated for the string by the rtMemAlloc function.

xd_16BitCharStr – Decode 16-bit Character String

The xd_16BitCharStr function will decode a variable an ASN.1 16-bit character string type. This includes the BMPString type.

Calling sequence:

```
status = xd_16BitCharStr (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	Asn116Bit CharString*	Pointer to a structure variable to receive the decoded string. The string as stored as an array of short integer characters. Memory is allocated for the string by the rtMemAlloc function.

xd_32BitCharStr – Decode 32-bit Character String

The xd_32BitCharStr function will decode a variable an ASN.1 32-bit character string type. This includes the UniversalString type.

Calling sequence:

```
status = xd_32BitCharStr (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding

		fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	Asn132Bit CharString*	Pointer to a structure variable to receive the decoded string. The string as stored as an array of unsigned integer characters. Memory is allocated for the string by the rtMemAlloc function.

xd_enum - Decode ENUMERATED

The xd_enum function will decode a variable of the ASN.1 ENUMERATED type.

Calling sequence:

```
status = xd_enum (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length

		is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1ENUM*	Pointer to a variable to receive the decoded enumerated value.

xd_null - Decode NULL

The xd_null function will decode an ASN.1 NULL placeholder.

```
status = xd_null (ctxt_p, tagging);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.

Output parameters:

None

xd_objid - Decode OBJECT IDENTIFIER

The xd_objid function will decode a variable of the ASN.1 OBJECT IDENTIFIER type.

Calling sequence:

```
status = xd_objid (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1OBJID*	Pointer to a variable to receive the decoded object identifier value. This structure contains an integer to hold the number of subidentifers in the object and an array to hold the subidentifier values.

xd_real - Decode REAL

The xd_real function will decode a variable of the ASN.1 REAL type.

Calling sequence:

```
status = xd_real (ctxt_p, object_p, tagging, length);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
tagging	ASN1TagType	An enumerated type whose value is set to either 'ASN1EXPL' (for explicit tagging) or 'ASN1IMPL' (for implicit). Controls whether the universal tag value for this type is decoded prior to decoding the field contents. Users will generally always set this value to 'ASN1EXPL'.
length	int	The length, in octets, of the contents field to be decoded. This parameter only has meaning if the tagging parameter specifies implicit decoding. If explicit, the length

		is obtained from the decoded length field.

Output parameters:

Name	Type	Description
object_p	ASN1REAL*	Pointer to a variable to receive the decoded real value.

xd_OpenType - Decode Open Type

The xd_OpenType function will decode a variable of an ASN.1open type. This includes the now deprecated ANY and ANY DEFINED BY types from the 1990 standard as well as other types defined to be open in the new standards (for example, a variable type declaration in an X.681 Information Object Class definition).

Decoding is accomplished by returning a pointer to the encoded message component at the current decode pointer location and skipping to the next field. The caller must then call additional decode functions to further decode the component.

The default behavior of returning a pointer to the location of the message component within the decode message buffer can be changed by setting the ASN1COPYVALUES flag within the context structure. This is done by calling the *rtSetCopyValues* run-time function. If this flag is set, memory is allocated for the message component using *xu_malloc* and the component is copied into the allocated memory.

Calling sequence:

```
status = xd_OpenType (ctxt_p, object_p);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output parameters:

Name	Type	Description
object_p	ASN1ANY*	A pointer to a pointer (**) to hold the address of a byte buffer. This buffer will contain the encoded message component located at the current decode pointer location or a copy of that value if the ASN1COPYVALUES flag is set within the context.

xd_OpenTypeExt – Decode Open Type Extension

The xd_OpenTypeExt function is similar to the xd_OpenType function except that it is used in places where open type extensions are specified. An open type extension is defined as an extensibility marker on a constructed type without any extension elements defined (for example, SEQUENCE { a INTEGER, … }). The difference is that this is an implicit field that can span one or more elements whereas the standard Open Type is assumed to be a single tagged field.

Calling sequence:

```
status = xd_OpenTypeExt (ctxt_p, object_p);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output parameters:

Name	Type	Description
object_p	ASN1ANY*	A pointer to a pointer (**) to hold the address of a byte buffer. This buffer will contain the encoded message component located at the current decode pointer location.

xd_chkend - Check for End of Context

The xd_chkend function determines if the decoder has reached the end of a message context block. The compiler generates calls to this function when decoding a SET or SEQUENCE OF/SET OF construct.

Calling sequence:

```
eoc = xd_chkend (ctxt_p);
```

Return value:

Name	Type	Description
eoc	int	Boolean value indicating whether or not the end-of-context has been reached.

Input parameters:

Name	Type	Description

ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output parameters:

None

xd_count - Count Message Components

The xd_count function looks ahead in the decode buffer and counts the number of message components that make up a SEQUENCE OF or SET OF construct. Calls to this function are generated by the compiler when decoding a SEQUENCE OF or SET OF construct.

Calling sequence:

```
status = xd_count (ctxt_p, length, count_p);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
length	int	The length, in octets, of the SEQUENCE OF or SET OF constructor field.

Output parameters:

Name	Type	Description
count_p	int*	Pointer to a variable to receive the count of elements in the SEQUENCE OF or SET OF construct.

xd_memcpy - Copy Decoded Contents

The xd_memcpy function copies data from the contents field of a message component into the target object.

Calling sequence:

```
status = xd_memcpy (ctxt_p, object_p, length);
```

Return value:

Name	Type	Description
status	int	Status of the copy operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
length	int	The number of bytes to copy from the contents field.

Output parameters:

Name	Type	Description
object_p	void*	A pointer to a memory structure to receive the copied data.

xd_NextElement – Move to Next Element

The xd_NextElement function moves the decode pointer to the next tagged element in the decode buffer. It is useful for use in an error handling callback function because it allows an unknown or bogus element to be skipped.

Calling sequence:

```
status = xd_NextElement (ctxt_p);
```

Return value:

Name	Type	Description
status	int	Status of the copy operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameter:

None

xd_indeflen – Calculate Indefinite Length

The xd_indeflen function calculates the actual length of a message block that was encoded using indefinite length encoding.

Calling sequence:

```
status = xd_indeflen (msg_p);
```

Return value:

Name	Type	Description
status	int	Status of the operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
msg_p	ASN1Const OctetPtr	Pointer a message component that was encoded using indefinite length encoding.

Output Parameters:

None

BER/DER C File Functions

The BER/DER file decode functions allow decode operations to be performed directly on encoded entities within a binary file as opposed to in memory. This makes it possible to parse tag and length variables to determine when pieces of a message can be read into memory. The "tap3batch" sample program provides a good illustration of how these functions are used. They can be applied to a TAP3 batch file to get at the call-detail records for sequential processing without having to read the entire file into memory.

These functions all begin with the prefix "xdf_" to distinguish them from the other decode functions. The following is a description of the various functions that make up this package:

xdf_tag – Decode Tag from File

The xdf_tag function decodes an ASN.1 tag from a file stream into a standard 16-bit ASN.1 tag structure.

Calling sequence:

```
status = xdf_tag (fp, ptag, buffer, pbufidx);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
fp	FILE*	File pointer of binary file to be decoded. It is expected that the current file position is at the first byte of the tag to be decoded.
pbufidx	int*	Pointer to current buffer index containing offset to location where decoded bytes should be copied in the output buffer.

Output parameters:

Name	Type	Description
ptag	ASN1TAG*	A pointer to an ASN.1 tag structure to receive decoded tag.
buffer	ASN1OCTET*	Buffer to receive parsed data.
pbufidx	int*	Updated buffer index set to point at first free byte in buffer after tag is parsed and copied to buffer.

xdf_len – Decode Length from File

The xdf_len function decodes an ASN.1 length from a file stream.

Calling sequence:

```
status = xdf_len (fp, plen, buffer, pbufidx);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
fp	FILE*	File pointer of binary file to be decoded. It is expected that the current file position is at the first byte of the item to be decoded.
pbufidx	int*	Pointer to current buffer index containing offset to location where decoded bytes should be copied in the output buffer.

Output parameters:

Name	Type	Description
plen	int*	A pointer to an integer to receive the decoded length value.
buffer	ASN1OCTET*	Buffer to receive parsed data.
pbufidx	int*	Updated buffer index set to point at first free byte in buffer after tag is parsed and copied to buffer.

xdf_TagAndLen – Decode Tag and Length from File

The xdf_TagAndLen function decodes an ASN.1 tag and length pair from a file stream.

Calling sequence:

```
status = xdf_TagAndLen (fp, ptag, plen, buffer, pbufidx);
```

Return value:

Name	Type	Description
status	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
fp	FILE*	File pointer of binary file to be decoded. It is expected that the current file position is at the first byte of the tag to be decoded.
pbufidx	int*	Pointer to current buffer index containing offset to location where decoded bytes should be copied in the output buffer.

Output parameters:

Name	Type	Description
ptag	ASN1TAG*	A pointer to an ASN1TAG variable to receive parsed tag value.
plen	int*	A pointer to an integer to receive the decoded length value.
buffer	ASN1OCTET*	Buffer to receive parsed data.
pbufidx	int*	Updated buffer index set to point at first free byte in buffer after tag and length are parsed and copied to buffer.

xdf_ReadPastEOC – Read Past End-of-Context (EOC) Marker

The xdf_ReadPastEOC function consumes bytes from the file stream until a matching end-of-context (EOC) marker is found. The bytes read from the file are stored in the given buffer for later processing. An indefinite length marker is assumed to have been parsed prior to calling this function.

Calling sequence:

```
status = xdf_ReadPastEOC (fp, buffer, bufsiz, pbufidx);
```

Return value:

Name	Type	Description
status	int	Status of the read operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters**:**

Name	Type	Description
fp	FILE*	File pointer of binary file to be decoded. It is expected that the current file position is the first byte following an indefinite length marker (0x80 byte).
bufsiz	int	Size of buffer to receive parsed data.
pbufidx	int*	Pointer to current buffer index containing offset to location where decoded bytes should be copied in the output buffer.

Output parameters**:**

Name	Type	Description
buffer	ASN1OCTET*	Buffer to receive parsed data.
pbufidx	int*	Updated buffer index set to point at first free byte in buffer after parsed data is copied to buffer.

xdf_ReadContents – Read Contents from File

This routine reads the contents of a BER tag-length-value (TLV) into the given buffer. The TLV can be of indefinite length.

Calling Sequence:

```
status = xdf_ReadContents (fp, len, buffer, bufsiz, pbufidx);
```

Return value:

Name	Type	Description
status	int	Status of the read operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
fp	FILE*	File pointer of binary file to be decoded. It is expected that the current file position is the first byte following an indefinite length marker (0x80 byte).
len	int	Length of data to be read from file. This can be an indefinite length constant (ASN_K_INDEFLEN) indicating all data up to the corresponding end-of-context (EOC) marker should be read.
bufsiz	int	Size of buffer to receive parsed data.
pbufidx	int*	Pointer to current buffer index containing offset to location where decoded bytes should be copied in the output buffer.

Output parameters:

Name	Type	Description
buffer	ASN1OCTET*	Buffer to receive parsed data.
pbufidx	int*	Updated buffer index set to point at first free byte in buffer after parsed data is copied to buffer.

BER/DER C Utility Functions

BER/DER C utility functions are provided for memory management, formatting output of ASN.1 messages, and error reporting. In many cases, these functions are closely coupled with the *rt* (run-time) series of common functions that are documented later. The common functions provide common functionality shared between the BER/DER and PER run-time libraries. In many cases, these function simply provide direct passthroughs to the *rt* functions to maintain compatibility with existing versions of the ASN1C compiler.

Memory Management Functions (xu_malloc and xu_freeall)

Memory management functions override the standard C malloc and free functions to improve decoding performance. The standard malloc and free functions are expensive in terms of performance. ASN.1 messages frequently contain a large number of small, unsized OCTET STRINGS, BIT STRINGS, and SEQUENCE OF/SET OF constructs. Each of these requires the decoder to allocate dynamic memory for the results. This can lead to poor performance. The ASN1C compiler overcomes this by allocating memory in larger chunks and then breaking it up in subsequent allocation requests. The BER/DER C functions *xu_malloc* and *xu_freeall* are used for this purpose. They provide passthroughs to the *rtMemAlloc* and *rtMemFreeFree* that provide memory management services for both the BER/DER and PER run-time libraries.

xu_malloc - Allocate Dynamic Memory

The xu_malloc function provides a front-end to the C malloc function to allocate dynamic memory for decoded message components. The following ASN.1 constructs may require the allocation of dynamic memory within the generated C structure:

- BIT STRING
- OCTET STRING
- SEQUENCE OF
- SET OF
- ANY

This function uses a nibble memory management scheme to make memory allocations more efficient. On an initial allocation request, malloc will be called to obtain a large block of memory. This memory will then be subdivided as subsequent calls to this function are made. When the block is expired, another call to malloc will be made to allocate another large block and the subdivision process repeated. All large memory block allocated from within the context are freed when *xu_freeall* function is called.

Note that the main logic for the *xu_malloc* function is now in the *rtMemAlloc* function in the common run-time library. This function is still maintained for compatibility purposes, but it acts as a pass-through to the *rtMemAlloc* function.

Calling sequence:

```
ptr = xu_malloc (ctxt_p, memsiz);
```

Return value:

Name	Type	Description
ptr	void*	Pointer to the allocated dynamic memory block. Will be null if a free block of the requested size does not exist.

Input parameters:

Name	Type	Description

ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
memsiz	int	The number of bytes to allocate.

Output parameters:

None

xu_alloc_array – Allocate Elements for an Array

The xu_alloc_array function will allocate space for a given count of fixed size elements.

Calling Sequence:

```
xu_alloc_array (ctxt_p, seqOf_p, recSize, recCount);
```

Return value:

None

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
recSize	int	The number of bytes in one record in the array.
recCount	int	Number of records to allocate.

Output parameters:

Name	Type	Description
seqOf_p	ASN1SeqOf*	Pointer to a generic sequence of structure variable to receive the returned memory. This structure contains a record count and data pointer element. The record count is populated with the recCount passed into the function. The data pointer is set to the value that is returned from the memory allocation function.

xu_freeall - Free Dynamic Memory

The xu_freeall function frees up any dynamic memory allocated by the decode functions in the course of decoding a message. A call to this function releases all memory previously allocated using xu_malloc within the given context.

Note that the main logic for the *xu_freeall* function call is now in the *rtMemFree* function in the common run-time library. This function is still maintained for compatibility purposes, but it acts as a pass-through to the *rtMemFree* function.

Calling sequence:

```
xu_freeall (ctxt_p);
```

Return value:

None

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

None

Output Formatting Functions

The output formatting functions allow BER/DER ASN.1 messages to be displayed in a human-readable format. They display information on the tag, length, and contents of each field in a message. The primary function within this class is the xu_dump function. A callback function mechanism is provided to allow the user to redirect formatted output to somewhere other then stdout (for example, to a syslog type logging daemon process).

xu_dump - Dump Encoded ASN.1 Message

The xu_dump function dumps an encoded ASN.1 message to the standard output device or to another interface in a formatted display. The display includes, for each message component, message tag (class, form, and ID code), length, and data (in hexadecimal and ascii formats).

Output to another interface is accomplished through the callback function parameter. The prototype for this function is as follows:

```
int callback_function (char* text_p, void* cbArg_p);
```

This function is invoked for each line of text formatted from the given message. The formatted line is passed on the text_p argument. The cbArg_p argument allows a user defined callback argument to be passed to the callback function. This argument is specified in the call to xu_dump.

Use of the callback function is optional. If dump to standard output (stdout) is desired, the argument should be specified as NULL (note: the macro XU_DUMP in asn1type.h can be used for this purpose).

Calling sequence:

```
status = xu_dump (msgptr, cbFunc, cbArg_p)
```

Return value:

Name	Type	Description
status	int	Status of the dump operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description

msgptr	ASN1OCTET*	Pointer to an encoded ASN.1 message.
cbFunc	CB Function	Callback function that gets invoked for each line of formatted output. For dump to standard output (stdout), this parameter can be specified as NULL.
cbArg_p	void*	Callback function argument, will be passed to the callback function.

Output parameters:

None

xu_fdump - Dump Encoded ASN.1 Message to a Text File

The xu_fdump function dumps an encoded ASN.1 message to a text file. The display includes, for each message component, message tag (class, form, and ID code), length, and data (in hexadecimal and ascii formats).

Calling sequence:

```
status = xu_fdump (file_p, msgptr)
```

Return value:

Name	Type	Description
status	int	Status of the dump operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters**:**

Name	Type	Description
file_p	FILE*	Text file pointer.
msgptr	ASN1OCTET*	Pointer to an encoded ASN.1 message.

Output parameters:

None

xu_hex_dump - Dump Binary Data

The xu_hex_dump function dumps binary data in raw hexadecimal and ascii formats. It can be used to examine data going in to or out of the run-time library encode/decode functions. This function outputs data only to the standard output device.

Calling sequence:

```
xu_hex_dump (data, numocts, hdrflg)
```

Return value:

None

Input parameters:

Name	Type	Description
data	ASN1OCTET*	Pointer to the start of the block of memory to be dumped.
numocts	int	Number of octets (bytes) to be dumped.
hdrflg	ASN1OCTET	Boolean variable indicating whether or not a header line should be dumped as the first line of the display.

Output parameters:

None

Run-Time Error Reporting Functions

Error reporting functions allow the ASN1C generated functions to report specific information on internal errors. These functions allow for parameter substitution with error strings. These are embedded within the code generated by the ASN1C compiler.

This class of functions provides a compatible passthrough to the *rtErr* common functions. These functions provide common error management services for both the BER/DER and PER run-time libraries.

xu_perror – Print Error Information

The xu_perror function prints information about the last recorded error within the given ASN.1 context structure. The display includes information on the module that generated the error, the source code line number, the status, and a parameterized error message.

Calling sequence:

```
xu_perror (ctxt_p)
```

Return value:

None

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls. This structure holds information on the last error that occurred during encoding or decoding.

Output parameters:

None

xu_log_error – Log Error Information

This function is identical to xu_perror except it allows the error output to be redirected to another interface (for example, to a syslog-type logging daemon). This is accomplished through the callback function parameter. The prototype for this function is as follows:

```
int callback_function (char* text_p, void* cbArg_p);
```

The text_p parameter contains the formatted error message. The cbArg_p argument allows a user defined callback argument to be passed to the callback function. This argument is specified in the call to xu_log_error.

The callback function is invoked twice on a call to xu_log_error. The first call contains error message text indicating the module, line number, and status of the error. The second call contains the parameterized error message text.

Calling sequence:

```
xu_log_error (ctxt_p, cbFunc, cbArg_p);
```

Return value:

None

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls. This structure holds information on the last error that occurred during encoding or decoding.
cbFunc	CB Function	Callback function that gets invoked for each line of formatted error text.
cbArg_p	void*	Callback function argument.

Output parameters:

None

xu_fmtErrMsg – Format Error Message

The xu_fmtErrMsg function provides the user with the parameterized error text corresponding to the last error recorded in the given ASN.1 context structure.

Calling sequence:

```
text_p = xu_fmtErrMsg (ctxt_p, bufp);
```

Return value:

Name	Type	Description
text_p	char*	Pointer to the formatted error message. This is the address of the buffer passed as the second input argument.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls. This structure holds information on the last error that occurred during encoding or decoding.

bufp	char*	Pointer to a text buffer in which the parameterized error message is to be formatted. The caller is responsible for allocating space for this message. A 128 byte buffer should be sufficient for all messages.

Output parameters:

None

< this page intentionally left blank >

PER Run-Time Library

The packed encoding rules low-level C encode/decode functions are another part of the ASN1C run-time library. These functions are identified by their prefixes: pe_ for PER encode, pd_ for PER decode, and pu_ for PER utility functions. The following sections describe these functions.

PER C Encode Functions

The PER low-level encode functions handle the PER encoding of the primitive ASN.1 data types. Calls to these functions are assembled in the C source code generated by the ASN1C compiler to accomplish the encoding of complex ASN.1 structures. These functions are also directly callable from within a user's application program if the need to accomplish a low level encoding function exists.

The procedure to call a low-level encode function is the same as the procedure to call a compiler generated encode function described above. The *pu_initContext* or *pu_newContext* function must first be called to set a pointer to the buffer into which the variable is to be encoded. A static encode buffer is specified by specifying a pointer to a buffer and buffer size. Setting the buffer address to NULL and buffer size to 0 specifies a dynamic buffer. The encode function is then invoked. The result of the encoding will start at the beginning of the specified buffer, or, if a dynamic buffer was used, can be obtained by calling *pe_GetMsgPtr*. The length of the encoded component is obtained by calling *pe_GetMsgLen*.

For example, the following code fragment could be used to encode a single, boolean value (i.e., a single bit).

```
ASN1OCTET buf[10], *msg_p;
ASN1BOOL  boolValue = 1;  /* true */
ASN1CTXT  ctxt;
ASN1BOOL  aligned = 1;
int msglen, stat;

pu_initContext (&ctxt, buf, sizeof(buf), aligned);

stat = pe_bit (&ctxt, &boolValue, ASN1EXPL);
if (stat != ASN_OK) {
   rtErrPrint (&ctxt.errInfo);
   exit (-1);
}
msglen = pe_GetMsgLen (&ctxt);
```

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

pe_GetMsgLen – Get Length of Encoded Message

The pe_GetMsgLen function will return the length of an encoded message. This function is called after a compiler generated encode function is called to get the length of the encoded component

Calling sequence:

```
len = pe_GetMsgLen (ctxt_p);
```

Return value:

Name	Type	Description
len	int	Length (in octets) of encoded message component.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

None

pe_GetMsgBitCnt – Get Count of Bits in Encoded Message

The pe_GetMsgBitCnt function will return the number of bits in an encoded message. This function is called after a compiler generated encode function is called to get the bit count of the encoded component

Calling sequence:

```
len = pe_GetMsgBitCnt (ctxt_p);
```

Return value:

Name	Type	Description
len	int	Length (in bits) of encoded message component.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

None

pe_GetMsgPtr – Get Encoded Message Pointer

The pe_GetMsgPtr function will return the message pointer and length of an encoded message. This function is called after a compiler generated encode function to get the pointer and length of the message. It is normally used when dynamic encoding is specified because the message pointer is not known until encoding is complete. If static encoding is used, the message starts at the beginning of the specified buffer and the pe_GetMsgLen function can be used to obtain the length of the message.

Calling sequence:

```
ptr = pe_GetMsgPtr (ctxt_p, pLength);
```

Return value:

Name	Type	Description
ptr	ASN1OCTET*	Pointer to start of encoded message.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pLength	int*	Pointer to variable to receive length of encoded message.

pe_bit - Encode a Single Bit Value

The pe_bit function will encode a variable of the ASN.1 BOOLEAN type in a single bit.

Calling sequence:

```
stat = pe_bit (ctxt_p, object);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1BOOL	The BOOLEAN value to be encoded.

Output Parameters:

None

pe_bits - Encode Bit Values

The pe_bits function will encode multiple bits.

Calling sequence:

```
stat = pe_bits (ctxt_p, value, nbits);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1UINT	Unsigned integer containing the bits to be encoded.
nbits	ASN1UINT	Number of bits in value to encode.

Output Parameters:

None

pe_octets - Encode Octets

The pe_octets function will encode an array of octets. The octets will be encoded unaligned starting at the current bit offset within the encode buffer.

Calling sequence:

```
stat = pe_octets (ctxt_p, pvalue, nocts);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pvalue	ASN1OCTET*	Pointer to array of octets to encode.
nocts	ASN1UINT	Number of octets to encode.

Output Parameters:

None

pe_byte_align – Align Encode Buffer on a Byte Boundary

The pe_byte_align function will position the encode bit cursor on the next byte boundary.

Calling sequence:

```
stat = pe_byte_align (ctxt_p);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

None

pe_NonNegBinInt – Encode a Non-negative Binary Integer

The pe_NonNegBinInt function will encode a non-negative binary integer as specified in Section 10.3 of the X.691 standard.

Calling sequence:

```
stat = pe_NonNegBinInt (ctxt_p, value);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1UINT	Unsigned integer value to be encoded.

Output Parameters:

None

pe_2sCompBinInt – Encode a Two's Complement Binary Integer

The pe_2sCompBinInt function will encode a two's complement binary integer as specified in Section 10.4 of the X.691 standard.

Calling sequence:

```
stat = pe_2sCompBinInt (ctxt_p, value);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1INT	Signed integer value to be encoded.

Output Parameters:

None

pe_ConsWholeNumber – Encode a Constrained Whole Number

The pe_ConsWholeNumber function will encode a constrained whole number as specified in Section 10.5 of the X.691 standard.

Calling sequence:

```
stat = pe_ConsWholeNumber (ctxt_p, adjusted_value, range_value);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
adjusted_ value	ASN1UINT	Unsigned adjusted integer value to be encoded. The adjustment is done by subtracting the lower value of the range from the value to be encoded.
range_ value	ASN1UINT	Unsigned integer value specifying the total size of the range. This is obtained by subtracting the lower range value from the upper range value.

Output Parameters:

None

pe_SmallNonNegWholeNumber – Encode a Small Non-negative Whole Number

The pe_SmallNonNegWholeNumber function will encode a small non-negative whole number as specified in Section 10.6 of the X.691 standard. This is a number that is expected to be small, but whose size is potentially unlimited due to the presence of an extension marker.

Calling sequence:

```
stat = pe_SmallNonNegWholeNumber (ctxt_p, value);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1UINT	Unsigned integer value to be encoded.

Output Parameters:

None

pe_Length – Encode a Length Determinant

The pe_Length function will encode a length determinant value.

Calling sequence:

```
stat = pe_Length (ctxt_p, value)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1UINT	Length value to be encoded.

Output Parameters:

None

pe_ConsInteger – Encode a Constrained Integer

The pe_ConsInteger function will encode an integer constrained either by a value or value range constraint.

Calling sequence:

```
stat = pe_ConsInteger (ctxt_p, value, lower, upper)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1INT	Value to be encoded.
lower	ASN1INT	Lower range value.
upper	ASN1INT	Upper range value.

Output Parameters:

None

pe_UnconsInteger – Encode an Unconstrained Integer

The pe_UnconsInteger function will encode an unconstrained integer.

Calling sequence:

```
stat = pe_UnconsInteger (ctxt_p, value)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1INT	Value to be encoded.

Output Parameters:

None

pe_ConsUnsigned – Encode a Constrained Unsigned Integer

The pe_ConsUnsigned function will encode an unsigned integer constrained either by a value or value range constraint. The constrained unsigned integer option is used if:

1. The lower value of the range is >= 0, and
2. The upper value of the range is >= MAXINT

Calling sequence:

```
stat = pe_ConsUnsigned (ctxt_p, value, lower, upper)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1UINT	Value to be encoded.
lower	ASN1UINT	Lower range value.
upper	ASN1UINT	Upper range value.

Output Parameters:

None

pe_UnconsUnsigned – Encode an Unconstrained Unsigned Integer

The pe_UnconsUnsigned function will encode an unconstrained unsigned integer.

Calling sequence:

```
stat = pe_UnconsUnsigned (ctxt_p, value)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

value	ASN1UINT	Value to be encoded.

Output Parameters:

None

pe_BigInteger – Encode Big Integer

The pe_BigInteger function will encode a variable of the ASN.1 INTEGER type. In this case, the integer is assumed to be of a larger size than can fit in a C or C++ long type (normally 32 or 64 bits). For example, parameters used to calculate security values are typically larger than these sizes.

Items of this type are stored in character string constant variables. They can be represented as decimal strings (with no prefixes), as hexadecimal strings starting with a "0x" prefix, as octal strings starting with a "0o" prefix or as binary strings starting with a "0b" prefix. Other radixes are currently not supported.

Calling sequence:

```
stat = pe_BigInteger (ctxt_p, pvalue);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pvalue	char*	A pointer to a character string containing the value to be encoded.

Output Parameters:

None

pe_BitString – Encode a Bit String

The pe_BitString function will encode a value of the ASN.1 bit string type.

Calling sequence:

```
stat = pe_BitString (ctxt_p, numbits, data)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
numbits	ASN1UINT	Number of bits in the string to be encoded.
data	ASN1OCTET*	Pointer to bit string data to be encoded.

Output Parameters:

None

pe_OctetString – Encode an Octet String

The pe_OctetString function will encode a value of the ASN.1 octet string type.

Calling sequence:

```
stat = pe_OctetString (ctxt_p, numocts, data)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
numocts	ASN1UINT	Number of octets in the string to be encoded.
data	ASN1OCTET*	Pointer to octet string data to be encoded.

Output Parameters:

None

pe_Real – Encode Real

The pe_Real function will encode a value of the ASN.1 real type.

Calling sequence:

```
stat = pe_Real (ctxt_p, value)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1REAL	Value to be encoded.

Output Parameters:

None

pe_ObjectIdentifier – Encode Object Identifier

The pe_ObjectIdentifier function will encode a value of the ASN.1 object identifier type.

Calling sequence:

```
stat = pe_ObjectIdentifier (ctxt_p, pvalue)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	ASN1OBJID*	Pointer to value to be encoded. The ASN1OBJID structure contains a numids fields to hold the number of subidentifiers and an array to hold the subidentifier values.

Output Parameters:

None

pe_ConstrainedString – Encode 8-bit Character String

The pe_ConstrainedString function will encode a constrained ASN.1 character string. This function is normally not called directly but rather is called from the Useful Type Character String encode functions discussed in the next section.

Calling sequence:

```
stat = pe_ConstrainedString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
string	char*	Pointer to character string value to be encoded. This is a pointer to a standard null-terminated C string value.
pCharSet	Asn1CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

None

ASN.1 8-bit Character String Encode Functions

The ASN.1 8-bit character string encode functions are used to the standard 8-bit character string types included in the standard. The following functions are included:

- pe_NumericString
- pe_PrintableString
- pe_VisibleString
- pe_IA5String
- pe_GeneralString

In addition, the following macros are provided that call to the above functions to encode other types:

- pe_GeneralizedTime
- pe_UTCTime
- pe_GraphicString
- pe_ObjectDescriptor

The calling sequence is the same for each of these routines. They take as arguments a context pointer, a pointer to null-terminated string to encode, and an optional pointer to a character set to further restrict the contents of the encoded string.

Calling sequence:

```
stat = pe_<string> (ctxt_p, string, pCharSet)
```

where <string> would be replaced with the character string name (for example, NumericString).

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
string	char*	Pointer to character string value to be encoded. This is a pointer to a standard null-terminated C string value.
pCharSet	Asn1CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters. This is an optional parameter, it can be set to NULL to specify no additional constraints.

Output Parameters:

None

pe_16BitConstrainedString – Encode 16-bit Character String

The pe_16BitConstrainedString function will encode a constrained ASN.1 character string. This function is normally not called directly but rather is called from Useful Type Character String encode functions that deal with 16-bit strings. The only function that does that in this release is the *pe_BMPString* function described in the next section.

Calling sequence:

```
stat = pe_16BitConstrainedString (ctxt_p, value, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	Asn116Bit CharString	Character string to be encoded. The structure includes a count field containing the number of characters to encode and an array of unsigned short integers to hold the 16-bit characters to be encoded.
pCharSet	Asn116Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

None

pe_BMPString – Encode BMP Character String

The pe_BMPString function will encode a variable of the ASN.1 BMP character string. This differs from the encode routines for the character strings previously described in that the BMP string type is based on 16-bit characters. A 16-bit character string is modeled using an array of unsigned short integers.

Calling sequence:

```
stat = pe_BMPString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
string	Asn116Bit CharString	Character string to be encoded. The structure includes a count field containing the number of characters to encode and an array of unsigned short integers to hold the 16-bit characters to be encoded.
pCharSet	Asn116Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

None

pe_32BitConstrainedString – Encode 32-bit Character String

The pe_32BitConstrainedString function will encode a constrained ASN.1 character string. This function is normally not called directly but rather is called from Useful Type Character String encode functions that deal with 32-bit strings. The only function that does that in this release is the *pe_UniversalString* function described in the next section.

Calling sequence:

```
stat = pe_32BitConstrainedString (ctxt_p, value, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the

		negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
value	Asn132Bit CharString	Character string to be encoded. The structure includes a count field containing the number of characters to encode and an array of unsigned integers to hold the 32-bit characters to be encoded.
pCharSet	Asn132Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

None

pe_UniversalString – Encode 32-bit Character String

The pe_UniversalString function will encode a variable of the ASN.1 Universal character string. This differs from the encode routines for the character strings previously described in that the Universal string type is based on 32-bit characters. A 32-bit character string is modeled using an array of unsigned integers.

Calling sequence:

```
stat = pe_UniversalString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
string	Asn132Bit CharString	Character string to be encoded. The structure includes a count field containing the number of characters to encode and an array of unsigned integers to hold the 32-bit characters to be encoded.
pCharSet	Asn132Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

None

pe_OpenType – Encode Open Type

The pe_OpenType function will encode an ASN.1 open type. This used to be the ASN.1 ANY type, but now is used in a variety of applications requiring an encoding that can be interpreted by a decoder without an prior knowledge of the type of the variable.

Calling sequence:

```
stat = pe_OpenType (ctxt_p, pOpenType)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pOpenType	ASN1OpenType *	Pointer to open type to be encoded. The open type structure contains a count of octets to be encoded and an array of the octets to be encoded.

Output Parameters:

None

pe_OpenTypeExt – Encode Open Type Extension

The pe_OpenTypeExt function will encode an ASN.1 open type extension. An open type extension field is the data that potentially resides after the … marker in a version-1 message. The open type structure contains a complete encoded bit set including optional element bits or choice index, length, and data. Typically, this data is populated when a version-1 system decodes a version-2 message. The extension fields are retained and can then be re-encoded if a new message is to be sent out (for example, in a store and forward system).

Calling sequence:

```
stat = pe_OpenTypeExt (ctxt_p, pOpenType)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description

ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pOpenType	ASN1OpenType *	Pointer to open type to be encoded. The open type structure contains a count of octets to be encoded and an array of the octets to be encoded.

Output Parameters:

None

pe_CheckBuffer – Check Encode Buffer Size

The pe_CheckBuffer function will determine if the given number of bytes will fit in the encode buffer. If not, either the buffer is expanded (if it is a dynamic buffer) or an error is signaled.

Calling sequence:

```
stat = pe_CheckBuffer (ctxt_p, nbytes)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
nbytes	int	Number of bytes of space required to hold the variable to be encoded.

Output Parameters:

None

pe_ExpandBuffer – Expand Encode Buffer

The pe_ExpandBuffer function will expand the encode buffer to hold the given number of bytes.

Calling sequence:

```
stat = pe_ExpandBuffer (ctxt_p, nbytes)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
nbytes	int	Number of bytes the buffer is to be expanded by. Note that the buffer will be expanded by ASN_K_ENCBUFSIZ or nbytes (whichever is larger).

Output Parameters:

None

PER C Decode Functions

PER run-time library decode functions handle the decoding of the primitive ASN.1 data types and length variables. Calls to these functions are assembled in the C source code generated by the ASN1C compiler to decode complex ASN.1 structures. These functions are also directly callable from within a user's application program if the need to decode a primitive data item exists.

The procedure to decode a primitive data item is as follows:

1. Call the *pu_newContext* or *pu_initContext* function to specify the address of the buffer containing the encoded ASN.1 data to be decode and whether the data is aligned or unaligned, and

2. Call the specific decode function to decode the value.

For example, to decode a message containing a single object identifier, the following code fragment could be used:

```
ASN1OBJID objId;  /* variable to receive decoded result */
ASN1CTXT  ctxt;
ASN1BOOL  aligned = TRUE;
int       stat;

/* assume 'buf' contains message fragment to be decoded.. */

pu_initContext (&ctxt, buf, sizeof(buf), aligned);

stat = pd_ObjectIdentifier (&ctxt, &objId);
if (stat != ASN_OK) {
   rtErrPrint (&ctxt.errInfo);
}
```

The objId variable now contains the decoded object identifier value.

pd_bit - Decode a Single Bit Value

The pd_bit function will decode a single bit and place the result in an ASN.1 BOOLEAN type variable.

Calling sequence:

```
stat = pd_bit (ctxt_p, pvalue);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pvalue	ASN1BOOL*	Pointer to BOOLEAN value to receive decoded result.

pd_bits - Decode Bit Values

The pd_bits function will decode a series of multiple bits and place the results in an unsigned integer variable.

Calling sequence:

```
stat = pd_bits (ctxt_p, pvalue, nbits);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
nbits	ASN1UINT	Number of bits to decode.

Output Parameters:

Name	Type	Description
pvalue	ASN1UINT*	Pointer to unsigned integer variable to receive decoded result.

pd_byte_align – Align Buffer on a Byte Boundary

The pe_byte_align function will position the decode bit cursor on the next byte boundary.

Calling sequence:

```
stat = pd_byte_align (ctxt_p);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

None

pd_ConsWholeNumber – Decode a Constrained Whole Number

The pd_ConsWholeNumber function will decode a constrained whole number as specified in Section 10.5 of the X.691 standard.

Calling sequence:

```
stat = pd_ConsWholeNumber (ctxt_p, padjusted_value, range_value);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
range_ value	ASN1UINT	Unsigned integer value specifying the total size of the range. This is obtained by subtracting the lower range value from the upper range value.

Output Parameters:

Name	Type	Description
padjusted_ value	ASN1UINT*	Pointer to unsigned adjusted integer value to receive decoded result. To get the final value, this value is added to the lower boundary of the range.

pd_SmallNonNegWholeNumber – Decode a Small Non-negative Whole Number

The pd_SmallNonNegWholeNumber function will decode a small non-negative whole number as specified in Section 10.6 of the X.691 standard. This is a number that is expected to be small, but whose size is potentially unlimited due to the presence of an extension marker.

Calling sequence:

```
stat = pd_SmallNonNegWholeNumber (ctxt_p, pvalue);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pvalue	ASN1UINT*	Pointer to unsigned integer value to receive decoded result.

pd_Length – Decode a Length Determinant

The pd_Length function will decode a length determinant value.

Calling sequence:

```
stat = pd_Length (ctxt_p, pvalue)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pvalue	ASN1UINT*	Pointer to unsigned integer variable to receive decoded length value.

pd_ConsInteger – Decode a Constrained Integer

The pd_ConsInteger function will decode an integer constrained either by a value or value range constraint.

Calling sequence:

```
stat = pd_ConsInteger (ctxt_p, pvalue, lower, upper)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
lower	ASN1INT	Lower range value.
upper	ASN1INT	Upper range value.

Output Parameters:

Name	Type	Description
pvalue	ASN1INT*	Pointer to integer variable to receive decoded value.

pd_UnconsInteger – Decode an Unconstrained Integer

The pd_UnconsInteger function will decode an unconstrained integer.

Calling sequence:

```
stat = pd_UnconsInteger (ctxt_p, pvalue)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pvalue	ASN1INT*	Pointer to integer variable to receive decoded result.

pd_ConsUnsigned – Decode a Constrained Unsigned Integer

The pd_ConsUnsigned function will decode an unsigned integer constrained either by a value or value range constraint.

Calling sequence:

```
stat = pd_ConsUnsigned (ctxt_p, pvalue, lower, upper)
```

Return value:

Name	Type	Description

stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
lower	ASN1UINT	Lower range value.
upper	ASN1UINT	Upper range value.

Output Parameters:

Name	Type	Description
pvalue	ASN1UINT*	Pointer to unsigned integer variable to receive decoded result.

pd_UnconsUnsigned – Decode an Unconstrained Unsigned Integer

The pd_UnconsUnsigned function will decode an unconstrained unsigned integer.

Calling sequence:

```
stat = pd_UnconsUnsigned (ctxt_p, pvalue)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pvalue	ASN1UINT*	Pointer to unsigned integer variable to receive decoded result.

pd_BigInteger – Decode a Big Integer

The pd_BigInteger function will decode a variable of the ASN.1 INTEGER type. In this case, the integer is assumed to be of a larger size than can fit in a C or C++ long type (normally 32 or 64 bits). For example, parameters used to calculate security values are typically larger than these sizes.

These variables are stored in character string constant variables. They are represented as decimal strings starting with no prefix. If it is necessary to convert a decimal string to another radix then use rtSetStrToBigInt / rtBigIntToString functions.

Calling sequence:

```
stat = pd_BigInteger (ctxt_p, ppvalue);
```

Return value:

Name	Type	Description
stat	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output parameters:

Name	Type	Description
ppvalue	char**	Pointer to a character pointer variable to receive the decoded unsigned value. Dynamic memory is allocated for the variable using the rtMemAlloc function. The decoded variable is represented as a decimal string starting with no prefix.

pd_BitString – Decode a Bit String

The pd_BitString function will decode a value of the ASN.1 bit string type whose maximum size is known in advance. The ASN1C compiler generates a call to this function to decode bit string productions or elements that contain a size constraint.

Calling sequence:

```
stat = pd_BitString (ctxt_p, numbits_p, buffer, bufsiz)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
bufsiz	ASN1UINT	Length (in octets) of buffer to receive decoded bit string.

Output Parameters:

Name	Type	Description
numbits_p	ASN1UINT*	Pointer to unsigned integer variable to receive decoded number of bits.
buffer	ASN1OCTET*	Pointer to fixed-size or pre-allocated array of bufsiz octets to receive decoded bit string.

pd_DynBitString - Decode a Dynamic Bit String

The pd_DynBitString function will decode a variable of the ASN.1 BIT STRING type. This function will allocate dynamic memory to store the decoded result. The ASN1C compiler generates a call to this function to decode an unconstrained bit string production or element.

Calling sequence:

```
stat = pd_DynBitString (ctxt_p, pBitStr)
```

Return value:

Name	Type	Description
stat	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output parameters:

Name	Type	Description
pBitStr	ASN1DynBitStr *	Pointer to a dynamic bit string structure to receive the decoded result. This structure contains a field to hold the number of decoded bits and a pointer to an octet string to hold the decoded data. Memory is allocated by the decoder using the rtMemAlloc function. This memory is tracked within the context and released when the pu_freeContext function is invoked.

pd_OctetString – Decode an Octet String

The pd_OctetString function will decode a value of the ASN.1 octet string type whose maximum size is known in advance. The ASN1C compiler generates a call to this function to decode octet string productions or elements that contain a size constraint.

Calling sequence:

```
stat = pd_OctetString (ctxt_p, numocts_p, buffer, bufsiz)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
bufsiz	ASN1UINT	Size of buffer to receive decoded result.

Output Parameters:

Name	Type	Description
numocts_p	ASN1UINT*	Pointer to unsigned integer to receive number of decoded octets.
data	ASN1OCTET*	Pointer to pre-allocated buffer of bufsiz octets to receive decoded data.

pd_DynOctString - Decode a Dynamic Octet String

The pd_DynOctString function will decode a variable of the ASN.1 OCTET STRING type. This function will allocate dynamic memory to store the decoded result. The ASN1C compiler generates a call to this function to decode an unconstrained octet string production or element.

Calling sequence:

```
stat = pd_DynOctString (ctxt_p, pOctStr)
```

Return value:

Name	Type	Description
stat	int	Status of the decode operation. Possible values are ASN_OK if decoding is successful or one of the negative status codes defined in Appendix A if decoding fails.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output parameters:

Name	Type	Description
pOctStr	ASN1DynOctStr *	Pointer to a dynamic octet string structure to receive the decoded result. This structure contains a field to hold the number of decoded octets and a pointer to an octet string to hold the decoded data. Memory is allocated by the decoder using the rtMemAlloc function. This memory is tracked within the context and released when

		the pu_freeContext function is invoked.

pd_Real – Decode Real

The pd_Real function will decode a value of the ASN.1 real type.

Calling sequence:

```
stat = pd_Real (ctxt_p, pvalue)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pvalue	ASN1REAL*	Pointer to real variable to receive decoded result.

pd_ObjectIdentifier – Decode Object Identifier

The pd_ObjectIdentifier function will decode a value of the ASN.1 object identifier type.

Calling sequence:

```
stat = pd_ObjectIdentifier (ctxt_p, pvalue)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description

pvalue	ASN1OBJID*	Pointer to value to receive decoded result. The ASN1OBJID structure contains a numids fields to hold the number of subidentifiers and an array to hold the subidentifier values.

pd_ConstrainedString – Decode 8-bit Character String

The pd_ConstrainedString function will decode a constrained ASN.1 character string. This function is normally not called directly but rather is called from the Useful Type Character String decode functions discussed in the next section.

Calling sequence:

```
stat = pd_ConstrainedString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pCharSet	Asn1CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

Name	Type	Description
string	char**	Pointer to character string pointer to receive decoded result. The string is returned as a standard null-terminated C string value. Memory is allocated by the decoder using the rtMemAlloc function. This memory is tracked within the context and released when the pu_freeContext function is invoked.

ASN.1 8-bit Character String Decode Functions

The ASN.1 8-bit character string decode functions are used to decode the standard 8-bit character string types included in the standard. The following functions are included:

- pd_NumericString
- pd_PrintableString
- pd_VisibleString
- pd_IA5String
- pd_GeneralString

In addition, the following macros are provided that call to the above functions to encode other types:

- pd_GeneralizedTime
- pd_UTCTime
- pd_GraphicString
- pd_ObjectDescriptor

The calling sequence is the same for each of these routines. They take as arguments a context pointer, the address of a character string pointer variable (i.e a char**) to receive the decoded result, and an optional pointer to a character set to further restrict the contents of the encoded string.

Calling sequence:

```
stat = pd_<string> (ctxt_p, string, pCharSet)
```

where <string> would be replaced with the character string name (for example, NumericString).

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pCharSet	Asn1CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters. This is an optional parameter, it can be set to NULL to specify no additional constraints.

Output Parameters:

Name	Type	Description
string	char**	Pointer to character string pointer to receive decoded result. The string is returned as a standard null-terminated C string value. Memory is allocated by the decoder using the rtMemAlloc function. This memory is tracked within the context and released when the pu_freeContext function is invoked.

pd_16BitConstrainedString – Decode 16-bit Character String

The pd_16BitConstrainedString function will decode a constrained ASN.1 16-bit character string. This function is normally not called directly but rather is called from Useful Type Character String decode functions that deal with 16-bit strings. The only function that does that in this release is the *pd_BMPString* function described in the next section.

Calling sequence:

```
stat = pd_16BitConstrainedString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description

stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pCharSet	Asn116Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

Name	Type	Description
string	Asn116Bit CharString*	Pointer to a structure variable to receive the decoded string. The string as stored as an array of short integer characters. Memory is allocated for the string by the rtMemAlloc function. This memory is tracked within the context and released when the pu_freeContext function is invoked.

pd_BMPString – Decode BMP Character String

The pd_BMPString function will decode a variable of the ASN.1 BMP character string. This differs from the decode routines for the character strings previously described in that the BMP string type is based on 16-bit characters. A 16-bit character string is modeled using an array of unsigned short integers.

Calling sequence:

```
stat = pd_BMPString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pCharSet	Asn116Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

Name	Type	Description
string	Asn116Bit CharString*	Pointer to character string structure to receive decoded result. The structure includes a count field containing the number of characters and an array of unsigned short

		integers to hold the 16-bit character values.

pd_32BitConstrainedString – Decode 32-bit Character String

The pd_32BitConstrainedString function will decode a constrained ASN.1 32-bit character string. This function is normally not called directly but rather is called from Useful Type Character String decode functions that deal with 32-bit strings. The only function that does that in this release is the *pd_UniversalString* function described in the next section.

Calling sequence:

```
stat = pd_32BitConstrainedString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pCharSet	Asn132Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

Name	Type	Description
string	Asn132Bit CharString*	Pointer to a structure variable to receive the decoded string. The string as stored as an array of unsigned integer characters. Memory is allocated for the string by the rtMemAlloc function. This memory is tracked within the context and released when the pu_freeContext function is invoked.

pd_UniversalString – Decode 32-bit Character String

The pd_UniversalString function will decode a variable of the ASN.1 32-bit character string. This differs from the decode routines for the character strings previously described in that the universal string type is based on 32-bit characters. A 32-bit character string is modeled using an array of unsigned integers.

Calling sequence:

```
stat = pd_UniversalString (ctxt_p, string, pCharSet)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pCharSet	Asn132Bit CharSet*	Pointer to the constraining character set. This contains an array containing all valid characters in the set as well as the aligned and unaligned bit counts required to encode the characters.

Output Parameters:

Name	Type	Description
string	Asn132Bit CharString*	Pointer to character string structure to receive decoded result. The structure includes a count field containing the number of characters and an array of unsigned integers to hold the 32-bit character values.

pd_OpenType – Decode Open Type

The pd_OpenType function will decode an ASN.1 open type. This used to be the ASN.1 ANY type, but now is used in a variety of applications requiring an encoding that can be interpreted by a decoder without an prior knowledge of the type of the variable.

Calling sequence:

```
stat = pd_OpenType (ctxt_p, pOpenType)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pOpenType	ASN1OpenType *	Pointer to open type variable to receive decoded data. The open type structure contains a count of octets and an octet data array to hold the encoded data.

pd_OpenTypeExt – Decode Open Type Extension

The pd_OpenTypeExt function will decode an ASN.1 open type extension. These are the extra fields in a version-2 message that may be present after the … extension marker. An open type structure (extElem1) is added to a message structure that contains an extension marker but no extension elements. The pd_OpenTypeExt function will populate this structure with the complete extension information (optional bits or choice index, length and data). A subsequent call to pe_OpenTypeExt will cause the saved extension fields to be included in a newly encoded message of the given type.

Calling sequence:

```
stat = pd_OpenTypeExt (ctxt_p, pOpenType)
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output Parameters:

Name	Type	Description
pOpenType	ASN1OpenType *	Pointer to open type variable to receive decoded data. The open type structure contains a count of octets and an octet data array to hold the encoded data.

PER C Utility Functions

The PER utility functions are common routines used by both the PER encode and decode functions. Among the services provided are:

- Encode/decode context initialization
- Setting constraint information within the context structure
- Diagnostics printing to examine an encoding
- Character string conversion

Encode/Decode Context Initialization

Before any PER encode or decode function can be invoked, a context structure must be initialized. The following PER utility functions are used for this purpose:

pu_initContext

The pu_initContext function is used to initialize a pre-allocated ASN1CTXT structure. This can be an ASN1CTXT variable declared on the stack or a pointer to an ASN1CTXT structure that was previously allocated. This function sets all internal variables within the structure to their initial values.

Calling sequence:

```
stat = pu_initContext (ctxt_p, bufaddr, bufsiz, aligned);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
bufaddr	ASN1OCTET*	For encoding, this is the address of a buffer to receive the encoded PER message (note: this is optional, if specified as NULL a dynamic buffer will be allocated). For decoding, this is that address of the buffer that contains the PER message to be decoded.
bufsiz	ASN1UINT	For encoding, this is the size of the encoded buffer (note: this is optional, if the bufaddr argument is specified as NULL, then dynamic encoding is in effect and the buffer size is indefinite). For decoding, this is the length (in octets) of the PER message to be decoded.
aligned	ASN1BOOL	Boolean value specifying whether aligned or unaligned encoding should be performed.

Output parameters:

None

pu_initContextBuffer

The pu_initContextBuffer function is used to initialize the buffer portion of an ASN1CTXT structure with buffer data from a second context structure. This function copies the buffer information from the source context buffer structure to the destination structure. The non-buffer related fields in the context remain untouched.

Calling sequence:

```
stat = pu_initContextBuffer (pTarget, pSource);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
pTarget	ASN1CTXT*	Pointer to target context structure. Buffer information within this structure is updated with data from the source context.
pSource	ASN1CTXT*	Pointer to source context structure. Buffer information from the source context structure is copied to the target structure.

Output parameters:

None

pu_newContext

The pu_newContext function is similar to the pu_initContext function in that it initializes a context variable. The difference is that this function allocates a new structure and then initializes it. It is equivalent to calling malloc to allocate a context structure and then calling pu_initContext to initialize it.

Calling sequence:

```
ctxt_p = pu_newContext (bufaddr, bufsiz, aligned);
```

Return value:

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to ASN1CTXT structure to receive the allocated structure. NULL is returned if an error occurs in allocating or initializing the context.

Input parameters:

Name	Type	Description
bufaddr	ASN1OCTET*	For encoding, this is the address of a buffer to receive the encoded PER message (note: this is optional, if specified as NULL a dynamic buffer will be allocated). For decoding, this is that address of the buffer that contains the PER message to be decoded.

bufsiz	ASN1UINT	For encoding, this is the size of the encoded buffer (note: this is optional, if the bufaddr argument is specified as NULL, then dynamic encoding is in effect and the buffer size is indefinite). For decoding, this is the length (in octets) of the PER message to be decoded.
aligned	ASN1BOOL	Boolean value specifying whether aligned or unaligned encoding should be performed.

Output parameters:

None

pu_freeContext

The pu_freeContext function releases all dynamic memory associated with a context. This function should be called even if the referenced context variable is not dynamic. The reason is because it frees memory allocated within the context as well as the context structure itself (it will only try to free the context structure if it detects that it was previously allocated using the pu_newContext function).

Calling sequence:

```
pu_freeContext (ctxt_p);
```

Return value:

None

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.

Output parameters:

None

Constraint Specification Functions

Unlike BER, PER encode/decode behavior can be affected by the presence of constraint specifications. The following PER utility functions allow constraint specifications to be added to the context prior to calling an encode or decode function. The ASN1C compiler adds these calls to the generated code when constraints are encountered in the ASN.1 specifications that are being compiled.

pu_addSizeConstraint

The pu_addSizeConstraint is used to add a size constraint to a context variable. A size constraint is specified using an Asn1SizeCnst structure. The definition of an Asn1SizeCnst is as follows:

```
struct Asn1SizeCnst {
   ASN1BOOL    extensible;
   ASN1INT     lower;
   ASN1INT     upper;
   struct Asn1SizeCnst* link;
```

```
} ;
```

The extensible boolean specifies whether or not it is an extensible constraint. The lower and upper fields are used to represent the actual size constraint bounds. The link field is used to chain multiple size constraint records together. This makes it possible to specify composite size constraints that are specified in multiple parts using ASN.1 union or extensibility syntax (for example, SIZE (1|3..5,…,7..10)).

Calling Sequence:

```
stat = pu_addSizeConstraint (ctxt_p, pSize);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

Input parameters:

Name	Type	Description
pSize	Asn1SizeCnst*	Pointer to size constraint to add to the context variable.

Output parameters:

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure. The referenced size constraint is added to this structure for use by a subsequent encode or decode function.

pu_setCharSet

The pu_setCharSet function sets a permitted alphabet character set. This is the resulting set of characters when the character associated with a standard character string type is merged with a permitted alphabet constraint.

Calling Sequence:

```
pu_setCharSet (pCharSet, permSet)
```

Return Value:

None

Input parameters:

Name	Type	Description
pCharSet	Asn1CharSet*	Pointer to character set structure describing the character set currently associated with the character string type.
permSet	char*	Null-terminated string of permitted characters.

Output parameters:

Name	Type	Description

pCharSet	Asn1CharSet*	Resulting character set structure after being merged with the permSet parameter.

pu_set16BitCharSet

The pu_set16BitCharSet function sets a permitted alphabet character set for 16-bit character string. This is the resulting set of characters when the character associated with a 16-bit character string type is merged with a permitted alphabet constraint.

Calling Sequence:

```
pu_set16BitCharSet (pCharSet, pAlphabet)
```

Return Value:

None

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
pCharSet	Asn116Bit CharSet*	Pointer to character set structure describing the character set currently associated with the character string type.
pAlphabet	Asn116Bit CharSet*	Pointer to structure describing 16-bit permitted alphabet.

Output parameters:

Name	Type	Description
pCharSet	Asn116Bit CharSet*	Resulting character set structure after being merged with the permSet parameter.

pu_set32BitCharSet

The pu_set32BitCharSet function sets a permitted alphabet character set for 32-bit character string. This is the resulting set of characters when the character associated with a 32-bit character string type is merged with a permitted alphabet constraint.

Calling Sequence:

```
pu_set32BitCharSet (pCharSet, pAlphabet)
```

Return Value:

None

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
pCharSet	Asn132Bit CharSet*	Pointer to character set structure describing the character set currently associated with the character string type.

pAlphabet	Asn132Bit CharSet*	Pointer to structure describing 32-bit permitted alphabet.

Output parameters:

Name	Type	Description
pCharSet	Asn132Bit CharSet*	Resulting character set structure after being merged with the permSet parameter.

Diagnostic Printing Functions

PER utility functions can be used to track the bit encoding or decoding of individual fields and get a detailed binary dump of the encoding. Several of these function calls are built directly into PER encode/decode functions.

pu_hexdump

The pu_hexdump function provides a standard hexadecimal dump of the contents of the buffer currently specified in the given context.

Calling Sequence:

```
pu_hexdump (ctxt_p)
```

Return Value:

None

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure. The contents of the encode or decode buffer that was specified in the call to pu_initContext or pu_newContext is dumped.

Output parameters:

None

pu_bindump

The pu_bindump function provides a detailed binary dump of the contents of the buffer currently specified in the given context. The list of fields dumped by this function was previously built up within the context using calls *pu_newField*, *pu_pushName*, and *pu_popName*. These calls are built into both compiler-generated and low-level PER encode/decode functions to trace the actual bit encoding of a given construct.

Calling Sequence:

```
pu_bindump (ctxt_p, varname)
```

Return Value:

None

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure. The contents of the encode or decode buffer that was specified in the call to pu_initContext or pu_newContext is dumped.
varname	char*	Name of the top-level variable name of the structure being dumped.

Output parameters:

None

Run-Time Common Library

The run-time common library is another set of common functions used by both the BER and PER low-level encode/decode functions. These functions are identified by their 'rt' prefixes. The following general categories of functions are provided:

- Context initialization functions
- Memory management functions
- Diagnostic trace functions
- Error formatting and print functions
- Formatted printing functions
- Object identifier helper functions
- Linked list and stack utility functions
- Character string conversion utility functions

The following sections describe these functions.

Context Initialization Functions

Context initialization functions handle the allocation, initialization, and destruction of ASN.1 context variables (variables of type ASN1CTXT). These variables hold all of the working data used during the process of encoding or decoding a message. They provide thread safe operation by isolating what would be otherwise be global variables within this structure that is passed from function to function.

In general, the BER and PER run-time libraries provide specific higher-level functions that invoke these functions (for example, the BER xe_setp function and the PER pu_initContext functions calls rtInitContext). Therefore, the average user would have little reason to call them directly. They are documented here for completeness.

rtInitContext – Initialize Context Block

The rtInitContext function initializes an ASN1CTXT block by setting all key working parameters to their correct initial state values.

Calling Sequence:

```
int rtInitContext (ctxt_p);
```

Return value:

Name	Type	Description
stat	int	Status of the operation. Possible values are ASN_OK if successful or one of the negative error status codes defined in Appendix A.

rtNewContext – Allocate New Context Block

The *rtNewContext* function allocates a new ASN1CTXT block and initializes it. Although the block is allocated from the standard heap, it should not be freed using free. The *rtFreeContext* function should be used because this frees items allocated within the block before freeing the block itself.

This is the preferred way of setting up a new encode or decode context because it ensures the block is properly initialized before it is used. If a context variable is declared on the stack, the user must first

remember to initialize it using *rtInitContext*. This function can be called directly when setting up a BER context or it will be invoked from within the *pu_newContext* call for PER.

Calling Sequence:

```
ctxt_p = rtNewContext ();
```

Return value:

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to newly allocated and initialized context structure.

Input Parameters:

None.

Output Parameters:

None.

rtFreeContext – Free Context Block

The *rtFreeContext* functions frees all dynamic memory associated with a context. This includes all memory inside the block (in particular, the list of memory blocks used by the *rtMem* functions described later) as well as the block itself if allocated with the *rtNewContext* function.

Calling Sequence:

```
rtFreeContext (ctxt_p);
```

Return value:

None.

Input Parameters:

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to newly allocated and initialized context structure.

Output Parameters:

None.

Memory Management Functions

Memory management functions handle the allocation and deallocation of dynamic memory used by the encode/decode functions. Specialized routines are used for performance reasons and to allow the allocated memory blocks to be tracked within the context for subsequent release.

These functions are designed to improve the performance of memory allocations within an application. Users with the standard version of the compiler can attain higher performance still by replacing these functions with their own specialized functions. For example, if it is known that only a certain peak

memory usage requirement will be necessary for a certain application, then the nibble allocation algorithm can be replaced with an algorithm that works on a sized static block.

rtMemAlloc – Allocate Dynamic Memory

The rtMemAlloc function allocates dynamic memory. This improves on the standard malloc function by allocating memory in larger chunks and then splitting up these chunks on subsequent calls. The pointers to the large memory blocks are maintained on a list within the context structure so that a free context call can release all memory at once.

Calling Sequence:

```
ptr = rtMemAlloc (ppMemBlk, nbytes)
```

Return value:

Name	Type	Description
ptr	void*	Pointer to allocated memory (note: the user should not call 'free' on this pointer as it points at memory within one of the larger allocated blocks. The rtMemFree function should be called to release all memory allocated using these functions).

Input Parameters:

Name	Type	Description
nbytes	int	Number of bytes of dynamic memory to allocate.

Output Parameters:

Name	Type	Description
ppMemBlk	ASN1MemBlk**	Pointer to pointer to a memory block structure that contains the list of dynamic memory block maintained by these functions. Typically, the address of the memory block list within the ASN1CTXT structure is passed as this parameter (i.e., &ctxt_p->pMemBlk).

rtMemFree – Release Dynamic Memory

The rtMemFree function frees dynamic memory. Unlike the standard C 'free' function, this function releases a set of dynamic memory pointers at once instead of a single pointer. It works this way because it is used by the decoder to keep track of all dynamic memory allocated within an ASN.1 C structure. This function is invoked from within the context free functions (xu_freeall for BER or pu_freeContext for PER).

Calling Sequence:

```
rtMemFree (pMemBlk)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pMemBlk	ASN1MemBlk*	Pointer to a memory block structure that contains the list of dynamic memory blocks maintained by these functions. Typically, the pointer to the memory block

		list within the ASN1CTXT structure is passed as this parameter (i.e., ctxt_p->pMemBlk).

Output Parameters:

None

Diagnostic Trace Functions

Diagnostic trace functions allow the output of trace messages to stdout that trace the execution of compiler generated functions. The primary function is **rtdiag**, a printf-like function that checks a global trace flag before writing to the standard output.

rtdiag – Output Trace Messagesy

The rtdiag function conditionally outputs diagnostic trace messages to stdout. The ASN1C compiler embeds calls to this function into the generated source code when the –trace option is specified on the command line.

Calling Sequence:

```
rtdiag (fmtspec, ...)
```

Return value:

None

Input parameters**:**

Name	Type	Description
fmtspec	char*	printf-like format specification string describing the message to be printed (for example, “string %s, ivalue %d\n”)
…	any	Variable list of arguments

Output parameters:

None

rtSetDiag – Set Diagnostic Tracing

The rtSetDiag function turns diagnostic tracing on or off.

Calling Sequence:

```
rtSetDiag (value)
```

Return value:

None

Input parameters**:**

Name	Type	Description

value	int	Boolean value indicating whether to enable or disable tracing. Zero disables tracing, any other value enables it.

Output parameters:

None

Error Formatting and Print Functions

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

rtErrPrint – Print Error Information

The rtErrPrint function prints error information to the standard output device. The error information is stored in an ASN1ErrInfo structure which is part of the ASN1CTXT structure.

Calling Sequence:

```
rtErrPrint (pErrInfo)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pErrInfo	ASN1ErrInfo*	Pointer to structure containing information on the error to be printed. Typically, the error info structure referred to is the one inside the ASN1CTXT structure (i.e., &ctxt_p->errInfo).

Output Parameters:

None

rtErrLogUsingCB – Log Using Callback Function

The rtErrLogUsingCB function logs error information using a callback function provided by the user. In many situations, it is not sufficient to write error information to stdout to debug problems. Examples are back-end server applications that run in the background and write diagnostic information to system log files and front-end applications that log error information to window displays. This function allows different error output methods to be accommodated.

The type definition of the callback function is as follows:

```
typedef int (*ASN1DumpCbFunc) (char* text_p, void* cbArg_p)
```

The given function is invoked with a line of text from the formatted error output provided in the text_p argument. The cbArg_p argument is used to pass in a user-defined parameter (specified in the cbArg argument to this function). The integer return status is not used at this time.

The callback function is invoked once for each formatted line of information in the error holding structure.

Calling Sequence:

```
rtLogUsingCB (pErrInfo, cb, cbArg)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pErrInfo	ASN1ErrInfo*	Pointer to structure containing information on the error to be printed. Typically, the error info structure referred to is the one inside the ASN1CTXT structure (i.e., &ctxt_p->errInfo).
cb	ASN1Dump CBFunc	Callback function as defined above to be invoked for each line of formatted error output in the error information structure.
cbArg	void*	User defined callback argument to be passed as a parameter to the callback function.

Output Parameters:

None

rtErrSetData – Set Error Information

The rtErrSetData function sets error information in an error information structure. The information set includes status code, module name, and line number. Location information (i.e., module name and line number) is pushed onto a stack within the error information structure to provide a complete stack trace when the information is printed out.

Calling Sequence:

```
stat = rtErrSetData (pErrInfo, status, module, lno)
```

Return Value:

Name	Type	Description
stat	int	Status value passed to the operation in the third argument. This makes it possible to set the error information and return the status value in one line of code.

Input Parameters:

Name	Type	Description
status	int	Error status code. This is one of the negative error status codes described in Appendix A.
module	char*	Name of the module (C or C++ source file) in which the module occurred. This is typically obtained by using the __FILE__ macro.
lineno	int	Line number at which the error occurred. This is typically obtained by using the __LINE__ macro.

Output Parameters:

Name	Type	Description
pErrInfo	ASN1ErrInfo*	Pointer to an error information structure to receive the details on the error. This is typically the error structure variable within the context (i.e., &ctxt_p->errInfo).

rtErrAdd<type>Param – Add Typed Error Parameter to Error Information

The rtErrAdd<type>Param functions add typed parameters to an error information structure. This section describes a series of functions whose name is formed by substituting a type identifier name for <type> in the above definition (for example, rtErrAddIntParam adds an integer parameter to an error structure).

Parameter substitution is done in much the same way as it is done in C printf statements. The base error message specification that goes along with a particular status code may have variable fields built in using '%' modifiers. These would be replaced with actual parameter data. The parameters they are replaced with are added using the functions described in this section.

Calling Sequence:

```
stat = rtErrAdd<type>Param (pErrInfo, errParm)
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
errParm	<type>	Typed error parameter.

Output Parameters:

Name	Type	Description
pErrInfo	ASN1ErrInfo*	Pointer to an error information structure to receive the details on the error. This is typically the error structure variable within the context (i.e., &ctxt_p->errInfo).

rtErrFreeParams – Free Error Parameter Memory

The rtErrFreeParams function frees memory associated with the storage of parameters associated with an error message. These parameters are maintained on an internal linked list maintained within the error information structure. The list memory must be freed when error processing is complete. This function is called from within rtErrPrint after the error has been printed out. It is also called in the pu_freeContext function.

Calling Sequence:

```
rtErrFreeParms (pErrInfo)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pErrInfo	ASN1ErrInfo*	Pointer to an error information structure to receive the details on the error. This is typically the error structure variable within the context (i.e., &ctxt_p->errInfo).

Output Parameters:

None

Formatted Printing Functions

This group of functions allows raw ASN.1 data fields to be formatted and printed to stdout and other output devices.

rtBoolToString – Convert ASN.1 Boolean Value to String

The rtBoolToString function converts an ASN.1 boolean value to a string. The string value returned is one of the keywords "TRUE" or "FALSE".

Calling Sequence:

```
string = rtBoolToString (value)
```

Return Value:

Name	Type	Description
string	char*	Converted value. This will be a string literal set to either "TRUE" or "FALSE".

Input Parameters:

Name	Type	Description
value	ASN1BOOL	Value to convert.

Output Parameters:

None

rtIntToString – Convert ASN.1 Integer Value to String

The rtIntToString function converts an ASN.1 integer value to a string.

Calling Sequence:

```
string = rtIntToString (value, buffer, bufsiz)
```

Return Value:

Name	Type	Description
string	char*	Converted integer value. This pointer will be equal to the buffer argument that was passed in.

Input Parameters:

Name	Type	Description
value	ASN1INT	Value to convert.
bufsiz	int	Size of buffer to receive stringified value.

Output Parameters:

Name	Type	Description

buffer	char*	Pointer to a buffer to receive stringified value.

rtUIntToString – Convert ASN.1 Unsigned Integer Value to String

The rtUIntToString function converts an ASN.1 integer value to a string. In this case, the ASN.1 value was represented in the C/C++ code as an unsigned integer based on a constraint.

Calling Sequence:

```
string = rtUIntToString (value, buffer, bufsiz)
```

Return Value:

Name	Type	Description
string	char*	Converted integer value. This pointer will be equal to the buffer argument that was passed in.

Input Parameters:

Name	Type	Description
value	ASN1UINT	Value to convert.
bufsiz	int	Size of buffer to receive stringified value.

Output Parameters:

Name	Type	Description
buffer	char*	Pointer to a buffer to receive stringified value.

rtBitStrToString – Convert ASN.1 Bit String Value to String

The rtBitStrToString function converts an ASN.1 bit string value to a string. The output format is ASN.1 value notation for a binary string (for example, '10010'B).

Calling Sequence:

```
string = rtBitStrToString (numbits, data, buffer, bufsiz)
```

Return Value:

Name	Type	Description
string	char*	Converted value. This pointer will be equal to the buffer argument that was passed in.

Input Parameters:

Name	Type	Description
numbits	ASN1UINT	Number of bits in the data argument to format.
data	ASN1OCTET*	Buffer containing the bit string to be formatted (note: in the case of BER/DER, this

		refers to the actual point in the string where the data starts, not where the contents field starts. The contents field contains an extra byte at the beginning that specifies the number of unused bits in the last byte).
bufsiz	int	Size of buffer to receive stringified value.

Output Parameters:

Name	Type	Description
buffer	char*	Pointer to a buffer to receive stringified value.

rtOctStrToString – Convert ASN.1 Octet String Value to String

The rtOctStrToString function converts an ASN.1 octet string value to a string. The output format is ASN.1 value notation for a hexadecimal string (for example, '1F8A'H).

Calling Sequence:

```
string = rtOctStrToString (numocts, data, buffer, bufsiz)
```

Return Value:

Name	Type	Description
string	char*	Converted value. This pointer will be equal to the buffer argument that was passed in.

Input Parameters:

Name	Type	Description
numocts	ASN1UINT	Number of octets (bytes) in the data argument to format.
data	ASN1OCTET*	Buffer containing the octet string to be formatted.
bufsiz	int	Size of buffer to receive stringified value.

Output Parameters:

Name	Type	Description
buffer	char*	Pointer to a buffer to receive stringified value.

rtOIDToString – Convert ASN.1 Object Identifier Value to String

The rtOIDToString function converts an ASN.1 object value to a string. The output format is ASN.1 value notation for an object identifier (ex. { 0 1 222 333 }). All subidentifiers are shown as integer numbers – no attempt is made to map the identifiers to symbolic names.

Calling Sequence:

```
string = rtOIDToString (numids, data, buffer, bufsiz)
```

Return Value:

Name	Type	Description
string	char*	Converted value. This pointer will be equal to the buffer argument that was passed in.

Input Parameters:

Name	Type	Description
numids	ASN1UINT	Number of subidentifiers in the OID value.
data	ASN1UINT*	Buffer containing the OID subidentifiers to be formatted.
bufsiz	int	Size of buffer to receive stringified value.

Output Parameters:

Name	Type	Description
buffer	char*	Pointer to a buffer to receive stringified value.

rtTagToString – Convert ASN.1 Tag to String

The rtTagToString function converts an ASN.1 tag to a string. The tag is represented using the compilers internal ASN1TAG structure. The output format is standard ASN.1 notation for a tag (for example, [0] = context 0 tag).

Calling Sequence:

```
string = rtTagToString (tag, buffer, bufsiz)
```

Return Value:

Name	Type	Description
string	char*	Converted value. This pointer will be equal to the buffer argument that was passed in.

Input Parameters:

Name	Type	Description
tag	ASN1TAG	Tag value to be converted.
data	ASN1OCTET*	Buffer containing the octet string to be formatted.
bufsiz	int	Size of buffer to receive stringified value.

Output Parameters:

Name	Type	Description
buffer	char*	Pointer to a buffer to receive stringified value.

rtPrint<type> – Print ASN.1 Values to Standard Output

The rtPrint<type> group of functions print ASN.1 values of various types to standard output (stdout). This section describes a series of functions whose name is formed by substituting a type identifier name for <type> in the above definition (for example, rtPrintBoolean prints a boolean value).

In general, these functions are very similar to the "ToString" functions described above. They simply print the output value to standard output in a "name = value" format. The value format is obtained by calling one of the "ToString" functions with the give value.

The following are the low-level print functions that are provided:

- rtPrintBoolean
- rtPrintInteger
- rtPrintUnsigned
- rtPrintBitStr
- rtPrintOctStr
- rtPrintCharStr
- rtPrint16BitCharStr
- rtPrint32BitCharStr
- rtPrintReal
- rtPrintOID
- rtPrintOpenType

These functions are assembled by the compiler print routine generator (-print command line option) to form the compiler-generated print functions.

The general calling sequence and parameters are as follows:

Calling Sequence:

```
rtPrint<type> (name, value);
```

Return Value:

None

Input Parameters:

Name	Type	Description
name	char*	Name of the variable to print
value	<various>	ASN.1 value to print (note: multiple arguments may be used to represent the value – for example a bit string would be represented by a numbits and data argument. See the function prototype for the exact calling sequence).

Object Identifier Helper Functions

Object identifier helper functions provide assistance in working with the object identifier ASN.1 type. Two functions are provided: one to populate an Object Identifier structure and one to print the contents.

rtSetOID – Populate Object Identifier Structure

The rtSetOID function populates an object identifier variable with data. It copies data from a source variable to a target variable. Typically, the source variable is a compiler-generated object identifier constant that resulted from an object identifier value specification within an ASN.1 specification.

Calling Sequence:

```
rtSetOID (ptarget, psource)
```

Return Value:

None

Input Parameters:

Name	Type	Description
psource	ASN1OBJID*	Pointer to source object identifier variable to copy to the target. Typically, this is a compiler-generated variable corresponding to an ASN.1 value specification in the ASN.1 source file.

Output Parameters:

Name	Type	Description
ptarget	ASN1OBJID*	Pointer to target object identifier variable to receive object identifier data. Typically, this is a variable within a compiler-generated C structure.

rtPrintOID – Print Object Identifier Structure

The rtPrintOID function formats and prints an object identifier value to stdout.

Calling Sequence:

```
rtPrintOID (pOID)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pOID	ASN1OBJID*	Pointer to object identifier variable to be printed.

Output Parameters:

None

Linked List and Stack Utility Functions

Linked list and stack utility functions are used to maintain linked lists and stacks used within the ASN.1 run-time library functions.

rtDListInit – Initialize a Doubly Linked List Structure

The rtDListInit function initializes a doubly linked list structure. It sets the number of elements to zero and sets all internal pointer values to NULL.

Calling Sequence:

```
rtDListInit (pList)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to linked list structure to be initialized.

Output Parameters:

None

rtDListAppend – Append an Item to a Doubly Linked List

The rtDListAppend function appends an item to linked list structure. The item is represented by a void pointer that can point to an object of any type. The *rtMemAlloc* function is used to allocate the memory for the list node structure, therefore, all internal list memory will be released whenever *rtMemFree* is called.

Calling Sequence:

```
pNode = rtDListAppend (pCtxt, pList, pData)
```

Return Value:

Name	Type	Description
pNode	Asn1RTDList Node*	Pointer to allocated node structure used to link the given data value into the list.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pList	Asn1RTDList*	Pointer to linked list structure onto which the data item is to be appended.
pData	void*	Pointer to data item to be appended to the list.

Output Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to updated linked list structure.

rtDListInsert – Insert an Item to a Doubly Linked List

The rtDListInsert function inserts an item to linked list structure. The item is represented by a void pointer that can point to an object of any type. The *rtMemAlloc* function is used to allocate the memory for the list node structure, therefore, all internal list memory will be released whenever *rtMemFree* is called.

Calling Sequence:

```
pNode = rtDListInsert (pCtxt, pList, index, pData)
```

Return Value:

Name	Type	Description
pNode	Asn1RTDList Node*	Pointer to allocated node structure used to link the given data value into the list.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pList	Asn1RTDList*	Pointer to linked list structure onto which the data item is to be inserted.
index	int	Index at which the specified item is to be inserted.
pData	void*	Pointer to data item to be inserted to the list.

Output Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to updated linked list structure.

rtDListInsertBefore – Insert an Item to a Doubly Linked List before specified node

The rtDListInsertBefore function inserts an item to linked list structure before specified node. The item is represented by a void pointer that can point to an object of any type. The *rtMemAlloc* function is used to allocate the memory for the list node structure, therefore, all internal list memory will be released whenever *rtMemFree* is called.

Calling Sequence:

```
pNode = rtDListInsertBefore (pCtxt, pList, pBefore, pData)
```

Return Value:

Name	Type	Description
pNode	Asn1RTDList Node*	Pointer to allocated node structure used to link the given data value into the list.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pList	Asn1RTDList*	Pointer to linked list structure onto which the data item is to be inserted.
pBefore	Asn1RTDList Node*	Pointer to node before which the specified item is to be inserted. It should be already in the linked list structure.
pData	void*	Pointer to data item to be inserted to the list.

Output Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to updated linked list structure.

rtDListInsertAfter – Insert an Item to a Doubly Linked List after specified node

The rtDListInsertBefore function inserts an item to linked list structure after specified node. The item is represented by a void pointer that can point to an object of any type. The *rtMemAlloc* function is used to allocate the memory for the list node structure, therefore, all internal list memory will be released whenever *rtMemFree* is called.

Calling Sequence:

```
pNode = rtDListInsertAfter (pCtxt, pList, pAfter, pData)
```

Return Value:

Name	Type	Description
pNode	Asn1RTDList Node*	Pointer to allocated node structure used to link the given data value into the list.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pList	Asn1RTDList*	Pointer to linked list structure onto which the data item is to be inserted.
pAfter	Asn1RTDList Node*	Pointer to node after which the specified item is to be inserted. It should be already in the linked list structure.
pData	void*	Pointer to data item to be inserted to the list.

Output Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to updated linked list structure.

rtDListFindByIndex –Find a node in the Doubly Linked List by index

The rtDListFindByIndex function gets a node from linked list structure, which has a specified index.

Calling Sequence:

```
pNode = rtDListFindByIndex (pList, index)
```

Return Value:

Name	Type	Description
pNode	Asn1RTDList Node*	Pointer to found node structure. NULL, if node is not found.

Input Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to linked list structure in which the node is to be found.
index	int	Index of the node to be returned.

Output Parameters:

None

rtDListFindByData –Find a node in the Doubly Linked List by index

The rtDListFindByData function gets a node from linked list structure, which contains a specified data.

Calling Sequence:

```
pNode = rtDListFindByData (pList, pData)
```

Return Value:

Name	Type	Description
pNode	Asn1RTDList Node*	Pointer to found node structure. NULL, if node is not found.

Input Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to linked list structure in which the node is to be found.
pData	void*	Pointer to data item to be found in the list.

Output Parameters:

None

rtDListFindIndexByData –Find an index of node in the Doubly Linked List by data

The rtDListFindIndexByData function gets a node's index from linked list structure, which contains a specified data.

Calling Sequence:

```
index = rtDListFindIndexByData (pList, pData)
```

Return Value:

Name	Type	Description
index	int	Index of found node that contains specified data.

Input Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to linked list structure in which the node is to be found.
pData	void*	Pointer to data item to be found in the list.

Output Parameters:

None

rtDListRemove – Remove a node from a Doubly Linked List

The rtDListRemove function removes a node from linked list structure. The *rtMemAlloc* function was used to allocate the memory for the list node structure, therefore, all internal list memory will be released whenever *rtMemFree* is called.

Calling Sequence:

```
rtDListRemove (pList, pNode)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to linked list structure from which the node is to be removed.
pNode	Asn1RTDList Node*	Pointer to node is to be removed. It should be already in the linked list structure.

Output Parameters:

Name	Type	Description
pList	Asn1RTDList*	Pointer to updated linked list structure.

rtSListInit – Initialize a Singly Linked List Structure

The rtSlistInit function initializes a singly linked list structure. It sets the number of elements to zero and sets all internal pointer values to NULL.

Calling Sequence:

```
rtSListInit (pList)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pList	Asn1RTSList*	Pointer to linked list structure to be initialized.

Output Parameters:

None

rtSListCreate – Create a Singly Linked List Structure

The rtSListCreate function creates a new linked list structure. It allocates memory for the structure and calls rtSListInit on it to initialize the structure.

Calling Sequence:

```
pList = rtSListCreate ()
```

Return Value:

Name	Type	Description
pList	Asn1RTSList*	Pointer to allocated linked list structure.

Input Parameters:

None

Output Parameters:

None

rtSListAppend – Append an Item to a Singly Linked List

The rtSListAppend function appends an item to linked list structure. The item is represented by a void pointer that can point to an object of any type.

Calling Sequence:

```
pNode = rtSListAppend (pList, pData)
```

Return Value:

Name	Type	Description

pNode	Asn1RTSList Node*	Pointer to allocated allocated node structure used to link the given data value into the list.

Input Parameters:

Name	Type	Description
pList	Asn1RTSList*	Pointer to linked list structure onto which the data item is to be appended.
pData	void*	Pointer to data item to be appended to the list.

Output Parameters:

Name	Type	Description
pList	Asn1RTSList*	Pointer to updated linked list structure.

rtStackInit – Initialize a Stack Structure

The rtStackInit function initializes a stack structure. It sets the number of elements to zero and sets all internal pointer values to NULL.

Calling Sequence:

```
rtStackInit (pStack)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pStack	Asn1RTStack*	Pointer to stack structure to be initialized.

Output Parameters:

None

rtStackCreate – Create a Stack Structure

The rtSListCreate function creates a new stack structure. It allocates memory for the structure and calls rtStackInit on it to initialize the structure.

Calling Sequence:

```
pStack = rtStackCreate ()
```

Return Value:

Name	Type	Description
pStack	Asn1RTSList*	Pointer to allocated stack structure.

Input Parameters:

None

Output Parameters:

None

rtStackPush – Push an Element onto the Stack

The rtStackPush function pushes an item onto the stack.

Calling Sequence:

```
stat = rtStackPush (pStack, pData)
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pStack	Asn1RTStack*	Pointer to stack structure onto which the data item is to be pushed.
pData	void*	Pointer to data item to be pushed onto the stack.

Output Parameters:

Name	Type	Description
pStack	Asn1RTStack*	Pointer to updated stack structure.

rtStackPop – Pop an Element from the Stack

The rtStackPop function pops an element from the stack.

Calling Sequence:

```
ptr = rtStackPop (pStack)
```

Return Value:

Name	Type	Description
ptr	void*	Pointer to item popped from the stack.

Input Parameters:

Name	Type	Description
pStack	Asn1RTStack*	Pointer to stack structure from which the value is to be popped.

Output Parameters:

Name	Type	Description
pStack	Asn1RTStack*	Pointer to updated stack structure.

Character String Conversion Functions

Common utility functions are provided to convert between standard null-terminated C strings and different ASN.1 string types.

rtCToBMPString

The rtCToBMPString function converts a null-terminated C string into a 16-bit BMP string structure.

Calling Sequence:

```
pString = rtCToBMPString (ctxt_p, cstring, pBMPString, pCharSet);
```

Return value:

Name	Type	Description
pString	ASN1BMP String*	Pointer to BMP string structure. This is the pBMPString argument parameter value.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
cstring	char*	Pointer to null-terminated C string to be converted into a BMP string.
pCharSet	Asn116Bit CharSet*	Pointer to character set structure describing the character set currently associated with the BMP character string type.

Output parameters:

Name	Type	Description
pBMPString	ASN1BMP String*	Pointer to BMP string structure to receive converted string.

rtBMPToCString

The rtBMPToCString function converts a BMP string into a null-terminated C string. Any characters that are not 8-bit characters are discarded.

Calling sequence:

```
pString = rtBMPToCString (pBMPString, cstring, cstrsize);
```

Return value:

Name	Type	Description
pString	char*	Pointer to returned string structure. This is the cstring argument parameter value.

Input parameters:

Name	Type	Description
pBMPString	ASN1BMP String*	Pointer to BMP string structure to be converted.
cstrsize	int	Size of the buffer to receive the converted string.

Output parameters:

Name	Type	Description
cstring	char*	Pointer to buffer to receive converted string.

rtBMPToNewCString

The rtBMPToNewCString function converts a BMP string into a null-terminated C string. Any characters that are not 8-bit characters are discarded. This function allocates dynamic memory to hold the converted string. The user is responsible for freeing this memory.

Calling sequence:

```
pString = rtBMPToCString (pBMPString)
```

Return value:

Name	Type	Description
pString	char*	Pointer to allocated null-terminated string. The user is responsible for freeing this memory.

Input parameters:

Name	Type	Description
pBMPString	ASN1BMP String*	Pointer to BMP string structure to be converted.

Output parameters:

None

rtCToUCSString

The rtCToUCSString function converts a null-terminated C string into a 32-bit UCS-4 (Universal Character Set, 4 bytes) string structure.

Calling Sequence:

```
pString = rtCToUCSString (ctxt_p, cstring, pUCSString, pCharSet);
```

Return value:

Name	Type	Description
pString	ASN1Universal String*	Pointer to Universal string structure. This is the pUCSString argument parameter value.

Input parameters:

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
cstring	char*	Pointer to null-terminated C string to be converted into a Universal string.
pCharSet	Asn132Bit CharSet*	Pointer to character set structure describing the character set currently associated with the Universal character string type.

Output parameters:

Name	Type	Description
pUCSString	ASN1Universal String*	Pointer to Universal string structure to receive converted string.

rtUCSToCString

The rtUCSToCString function converts a Universal 32-bit string into a null-terminated C string. Any characters that are not 8-bit characters are discarded.

Calling sequence:

```
pString = rtUCSToCString (pUCSString, cstring, cstrsize);
```

Return value:

Name	Type	Description
pString	char*	Pointer to returned string structure. This is the cstring argument parameter value.

Input parameters:

Name	Type	Description
pUCSString	ASN1Universal String*	Pointer to Universal string structure to be converted.
cstrsize	int	Size of the buffer to receive the converted string.

Output parameters:

Name	Type	Description
cstring	char*	Pointer to buffer to receive converted string.

rtUCSToNewCString

The rtUCSToNewCString function converts a Universal 32-bit string into a null-terminated C string. Any characters that are not 8-bit characters are discarded. This function allocates dynamic memory to hold the converted string. The user is responsible for freeing this memory.

Calling sequence:

```
pString = rtUCSToCString (pUCSString)
```

Return value:

Name	Type	Description
pString	char*	Pointer to allocated null-terminated string. The user is responsible for freeing this memory.

Input parameters**:**

Name	Type	Description
pUCSString	ASN1Universal String*	Pointer to Universal 32-bit string structure to be converted.

Output parameters:

None

rtUCSToWCSString

The rtUCSToWCSString function converts a 32-bits encoded string to a wide-character string.

Calling Sequence:

```
len = rtUCSToWCSString (inbuf, outbuf, outbufsiz);
```

Return value:

Name	Type	Description
len	int	Character count or error status. Will be negative if conversion fails. If positive, indicates number of character s written to outbuf.

Input parameters**:**

Name	Type	Description
inbuf	ASN1Universal String*	Pointer to Universal string structure.
outbufsiz	int	Number of wide characters (wchar_t) the output buffer can hold.

Output parameters:

Name	Type	Description
outbuf	wchar_t*	Pointer to buffer to receive converted string.

rtWCSToUCSString

The rtWCSToUCSString function converts a wide-character string to a Universal 32-bits encoded string.

Calling Sequence:

```
len = rtWCSToUCSString (ctxt_p, inbuf, outbuf, pCharSet);
```

Return value:

Name	Type	Description
len	int	If conversion of WCS to UTF-8 is successful, the number of bytes in the converted string is returned. If the encoding fails, a negative status value is returned.

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
inbuf	wchar_t*	Pointer to wide-character (Unicode) string to convert.
pCharSet	Asn132Bit CharSet*	Pointer to character set structure describing the character set currently associated with the Universal character string type.

Output parameters:

Name	Type	Description
outbuf	ASN1Universal String*	Pointer to Universal String structure to receive converted string.

rtWCSToUTF8

The rtWCSToUTF8 function converts a wide-character string to a UTF-8 encoded string.

Calling Sequence:

```
len = rtWCSToUTF8 (ctxt_p, inbuf, inlen, outbuf, outbufsiz);
```

Return value:

Name	Type	Description
len	int	If conversion of WCS to UTF-8 is successful, the number of bytes in the converted string is returned. If the encoding fails, a negative status value is returned.

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
inbuf	wchar_t*	Pointer to wide-character (Unicode) string to convert.

inlen	int	Number of characters in the Unicode string.
outbufsiz	int	Size (in bytes) or the output buffer to receive the encoded string

Output parameters:

Name	Type	Description
outbuf	ASN1OCTET*	Pointer to buffer to receive converted string.

rtUTF8ToWCS

The rtUTF8ToWCS function converts a UTF-8 encoded string to a wide-character string.

Calling Sequence:

```
len = rtUTF8ToWCS (ctxt_p, inbuf, outbuf, outbufsiz);
```

Return value:

Name	Type	Description
len	int	Character count or error status. Will be negative if conversion fails. If positive, indicates number of character s written to outbuf.

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
inbuf	char*	Pointer to null-terminated UTF-8 encoded string.
outbufsiz	int	Number of wide characters (wchar_t) the output buffer can hold.

Output parameters:

Name	Type	Description
outbuf	wchar_t*	Pointer to buffer to receive converted string.

rtValidateUTF8

The rtValidateUTF8 function will validate a UTF-8 encoded string to ensure that it us encoded correctly.

Calling Sequence:

```
stat = rtUTF8ToWCS (ctxt_p, inbuf);
```

Return value:

Name	Type	Description
stat	int	Status of validation. Will be ASN_OK (zero) if validation successful or a negative

		status value if an error is detected.

Input parameters**:**

Name	Type	Description
ctxt_p	ASN1CTXT*	Pointer to a context structure.
inbuf	char*	Pointer to null-terminated UTF-8 encoded string.

Output parameters:

None

Big integer helper functions

Arbitrary-precision integers' manipulating functions are used to maintain big integers used within the ASN.1 run-time library functions.

rtBigIntInit – Initialize a big integer Structure

The rtBigIntInit function initializes a big integer structure. This function should be called before the first use of the big integer.

Calling Sequence:

```
rtBigIntInit (pBigInt)
```

Return Value:

None

Input Parameters:

Name	Type	Description
pBigInt	ASN1BigInt*	Pointer to big integer structure to be initialized.

Output Parameters:

None

rtSetStrToBigInt – Convert string to a big integer

The rtSetStrToBigInt function converts the character string to a big integer structure.

Calling Sequence:

```
stat = rtSetStrToBigInt (pCtxt, pInt, pString, radix)
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pInt	ASN1BigInt*	Pointer to big integer structure onto which the value is to be stored.
pString	ASN1ConstCharPtr	Pointer to character string to be converted.
radix	int	Base of value in pString. Must be 2, 8, 10 or 16.

Output Parameters:

Name	Type	Description
pInt	ASN1BigInt *	Pointer to updated big integer structure.

rtSetInt64ToBigInt – Convert ASN1INT64 value to big integer

The rtSetInt64ToBigInt function converts the ASN1INT64 value to big integer structure. An ASN1INT64 type is a 64-bit integer type, if platform supports 64-bit integers. In other case it will be a simple 32-bit integer.

Calling Sequence:

```
stat = rtSetStrToBigInt (pCtxt, pInt, i64value)
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pInt	ASN1BigInt*	Pointer to big integer structure onto which the value is to be stored.
i64value	ASN1INT64	The 64-bit integer value to be converted.

Output Parameters:

Name	Type	Description
pInt	ASN1BigInt *	Pointer to updated big integer structure.

rtSetBytesToBigInt – Convert sequence of octets to big integer

The rtSetBytesToBigInt function translates an octet array containing the two's-complement binary representation of an arbitrary-precision integer into a big integer structure. The input array is assumed to be in big-endian octet-order: the most significant octet is in the zeroth element.

Calling Sequence:

```
stat = rtSetBytesToBigInt (pCtxt, pInt, pOctets, len)
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pInt	ASN1BigInt*	Pointer to big integer structure onto which the value is to be stored.
pOctets	ASN1OCTET*	Pointer to an octet array with two's-complement binary representation of an arbitrary-precision integer.
len	int	Length of an octet array.

Output Parameters:

Name	Type	Description
pInt	ASN1BigInt *	Pointer to updated big integer structure.

rtGetBigIntLen– Get big integer length

The rtGetBigIntLen function returns the number of octets necessary for the storing a big integer value in the octet array. This function may be used for the calculating the necessary buffer size for the rtGetBigInt function.

Calling Sequence:

```
len = rtGetBigIntLen (pInt)
```

Return Value:

Name	Type	Description
len	int	Number of octets, necessary for the storing a big integer value in the octet string.

Input Parameters:

Name	Type	Description
pInt	ASN1BigInt*	Pointer to big integer structure onto which the value is to be stored.

Output Parameters:

None

rtGetBigInt – Copy big integer value into an octet array

The rtGetBigInt function copies the two's-complement binary representation of a big integer into an octet string. The output array will be in big-endian octet-order: the most significant octet will be in the zeroth element.

Calling Sequence:

```
stat = rtGetBigInt (pCtxt, pInt, pOctets, bufsize)
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pInt	ASN1BigInt*	Pointer to big integer structure.
bufsize	int	Length of an octet array.

Output Parameters:

Name	Type	Description
pOctets	ASN1OCTET*	Pointer to octet array to receive two's-complement binary representation of a arbitrary-precision integer.

rtBigIntDigitsNum – Return the approximated number of digits of the big integer

The rtBigIntDigitsNum function returns the approximated number of digits of the big integer value according to the specified radix. This function may be used to calculate size of buffer for the rtBigIntToString function. The number of digits might be slightly greater than really necessary, but never less.

Calling Sequence:

```
dnum = rtBigIntDigitsNum (pInt, radix)
```

Return Value:

Name	Type	Description
dnum	int	The approximated number of digits of the big integer.

Input Parameters:

Name	Type	Description
pInt	ASN1BigInt*	Pointer to a big integer structure.
radix	int	Base of value. Must be 2, 8, 10 or 16.

Output Parameters:

None

rtBigIntToString – Convert a big integer to a string

The rtBigIntToString function converts a big integer to a string according to the specified radix.

Calling Sequence:

```
stat = rtBigIntToString (pCtxt, pInt, radix, pBuf, bufsize)
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pInt	ASN1BigInt*	Pointer to big integer structure to be converted into a string.
radix	int	Base of value for conversion. Must be 2, 8, 10 or 16.
bufsize	int	The size of buffer pBuf.

Output Parameters:

Name	Type	Description
pBuf	char*	Pointer to buffer to receive the converted character string.

rtPrintBigInt – Print big integer value to Standard Output

The rtPrintBigInt function prints big integer value to standard output (stdout), according to specified radix.

In general, this function is very similar to the "rtBigIntToString" function described above. It simply prints the output value to standard output in a "name = value" format.

Calling Sequence:

```
rtPrintBigInt (name, value, radix);
```

Return Value:

None

Input Parameters:

Name	Type	Description
name	char*	Name of the variable to print
value	ASN1BigInt*	Big integer value to print.
radix	int	Base of value for conversion to print. Must be 2, 8, 10 or 16.

rtCompareBigInt – Compare two big integer values

The rtCompareBigInt function compares two big integer values. The result of comparison can be –1 (first value less than second one), 0 (both are equal) and 1 (first value greater than second one).

Calling Sequence:

```
res = rtPrintBigInt (pInt1, pInt2);
```

Return Value:

Name	Type	Description
res	int	The result of comparison (-1, 0 or 1).

Input Parameters:

Name	Type	Description
pInt1	ASN1BigInt*	The first big integer value to compare.
pInt1	ASN1BigInt*	The second big integer value to compare.

rtBigIntCopy – Copy one big integer structure into another

The rtBigIntCopy function copies one big integer structure into another one. The destination big integer structure **should not be** initialized yet.

Calling Sequence:

```
stat = rtBigIntCopy (pCtxt, pSrc, pDst);
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pSrc	ASN1BigInt*	The source big integer value to copy.
pDst	ASN1BigInt*	The destination big integer value to receive copied value. **Should not be initialized.**

rtBigIntFastCopy – Fast copy of one big integer structure into another

The rtBigIntFastCopy function copies one big integer structure into another one. The destination big integer structure **should be** initialized before use of this function. This function might be faster than rtBigIntCopy because if the destination big integer already has enough allocated memory then memory will be reused without allocation.

Calling Sequence:

```
stat = rtBigIntFastCopy (pCtxt, pSrc, pDst);
```

Return Value:

Name	Type	Description
stat	int	Status value of the operation. This is one of the status values described in Appendix A.

Input Parameters:

Name	Type	Description
pCtxt	ASN1CTXT	Pointer to a context structure. This provides a storage area for the function to store all working variables that must be maintained between function calls.
pSrc	ASN1BigInt*	The source big integer value to copy.
pDst	ASN1BigInt*	The destination big integer value to receive copied value. **Should be initialized.**

APPENDIX A

The following error status codes can be returned by the run-time library functions or by generated ASN1C code:

Error Constant	Value	Description
ASN_OK	0	Decode successful (successful encode returns a positive value equal to the number of bytes encoded).
ASN_OK_FRAG	2	OK fragment. This is a success status returned by some PER functions to indicate encode or decode was successful, but that a message fragment was encountered. The results are not yet complete.
ASN_E_BUFOVFLW	-1	Encode buffer overflow. Occurs when the size of a static encode buffer is exceeded.
ASN_E_ENDOFBUF	-2	Unexpected end-of-buffer on decode. Occurs when a decode function encounters the end of the buffer when expecting to find more data.
ASN_E_IDNOTFOU	-3	Identifier not found on decode. Occurs when an unexpected tag is encountered during message decoding.
ASN_E_INVOBJID	-4	Invalid object identifier code. Occurs when the ASN.1 rules for specifying an object identifier value are violated.
ASN_E_INVLEN	-5	Invalid length component on decode. The actual length of data in a given field did not match the encoded length.
ASN_E_INVENUM	-6	Enumerated value parsed from a field was not in the defined set for that field.
ASN_E_SETDUPL	-7	A duplicate occurrence of a tagged element within a SET was encountered during decoding.
ASN_E_SETMISRQ	-8	Decoding of a SET construct was completed and one or more required (i.e., not OPTIONAL) SET elements were missing.
ASN_E_NOTINSET	-9	A tagged element was encountered during the decoding of a SET that was not a member of the defined SET.
ASN_E_SEQOVFLW	-10	More elements were encountered in a sized SEQUENCE OF or SET OF construct then were specified in the SIZE specification.
ASN_E_INVOPT	-11	An element was encountered during decoding of a CHOICE construct that was not defined to be an option for the CHOICE.
ASN_E_NOMEM	-12	No dynamic memory available.
ASN_E_INVHEXS	-14	Invalid ASN.1 hexadecimal string value. This error occurs if a string is passed to a run-time function that is expecting an ASN.1 hex string value in ASN.1 value notation format and the string is not properly formatted (i.e., in the form ‘xxxx’H).
ASN_E_INVBINS	-15	Invalid ASN.1 binary string value. This error occurs if a string is passed to a run-time function that is expecting an ASN.1 binary string value in ASN.1 value notation format and the string is not properly formatted (i.e., in the form ‘xxxx’B).

ASN_E_INVREAL	-16	Invalid real value. This is returned by the ASN.1 REAL type decode functions if an encoded real value violates any of the ASN.1 encoding rules for a REAL value.
ASN_E_STROVFLW	-17	More OCTETs or BITS were encountered in a sized OCTET or BIT STRING field then were specified in the SIZE specification.
ASN_E_BADVALUE	-18	Invalid value specification.
ASN_E_UNDEFVAL	-19	Definition not found for referenced defined value.
ASN_E_UNDEFTYPE	-20	Definition not found for referenced defined type.
ASN_E_BADTAG	-21	An ID number in a tag value was encountered in decoding which was greater then the maximum value supported by the ASN1C compiler (8191).
ASN_E_TOODEEP	-22	Message contains nested constructs greater then the maximum defined depth.
ASN_E_CONSVIO	-23	Value constraint violation (for example, integer value not within the given range).
ASN_E_RANGERR	-24	Invalid range specification (lower value greater than upper value or endpoints are of different types).
ASN_E_ENDOFFILE	-25	This is returned from the decode from file functions (xdf) if an unexpected end-of-file condition is encountered. An example would be an indefinite length encoding in which the EOC marker was not found before end-of-file occurred.
ASN_E_INVUTF8	-26	Invalid UTF-8 encoded string.
ASN_E_CONCMODF	-27	Concurrent list modification. This is returned by ASN1CSeqOfListIterator's methods if the list was modified during the usage of the iterator.
ASN_E_ ILLSTATE	-28	Illegal state error. This is returned by ASN1CSeqOfListIterator's methods if the iterator is in illegal state for some operations (for example, call of remove() or set() method before the first call of next() or prev()).
ASN_E_ OUTOFBND	-29	Out of bounds. This is returned if some indices are out of bounds (of array, for example).
ASN_E_ INVPARAM	-30	Invalid parameter.
ASN_E_INVFORMAT	-31	Invalid time string format.
ASN_E_NOTINIT	-32	Not initialized. This is returned if some data is not initialized before use (for example, if ASN1CTXT is not initialized by rtInitContext function before use of this context).
ASN_E_NOTSUPP	-99	Non-supported ASN.1 construct encountered.

APPENDIX B

This version of the ASN1C compiler can parse all syntax as set forth in the 1997 ITU-T recommendations X.680 through X.683. However, all syntax does not result in the generation of corresponding C or C++ code. In general, the approach is to extract what is needed to form accurate C/C++ types and encode/decode functions for encoding the base types in a specification. It is up to the user to accurately put them together in places where layered messages are necessary to form a complete PDU.

The following ASN.1 constructs result in the generation of limited or no C/C++ code in this version of the ASN1C compiler:

- Constructed ASN.1 value specifications (including DEFAULT on a SEQUENCE).
- General constraints and table constraints.
- Value set specifications.
- EMBEDDED PDV type
- Selection Type
- Macros from the X.208 specification (note: a special version of the compiler is included that can parse ROSE OPERATION and ERROR macros).

Index