ASN1C

Objective Systems, Inc. version 7.1 — January 2017

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

Copyright Notice

Author's Contact Information

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

Table of Contents

Chapter 1. Overview of ASN1C for Java

The ASN1C code generation tool translates an Abstract Syntax Notation 1 (ASN.1) or XML Schema Definitions (XSD) source file into computer language source files that allow typed data to be encoded/decoded. This release of ASN1C includes options to generate code in the following languages: C, C++, C# or Java. This manual discusses the Java code generation capabilities. The following manuals discuss the other language code generation capabilities:

• *ASN1C C/C++ Compiler User's Manual* : C/C++ code generation

• *ASN1C C# Compiler User's Manual* : C# code generation

Each module or namespace that is encountered in an ASN.1 or XSD source file results in the generation of a series of Java source files. A separate Java file is generated for each production (type or global element) in the source file. Additional files are generated for compiler-generated productions and to hold value specification constants.

There is also a set of classes that form the run-time component of the Java package. These classes provide the primitive component building blocks that are assembled by the compiler to encode/decode complex structures. They also provide support for managing message buffers that hold the encoded message components.

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

• Basic Encoding Rules (BER), Distinguished Encoding Rules (DER), and Canonical Encoding Rules (CER) as published in the ITU-T X.690 standard.

• Packed Encoding Rules (PER) as published in the ITU-T X.691 standard.

• XML Encoding Rules (XER) as published in the ITU-T X.693 standard.

• XML Schema to ASN.1 translation as published in the ITU-T X.694 standard.

• Octet Encoding Rules (OER) as published in the ITU-T X.696 standard.

The compiler is capable of parsing all ASN.1 syntax as defined in the standards. It is capable of parsing advanced syntax including Information Object Specifications as defined in the ITU-T X.681 standard as well as Parameterized Types as defined in ITU-T X.683.

Note that XER support does not include support for the EXTENDED-XER syntax. This is accomplished through direct compilation of XSD files. An internal translation of XSD to ASN.1 based on the rules in the X.694 standard is done within the compiler and the resulting ASN.1 syntax is compiled into Java classes.

This release of the compiler contains a special compiler option (*-asnstd x208*) that is 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 and SNMP.

Chapter 2. Using the Compiler

Running ASN1C

The ASN1C compiler distribution contains command-line compiler executables as well as a graphical user interface (GUI) wizard that can aid in the specification of compiler options. Please refer to the *ASN1C C/C++ Compiler User's Manual* for instructions on how to run the compiler. The remaining sections describe options and configuration items specific to the Java version.

ASN1C Java Command Line Options

The following table shows a summary of the command line options that have meaning when Java code generation is selected:

Option	Argument	Description
-allow-ambig-tags		This option suppresses the check that is done for ambiguous tags within a SEQUENCE or SET type within a specification. Special code is generated for the decoder that assigns values to ambiguous elements within a SET in much the same way as would be done if the elements were declared to be in a SEQUENCE.
-asnstd	x680 x208 mixed	This option instructs the compiler to parse ASN.1 syntax conforming to the specified standard. x680 (the default) refers to modern ASN.1 as specified in the ITU-T X.680-X.690 series of standards. x208 refers to the now deprecated X.208 and X.209 standards. This syntax allowed the ANY construct as well as unnamed fields in SEQUENCE, SET, and CHOICE constructs. This option also allows for parsing and generation of code for ROSE OPERATION and ERROR macros and SNMP OBJECT-TYPE macros. The mixed option is used to specify a source file that contains modules with both X.208 and X.680 based syntax.
-ber	None	This option instructs the compiler to generate functions that implement the Basic Encoding Rules (BER) as specified in the ASN.1 standards.
-cer	None	This option instructs the compiler to generate functions that implement the Canonical Encoding Rules (CER) as specified in the ASN.1 standards.

Option	Argument	Description
-cldc	None	This option instructs the compiler to generate Java Micro Edition Connected Limited Device Configuration (CLDC) 1.1 compatible code. See the chapter *Java Micro Edition Support* for more information.
-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.
-compare	None	This option is used to generate a comparison method (*Equals*) in the generated classes.
-compat	<versionNumber>	Generate code compatible with an older version of the compiler. The compiler will attempt to generate code more closely aligned with the given previous release of the compiler. <versionNumber> is specified as x.x (for example, -compat 5.2)
-config	<filename>	This option is used to specify the name of a file containing configuration information for the source file being parsed. A full discussion of the contents of a configuration file is provided in the *Compiler Configuration File* section.
-copy	None	This option instructs the compiler to generate clone methods in each of the generated types. These methods perform a deep copy.
-depends	None	This option instructs the compiler to generate a full set of Java source files that contain only the productions in the main file being compiled and items those productions depend on from IMPORT files.
-der	None	This option instructs the compiler to generate functions that implement the Distinguished Encoding Rules (DER) as specified in the ASN.1 standards.
-dirs	None	This is a Java option that causes a subdirectory to be created to hold each of the generated Java source files for each module in an ASN.1 source file.

Option	Argument	Description
-events	None	Generate extra code to invoke user defined event and error handler callback methods (see the *Event Handlers* section).
-genant	None	Generate a build script (build.xml) that is compatible with the Ant toolchain.
-genbuild	None	This option is used to generate a build script for compiling generated classes.
-genmetadata	None	This option is used to generate methods to permit accessing metadata about SEQUENCE and SET types. Users may query value ranges (represented as Asn1ValueRange objects) and optional/required status for elements in the given SEQUENCE or SET.
-genPrint -print	None	This option specifies that print methods should be generated. Print functions are debug functions that allow the contents of generated type variables to be written to stdout.
-genPrtToStr -prtToStr	None	This option produces additional print methods whose primary parameter is a StringBuilder. This provides somewhat more efficient output to strings if desired. The generated methods are not synchronized.
-getset	None	This option is used to generate protected member variables and get/set methods for accessing the variables. By default, member variables are declared to be public and they are accessed directly by application code.
-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
-lax	None	This option instructs the compiler to not generate code to check constraints. When used in conjunction with the *–compact* option, it produces the smallest code base for a given ASN.1 specification
-list	None	Generate listing. This will dump the source code to the standard output

Option	Argument	Description
		device as it is parsed. This can be useful for finding parse errors.
-nodecode	None	This option suppresses the generation of decode functions.
-noencode	None	This option suppresses the generation of encode functions.
-noevents	None	This option supresses generation of event handling code that is generated by default to enable element tracking, a feature used to enhance error reporting.
-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.
-noOpenExt	None	This option instructs the compiler to not add an open extension element in constructs that contain extensibility markers. The purpose of the element is to collect any unknown items in a message. If an application does not care about these unknown items, it can use this option to reduce the size of the generated code.
-o	<directory>	This option is used to specify the name of a directory to which all of the generated files will be written.
-oer	None	Generate functions that implement the Octet Encoding Rules (OER) as specified in the ASN.1 standards.
-pdu	<typeName>	Designate given type name to be a Protocol Definition Unit (PDU) type. By default, PDU types are determined to be types that are not referenced by any other types within a module. This option allows that behavior to be overridden. The * wildcard character may be specified for <typeName> to indicate that all productions within an ASN.1 module should be treated as PDU types.
-per	None	This option instructs the compiler to generate functions that implement

Option	Argument	Description
		the Packed Encoding Rules (PER) as specified in the ASN.1 standards.
-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.
-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.
-reader	None	This option is used to generate a sample reader program to decode data.
-sax	None	This option instructs the compiler to generate code that uses the old SAX XML parser API. The default is to use the XmlPull API.
-shortnames	None	This option is used to change the names generated by compiler for embedded types in constructed types. This option is required to handle the limit on the size of filenames in certain situations. With this option, the generated code filenames would be shorter than without this option.
-stax	None	This option instructs the compiler to generate code that uses the StAX XML parser API. The default is to use the XmlPull API.
-stream	None	This option instructs the compiler to generate stream-based encoders/ decoders instead of memory buffer based. This makes it possible to encode directly to or decode directly from a source or sink such as a file or socket. In the case of BER, it will also cause forward encoders to be generated, which will use indefinite lengths for all constructed elements in a message.
-tables	None	This option is used to generate additional code for the handling of

Option	Argument	Description
		table constraints as defined in the X.682 standard.
-uniquenames	None	This option instructs the compiler to automatically generate unique names to resolve name collisions in the generated code. Name collisions can occur, for example, if two modules are being compiled that contain a production with the same name. A unique name is generated by prepending the module name to one of the productions to form a name of the form <module>_<name>. Note that name collisions can also be manually resolved by using the typePrefix, enumPrefix, and valuePrefix configuration items (see the Compiler Configuration File section for more details)
-warnings	None	Output information on compiler generated warnings.
-writer	None	This option is used to generate a sample writer program to encode data.
-xer	None	This option instructs the compiler to generate functions that implement the XML Encoding Rules (XER) as specified in the ASN.1 standards. Related XML Schema can be produced by using the -xsd command line option.
-xml	None	This option instructs the compiler to generate functions that implement the XML Encoding Rules (XML) as specified in the World-Wide Consortium (W3C). Related XML Schema can be produced by using the -xsd command line option.
-xmlpull	None	This option instructs the compiler to generate code that uses the XmlPull XML parser API. This is the default behavior.

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 or by specifying the name as an attribute (for example, <module name="MyModule">). Other attributes specified within the <module> section apply only to that module and not to other modules specified within the specification. A complete list of all module attributes is provided in the table at the end of this section.

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

When an attribute is specified in more than one section, the most specific application is always used. For example, assume a <typePrefix> qualifier is used within a module specification to specify a prefix for all generated types in the module and another one is used to 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

There are no attributes that are specific to Java that can be specified at the global level.

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

Name	Values	Description
		a type or value referenced by another item, the directive will be ignored.
<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.
<pkgName>	Java package name	Name of the Java package associated with this module. This will cause a Java import statement to be generated for the module if this name is not the same as that of the package being compiled.

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.
<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 Java long type (64 bits). A Java BigInteger class will be used to hold 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.
<listclass> </listclass>	Java List Class	This instructs the compiler to use a different class than the default `LinkedList`; it would be possible, for example, to use `java.util.ArrayList`.

Element Level

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

Name	Values	Description
<isBigInteger/>	n/a	This is a flag variable (an 'empty element' in XML terminology) that

Name	Values	Description
		specifies that this production will be used to store an integer larger than the Java long type (64 bits). A Java BigInteger class will be used to hold 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.
<isOpenType/>	n/a	This flag variable specifies that this element will be decoded as an open type (i.e. skipped). Refer to the section on deferred decoding and partial decoding for further information. Note that this variable can only be used with BER, CER, or DER encoding rules.
<skip/>	n/a	This flag variable specifies that this element will be skipped during decoding. Refer to the section on deferred decoding and partial decoding for further information. Note that this variable can only be used with BER, CER, or DER encoding rules.

Compiler Error Reporting

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

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

The most common types of syntax errors are as follows:

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

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

- Invalid special characters: only letters, numbers, and the hyphen (-) character are allowed. Programmers tend to like to use the underscore character (_) in identifiers. This is not allowed in ASN.1. Conversely, C or C# does not allow hyphens in identifiers. To get around this problem, ASN1C converts all hyphens in an ASN.1 specification to underscore characters in the generated code.

Semantics errors occur on the compiler back-end as the code is being generated. In this case, parsing was successful, but the compiler does not know how to generate the code. These errors are flagged by embedding error messages

directly in the generated code. The error messages always begin with an identifier with the prefix **'%ASN**-',. 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.

Chapter 3. ASN1C GUI Users Guide

Quick Start

In this section, we will demonstrate running ACGUI, creating a new ASN.1 schema, and compiling it to C for BER data. The process is similar for other languages.

First, start ACGUI. Once the program is running, we'll create a new project to store all of our settings. To do this, select "Project->New Project..." from the menus.
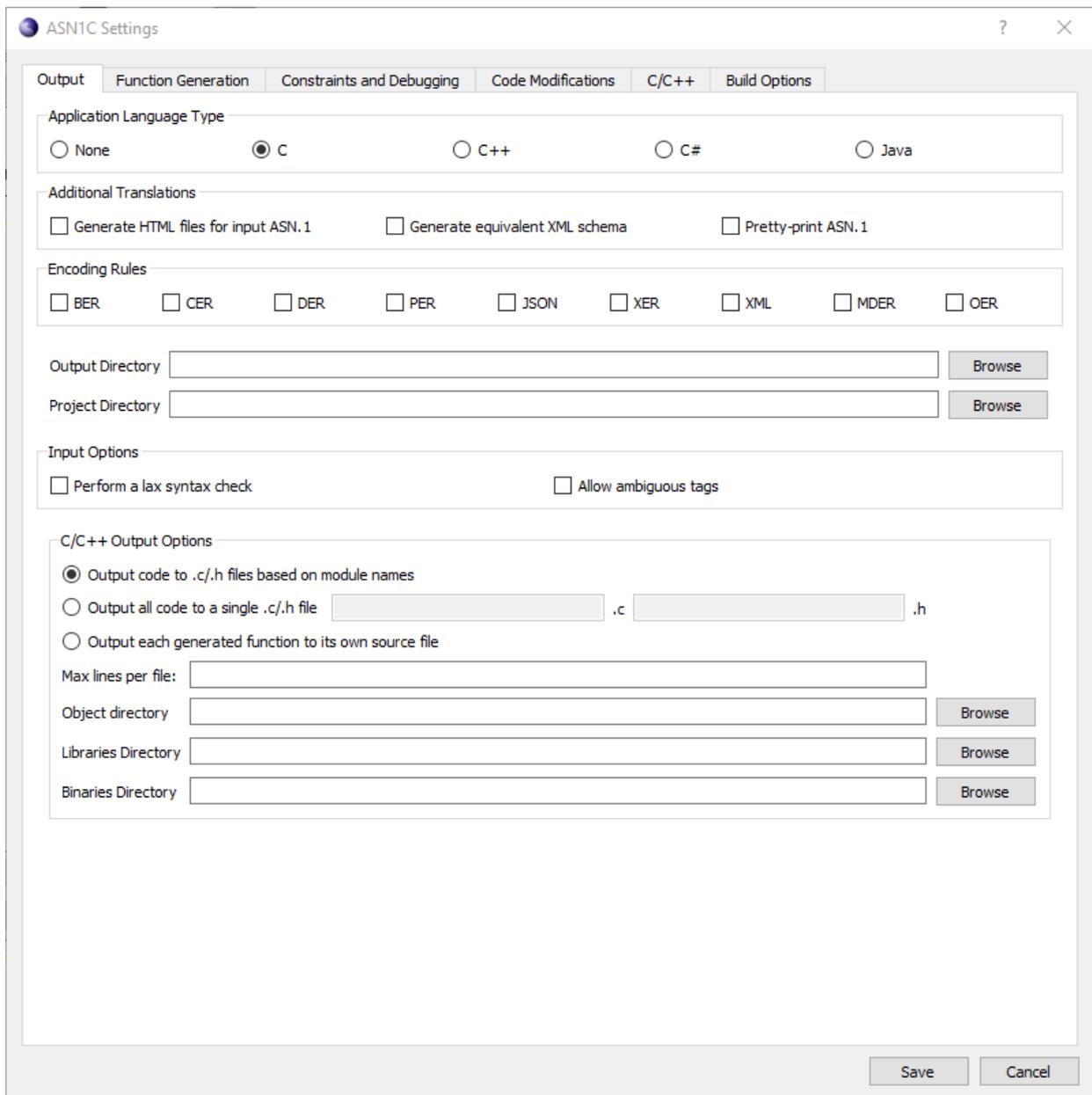
In the first tab, "Output", under "Application Language Type", select "C". Below that, under "Encoding Rules", select "BER". Finally, choose an output directory for the generated files. Once the settings are correct, click "Save". These settings can be changed at any time by selecting "Project->Project Settings..." from the menus.

Next, we'll create a new schema file for our project. Click the "New" button in the toolbar or select "File->New Schema File" from the menus. A dialog box will be shown to provide a name for the new file. Once entered, the file will be added to the project window under "Schema/ASN.1 files" and its empty contents will be shown in the editor.
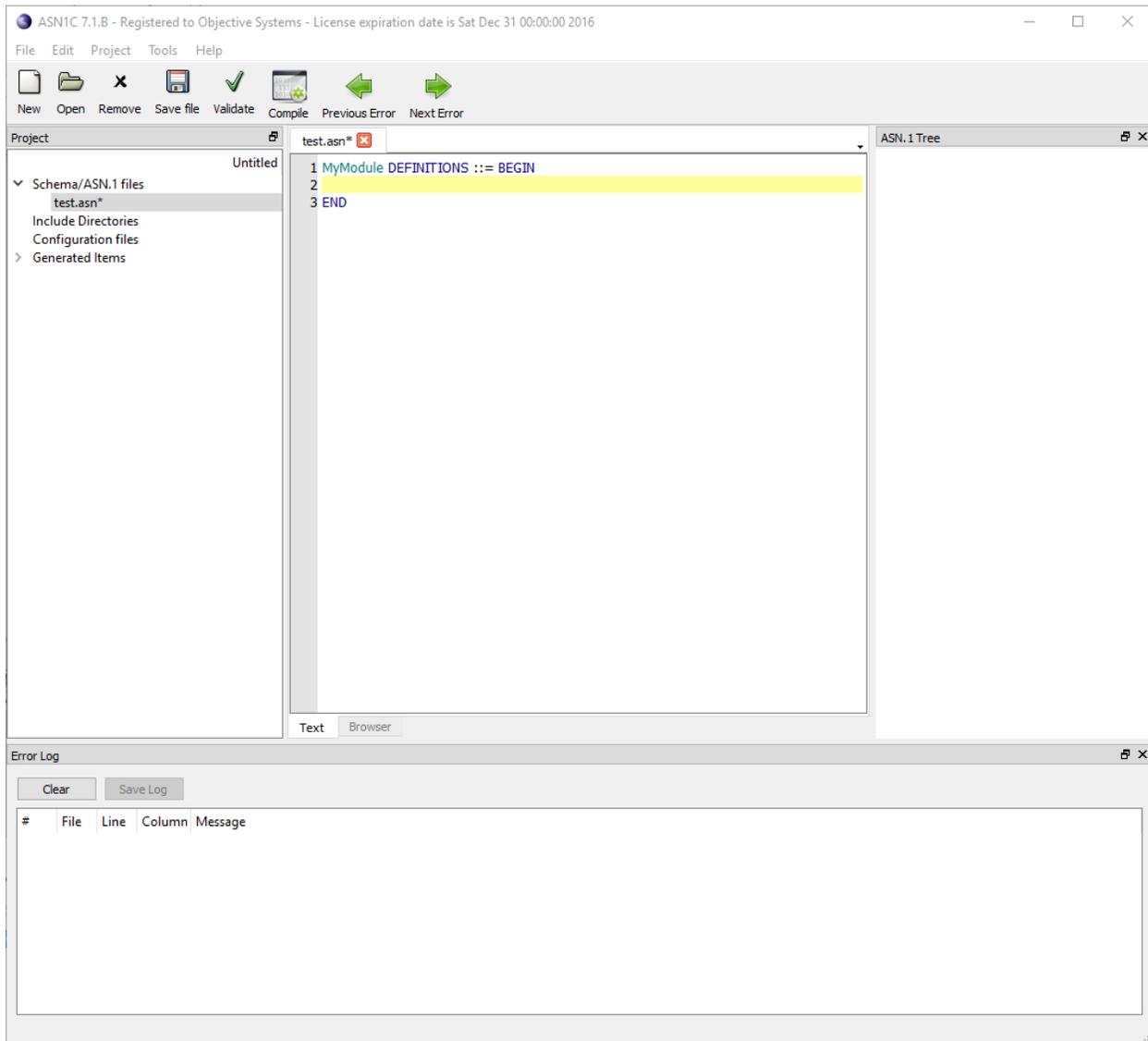
Now we need to write our schema between the "DEFINITIONS ::= BEGIN" and "END" statements in the file. Enter something like:

```
MySequence ::= SEQUENCE {
    ingredient PrintableString,
    count INTEGER,
    units PrintableString
}
```

When you are satisfied with the schema you've created, click the "Validate" button to make sure there are no errors. Since the new schema file hasn't been saved yet, ACGUI will ask if it should be. Save the new file and ACGUI will then validate it. If the schema has errors, the log at the bottom of the window will show them. Otherwise, it is safe to move on.

Since we've already set up our project, we can click the "Compile" button to generate code according to our project settings. If all goes well, the project window will show a list of generated files under "Generated Items" in the section for the selected language. If there were any errors, they will be shown in the log.
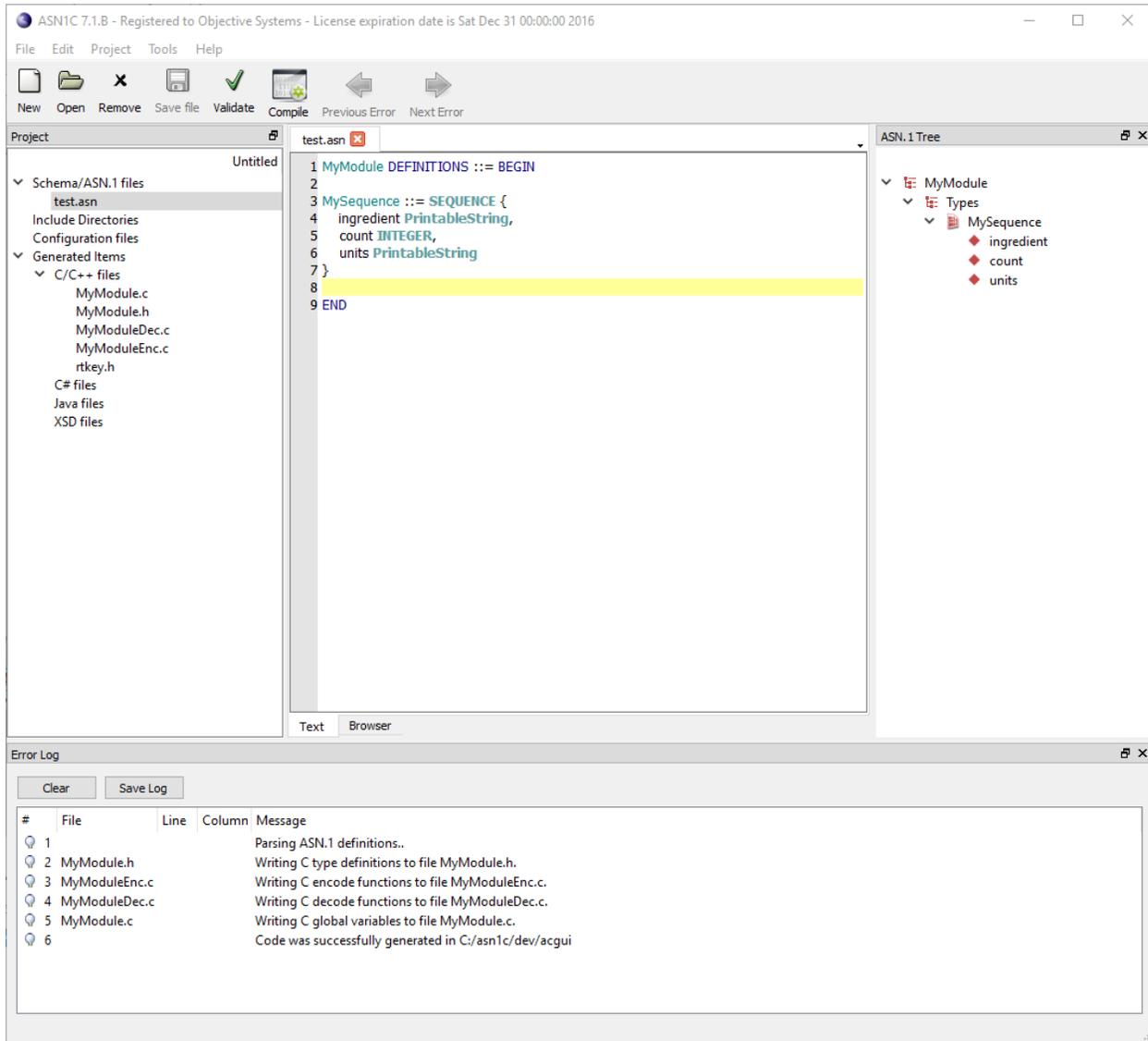
At this point, project settings can be changed and schema files can be edited as needed.

Creating a Project

Since there are a large number of options available in the code generation process, ACGUI allows settings to be saved in project files for reuse. Project files can be created, opened, and saved from the "Project" menu. If no project file is explicitly used, a dummy project will be implicitly created and can be saved to a file at a later time.

Project assets, such as ASN.1 schemas and generated source files, are visible in the "Project" window. Project settings can be changed via the Project Settings window, accessible by selecting "Project->Project Settings..." from the menubar.

Editing Schemas

The central area of the ACGUI window is dedicated to editing ASN.1 schema definition files. To create a new file, click the "New" button in the toolbar or select "File->New Schema File" in the menus. To open an existing schema file, either click the "Open" button in the toolbar or select "File->Open File..." from the menus. In both cases, the file will be added to the project and the editor will show it. Clicking on an ASN.1 schema file name in the project window will also display that file in a tab in the editor.

At any point during editing, the schema can be saved and checked for proper syntax by clicking the "Validate" button in the toolbar.

Compiling

Once a project has been created and schemas added, the schemas may be compiled. This assumes that a target language has been selected in the project. Click the "Compile" button or select "Tools->Compile" from the menus. If any files have unsaved changed, a dialog box will be displayed prompting the user to save them. Once saved, the compiler will run, generating source code and related files. From here, changes can continue to be made to the schema and to project settings and recompilation done as needed.

Interface

Editor

The central part of the ACGUI window is the schema editor. From here, schema files can be viewed and edited. To begin editing an ASN.1 schema, create a new file or open an existing file via the toolbar or menu. The file will be added to the current project and shown in the editor.

The editor window is also used to display a schema browser for navigating a validated schema. To display the browser after validating a schema, click on an item in the ASN.1 Tree window. The browser will display a hyperlinked version of the schema, centered on the definition of the selected item. Clicking the names of other defined types in the browser will cause their definitions to be shown.

By default, documents are displayed in tabs in the editor. Tabs "Text" and "Browser" at the bottom of the window are for schema editing and hyperlinked schema browsing, respectively. At the top of the "Text" tab, each schema file currently being edited has a tab.

Alternatively, ACGUI can display documents in separate subwindows. To change this, select "Tools->Options..." from the menus. In the "General" tab of the options window, change "Open files" from "In tabs" to "In MDI windows". Click "OK" and restart ACGUI. Now, open files will be displayed as separate windows within the main ACGUI window. This option is useful for viewing two files simultaneously, for example.

Project Window
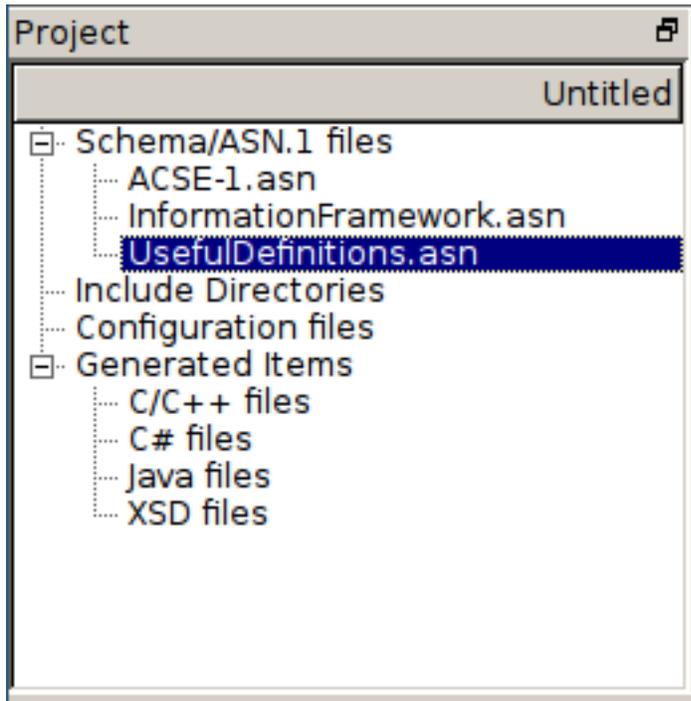
The project window allows the user to interact with project assets.

Schema/ASN.1 files	Files containing the current project's ASN.1 schema definitions.
Include directories	Directories containing auxiliary ASN.1 schema files. The current project's schema may import definitions from modules defined in an included directory.
Configuration file	An ASN1C compiler configuration file.
Generated Items	A listing of the files generated by the compiler, separated by target language.

Clicking on a schema file or configuration file in the project window will open that file in the editor. A right-click context menu is also provided for schema files, include directories, and configuration file for adding or removing these assets from the project.

ASN.1 Tree Window
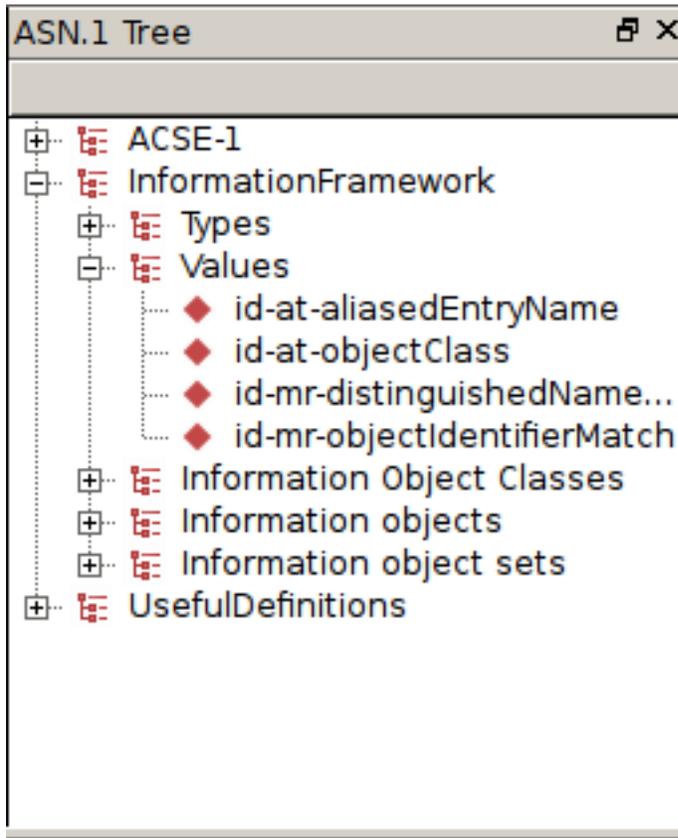
Once a schema has been validated or compiled in ACGUI, the ASN.1 Tree window provides an interactive view of the ASN.1 types defined in it. At the top level of the tree, the modules of the schema are shown. Each of these modules can be expanded to reveal branches for the types, values, information objects, etc. defined in each of them. Clicking on any node of the tree will show the relevant ASN.1 definition in a built-in browser in the editor window.

Error Log Window
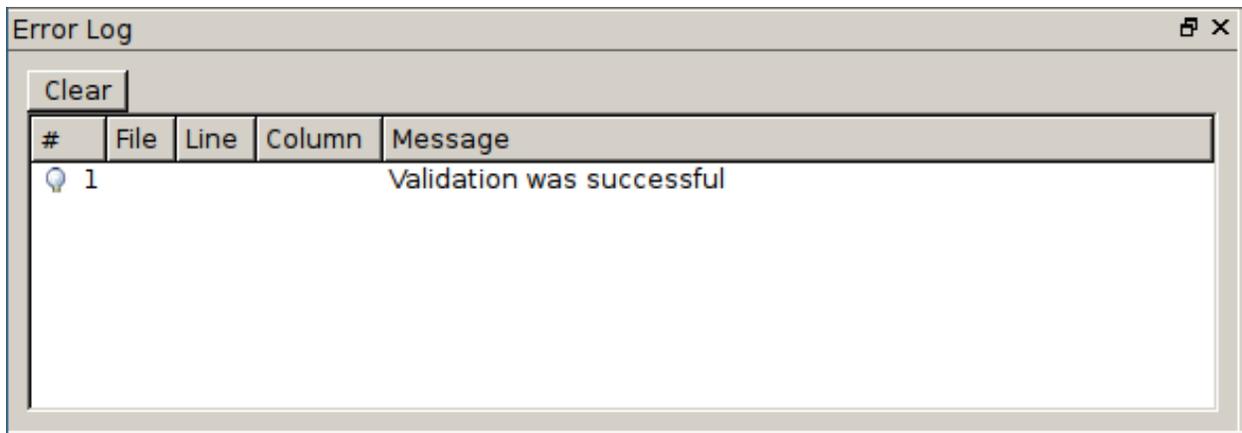
The Error Log window displays messages related to schema validation and compilation. Whenever a schema is successfully validated or compiled, the Error Log will report a success. If an error occurs, an error message will be displayed.

In many cases, an error will be associated with a particular portion of the schema being compiled. In these cases, clicking on an error will open the schema editor to the location that the error occurred. If more than one error is reported, clicking the "Next Error" and "Previous Error" buttons in the toolbar will move the editor window to the part of the schema where the next or previous error occurred.

When the reported errors are no longer needed, they can be cleared by clicking the "Clear" button in the Error Log window.

Project Settings

The project settings window is where details such as encoding rules, target language, and code features to generate are modified.

Output tab
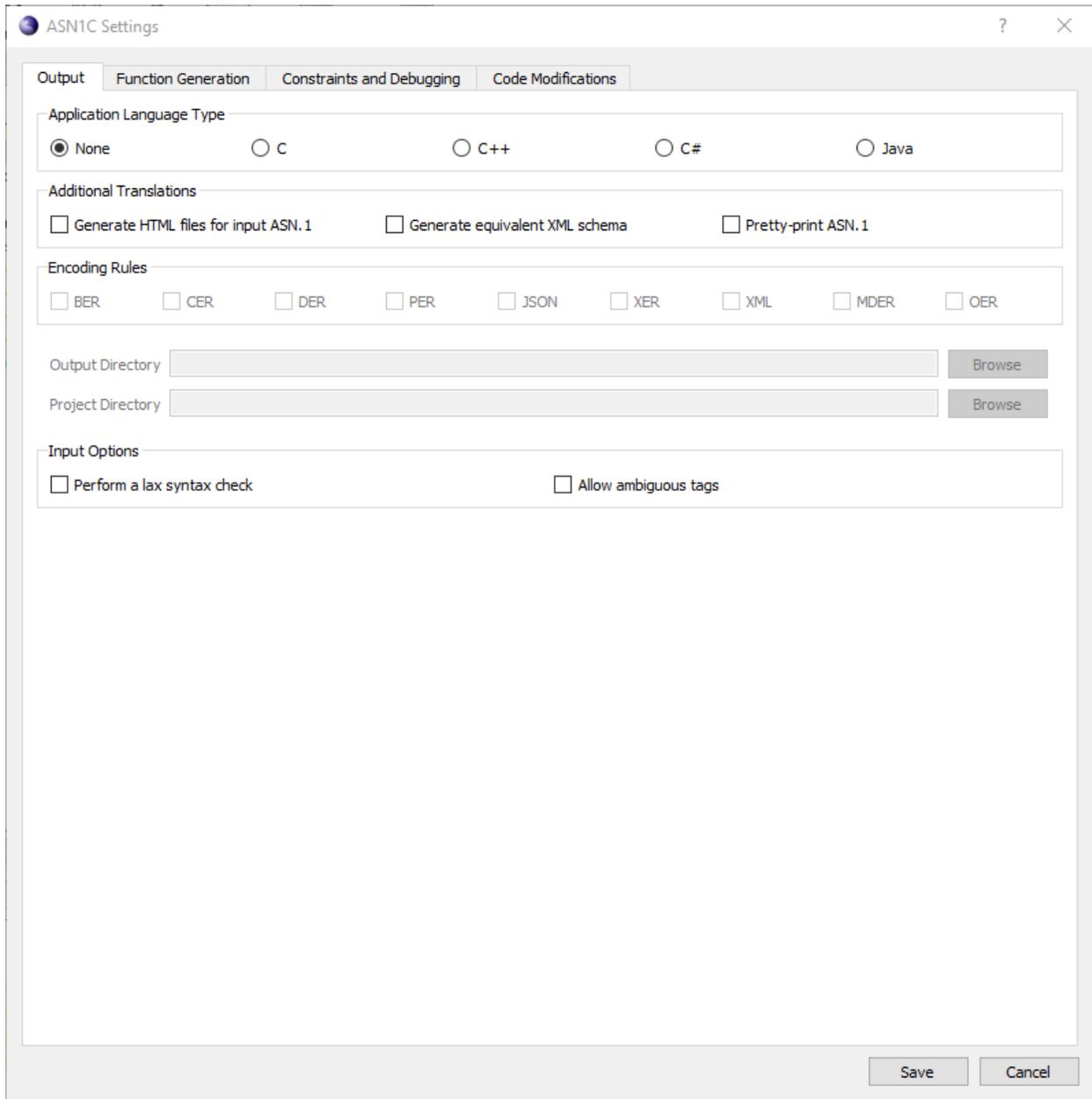
The "Output" tab contains options for selecting a target language, encoding rules, output directory. In order to compile a schema, a target language must be selected under "Application Language Type".

"Additional Translations" contains several options for generating transformed versions of the input schema, such as HTML or pretty-printed.

"Encoding Rules" allows for one or more encoding rule set to be selected for generated code.

"Input Options" affect how strict the compiler is when parsing the ASN.1 schema.

Depending on the target language selected, additional options are shown.

For C or C++ target languages, "C/C++ Output Options" controls how generated code is distributed across source files.

For C#, "C# Code Organization" controls how generated code is distributed across source files and how files are organized into directories.

For Java, "Java Code Organization" allows generated code to be organized into directories based on the ASN.1 module for which they were generated. Alternatively, generated files will be placed directly into the output directory.

Function Generation tab
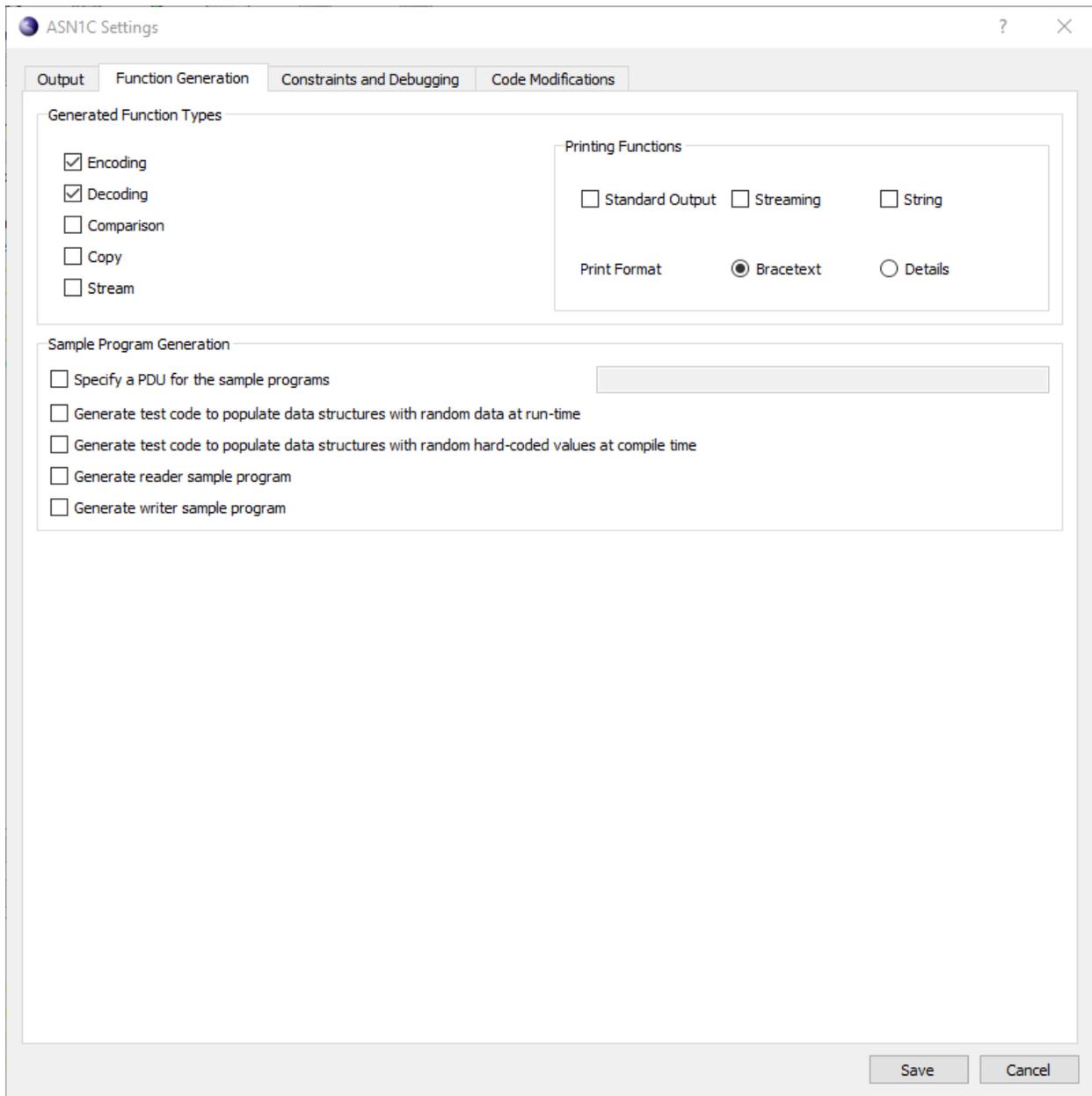
The "Function Generation" tab provides settings for what functionality to include in generated code. Options under "Generated Function Types" provide granular control of what functions to generate. The printing functions allow for various printing schemes to be generated, such as print-to-string and print-to-standard-output, and how the printed data should be formatted.

"Sample Program Generation" allows simple encoding and decoding programs, which demonstrate using the generated code, to be generated. The sample writer program can optionally encode randomly-generated test data.

Depending on the target language selected, additional options may be shown here.

For C and C++, additional functions for memory management and macros for dealing with named bits in BIT STRINGs can be generated. Initialization functions are generated by default. They may be turned off in the window.

For Java, get and set methods can be generated for members of generated classes. It is also possible to generate methods that can fetch certain types of metadata (for example, if an element is optional). A similar option exists for C#.

Constraints and Debugging tab
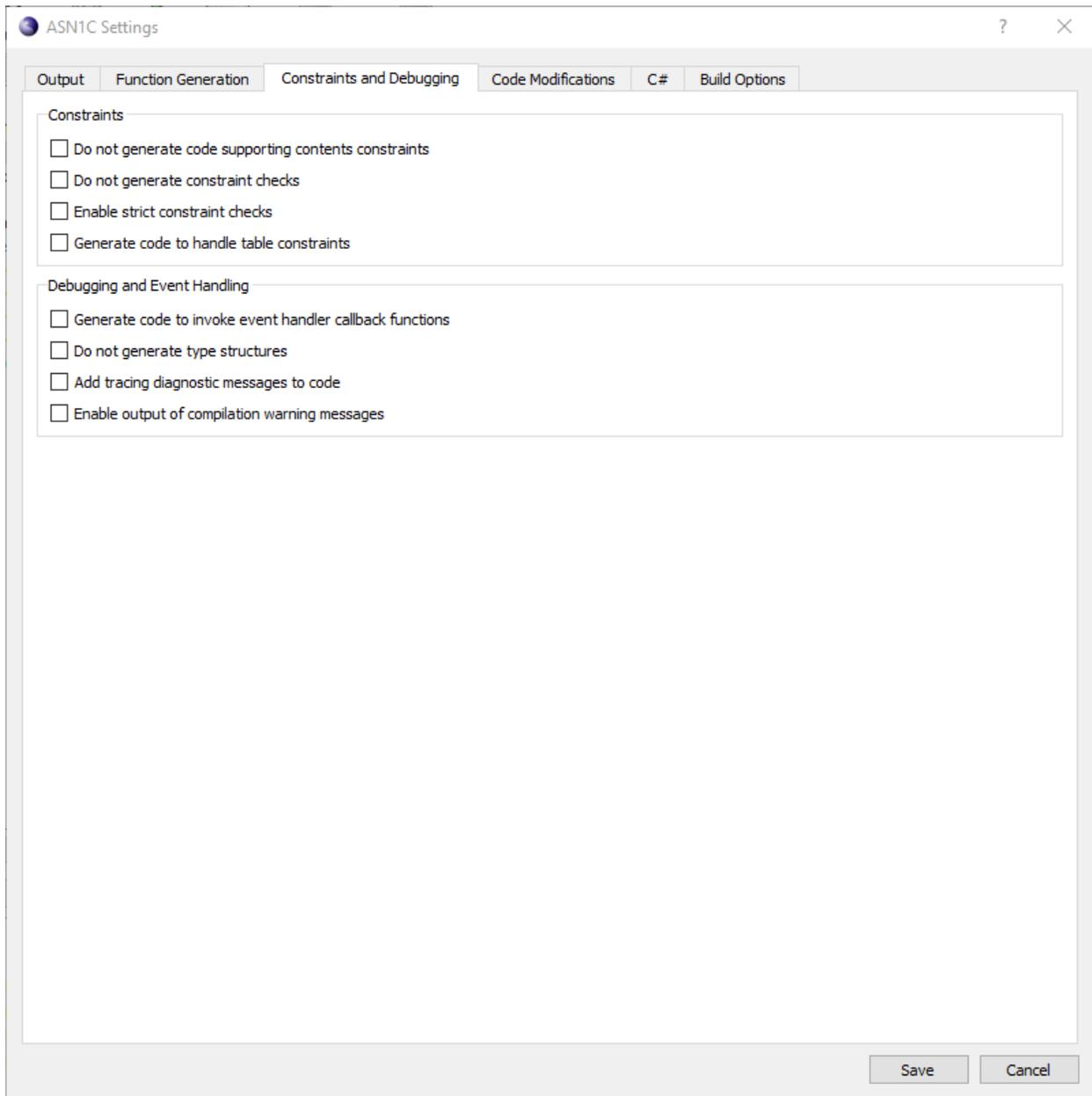
The "Constraints and Debugging" tab holds settings related to constraint handling, event handling, and logging in generated code. Under "Constraints", various types of constraint checks can be added or removed from generated code.

In "Debugging and Event Handling", settings for adding debug tracing and event hooks are available. In addition to enabling event callbacks, generation of type structures can also be disabled, in which case generated decode functionality will simply call user-created event handlers and not perform its own decoding operation.

Code Modifications tab
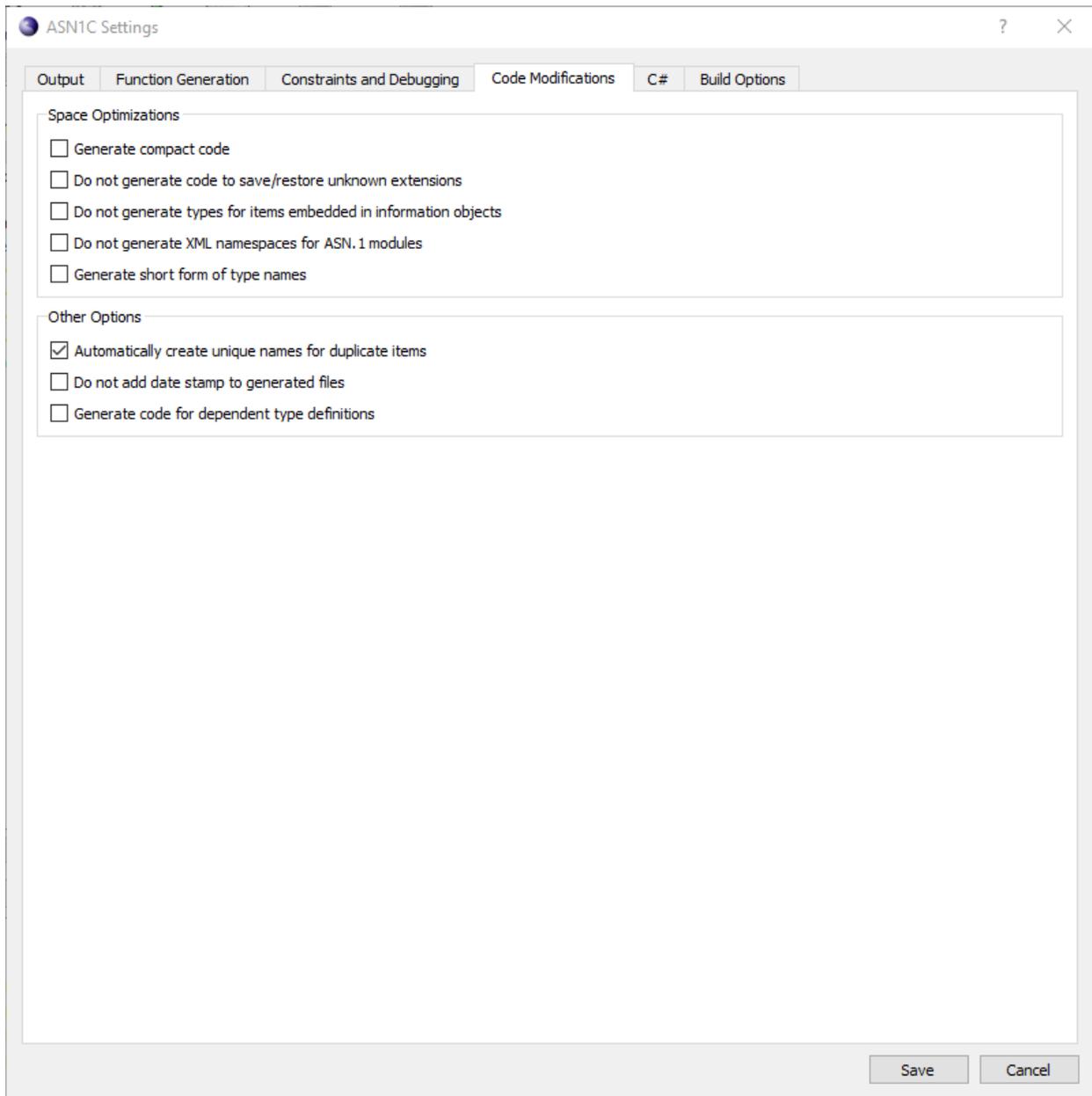
Under the "Code Modifications" tab are a number of options for generating simplified code. In "Space Optimizations", these mainly regard the removal of unwanted or unneeded functionality and shortening names of generated types.

"Other Options" provide several miscellaneous settings, including the option of generating code for types that have been imported into the current schema.

Additional code modification options that are language-specific are shown in a separate tab next to the "Code Modifications" tab. The lable on this tab will change based on the language selected. For C/C++, it is as follows:
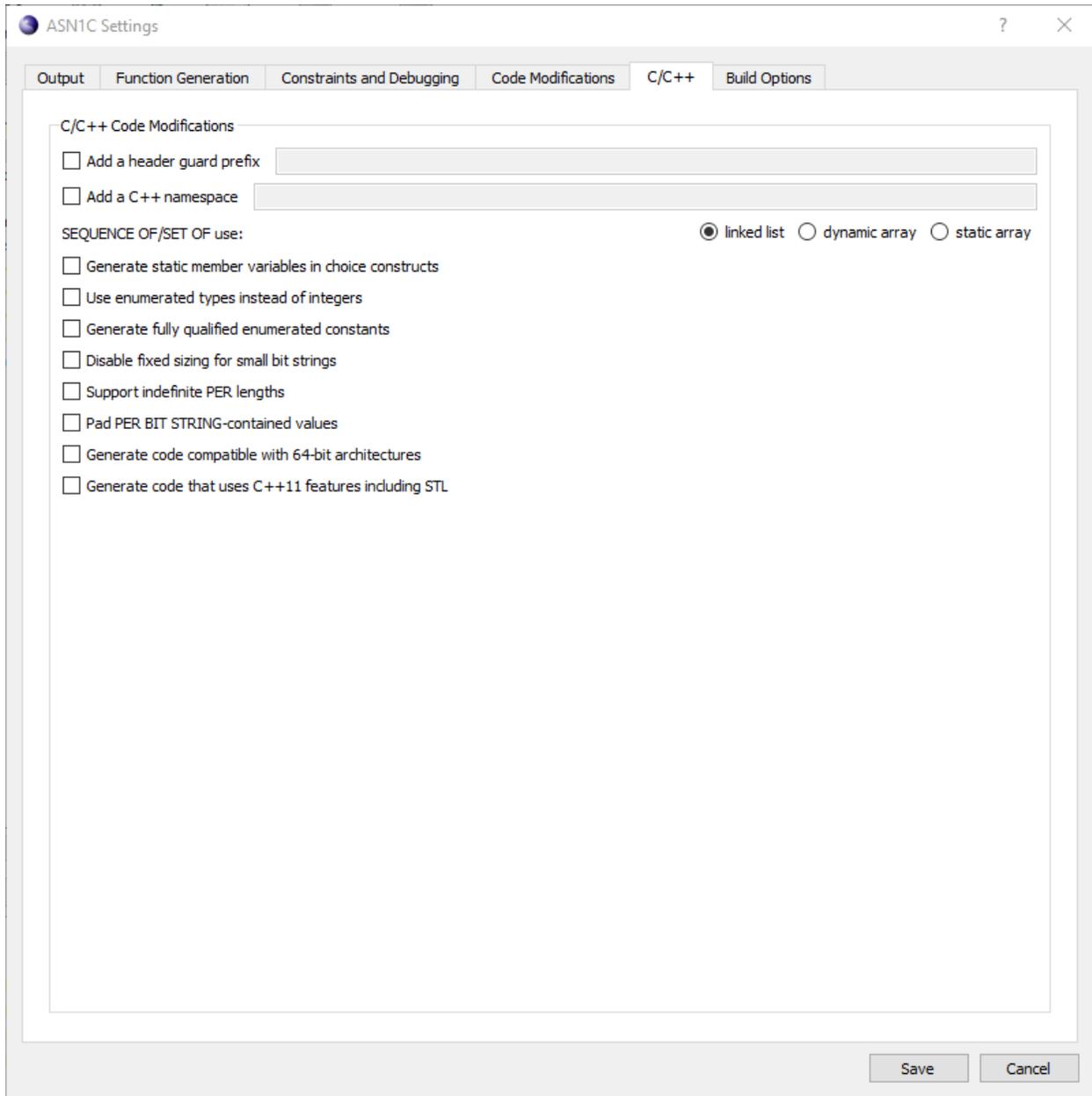
In this case, code modifications include several settings for adjusting how ASN.1 types are mapped to native C/C++ types.

For C#, the tab is as follows:
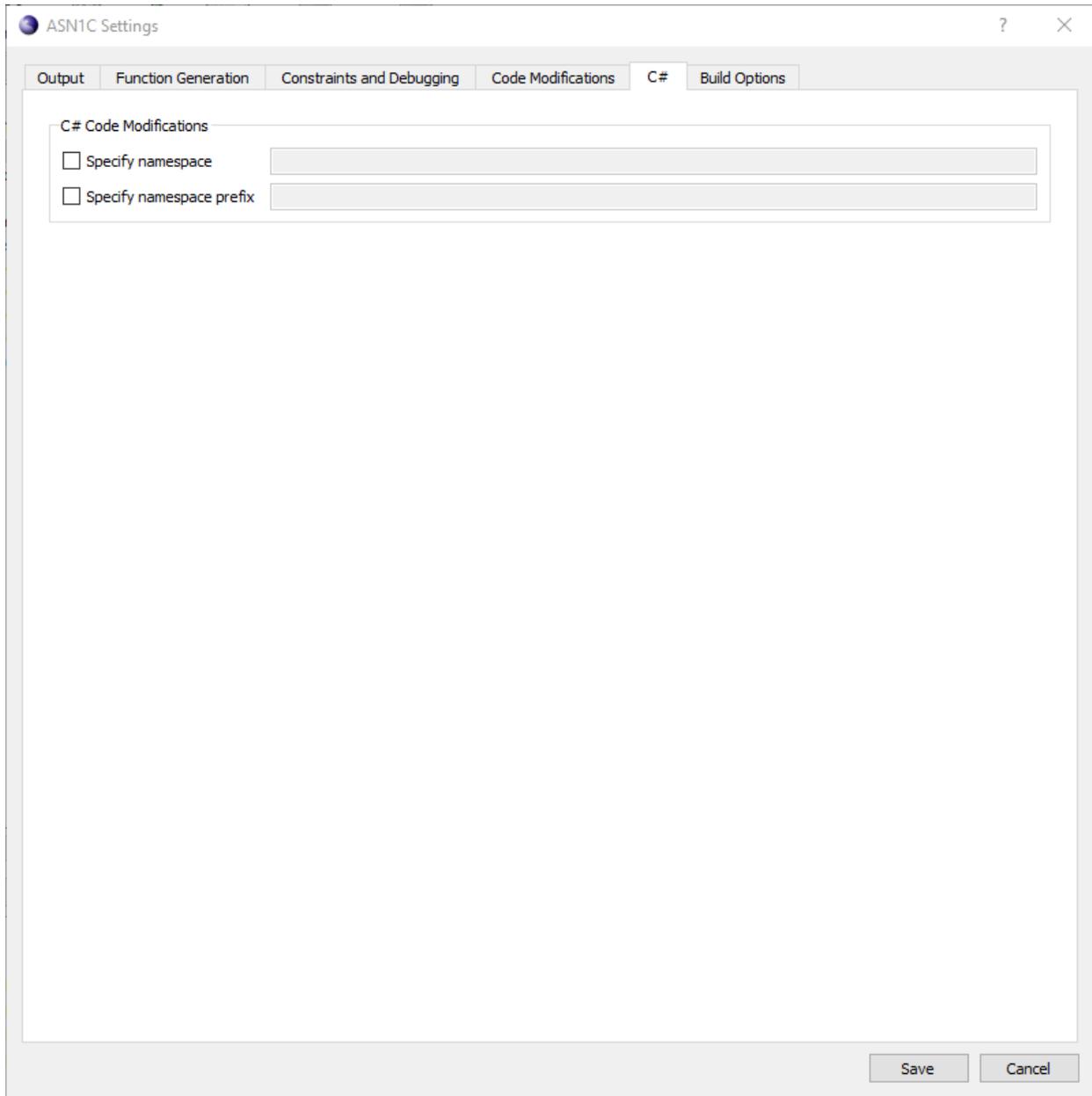
In this, modifications allow for manipulating the namespace into which code is generated. The Java tab contains similar options.

Build Options tab

When a target language other than "None" is selected, an additional "Build Options" tab contains language environment-specific settings for generating makefiles and build scripts.

For C or C++, the window is as follows:
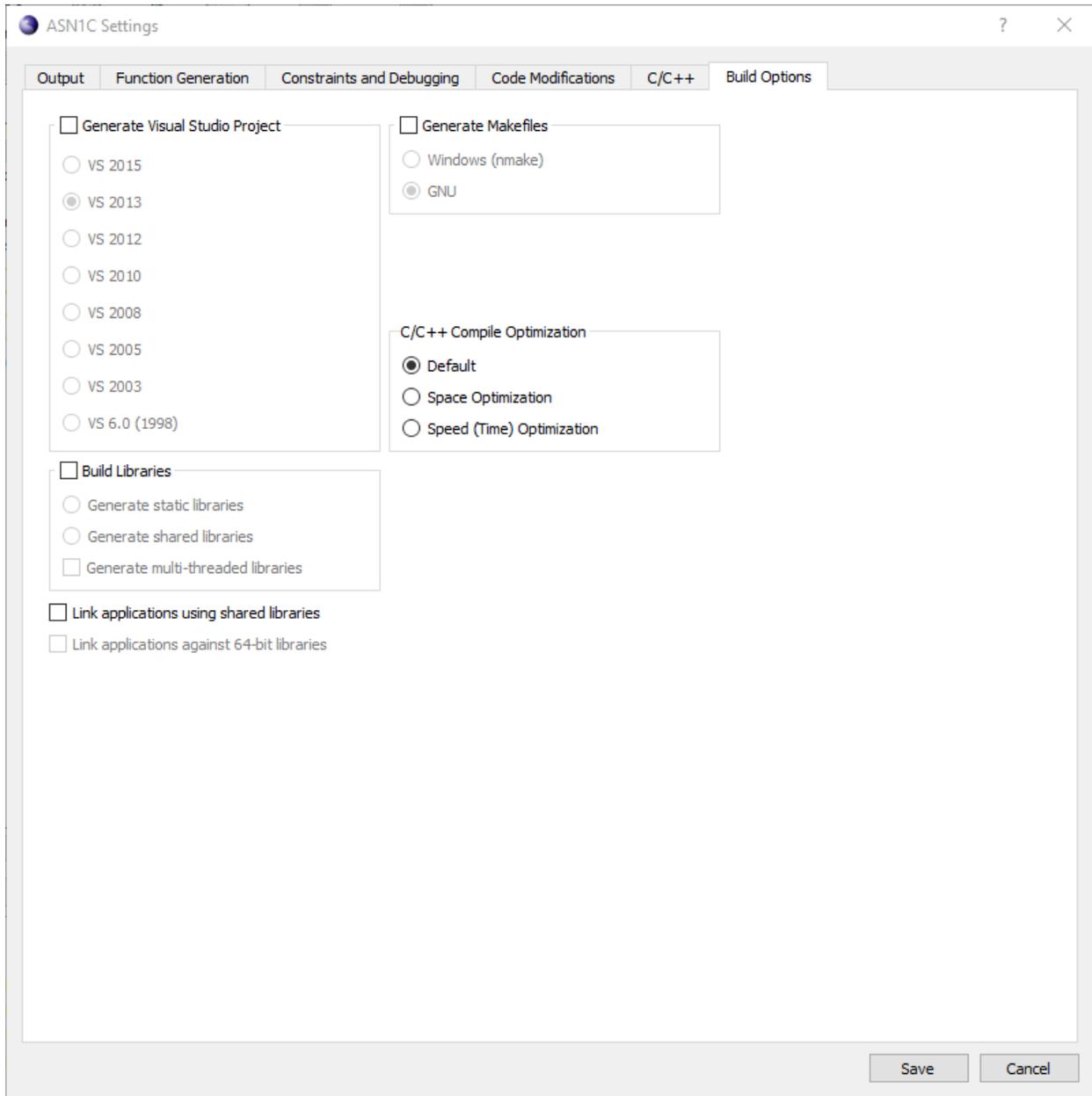
A makefile can be generated in either Windows or GNU format. For Windows, a Visual Studio project can also be generated. Under "Build Libraries", which will generate the build script to build a library rather than an executable, the desired variety of library can be selected.

"C/C++ Compile Optimization" allows for setting whether Space or Time optimization qualifiers should be added to the C compilation command-line in the makefile.

For C#, the following options are displayed:
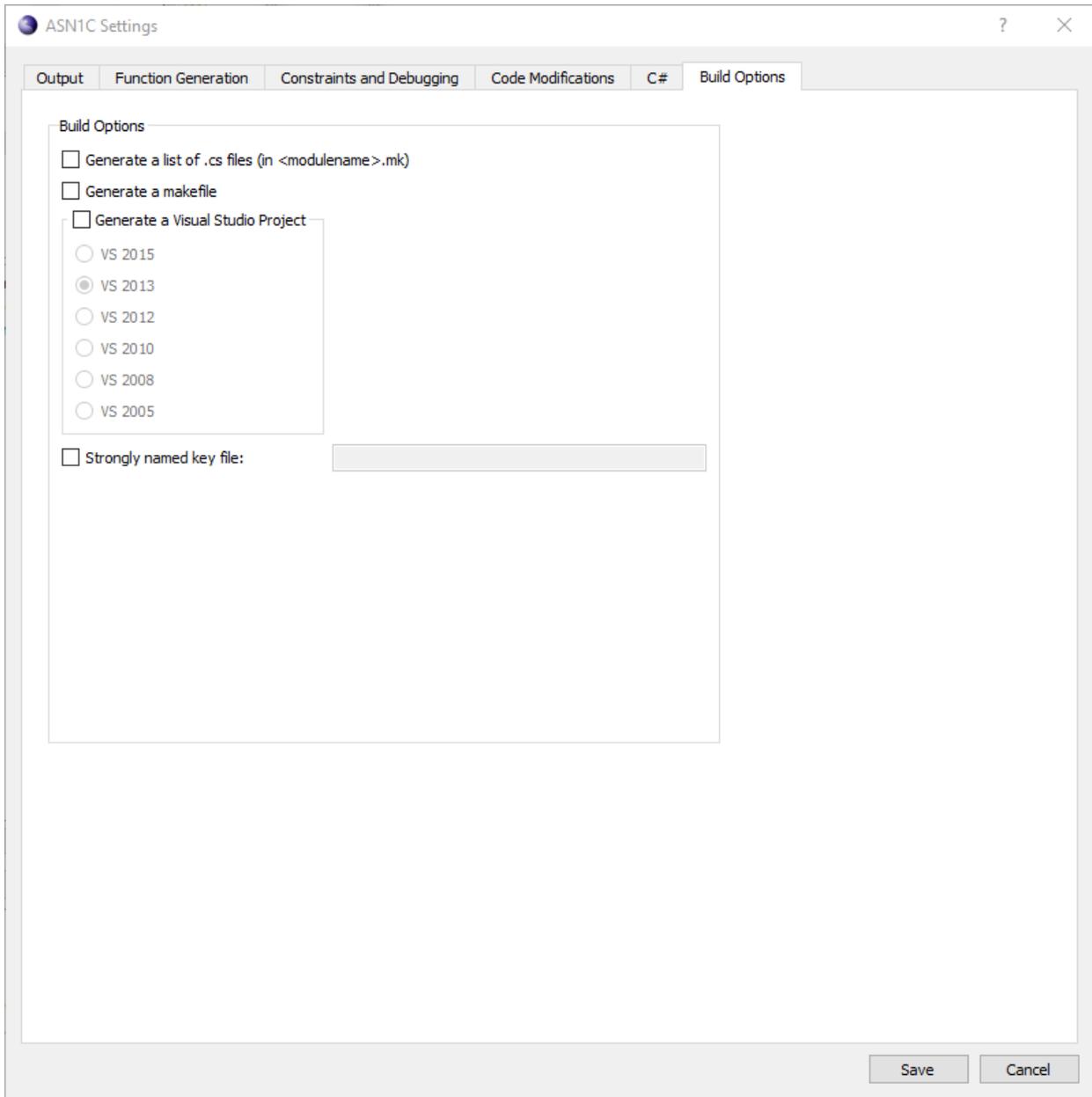
Similarly for C#, a makefile or Visual Studio project can be created, optionally including a *.mk file listing the files generated. An option to specify a strongly named key file also exists.

For Java, the following is displayed:
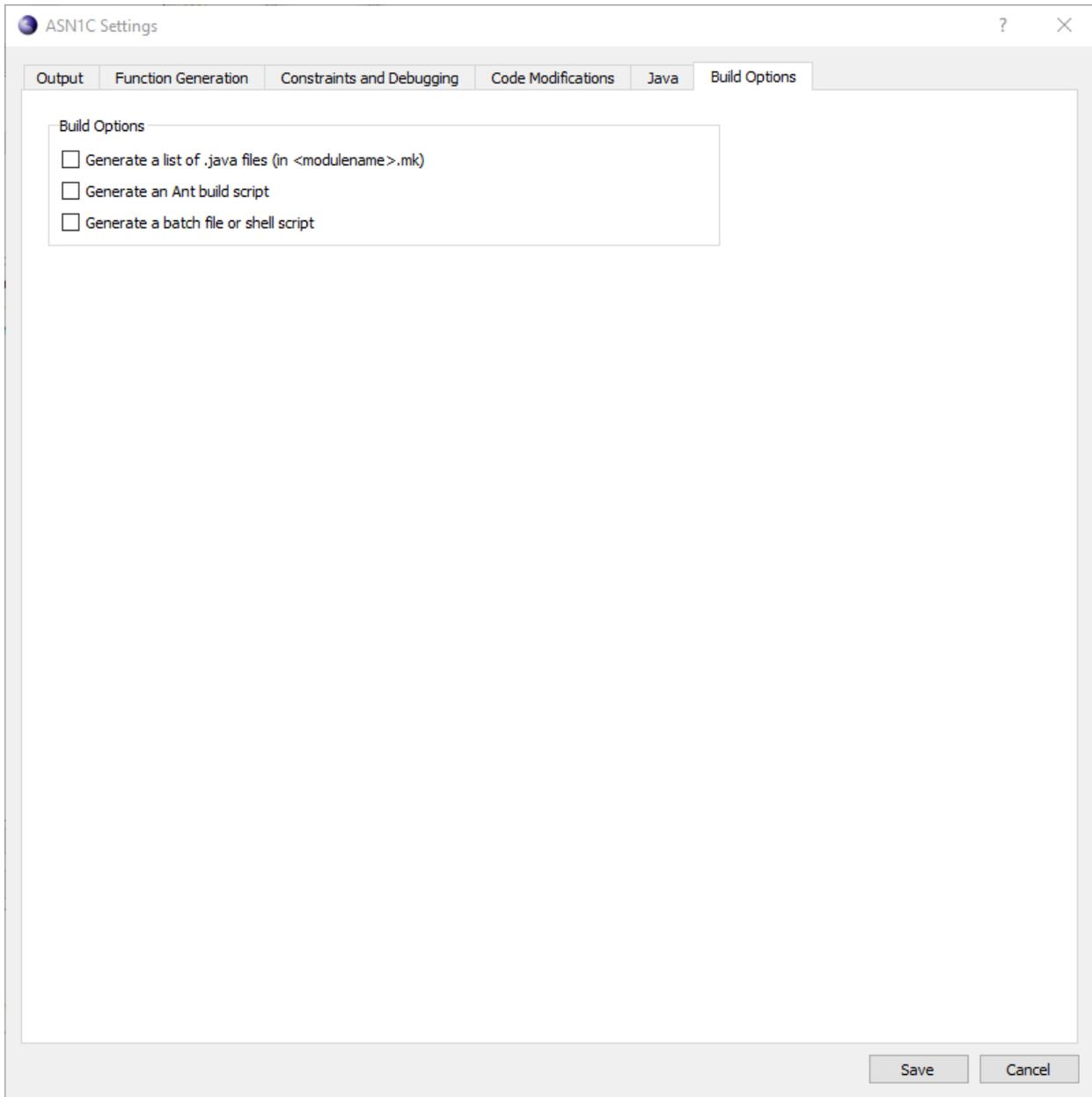
Java can also provide a *.mk generated file list, as well as Ant build script and a batch or shell script.

Chapter 4. Generated Java Source Code Overview

A separate Java source file with extension '.java' is generated for each production encountered within an ASN.1 source file. Every ASN.1 type is mapped to a Java class. This is true even at the lowest levels – types such as BOOLEAN, INTEGER, and NULL all have wrapper classes.

General Form of a Generated Java Source File

The following items may be present in a generated java file:

- Package specification

- Import statements

- Class declaration

- A tag constant object declaration

- Public member variables

- Constructors

- Public decode() method

- Public encode() method

- Other methods

- Inner SAX Handler class (XER or XML only)

Additional specialized items may be present as well depending on the base type of the target production. These specialized items are discussed in the sections on ASN.1 to Java mappings for the various ASN.1 types.

A complete generated Java source file for the 'EmployeeNumber' production within the production within the ASN.1 sample file 'employee.asn' can be found on the following page. The ASN.1 production from which this file was generated is as follows:

```
EmployeeNumber ::= [APPLICATION 2] IMPLICIT INTEGER
```

The generated code is as follows:

```
package sample_ber.Employee;

import com.objsys.asn1j.runtime.*;
import java.io.*;
import java.util.*;

public class EmployeeNumber extends Asn1Integer {
  public final static Asn1Tag TAG =
     new Asn1Tag (Asn1Tag.APPL, Asn1Tag.PRIM, 2);

  public EmployeeNumber() {
     super();
  }
```

```
    public EmployeeNumber (int value_) {
        super (value_);
    }

    pubilc void decode
        (Asn1BerDecodeBuffer buffer, boolean explicit, int implicitLength)
         throws Asn1Exception, java.io.IOException
    {
        final int llen = (explicit) ?
            matchTag (buffer, TAG) : implicitLength;

        super.decode (buffer, false, llen);

        if (explicit && llen == Asn1Status.INDEFLEN) {
            matchTag (buffer, Asn1Tag.EOC);
        }
    }

    public int encode (Asn1BerEncodeBuffer buffer, boolean explicit)
        throws Asn1Exception
    {
        int aal = super.encode (buffer, false);

        if (explicit) {
            aal += buffer.encodeTagAndLength (TAG, aal);
        }

        return (aal);
    }
}
```

Package Specification

The package specification is the first item in the file and is declared using the 'package' keyword. By default, this is set to the name of the ASN.1 module that is being compiled. However, this can be modified by using the –pkgpfx and – pkgname command line options. The –pkgpfx option adds the specified prefix before the module name. For example, if an ASN.1 module named 'Employee' is being compiled and '-pkgpfx test.' is specified on the command line, the package name in the generated source files would be 'test.Employee'. The –pkgname switch takes this a step further. It allows specification of the full package name. In the sample specification above, '-pkgpfx sample_ber.' was specified on the compiler command line.

Standard import statements are added for the ASN1C Java run-time classes and Java utility classes. Import statements may also be added for items imported from other ASN.1 modules if they don't exist within the package being generated.

Class Declaration

Next comes the class declaration. It is of the following form:

```
    public class <ProdName> extends <BaseClass>
```

<ProdName> is the name of the production in the ASN.1 source file. <BaseClass> is a class from which the type is derived. This can either be a standard run-time or compiler-generated class. In our example, the EmployeeNumber

is an INTEGER, so we can directly extend the Asn1Integer run-time base class. If we had a declaration such as the following:

```
EmployeeSSNumber ::= [APPLICATION 22] EmployeeNumber
```

Our EmployeeSSNumber class would be derived from the compiler-generated EmployeeNumber class as follows:

```
public class EmployeeSSNumber extends EmployeeNumber
```

Note: the preceding example is not true if –compact is specified. In that case, all intermediate classes would be removed so *EmployeeSSNumber* would extend *Asn1Integer* as in the first case.

Tag Constant

The next item in the generated source file is a tag constant. This is only generated if the production is tagged. The runtime class *Asn1Tag* is used for this constant. This class contains methods for operating on ASN.1 tag values. In the sample above, the [APPLICATION 2] tag that is present in the ASN.1 production definition is represented by the generated tag constant.

Public Member Variables

The next section of the file would be public member variables. In our example above, no member variables are present. This is because INTEGER is a primitive type, so the member variable in which the integer value is stored can be found in the *Asn1Integer* base class from which this class is derived. This is true for all primitive types – the value will be contained within the run-time base class.

Constructed types will contain public member variables to represent the elements that make up the type. For example, the following SEQUENCE production:

```
Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
    givenNameIA5String,
    initial IA5String,
    familyNameIA5String
}
```

will result in the following public member variables being added to the generated class:

```
public Asn1IA5String givenName;
public Asn1IA5String initial;
public Asn1IA5String familyName;
```

Note that the member variables are public. They were declared this way to make access easier. A trade-off existed between ease-of-use and secure encapsulation. The ease-of-use approach was chosen because it was felt that the repeated use of get/set methods within deeply nested structures would be too clumsy and bulky in most applications. Therefore, the variables were made public to make the encapsulated values easier to set and retrieve. Consistency checks have been added in some methods to make sure values of the correct types are specified for these elements. These checks are discussed in the sections on the specific constructed types.

This behavior can be overridden by using the *-getset* command-line option. This will cause the member variables to be declared as protected variables and accessor/mutator methods (i.e. get/set) methods added to access the variables.

Constructors

Constructors are generated to allow an object to be initialized in a number of different ways. All productions have a default constructor with no parameters. This creates an empty object that can be populated at a later time. Constructors

are also created that take a parameter of the base type value to allow direct population upon creation of an object. In our example code, two constructors were generated:

```
public EmployeeNumber () {
    super();
}

public EmployeeNumber (int value_) {
    super (value_);
}
```

More complex constructed ASN.1 types such as a SEQUENCE would have a constructor that would have an argument for each defined element. A CHOICE on the other hand would have a unique constructor for each of the possible choice items. See the sections on specific ASN.1 types to find out exactly what constructors are generated for a given type.

Decode Method

The generated decode method for BER/DER has the following general form:

```
public void decode (Asn1BerDecodeBuffer buffer,
                    boolean explicit, int implicitLength);
```

Users of the C and C++ version of the product might recognize this form. It is very similar to the C function prototype. A reference to an *Asn1BerDecodeBuffer* object is passed that specifies the message being decoded. This is similar to the context variable in the C version of the product.

The *explicit* and *implicitLength* arguments should be of no concern to the average user. The *explicit* argument should be set to `true` and the *implicitLength* argument set to zero. These arguments are only used in internal calls generated by the compiler when implicit tagging is used. In this case, the decoder will at times only be concerned with decoding the contents of a field and not the tag information. At the outer levels, it will always be necessary to decode a tag and length.

The Java decode method reports errors by throwing exceptions. This is a change from the C/C++ version that returned a status value. The method signature includes the following throws clause:

```
throws Asn1Exception, java.io.IOException
```

The *Asn1Exception* class is the base class for all exceptions defined for ASN1C. A complete list of these exceptions can be found in the ASN1C Exceptions section.

For PER, the signature is similar:

```
public void decode (Asn1PerDecodeBuffer buffer);
```

In this case, the *explicit* and *implicitLength* arguments are not required since PER has no tagging. The only required argument is a reference to a decode buffer object.

For XER or XML, the default behavior is to generate code using the `org.xmlpull.v1.XmlPullParser` class to parse the XML. Other parsers can be specified using `-stax` or `-sax`. When the XmlPull API is used, the following methods are generated:

```
public void decodeDocument(org.xmlpull.v1.XmlPullParser reader)

public void decode(org.xmlpull.v1.XmlPullParser reader, boolean asGroup)
```

If `-stax` is specified on the command line, the signatures change slightly:

```
public void decodeDocument(javax.xml.stream.XMLStreamReader reader)
                    throws Asn1Exception, javax.xml.stream.XMLStreamException

public void decode(javax.xml.stream.XMLStreamReader reader, boolean asGroup)
                    throws Asn1Exception, javax.xml.stream.XMLStreamException
```

If `-sax` is specified on the command line, the SAX API is used and two overloaded decode methods are generated:

```
public void decode (XMLReader reader, String xmlURI);

public void decode (XMLReader reader, InputStream byteStream);
```

These take as arguments an XML reader object reference and a reference to an input source object. The XML reader object is a standard class within an XML parser that reads and parses an XML document. The input source can either be a URI (this can be a local filename) or an in-memory byte stream.

For OER, the decode signature is:

```
public void decode(Asn1OerDecodeBuffer buffer)
                        throws java.io.IOException
```

Encode Method

The generated encode method for BER/DER has the following general form:

```
public int encode (Asn1BerEncodeBuffer buffer, boolean explicit);
```

The *Asn1BerEncodeBuffer* argument specifies the buffer into which the message will be encoded. The explicit argument is primarily for use by the compiler for generating internal calls to handle implicitly tagged elements in constructed types. Users should always set this argument to `true`.

The encode method returns the length of the encoded component. Unlike the C /C++ version, this return value does no double as a status value as well. Any errors that occur in the encode process are reported by throwing an ASN1C exception. A complete list of these exceptions can be found in the ASN1C Exceptions section.

The general form of a PER encode method is as follows:

```
public void encode (Asn1PerEncodeBuffer buffer);
```

In this case, the *explicit* argument is not required since PER has no tagging. The only required argument is a reference to an encode buffer object. Also note that the return value is void instead of int. No intermediate lengths are returned during the encoding of a PER message. Any errors that occur are reported as an exception; hence there I no need for a return value.

The general form of an XER or XML encode method is as follows:

```
public void encode (Asn1XerEncoder buffer, String elemName);
```

In this case, the buffer reference is to an XER encoder object and an element name argument is added. The *Asn1XerEncoder* reference is to an interface that allows either a message buffer or output stream object to be passed into the method. In the case of XML, this object reference would be to an *Asn1XmlEncoder* interface.

The element name is the name of the element that is to bracket the XML encoded value (i.e. <elemName>value</elemName>).

The method return type is void because errors are reported through the exception mechanism.

The general form of an OER encode method is as follows:

```
public void encode (Asn1OerEncodeBuffer buffer)
        throws java.io.IOException
```

Other Methods

Other generated methods include the following; these are documented more fully in separate sections:

• Get and set methods are generated when the `-getset` command-line swith is set. These methods allow bean-style access to each element in a container type (`SEQUENCE`, `SET`, and `CHOICE`).

• Print methods are generated when the `-print` or `-prtToStr` command-line switches are specified. Users can call these methods to write a formatted representation of the object to a print writer, a print stream, or a string.

• Public `set_<element>` methods are generated for all `CHOICE` types. These allow users to select the desired element to be encoded in a `CHOICE`. If the `-getset` switch is specified, these methods are replaced by set methods as described above.

• If `-copy` is specified on the command-line, ASN1C generates overrides to the standard `clone` method.

• ASN1C generates `equals` and `hashCode` methods when `-compare` is specified on the command line.

• Specifying `-genmetadata` on the command line causes ASN1C to generate methods that allow users access to some of the meta data for containers, like whether elements are required to be present or the value range specified for an element.

Inner Classes

The generation of code for XER or XML may cause the following inner class definition to be generated:

```
public class SaxHandler extends Asn1XerSaxHandler {
    Asn1XerSaxHandler mElemSaxHandler;
    StringBuffer mCurrElemValue;

    SaxHandler() {
        <code ..>
    }

    public void startElement (String namespaceURI, String localName,
                              String qName, Attributes atts)
        throws SAXException
    {
        <code ..>
    }

    public void characters (char[] ch, int start, int length)
        throws SAXException
    {
```

```
            <code ..>
        }

    public void endElement (String namespaceURI,
                            String localName, String qName)
        throws SAXException
    {
            <code ..>
        }
    }
```

This is an implementation of a standard SAX content handler class. As the XML parser software parses messages, the methods within this class are invoked with the parsed content. The *startElement* method is invoked after a start element tag (<tag>) is parsed. The *characters* method is invoked one or more times to pass the content between tags into the application. The *endElement* method is invoked when an end element tag (</tag>) is encountered.

The ASN1C compiler generates custom code for each ASN.1 type within a given specification to parse the XML contents and fill in the generated Java objects.

Error Handling

As noted elsewhere in this manual, the asn1c runtime and generated code will throw exceptions that are, or derive from, Asn1Exception. The exception tells you what was wrong and provides a stack trace. If you need additional information you may try the following:

• Identify the byte location in the input data using Asn1DecodeBuffer.getByteCount().

• Identify the problem element using element name tracking. The procedure to do this is simple:

  1. Add "-events" to your asn1c command line. This is necessary for element name tracking to work.

  2. Obtain an Asn1Context object from your encode/decode buffer using Asn1MessageBuffer.getContext().

  3. Enable element name tracking before encoding/decoding by invoking Asn1Context.enableElementTracking().

  4. Get the name of the problem element by invoking Asn1Context.getCurrentElement() inside your exception handling code.
  Element name tracking is demonstrated in the sample Writer in sample_ber/EventHandler.

Chapter 5. ASN.1 Type to Java Class Mappings

The following sections discuss the specific mappings of ASN.1 and XSD types to Java classes.

BOOLEAN

The ASN.1 BOOLEAN type is converted to a Java class that extends the *Asn1Boolean* run-time class. This base class encapsulates the following public member variable:

```
public boolean value;
```

This is where the Boolean value to be encoded is stored. It also contains the result of a decode operation. Since it is public, it can be accessed directly to get or set the value. The generated constructors can also be used to set the value.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

XSD Type

```
<xsd:boolean>
```

Generated Java class

```
public class <name> extends Asn1Boolean {
    public <name> ()
    {
        super();
    }

    public <name> (boolean value_)
    {
        super (value_);
    }
}
```

This definition assumes a simple assignment of the form "<name> ::= BOOLEAN" (i.e., no tagging or subtypes have been added to the BOOLEAN declaration). In this case, no specific encode or decode methods are generated – calls to these methods pass through to the generic calls defined in the base class. This is true of all other primitive type declarations as well unless otherwise noted.

INTEGER

The ASN.1 INTEGER type is converted to a Java class that extends the *Asn1Integer* run-time class. This base class encapsulates the following public member variable:

```
public long value;
```

This is where the integer value to be encoded is stored. It also contains the result of a decode operation. Since it is public, it can be accessed directly to get or set the value. The generated constructors can also be used to set the value.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

XSD Types

```
<xsd:integer>,          <xsd:byte>,          <xsd:short>,          <xsd:int>,
<xsd:long>,   <xsd:unsignedByte>,   <xsd:unsignedShort>,   <xsd:unsignedInt>,
<xsd:unsignedLong>, <xsd:positiveInt>, <xsd:nonPositiveInt>, <xsd:negativeInt>,
<xsd:nonNegativeInt>
```

Generated Java class:

```
public class <name> extends Asn1Integer {
    public <name> () {
        super();
    }
    public <name> (long value_) {
        super (value_);
    }
}
```

This shows the class generated for a simple INTEGER assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

Large Integer Support

The maximum size for a Java long integer type is 64 bits. 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 class to be derived from the *Asn1BigInteger* class instead of the *Asn1Integer* class. The *Asn1BigInteger* class encapsulates an object of the Java *BigInteger* class. This provides full support for working with integers of arbitrary lengths.

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 class header:

```
class SecurityKeyType extends Asn1BigInteger
```

The value field is populated by creating a Java BigInteger object and either passing it in through the constructor or using it to directly populate the public member variable named **value** declared in the base class.

BIT STRING

The ASN.1 BIT STRING type is converted to a Java class that extends the *Asn1BitString* run-time class. This base class encapsulates the following two public member variables:

```
public int numbits;
public byte[] value;
```

These describe the bit string to be encoded or decoded.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

Generated Java class:

```
public class <name> extends Asn1BitString {
    public <name> () {
        super();
    }

    public <name> (int numbits_, byte[] data) {
        super (numbits_, data);
    }

    public <name> (boolean[] bitValues) {
        super (bitValues);
    }

    public <name> (String value_)
    throws Asn1ValueParseException {
        super (value_);
    }
}
```

This shows the class generated for a simple BIT STRING assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

The constructors generated for this type provide additional options for populating the member variables in the base class. In addition to passing the string using the numbits and data arguments to specify a bit string in native format, the string can be specified as an array of boolean values or as a string. The string form expects the string to be passed in the ASN.1 value notation format for either a binary string (i.e., 'xxxx'B) or a hexadecimal string (i.e., 'xxxx'H).

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 simply the bit names and values in the following general form:

```
public final static int <name> = <value>;
```

The base class contains the following methods for using these generated constants:

- `set` : This method can be used to set a bit in the bit string to be set. There is also an overloaded version that takes a boolean value argument that can be used to set the bit to the given boolean value.

- `clear` : This method can be used to clear the named bit in the bit string.

- `isSet` : This method can be used to test if the named bit is set or clear.

See the *Asn1BitString* class description in the run-time section for more details on these methods.

OCTET STRING

The ASN.1 OCTET STRING type is converted to a Java class that extends the *Asn1OctetString* run-time class. This base class encapsulates the following public member variable:

```
public byte[] value;
```

The number of octets to be encoded or that were decoded is specified in the built-in length component of the array object (i.e., value.length).

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

XSD Types

`<xsd:hexBinary>`, `<xsd:base64Binary>`

Generated Java class

```java
public class <name> extends Asn1OctetString {
    public <name> () {
        super();
    }

    public <name> (byte[] data) {
        super (data);
    }

    public <name> (byte[] data,
                   int offset,
                   int nbytes)
    {
        super (data, offset, nbytes);
    }

    public <name> (String value_)
    throws Asn1ValueParseException {
        super (value_);
    }
}
```

This shows the class generated for a simple OCTET STRING assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

The constructors generated for this type provide additional options for populating the member variables in the base class. In addition to passing the string directly using the data argument, the string form can be used. The string is passed in ASN.1 value notation format for either a binary string (i.e., 'xxxx'B), hexadecimal string (i.e., 'xxxx'H), or a character string (i.e., 'xxxx'). A constructor also exists that allows a portion of a byte array starting at a given offset and consisting of a given number of bytes to be used to populate the variable.

Character String Types

The Java version of the compiler contains support for the various ASN.1 character string types including the BMP, Universal and UTF-8 string types. All character strings in Java are based on 16-bit Unicode characters except for UniversalString which is based on a 32-bit character set.

All character string types are derived from the *Asn1CharString* base class (except the UniversalString). This class contains the following public member variable that holds the character string contents:

```
public String value;
```

Each of the specific ASN.1 character string types except UniversalString has an associated Java class that is derived from the Asn1CharString base class. The general form of the Java class name for each of the ASN.1 string types is *Asn1* followed by the ASN.1 string type name. For example, *IA5String* is represented by the *Asn1IA5String* class, *NumericString* by the *Asn1NumericString*, etc.

The UniversalString associated Java class is derived from Asn1Type and it contains the following public member that holds the character string contents:

```
public int value[];
```

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

```
<name> ::= <CharStrType>
```

XSD Types

`<xsd:string>` and all related types including date/time types and duration.

Generated Java class

```
public class <name> extends Asn1<CharStrType> {
    public <name> () {
        super();
    }
    public <name> (String value_) {
        super (value_);
    }
}
```

ENUMERATED

The ASN.1 ENUMERATED type is converted into a Java class that extends the *Asn1Enumerated* run-time class. In version 6.1, the generated code was changed to conform to Joshua Bloch's static enumeration pattern (as explained in *Effective Java*). Enumerated values are created as singletons to allow for lazy initialization. A specially named object,

dec, is created to hold decoded values. In combination, these changes improve application performance, since only a fixed number of objects are allocated for any execution of the application.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

```
<name> ::= ENUMERATED { <e1>, <e2>, ..., <en> }
```

XSD Types

Any type with an `<xsd:enumeration>` restriction.

Generated Java class

```
public class <name> extends Asn1Enumerated {
    private static <name> <e1> = null;
    private static <name> <e2> = null;
    ...
    private static <name> dec = new <name> (-1);

    protected <name> (int value_) {
        super (value_);
    }

    public static <name> <e1>() {
        if (<e1> == null) <e1> = new <name>(<v1>);

        return <e1>;
    }
    ...

    public static <name> valueOf(int value_) { ... }

    protected static <name> dec() { return dec; }

    public void decode () { ... }
    public int encode () { ... }
    public void print () { ... }
}
```

Note

1. The ... notation used in the ASN.1 definition above does not represent the ASN.1 extensibility notation. It is used to show a continuation of the enumerated sequence of values.

2. The <e1>, <e2>, etc. items denote enumerated constants. These can be in identifier only format or identifier(value) format. The <v1>, <v2>, etc. items denote the enumerated values. These are sequential numbers starting at zero if no values are provided. Otherwise, the actual enumerated values are used.

3. The public methods that are generated are shown without arguments or function bodies for brevity.

In the case of the enumerated type, encode/decode methods are always generated. These verify that the given value is within the defined set. An *Asn1InvalidEnumException* is thrown if the value is not in the defined set unless the enumeration is extensible. In this case, no exception is thrown.

If an extensibility marker (...) is present in the ASN.1 definition, it will not affect the generated constants. A constant will be generated for all options – both root and extended. However, in the *ValueOf* method, an "undefined" constant will be returned to indicate that the value is not in the original specification.

NULL

The ASN.1 NULL type is converted into to a Java class that extends the *Asn1Null* run-time class. This base class does not contain a public member variable for a value because the NULL type has no associated value.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

```
<name> ::= NULL
```

Generated Java class

```
public class <name> extends Asn1Null {
    public <name> () {
        super();
    }
}
```

This shows the class generated for a simple NULL assignment. If a tagged type is specified, specific encode and decode methods will be generated as well.

OBJECT IDENTIFIER

The ASN.1 OBJECT IDENTIFIER type is converted to a Java class that extends the *Asn1ObjectIdentifier* run-time class. This base class encapsulates the following public member variable:

```
public int[] value;
```

The number of subidentifiers to be encoded or that were decoded is specified in the built-in length component of the array object (i.e., value.length).

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

Generated Java class

```
public class <name> extends Asn1ObjectIdentifier {
    public <name> () {
        super();
    }
    public <name> (int[] value_) {
        super (value_);
    }
}
```

This shows the class generated for a simple OBJECT IDENTIFIER assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

RELATIVE-OID

The ASN.1 RELATIVE-OID type is converted to a Java class that extends the *Asn1RelativeOID* run-time class. This class extends the *Asn1ObjectIdentifier* class defined above. The storage of the relative OID value is the same as described for OBJECT IDENTIFIER. The only difference is the extended class defines different implementations of the encode/decode methods that apply the rules associated with the RELATIVE-OID type.

REAL

The ASN.1 REAL type is converted to a Java class that extends the *Asn1Real* run-time class.

The *Asn1Real* base class is used for standard ASN.1 REAL specifications or XSD float or double types. This class encapsulates the following public member variable:

```
public double value;
```

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

XSD Types

```
<xsd:float>, <xsd:double>
```

Generated Java class

```
public class <name> extends Asn1Real {
    public <name> () {
        super();
    }
    public <name> (double value_) {
        super (value_);
    }
}
```

This shows the class generated for a simple REAL assignment. If a tagged or constrained type is specified, specific encode and decode methods will be generated as well.

REAL (Base 10)

The ASN.1 Base 10 REAL type is converted to a Java class that extends the *Asn1Real10* run-time class. A base 10 real is specified in ASN.1 using a WITH COMPONENTS clause such as the following:

```
REAL(WITH COMPONENTS {
    ...,
    base (10)
})
```

It is also used for XSD decimal type specifications.

In this case, the real number is stored as a Java character string in the character string base class:

```
public String value;
```

ASN.1 Production:

```
<name> ::= REAL (WITH COMPONENTS { base(10) })
```

XSD Types

```
<xsd:decimal>
```

Generated Java class

```
public class <name> extends Asn1Real10 {
    public <name> () {
        super();
    }

    public <name> (String value_) {
        super (value_);
    }
}
```

SEQUENCE

The ASN.1 SEQUENCE type is converted to a Java class that extends the *Asn1Type* run-time base class. Public member variables are generated for each of the elements defined in the SEQUENCE. Each of these member variables represents an object reference since all of the ASN.1 types are mapped to Java objects.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

XSD Types

```
<xsd:sequence>, <xsd:all>
```

Generated Java class

```
public class <name> extends Asn1Type {
    public <type1> <element1-name>
    public <type2> <element2-name>
    ...

    public <name> () {
        super();
    }

    public <name> (<type1> <arg1>, <type2> <arg2>, ...) {
        super();
        <element1-name> = <arg1>;
```

```
        <element2-name> = <arg2>;
        ...
    }

    public <name> (<basetype1> <arg1>,
                   <basetype2> <arg2>, ...)
    {
        super();
        <element1-name> = new <type1> (<arg1>);
        <element2-name> = new <type2> (<arg2>);
        ...
    }

    private void init () { ... }
    public void decode () { ... }
    public int encode () { ... }
    public void print () { ... }
```

Note

1. The ... notation used in the ASN.1 definition above does not represent the ASN.1 extensibility notation. It is used to show a continuation of the sequence elements.

2. The <type1>, <type2>, etc. items denote the equivalent Java types generated from the ASN.1 <element-type1>, <element-type2>, etc. definitions.

3. The public and private methods that are generated are shown without arguments or function bodies for brevity.

The compiler first generates a public member variable for each of the elements defined in the SEQUENCE. The decision was made to make these variables public to make them easier to populate for encoding. The alternative was to use protected or private variables with get/set methods for setting or examining the values. It was felt that this approach would be too cumbersome for setting values in deeply nested constructed types.

A default constructor is then generated followed by overloaded constructors for setting the element values. The first form is simply a direct mapping of each of the element types to a constructor argument. The second form only contains arguments for the required types in the SEQUENCE (i.e. OPTIONAL and DEFAULT elements are omitted). The third form uses the base type of each of the elements as the type for each argument. This makes it possible to construct a SEQUENCE or SET using literal variables instead of always having to create an object. Finally, another variant of this constructor with primitive types is generated for required elements only. It is possible that you will not see all of these variations in a given generated class. It depends on a) whether or not the SEQUENCE or SET contains optional items and b) whether or not it contains primitive data items.

For example, the following shows how a variable of a generated class containing two IA5String elements could be constructed:

```
    v1 = new HelloWorld ("hello", "world");
```

Without this second form of constructor, the following would need to be done:

```
    v1 = new HelloWorld (new Asn1IA5String("hello"),
                         new Asn1IA5String("world"));
```

Also note that since all member variables are public, it is not necessary to use any of the argument-based constructors at all. A variable can be created using the default constructor and each of the elements populated directly.

Creation of Temporary Types

Temporary types are created when a SEQUENCE (or any other constructed type) definition contains other embedded constructed types. An example of this is 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 names of the temporary types are of the form <name>_<element-name1>_<element-name2>_ ... <element-nameN>. Using this algorithm, the ASN.1 type defined above would be reduced to the following equivalent ASN.1 types:

```
A-x ::= SEQUENCE {
    a1 INTEGER,
    a2 BOOLEAN
}

A ::= SEQUENCE {
    x A-x,
    y OCTET STRING SIZE (10)
}
```

The mapping of the ASN.1 types to Java classes would then be done.

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 the 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.

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.

Optional elements are accounted for in the Java version of the compiler by simply using null object references to denote the absence of an element. Remember that even the simplest primitive ASN.1 type definitions are wrapped in a Java class definition. Therefore an object must be created for any type defined as an element within a SEQUENCE.

To populate a SEQUENCE object for encoding that contains optional elements, the special constructor(s) for required elements only can be used. The default constructor also can be used followed by the manual creation and setting of the individual element values. The default constructor will initialize all element object references to null, so only the items to be encoded need be populated.

DEFAULT keyword

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

For DER or PER, it is a requirement that no value be present in the encoding for the default value. For integer and boolean default values, the compiler automatically generates code to handle this requirement based on the value in the structure. For other values, the default value is handled the same as an optional element (i.e., a null object reference indicates that nothing should be transmitted). The programmer must set the element object reference to null 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 a null object reference. If this is the case, the default value specified in the ASN.1 specification is used.

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 *ASN1OpenExt* 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.

XSD <xsd:all> Type Mapping

As per the X.694 standard, the XSD all type is mapped to an ASN.1 SEQUENCE type with a special element add named *order*. This is added as a special element to the generated Java class with the name *_order*. This contains an index entry for each element that identifies the order to elements are to be serilaized in when encoded in XML. By default, the array is initialized to encode the elements in the same order as specified in the type. When an XML document of this type is decoded, the order in which the elements are received in recorded in this array. If the data is serialized out in binary form (BER or PER) the array is included in the encoding. If is only transparent in XML encode/decode operations to mimic the behavior of its handling in XSD.

An example of how this is used might be a gateway application that read XML data and then translated to binary form for transmission over a low bandwidth network. When received on the other end, the receiving application would

transcode back from binary to XML. Suppose the item being transmitted was described using an xsd:all type that had three elements: a, b, and c. When the original XML document was received by the sending application, suppose the elements were received in the order c, b, a. The order array would record this fact and it would be included in the binary serialization. When the receiver decoded the message on the other end, the order information would be available along with the element data. The receiver could then reconstruct the XML document with the items in the same order as received.

SET

The ASN.1 SET type is converted into a Java class 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 to a Java class that extends the *Asn1Type* run-time base class. An array public member variable named *elements* is generated to hold the elements of the defined type.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

XSD Types

Elements or content group definitions containing the minOccurs and/or maxOccurs facets. Also, <xsd:list> types use this model.

Generated Java class

```
public class <name> extends Asn1Type {
   public <type>[] elements

   public <type> () {
      elements = null;
   }

   public <type> (int numRecords) {
      elements = new <type> [numRecords];
   }

   public void decode () { ... }
   public int encode () { ... }
   public void print () { ... }
}
```

The compiler first generates a public member variable to hold the SEQUENCE OF elements. The decision was made to make the variable public to make it easier to populate for encoding. The alternative was to use protected or

private variables with get/set methods for setting or examining the values. It was felt that this approach would be too cumbersome for setting values in deeply nested constructed types.

Two constructors are generated: a default constructor and a constructor that takes a number of *elements* argument. The default constructor will set the elements variable to null. The second constructor will allocate space for the given number of elements. The recommended way to populate a variable of this type for encoding is to use the second form of the constructor to allocate the required number of elements and then directly set the element object values. For example, to populate the following construct:

```
IntSeq ::= SEQUENCE OF INTEGER
```

with 3 integers, the following code could be used:

```
IntSeq intSeq = new IntSeq (3);
intSeq.elements[0] = new Asn1Integer (1);
intSeq.elements[1] = new Asn1Integer (2);
intSeq.elements[2] = new Asn1Integer (3);
```

Note that each of the integer element values is wrapped in an *Asn1Integer* wrapper 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 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 Java classes 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 Java class that is identical to that for SEQUENCE OF as described in the previous section.

CHOICE

The ASN.1 CHOICE type is converted to a Java class that extends the *Asn1Choice* run-time base class. This base class contains protected member variables to hold the choice element object and a selector value to specify which item in the CHOICE was chosen. Methods are generated to get and set the base class members.

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

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

```
        <element2-name> <element2-type>,
        ...
    }
```

XSD Types

```
<xsd:choice>, <xsd:union>
```

Generated Java class

```
public class <name> extends Asn1Choice {
    public final static byte _<ELEMENT1-NAME> 1
    public final static byte _<ELEMENT2-NAME> 2
    ...

    public <name> () { super(); }

    public String getElemName() { ... }

    public void set_<element1-name> () { ... }
    public void set_<element2-name> () { ... }
    ...
    public void decode () { ... }
    public int encode () { ... }
    public void print () { ... }
}
```

Note

1. The ... notation used in the ASN.1 definition above does not represent the ASN.1 extensibility notation. It is used to show a continuation of the sequence elements.

2. The public and private methods that are generated are shown without arguments or function bodies for brevity.

The compiler generates sequential identification constants for each of the defined elements in the CHOICE construct. The format used is the element names converted to all uppercase characters and preceded by an underscore. The constants represent the values returned by the base class *getChoiceID* method can therefore be used to determine what type of choice element was received in a decode operation.

The *getElemName* method is generated by the compiler and returns the name of the selected element.

A series of *set_<element>* methods are generated for setting the element value. In these declarations, <element> would be replaced with the actual element names. This is the only way an element value can be set for encoding; these methods ensure a consistent setting of both the element identifier and object reference values.

To access the value of a generated CHOICE object, the *getChoiceID* and *getElement* methods within the base class are used. This is generally done with an if or switch statement as follows:

```
Asn1BMPString element;
if (aliasAddress.getChoiceID() == AliasAddress._H323_ID) {
    element = (Asn1BMPString) aliasAddress.getElement();
}
```

In this case, *getChoiceID* is invoked and the result tested to see if the expected value was received. If it was, the element is assigned using *getElement* with a cast operation.

Creation of Temporary Types

The rules for the generation of CHOICE temporary type variables are the same as they were for SEQUENCE and SET variables. Complex nested types are pulled out of the definitions and used to create additional types to reduce the nesting levels. An example of this is as follows:

```
TestChoice ::= CHOICE {
    a INTEGER,
    b BOOLEAN,
    c SEQUENCE { aa IA5String, bb NULL }
}
```

This would be reduced to the following equivalent ASN.1 productions:

```
TestChoice-c ::= SEQUENCE { aa IA5String, bb NULL }

TestChoice ::= CHOICE {
    a INTEGER,
    b BOOLEAN,
    c TestChoice-c
}
```

In this case, the embedded constructed element for option c was pulled out to form the *TestChoice-c* production and then this new production is referenced in the original definition.

Populating Generated Choice Structures for Encoding

The only way a CHOICE construct can be populated for encoding is by using one the generated *set_<element>* methods. It is necessary to do it this way because the base class contains two protected member variables (*choiceID* and *element*) that must be set consistently. This is the only instance of a mapped type where the mapped element values do not have public access.

The following demonstrates setting a variable of the TestChoice structure defined above to use the first option:

```
TestChoice testChoice = new TestChoice ();
testChoice.set_a (new Asn1Integer (222));
```

Accessing the Choice Element Value after Decoding

To access the element in a choice construct after decoding, the following two methods can be used (both are defined in the Asn1Choice base class):

1. *getChoiceID* – this returns an identifier equal to one the generated choice identifier constants, and

2. *getElement* – this returns a reference to the decoded element object. It is of type Asn1Type but it can be upcast to the correct element type using information from the getChoiceID call.

In addition, the compiler generates a *getElemName* method that can be used to get the textual name of the decoded element.

XSD <xsd:union> Type Mapping

The <xsd:union> type is handled in a similar fashion to a choice type. The main difference is that the items in a union are not tagged. As per X.694, special element names are generated for these items for use in an ASN.1 CHOICE type.

These names are based on the base name *alt* and progress with sequential digits added for each addional union item (*alt-1*, *alt- 2*, etc.). XML decoding is accomplished by attempting to decode the content of each alternative in the union and setting the value to the first alternative that can be decoded successfully.

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 Java class that extends the *Asn1OpenType* class. This class in turn extends the *Asn1OctetString* class and provides the following public member variable for storing the encoded message component:

```
public byte[] value;
```

The number of octets to be encoded or that were decoded is specified in the built-in length component of the array object (i.e., value.length).

The following shows the basic mapping from ASN.1 type to Java class definition:

ASN.1 Production

```
<name> ::= <openType>
```

Generated Java class

```
public class <name> extends Asn1OpenType {
    public <name> () {
        super();
    }

    public <name> (byte[] data) {
        super (data);
    }

    public <name> (byte[] data,
                     int offset,
                     int nbytes)
    {
        super (data, offset, nbytes);
    }

    public <name> (Asn1EncodeBuffer buffer) {
        super ();
    }
}
```

The <openType> placeholder is to be replaced with any type of open type specification. It could be the ANY or ANY DEFINED BY keywords from the X.208 specification or an open type from X.681 (for example, TYPEIDENTIFIER.& Type).

The last form of the constructor shown above is for an optimized form of Open Type encoding. When encoding is done using BER, an open type header can be directly added to the beginning of an encoded message component. By using this form of the constructor, you are indicating to the run-time encoder that the encoded message component

onto which a header is to be added is already present in the message buffer. The advantage is that binary copies of the encoded message components are avoided both from the encode buffer to the open type object and from the open type object back to the encode buffer.

For XER, a new class derived from the *Asn1OpenType* class was created. This is the *Asn1XerOpenType* class and this must be used whenever an open type is required for XER. The reason for creating a special derived class is because of dependencies on XML parser classes defined within this class. If these were added directly to the *Asn1OpenType* class, a user would need to always have XML parser .jar files included in their classpath – even if working with BER, DER, or PER only.

If the –tables command line option is selected and the ASN.1 type definition references a table constraint, the code generated is different. In this case, *Asn1OpenType* above is replaced with *Asn1Type*. This the base class for all ASN.1 types. This allows a value of any ASN.1 type to be specified. On the encoding side, a user can assign an object of any ASN.1 type to this variable and the encoding routine will call the appropriate encoder according to the table index value. If the variable type is not present in the table and the Object Set is extensible, than it can be encoded as an open type. Otherwise an exception will be thrown. On the decoding side, the appropriate variable type is populated from the table based on the decoded index parameters. The user can determine the variable type from the table index value. If the variable type is not present in table, then it will be decoded as an open type if the Object Set is extensible; otherwise and exception will be thrown.

<xsd:any> Handling

The XSD any wildcard item is similar to an ASN.1 open type in semantics in that it allows any valid content to be present in that position in an XML document. However, an ASN.1 open type is not used to model an <xsd:any>. Instead, a character string variable is used. This stores the full XML text of the field in native XML form (i.e. angle brackets and the like are not escaped). Note that the XML text is not converted to different form when using binary encoding rules - it is maintained as XML text.

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.java (or *Asn1XerExternal.java* for XER). An object created from the resulting Java class is populated just like any other compiler-generated structure for working with ASN.1 data.

EmbeddedPDV Type

The ASN.1 EMBEDDED PDV type is a useful type used to include non-ASN.1 or other data within an ASN.1 encoded message. It was introduced in 1994 to replace EXTERNAL by removing unneeded fields and adding a few new ones to hold information that was missing. This type is described using the following ASN.1 SEQUENCE:

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

The ASN.1 compiler is used to create a meta-definition for this structure. The definition is stored in the file *Asn1EmbeddedPDV.java* (or *Asn1XerEmbeddedPDV.java* for XER). An object created from the resulting Java class is populated just like any other compiler-generated structure for working with ASN.1 data.

Parameterized Types

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

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

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

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

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

The compiler would handle this in the same way as if the original type was declared to be an octet string of size 1 to 32. In the case of Java, this would result in size constraint checks being added to the generated encode and decode methods for the type.

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.

ASN1C performs the substitution to create the proper Java class definition for SignedName:

```
public class SignedName extends Asn1Type {
  public Name toBeSigned;
  public Asn1ObjectIdentifier algorithmOID;
  public Params paramS;
  public Asn1BitString signature;
  ...
}
```

When processing parameterized type definitions, ASN1C 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 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 *Information Objects* section 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 generated *ProtocolID-Field* class.

Value Specifications

The ASN1C compiler can parse any type of ASN.1 value specification, however, the basic version will only generate code for the following types of value specifications:

• BOOLEAN

• INTEGER

• ENUMERATED

• Binary String

- Hexadecimal String

- Character String

- OBJECT IDENTIFER

The Pro version of the compiler will generate code for the following remaining types of value specifications:

- Enumerated

- Real

- Sequence

- Set

- Sequence Of

- Set Of

- Choice

If any of the above types of value specifications are detected in an ASN.1 module, the compiler will generate a Java source file with a special class to hold the values. The name of the source file and class is of the following format:

```
_<ModuleName>Values
```

In this definition, <ModuleName> would be replaced with the name of the ASN.1 module in which the values are defined.

The following sections provide details on the Java constants generated for the various types of ASN.1 value specifications.

INTEGER Value Specification

An INTEGER value specification causes a Java integer constant to be generated.

ASN.1 production:

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

Generated Java constant:

```
public static final long <name> = <value>;
```

BOOLEAN Value Specification

A BOOLEAN value specification causes a Java boolean constant to be generated.

ASN.1 production:

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

Generated Java constant:

```
public static final boolean <name> = <value>;
```

Binary String Value Specification

This value specification causes two Java constants to be generated: a 'numbits' constant specifying the number of bits in the string and a 'data' constant that hold the actual bit values.

ASN.1 production:

```
<name> BIT STRING ::= 'bbbbbbb'B
```

Generated Java constants:

```
public static final int <name>_numbits = <numbits>;
public static final byte[] <name>_data = { 0xhh, 0xhh, ... };
```

In the ASN.1 production definition, the lowercase 'b's above represent binary digits (1's or 0's). The generated code contains a *numbits* constant set to the number of bits (binary digits) in the string. The *data* constant specifies the binary data using hexadecimal byte values.

Hexadecimal String Value Specification

This value specification causes a Java constant to be generated containing a byte array of the hexadecimal byte values.

ASN.1 production:

```
<name> OCTET STRING ::= 'hhhhhh'H
```

Generated Java constants:

```
public static final byte[] <name> = { 0xhh, 0xhh, ... };
```

In the ASN.1 production definition, the lowercase 'h's above represent hexadecimal digits (0-9, a-f, or A-F). The generated constant specifies the binary data using hexadecimal byte values.

Character String Value Specification

A character string declaration causes a Java String constant to be generated.

ASN.1 production:

```
<name> <StringType> ::= 'ccccccc'
```

Generated Java constants:

```
public static final String <name> = "ccccccc";
```

In the ASN.1 production definition, <StringType> would be replaced with one of the ASN.1 character string types (for example, IA5String). The lowercase 'c's represent string characters. The generated constant is simply the string in Java form.

Object Identifier Value Specification

An object identifier value specification causes a Java integer array to be generated containing the subidentifier values.

ASN.1 production:

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

Generated Java constants:

```
public static final int[] <name> = { id1, id2, ..., idn };
```

For example, consider the following declaration:

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

This would result in the following Java constant being generated:

```
public static final int[] oid = { 0, 5, 10 };
```

ENUMERATED Value Specification

An ENUMERATED value specification causes a Java integer constant to be generated.

ASN.1 production:

```
<name> <enumtype> ::= <enumitem>
```

Generated Java constants:

```
public static final int <name> = <enumvalue>;
```

enumvalue will be the sequential integer value corresponding to the *enumitem* in *enumtype*.

REAL Value Specification

A REAL value specification causes a Java double constant to be generated.

ASN.1 production:

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

Generated Java constants:

```
public static final double <name> = <value>;
```

SEQUENCE Value Specification

A SEQUENCE value specification causes a final static instance of the Java class generated for the SEQUENCE to be generated.

ASN.1 production:

```
<name> <SequenceType> ::= <value>
```

Generated Java constants:

```
public static final <SequenceType> <name> =
   new <SequenceType> ( new <Elem1Type> (<elem1value>),
                        new <Elem2Type> (<elem2value>),
                        ... );
```

For example, consider the following declaration:

```
SeqType ::= SEQUENCE {
    oid OBJECT IDENTIFIER,
    id INTEGER
}

value SeqType ::= { oid { 0 1 1 }, id 12 }
```

This would result in the following Java constant being generated for value:

```
public static final SeqType value = new SeqType (
    new Asn1ObjectIdentifier( new int[]{0, 1, 1}),
    new Asn1Integer(12)
);
```

SET Value Specification

The value code generation for the ASN.1 SET type is that same as that for SEQUENCE described above.

SEQUENCE OF Value Specification

A SEQUENCE OF value specification causes a Java array constant to be generated.

ASN.1 production:

```
<name> <SequenceOfType> ::= <value>
```

Generated Java constants:

```
public static final <SequenceOfType> <name> =
    new <SequenceOfType>[] {
    new <ElemType> (<elem1value>),
    new <ElemType> (<elem2value>),
    ... };
```

For example, consider the following declaration:

```
SeqOfType ::= SEQUENCE OF INTEGER

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

This would result in the following Java constant being generated for value:

```
public static final SeqOfType value = new SeqOfType[] {
    new Asn1Integer(1),
    new Asn1Integer(2)
};
```

SET OF Value Specification

The value code generation for the ASN.1 SET OF type is that same as that for SEQUENCE OF described above.

CHOICE Value Specification

A CHOICE value specification causes a final static instance of the Java class generated for the CHOICE to be generated.

ASN.1 production:

```
<name> <ChoiceType> ::= elemname : <elemvalue>
```

Generated Java constants:

```
public static final <ChoiceType> <name> =
   new <ChoiceType> (<ElemCode>,
                     new <ElemType> (<elemvalue>));
```

For example, consider the following declaration:

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

value ChoiceType ::= id: 1
```

This would result in the following Java constant being generated:

```
public static final ChoiceType value =
   new ChoiceType (ChoiceType._ID, new Asn1Integer(1));
```

Chapter 6. Generated BER/DER/CER Encode Methods

Two different types of BER (Basic Encoding Rules) encode methods may be generated using the ASN1C compiler:

- Memory-buffer based definite length backward encoders

- Stream-based indefinite length forward encoders

For DER (Distinguished Encoding Rules), only the first option is available because a requirement of DER is that all lengths must be in definite form. For CER (Canonical Encoding Rules), only the second option is available because all constructed element lengths must be in indefinite length form. Each of these methods are described in the following sections.

Memory-buffer Based Definite Length Encoders

For each ASN.1 production defined in an ASN.1 source file, a Java encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the encoding of the base class method. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types. These default implementations include support for adding the universal tags associated with the types as defined in the X.680 standard.

So for simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method:

```
X ::= INTEGER
```

In this case, the generated Java class extends the *Asn1Integer* base class and the default encode method within this class is sufficient to encode a value of the generated type.

However, if the type is altered to contain a tag or constraint, then a custom encode method would be generated:

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

In this case, special logic is necessary to apply the tag value.

Some types will always cause encode methods to be generated. At the primitive level, this is true for the ENUMERATED type. This type will always contain a custom set of enumerated values. All constructed types (SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE) will cause encode methods to be added to the generated classes.

Generated Java Method Format and Calling Parameters

The signature for a Java BER encode method is as follows:

```
public int encode (Asn1BerEncodeBuffer buffer, boolean explicit)
```

```
throws Asn1Exception
```

The *buffer* argument is a reference of an *Asn1BerEncodeBuffer* object that describes the buffer into which a message is being encoded. This must be created and initialized before calling any encode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used.

The return value is the length in octets of the encoded message component. Unlike the C/C++ version, a negative value is never returned to indicate an encoding failure. That is handled by the exception mechanism. All ASN1C Java exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown.

Populating Generated Variables for Encoding

Populating generated variables for encoding can be done in most cases either through the object constructors or directly by assigning an object reference to a public member variable.

Constructors are provided for most generated types to allow direct population of the encapsulated member variable(s) on initialization. The exception is the classes generated for SEQUENCE OF or SET OF. These only allow the size of an array to be specified – population of the array elements must be done manually.

All of the base run-time classes except *Asn1Null* contain public member variables. In practically all cases there is a single variable called *value* that is of the base type that needs to be populated. For example, the *Asn1Integer* base class contains the following item:

```
public long value;
```

Therefore, population of any class variable derived from INTEGER can be done by adding.*value* to the end of the lefthand side of the assignment and an integer value on the right. So for the following assignment:

```
X ::= INTEGER
```

A variable of the type can either be populated using the constructor with the following statement:

```
X x = new X (25);
```

or via direct access of the member variable as follows:

```
X x = new X ();
x.value = 25;
```

The only primitive type that does not have a single member called *value* to represent its value is BIT STRING. In this case, the *Asn1BitString* class contains a second variable called *numbits* to specify the number of bits in the string.

Procedure for Calling Java BER Encode Methods

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode message buffer object into which the value will be encoded.

2. Invoke the encode method.

3. Invoke encode message buffer methods to access the encoded message component.

The first step is the creation of an encode message buffer object. Unlike the C/C++ version of the product, there is no choice to be made between a static or dynamic encode buffer. In Java, everything is dynamic. There are two forms of

the constructor: a default constructor and one that allows specification of a message buffer size increment. The size increment will determine how often the buffer will need to be resized to hold large messages. If you know that you will be encoding large messages, then this object should be constructed with a large value for the increment. If you know that you will be encoding small messages in a constrained environment, then this value can be set very low. The default constructor sets the value to a reasonable mid-range value (see SIZE_INCREMENT in *Asn1EncodeBuffer.java*, as of this writing the value was set to 1024).

The second step is the invocation of the encode method. The calling arguments were described earlier. As per the Java standard, this method must be invoked from within a try/catch block to catch the possible *Asn1Exception* that may be thrown. Alternatively, the method from which the encode method is called can declare that it throws an Asn1Exception leaving it to be dealt with at a higher level.

Finally, encode buffer methods can be called to access the encoded message component. The encode method itself returns the length of the component, so this item is already known (however, there is a *getMsgLength* method available if you want to access this length from a different location). Unlike C or C++, a pointer to where the message starts in the encode buffer cannot be returned (recall that BER encoding is done from back to front, so a message rarely starts at the beginning of a buffer). However, the Java API provides an object called a *ByteArrayInputStream* that provides a way to look at the encoded component as a stream. The encode buffer object therefore provides a method called *getByteArrayInputStream* which is the preferred way to access the encoded component.

In addition to *getByteArrayInputStream* there is a *getMsgCopy* function that will retrieve a copy of the generated message into a byte array object. This is somewhat slower because a copy needs to be done. The encode buffer class also contains other methods for operating directly on the encoded component (for example, the *write* method can be used to write it to a file or other medium). And of course, one could derive their own special encode buffer class from this class to add more functionality. See the description of the *Asn1BerEncodeBuffer* class in the run-time section for a full description of the available methods.

A complete example showing how to invoke an encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

// Step 1: Create a message buffer object. This object uses the
// default size increment for buffer expansion..

Asn1BerEncodeBuffer encodeBuffer = new Asn1BerEncodeBuffer();

// Step 2: Invoke the encode method. Note that it must be done
// from within a try/catch block..

try {
    personnelRecord.encode (encodeBuffer, true);

    if (trace) {
        System.out.println ("Encoding was successful");
        System.out.println ("Hex dump of encoded record:");
        encodeBuffer.hexDump ();
        System.out.println ("Binary dump:");
        encodeBuffer.binDump ();
    }

    // Step 3: Access the encoded message component. In this
    // case, we use methods in the class to write the component
    // to a file and output a formatted dump to the message.dmp
    // file..
```

```
        // Write the encoded record to a file

        encodeBuffer.write (new FileOutputStream (filename));

        // Generate a dump file for comparisons

        encodeBuffer.hexDump
            (new PrintStream (new FileOutputStream ("message.dmp")));
    }
    catch (Exception e) {
        System.out.println (e.getMessage());
        e.printStackTrace();
        return;
    }
```

Reuse of Java Encoding Objects

The simple example above showed the procedure to encode a single record. But what if you had to decode a series of the same record over and over again? This is a common occurrence in a BER encoding application.

You would not want to recreate the data holder and message buffer objects on each pass of the loop. This would have an adverse effect on the performance of the application. What you would want to do is only create the objects a single time and then reuse them to encode each message instance.

It turns out that this is an easy thing to do. The public member variable access to the data holder object makes it easy to change the variables on each given pass. And the encode buffer object contains a reset method for resetting the encode buffer for subsequent encodings. The use of this method has the advantage of not releasing any of the memory that had been accumulated to this point for previous encodings.

To show an example of object reuse, suppose we were going to encode a series of names. The ASN.1 type for the names would be as follows:

```
Name ::= [APPLICATION 1] IMPLICIT SEQUENCE {
    givenNameIA5String,
    initial IA5String,
    familyNameIA5String
}
```

The generated Java class would contain public member variables for each of the string objects:

```
public Asn1IA5String givenName;
public Asn1IA5String initial;
public Asn1IA5String familyName;
```

The most efficient way to repopulate these variables within a loop would be simply to assign each of the new strings to be encoded directly to the public value member variables contained within the *Asn1IA5String* objects (i.e., the *Name* or *Asn1IA5String* objects should not be reconstructed each time).

A code snippet showing how this could be done is as follows:

```
// Step 1: Create Name and Asn1BerEncodeBuffer objects for use in
// the loop..

Name name = new Name ("", "", ""); // creates empty string objects
Asn1BerEncodeBuffer encodeBuffer = new Asn1BerEncodeBuffer ();
```

```
for (;;) {

    // logic here to read name components from a DB or other medium

    ...

    // populate string variables (assume string1, 2, and 3 are string
    // variables read from DB above)..

    name.givenName.value = string1;
    name.initial.value = string2;
    name.familyName.value = string3;

    // encode

    try {
        len = name.encode (encodeBuffer, true);

        // do something with the encoded message component

        ...

        // reset encode buffer for next pass

        encodeBuffer.reset ();
    }
    catch (Asn1Exception e) {
        // handle error ..
    }
}
```

Stream-Oriented Indefinite Length Encode Methods

BER messages can be encoded directly to an output stream such as a file, network or memory stream. The ASN1C compiler has the –**stream** option to generate encode functions of this type. For each ASN.1 production defined in the ASN.1 source file, a Java encode method may be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

The basic principles of the generation of the encode methods are the same as for ordinary BER/DER encode methods. Stream-oriented BER encoding starts from the beginning of the message until the message is complete. This is sometimes referred to as "forward encoding". This differs from regular BER where encoding that is done from back-tofront. Indefinite lengths are used for all constructed elements in the message. Also, there is no permanent buffer for stream-oriented encoding, all octets are written directly to the output stream.

Generated Java Method Format and Calling Parameters

The signature for a Java BER stream-oriented encode method is as follows:

```
public void encode (Asn1BerOutputStream out, boolean explicit)
    throws Asn1Exception, java.io.IOException
```

The out argument is a reference of an *Asn1BerOutputStream* object that describes the output stream into which a message is being encoded. This must be created and initialized before calling any encode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used

The *explicit* argument specifies whether or not an explicit tag should be applied to the encoded contents. The average user will almost always want to set this argument to true. The only time it would not be set to true is if a user wanted to just encode a contents field with no tag. This argument is used primarily by the compiler when generating internal calls to properly handle implicit and explicit tagging.

Unlike the C/C++ version, a negative value is never returned form encode methods to indicate an encoding failure. That is handled by the exception mechanism. All ASN1C Java exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown. If I/O error occurs then the *java.io.IOException* is thrown.

Procedure for Calling Java BER Stream-Oriented Encode Methods

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an output stream object into which the value will be encoded

2. Invoke the encode method

3. Close the output stream.

The first step is the creation of an output stream object. There are two forms of the constructor: a constructor with one parameter (*OutputStream* reference) and one that allows specification of an internal buffer size. A larger internal buffer size generally provides better performance at the expense of increased memory consumption. The first constructor sets the value to a reasonable mid-range value.

The second step is the invocation of the encode method. The calling arguments were described earlier. As per the Java standard, this method must be invoked from within a try/catch block to catch the possible *Asn1Exception* and *java.io.IOException*, which may be thrown. Alternatively, the method from which the encode method is called can declare that it throws *Asn1Exception* and java.io.IOException leaving it to be dealt with at a higher level.

Finally, close the output stream.

A complete example showing how to invoke a stream-based encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

Asn1BerOutputStream out = null;

try {
   // Step 1: Create an output stream object. This object uses the
   // default size increment for buffer expansion..

   out = new Asn1BerOutputStream
      (new FileOutputStream (filename));

   // Step 2: Invoke the encode method. Note that it must be done
   // from within a try/catch block..

   personnelRecord.encode (out, true);
```

```
    if (trace) {
        System.out.println ("Encoding was successful");
        System.out.println ("Hex dump of encoded record:");
        encodeBuffer.hexDump ();
        System.out.println ("Binary dump:");
        encodeBuffer.binDump ();
    }
}
catch (Exception e) {
    System.out.println (e.getMessage());
    e.printStackTrace();
    return;
}
finally {

    // Step 3: Close the output stream, if opened

    try {
        if (out != null)
            out.close ();
    }
    catch (Exception e) {}
}
```

If you compare this example with the BER encoding example in Figure 2, you will see the encoding procedure is almost identical. This makes it very easy to switch encoding methods should the need arise. All you need to do is change *Asn1BerEncodeBuffer* to *Asn1BerOutputStream* and remove the explicit code that writes the messages into the stream. Also closing of the stream should be added.

Chapter 7. Generated BER/DER/CER Decode Methods

For each ASN.1 production defined in the ASN.1 source file, a Java decode method *may* be generated. This method will decode an ASN.1 message into public member variables within the Java object.

As was the case for encode methods, a decode method is only generated if it is required to alter the default method in the base class. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types. These default implementations include support for handling the universal tags associated with the types as defined in the X.680 standard.

Generated Java Method Format and Calling Parameters

The signature for a Java BER decode method is as follows:

```
public void decode (Asn1BerDecodeBuffer buffer, boolean explicit,
                    int implicitLength)
    throws Asn1Exception, java.io.Exception
```

The *buffer* argument is a reference of an *Asn1BerDecodeBuffer* object that describes the message that is being decoded. This must be created and initialized before calling any decode method. See the description of this class in the *Java Run- Time Classes* section for details on how this class is used.

The *explicit* and *implicitLength* arguments specify whether or not an explicit tag should be parsed from the encoded contents. The average user will almost always want to set *explicit* to true and *implicitLength* to zero. The only time these arguments would not be set this way is if a user wanted to directly decode contents with no tag/length information. These arguments are used primarily by the compiler when generating internal calls to properly handle implicit and explicit tagging.

The decode method returns no result. Unlike the C/C++ version, a negative status value is not returned to indicate a failure. That is handled by the exception mechanism. All ASN1C Java exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various ASN.1 exceptions that can be thrown. The *java.io.Exception* that can be thrown is in the *read* method within the decode buffer base class. This method attempts to read data from an input stream using the methods in the java.io package.

Procedure for Calling Java BER Decode Methods

The general procedure to decode an ASN.1 BER message involves the following three steps:

1. Create a decode message buffer object to describe the message to be decoded

2. Invoke the decode method

3. Process the decoded data values

The first step is the creation of a decode message buffer object. The *Asn1BerDecodeBuffer* object contains constructors that can either accept a message as a byte array or as an I/O input stream. The input stream option makes it possible

to decode messages directly from other mediums other than a memory buffer (for example, a message can be decoded directly from a file).

The *Asn1BerDecodeBuffer* object contains a method called peekTag that can be used to determine the outer-level tag on a message. This can be used to determine the type of message received in applications that must deal with multiple message types.

The generated decode method can then be invoked to decode the message. The calling arguments were described earlier. As per the Java standard, this method must be invoked from within a try/catch block to catch the possible exceptions that may be thrown. Alternatively, the method from which the decode method is called can declare that it throws the exceptions leaving them to be dealt with at a higher level.

The final step is to process the data. All data is contained within public member variables so access is quite easy. And of course Java has the distinct advantage of not requiring any clean-up once you are done with the data. The garbage collector will collect the unused memory when it is no longer referenced.

A complete example showing how to invoke a decode method is as follows:

```
try {

    // Step 1: create a decode message buffer object to describe the
    // message to be decoded. This example will use a file input
    // stream to decode a message directly from a binary file..

    // Create an input file stream object

    FileInputStream in = new FileInputStream (filename);

    // Create a decode buffer object

    Asn1BerDecodeBuffer decodeBuffer = new Asn1BerDecodeBuffer (in);

    // Step 2: create an object of the generated type and invoke the
    // decode method..

    PersonnelRecord personnelRecord = new PersonnelRecord ();
    personnelRecord.decode (decodeBuffer);

    // Step 3: process the data

    if (trace) {
        System.out.println ("Decode was successful");
        personnelRecord.print (System.out, "personnelRecord", 0);
    }
}
catch (Exception e) {
    System.out.println (e.getMessage());
    e.printStackTrace();
    return;
}
```

Reuse of Java Decoding Objects

The sample above showed the BER decoding of a single message. In a typical application, a loop would be involved to decode a series of messages. While it would be possible to use the code shown above in a loop, it would not be

the most efficient way to decode the messages. Objects should be reused where possible to avoid the overhead of excessive memory allocations and garbage collection.

A single decode buffer object can be used to process a stream of messages. If the decode message buffer is created using an input stream object that contains a series of messages (for example, a file containing multiple records or a communications device), all that needs to be done is the continuous invocation of the BER decode method for the given message type.

Nothing special needs to be done to reuse the generated type object for decoding. The decoder will automatically all the internal *init()* method before decoding to make sure all items are reset to their starting state.

In the example above, all that would need to be done to decode a series of personnel records is the inclusion of a loop after the PersonnelRecord object was created in step 2:

```
for (;;) {
    personnelRecord.decode (decodeBuffer);

    if (trace) {
        System.out.println ("Decode was successful");
        personnelRecord.print (System.out, "personnelRecord", 0);
    }
}
```

Deferred Decoding and Partial Decoding

Another way to improve decoding performance of large messages is through the use of either *deferred decoding* or *partial decoding*. This allows for the selective decoding of only parts of a message in a single decode function call. This can significantly reduce decoding time because large parts of messages can be skipped.

Deferred decoding can be done on elements defined within a SEQUENCE, SET or CHOICE construct. It is done by designating an element to be an open type by using the *<isOpenType/>* configuration setting. This setting causes the ASN1C compiler to insert an *Asn1OpenType* placeholder in place of the type that would have normally been used for the element. The data in its original encoded form will be stored in the open type container when the message is decoded. The data within the open type container can be fully decoded later by using the normally-generated decode function generated by the ASN1C compiler. (This stands in contrast to C and C++ code generation, which requires a special decode function for this purpose.)

Partial decoding is similar to deferred decoding, except that where deferred decoding captures the encoded data for later decoding, partial decoding skips over it and discards the encoded data. Since the encoded data is not retained, partial decoding uses less memory. Partial decoding is done by designating an element to be skipped using the *<skip/>* configuration setting. This setting causes the ASN1C compiler to generate decoders that simply skip over that element.

In following example, decoding of the element *id* is deferred, while *details* is skipped:

```
Identifier ::= SEQUENCE {
    id INTEGER,
    oid OBJECT IDENTIFIER,
    details DetailedStuff
}
```

The following configuration file is required to indicate the element id is to be processed as an open type (i.e. that it will be decoded later) and that element details should be skipped:

```
<asn1config>
    <module>
```

```
        <name>modulename</name>
        <production>
            <name>Identifier</name>
            <element>
                <name>id</name>
                <isOpenType/>
            <element/>
            <element>
                <name>details</name>
                <skip/>
            <element/>
        <production/>
    <module/>
<asn1config/>
```

In the generated code, the element *id* type will be replaced with an open type (*Asn1OpenType*), and the type will be decoded as such. When the top-level decoding has finished, the element may be decoded by taking the open type value (*id.value*) and using it as a source for a new *Asn1BerDecodeBuffer*. The element can then be decoded by creating a new object for the element (in this case, a new instance of *Asn1ObjectIdentifier*) and calling its decode method. The element *details* will simply be skipped.

Chapter 8. Generated PER Encode Methods

The generation of methods to encode data in accordance with the Packed Encoding Rules (PER) is similar to how methods were generated in the BER/DER case discussed previously. For each ASN.1 production defined in the ASN.1 source file, a Java encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the encoding of the base class method. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method:

```
X ::= INTEGER
```

In this case, the generated Java class extends the *Asn1Integer* base class and the default encode method within this class is sufficient to encode a value of the generated type.

In the case of BER/DER, a custom encode method was generated if a) the type was tagged, or b) it contained a testable constraint. In the case of PER, only the latter condition will cause a custom method to be generated. The reason is because PER basically ignores the tags on tagged types and they therefore have no effect on the final decoded message component.

For example, the following declaration will cause a custom encode method to be generated because the value range constraint is a PER-visible that will alter the encoding:

```
X ::= INTEGER (1..255)
```

In this case, special logic is necessary to apply the value range constraint.

Some types will always cause encode methods to be generated. At the primitive level, this is true for the ENUMERATED type. This type will always contain a custom set of enumerated values. All constructed types (SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE) will cause encode methods to be added to the generated classes.

Generated Java Method Format and Calling Parameters

The signature for a Java PER encode method is as follows:

```
public void encode (Asn1PerEncodeBuffer buffer)
    throws Asn1Exception, java.io.IOException
```

The *buffer* argument is a reference of an *Asn1PerEncodeBuffer* object that describes the buffer into which a message is to be encoded. This must be created and initialized before calling any encode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used.

The PER encode methods do not return a value. This is different than the C/C++ version that returns a negative status value to indicate an encoding failure. For Java, errors are reported via the exception mechanism. All ASN1C Java exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown.

Procedure for Calling Java PER Encode Methods

The Java class variables corresponding to each of the ASN.1 types and method of population are the same as they were in the BER encoding case. See the section on BER encoding for instructions on how to populate the variables prior to encoding.

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode message buffer object into which the value will be encoded

2. Invoke the encode method

3. Invoke encode message buffer methods to access the encoded message component

The first step is the creation of an encode message buffer object. For PER encoding, this is an object of the *Asn1PerEncodeBuffer* class. The following constructors are available for creating a PER encode buffer object:

```
public Asn1PerEncodeBuffer (boolean aligned);

public Asn1PerEncodeBuffer (boolean aligned, int sizeIncrement);
```

The first argument indicates whether PER aligned or unaligned encoding should be done. The second form of the constructor contains a size increment argument. This argument will determine how often the buffer will need to be resized to hold large messages. If you know that you will be encoding large messages, then this object should be constructed with a large value for the increment. If you know that you will be encoding small messages in a constrained environment, then this value can be set very low. The default constructor sets the value to a reasonable mid-range value (see SIZE_INCREMENT in *Asn1EncodeBuffer.java*, as of this writing the value was set to 1024).

The second step is the invocation of the encode method. The calling arguments were described earlier. As per the Java standard, this method must be invoked from within a try/catch block to catch the possible exceptions that may be thrown. Alternatively, the method from which the encode method is called can declare that it throws an *Asn1Exception* leaving it to be dealt with at a higher level.

Finally, encode buffer methods can be called to access the encoded message component. The Java API provides an object called a *ByteArrayInputStream* that provides a way to look at the encoded component as a stream. The encode buffer object provides a method called *getInputStream* that returns a byte array input stream representing the message component. This is the preferred way to access the encoded component.

In addition to getInputStream there is a *getMsgCopy* function that will retrieve a copy of the generated message into a byte array object. This is somewhat slower because a copy needs to be done. Another option that is only available when doing PER encoding is the *getBuffer* method. This returns a reference to the actual message buffer into which the message was encoded. Since a PER message is encoded front-to-back (unlike the back-to-front used in BER/DER encoding), the buffer reference returned will point to the start of the encoded message. The *getMsgByteCnt* method can then be used to get the message length in bytes or the *getMsgBitCnt* method can be called to get the length in bits.

The encode buffer class also contains other methods for operating directly on the encoded component (for example, the *write* method can be used to write it to a file or other medium). And of course, one could derive their own special encode buffer class from this class to add more functionality. See the description of the *Asn1PerEncodeBuffer* class in the runtime section for a full description of the available methods.

A complete example showing how to invoke a PER encode method is as follows:

```
// Note: personnelRecord object was previously populated with data
```

```
    // Step 1: Create a message buffer object. This object uses the
    // default size increment for buffer expansion..

    Asn1PerEncodeBuffer encodeBuffer = new Asn1PerEncodeBuffer();

    // Step 2: Invoke the encode method. Note that it must be done
    // from within a try/catch block..

    try {
        personnelRecord.encode (encodeBuffer);

        if (trace) {
            System.out.println ("Encoding was successful");
            System.out.println ("Hex dump of encoded record:");
            encodeBuffer.hexDump ();
            System.out.println ("Binary dump:");
            encodeBuffer.binDump ("personnelRecord");
        }

        // Step 3: Access the encoded message component. In this
        // case, we use methods in the class to write the component
        // to a file and output a formatted dump to the message.dmp
        // file..

        // Write the encoded record to a file

        encodeBuffer.write (new FileOutputStream (filename));

        // Generate a dump file for comparisons

        encodeBuffer.hexDump
            (new PrintStream (new FileOutputStream ("message.dmp")));

        // We can also directly access the buffer as follows:

        byte[] buffer = encodeBuffer.getBuffer();
        int msglen = encodeBuffer.getMsgByteCnt();
    }
    catch (Exception e) {
        System.out.println (e.getMessage());
        e.printStackTrace();
        return;
    }
```

If you compare this example with the BER encoding example in Figure 2, you will see the encoding procedure is almost identical. This makes it very easy to switch encoding methods should the need arise. All you need to do is change *Asn1BerEncodeBuffer* to *Asn1PerEncodeBuffer* and remove the explicit argument from the encode method call.

Reuse of Java Encoding Objects

The concept of reusing Java objects for PER encoding is the same as was described previously for BER encoding. Basically, all that needs to be done is the creation of a single PER encode buffer object and an object corresponding

to the ASN.1 data type to be encoded outside of the processing loop. These objects can then be reused to encode each instance of the messages to be sent. After each message is encoded, the PER buffer must be reset for the next message by using the reset method. See the section on reuse of objects in the BER encoding section for a more thorough discussion and sample code on using this capability.

Chapter 9. Generated PER Decode Methods

For each ASN.1 production defined in the ASN.1 source file, a Java decode method *may* be generated. This method will decode an ASN.1 message into public member variables within the Java object.

As was the case for encode methods, a decode method is only generated if it is required to alter the default method in the base class. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For primitive types, a custom PER decode method is only generated if one or more of the following is true:

1. The type contains a PER-visible constraint

2. The generation of event handlers was specified

The exception to this rule is the ENUMERATED primitive type (or likewise, INTEGER type with a named number list) that will always cause a decode method to be generated.

Constructed types will always cause custom PER decode methods to be generated.

Generated Java Method Format and Calling Parameters

The signature for a Java PER decode method is as follows:

```
public void decode (Asn1PerDecodeBuffer buffer)
    throws Asn1Exception, java.io.Exception
```

The *buffer* argument is a reference of an *Asn1PerDecodeBuffer* object that describes the message that is being decoded. This must be created and initialized before calling any decode method. See the description of this class in the *Java Run- Time Classes* section for details on how this class is used.

The decode method returns no result. Unlike the C/C++ version, a negative status value is not returned to indicate a failure. That is handled by the exception mechanism. All ASN1C Java exceptions are derived from the Asn1Exception base class. See the section on exceptions for a complete list and description of the various ASN.1 exceptions that can be thrown. The *java.io.Exception* that can be thrown is in the *read* method within the decode buffer base class. This method attempts to read data from an input stream using the methods in the java.io package.

Procedure for Calling Java PER Decode Methods

The general procedure to decode an ASN.1 PER message involves the following three steps:

1. Create a decode message buffer object to describe the message to be decoded

2. Invoke the decode method

3. Process the decoded data values

The first step is the creation of a decode message buffer object. The *Asn1PerDecodeBuffer* object contains constructors that can either accept a message as a byte array or as an I/O input stream. The input stream option makes it possible

to decode messages directly from other mediums other than a memory buffer (for example, a message can be decoded directly from a file or a socket).

Unlike BER or DER, no mechanism exists in PER to peek at an outer level tag or identifier to identify the message type. This type must be known beforehand. Most protocols that employ PER have a specific outer level type know as a "Protocol Data Unit" (PDU) that encompasses all of the different message types that might be received. This is typically a CHOICE construct with each option representing a different type of message.

The generated decode method for the PDU is invoked to decode the message. The calling arguments were described earlier. As per the Java standard, this method must be invoked from within a try/catch block to catch the possible exceptions that may be thrown. Alternatively, the method from which the decode method is called can declare that it throws the exceptions leaving them to be dealt with at a higher level.

The final step is to process the data. All data is contained within public member variables so access is quite easy. All of the primitive data type classes contain a public member variable called *value* that contains decoded data. This can be accessed in nested structures by prefixing *value* with each of the element names from the top down. For example, the given name element in the Name type shown earlier would be accessed as follows: *name.givenName.value* (this assumes an instance of the Name class was created using the variable name *name*).

A complete example showing how to invoke a decode method is as follows:

```java
try {

    // Step 1: create a decode message buffer object to describe the
    // message to be decoded. This example will use a file input
    // stream to decode a message directly from a binary file..

    // Create an input file stream object

    FileInputStream in = new FileInputStream (filename);

    // Create a decode buffer object

    Asn1PerDecodeBuffer decodeBuffer = new Asn1PerDecodeBuffer (in);

    // Step 2: create an object of the generated type and invoke the
    // decode method..

    PersonnelRecord personnelRecord = new PersonnelRecord ();
    personnelRecord.decode (decodeBuffer);

    // Step 3: process the data

    if (trace) {
        System.out.println ("Decode was successful");
        personnelRecord.print (System.out, "personnelRecord", 0);
    }
}
catch (Exception e) {
    System.out.println (e.getMessage());
    e.printStackTrace();
    return;
}
```

Reuse of Java Decoding Objects

Java objects can be reused for decoding PER messages in the same way they were for BER messages. The decode buffer and message type objects are created outside of the main decoding loop. Then in the main loop these objects are reused to process each input message. Data must be saved from the message type object after each iteration because the contents of the object will be overwritten on each consecutive loop iteration. Nothing special needs to be done at the bottom of the loop to ready the decoder for the next message. All necessary initialization will be handled internally.

Chapter 10. Generated XML Methods

Overview

X.693 specifies XER ("XML Encoding Rules"). There are three variants of XER given: BASIC-XER (often just XER for short), canonical XER, and EXTENDED-XER. Into this mix, Objective Systems has added its own encoding rules which we'll call OSys-XER. OSys-XER is very similar to XER, but has a few variations that are meant to produce XML documents more closely aligned with what you might get if you were using XML Schema to specify your abstract syntax. Generally, OSys-XER produces fewer tags. The differences between these two sets of encoding rules are discussed in more detail below.

ASN1C supports BASIC-XER, canonical XER, and OSys-XER. It has for some time supported EXTENDED-XER via direct compilation of XSD. In version 6.5.0, we have begun to add direct support for EXTENDED-XER by adding support for some of the XER encoding instructions. Nonetheless, EXTENDED-XER is most fully support today via direct compilation of XSD. By compiling XSD, you can obtain behavior much the same as with OSys-XER, and more.

ASN1C has two runtimes for XML:

- The "XER" runtime. This is used for basic and canonical XER.

- The "XML" runtime. This is used for OSys-XER and for EXTENDED-XER (whether compiling XSD or compiling ASN.1 with XER encoding instructions).

Because these two runtimes are so similar, they are discussed in this chapter together. As you read this chapter, it is important to keep in mind when each of these runtimes is used so that you know which cases apply to you.

Note that you may use the -xsd switch when generating XML encoders and decoders. The XML schema produced from the ASN.1 specification using the -xsd switch can be used to validate the XML messages generated using the XML encode functions. Similarly, an XML instance can be validated using the generated XML schema prior to decoding.

Compiler Invocation	What is Generated
-xer flag is used to compile ASN.1 without XER encoding instructions	Generated code supports BASIC-XER and canonical XER -xsd produces schema that validates BASIC-XER encodings.
-xer or -xml flag is used to comple ASN.1 with supported XER encoding instructions (if any instructions are not supported, all instructions are ignored, and the above entry in this table applies)	Generated code supports EXTENDED-XER only. -xsd produces schema that validates EXTENDED-XER encodings. As of this writing, this is not fully supported.
-xml used to compile ASN.1	Generated code suppports OSys-XER encodings. -xsd produces schema that validates OSys-XER encodings.
-xml used to compile XSD	Generated code supports EXTENDED-XER only.

Differences between OSys-XER and XER (BASIC-XER)

OSys-XER differs from (BASIC-)XER in the following ways:

- Lists of numbers, enumerated tokens, and named bits are expressed in space-separated list form instead of as individually wrapped elements or value lists.

For example, the ASN.1 specification "A ::= SEQUENCE OF INTEGER" with value "{ 1 2 3 }" would produce the following encoding in XER:

<A><INTEGER>1</INTEGER><INTEGER>2</INTEGER><INTEGER>3</INTEGER></A>

in XML, it would be the following:

<A>1 2 3</A>

- The OSys-XER encoding for a SEQUENCE OF CHOICE wraps each repetition in an element. In XER, each occurrence of the CHOICE is an XML element and it is not further wrapped.

  For example, given `MyType ::= SEQUENCE OF CHOICE { a A, b B }`, XER might produce a sequence of <a> and <b> elements, while OSys-XER will produce a series of <CHOICE> elements, or, if the CHOICE type were named `MyChoice`, a series of <MyChoice> elements.

- The values of the BOOLEAN data type are expressed as the lower case words "true" or "false" with no delimiters. In XER, the values are <true/> and <false/>.

- Enumerated token values are expressed as the identifiers themselves instead of as empty XML elements (i.e. elements wrapped in '< />'). For example, a value of the ASN.1 type "Colors ::= ENUMERATED { red, blue, green }" equal to "red" would simply be "<color>red</color>" instead of "<color><red/></color>".

- The special REAL values <NOT-A-NUMBER/>, <PLUS-INFINITY/> and <MINUS-INFINITY/> are represented as NaN, INF and -INF, respectively.

- GeneralizedTime and UTCTime values are transformed into the XSD representation for dateTime (YYYY-MMDDTHH: MM:SS[.SSSS][(Z|(+|-)HH:MM)]) when encoded to XML. When an XML document is decoded, the time format is transformed into the ASN.1 format.

- When encoding/decoding a type as the root element of an XML document, OSys-XER will typically give the root element name a lowercase first letter. By contrast, XER uses the NonParameterizedTypeName, which will have an uppercase first letter, for the root element.

EXTENDED-XER

EXTENDED-XER (specified in X.693) allows you to vary the XML encoding of ASN.1 by using XER encoding instructions. ASN1C supports EXTENDED-XER in two different ways: by compiling XSD and by compiling ASN.1 with XER encoding instructions. Support for XER encoding instructions in ASN.1 is limited.

This section relates to our support for XER encoding instructions. If some features you need are not supported, you might consider using direct compilation of XSD.

How to Generate Code for EXTENDED-XER

If your ASN.1 contains XER encoding instructions, ASN1C will automatically generate code for EXTENDED-XER instead of BASIC-XER. This is true whether you use `-xer` or `-xml` on the command line. If, however, any unsupported encoding instructions are found, ASN1C will ignore all XER encoding instructions, since it would not be capable of supporting EXTENDED-XER for that specification.

Supported Instructions and Brief Summary

ASN1C supports these instructions: `ATTRIBUTE` and `BASE64`. Very brief summaries of the effects of these instructions follow.

- `ATTRIBUTE`: This instruction causes a component of a sequence to be encoded as an XML attribute.

- `BASE64`: This instruction causes octet strings to be encoded in a base64 representation, rather than a hexadecimal one.

Limitations

The following are limitations related to EXTENDED-XER:

- For `BASE64`: ASN1C only supports `BASE64` on octet strings. Using `BASE64` with octet stings having contents constraints, open types, or restricted character strings is not supported.

- For encoder's options: ASN1C decoders do not support the following encoder's options allowed by EXTENDED-XER:

  - encoding named bits as empty elements

  - encoding named numbers as empty elements

- Enforcement of Encoding Instruction Restrictions: ASN1C does not check that you are using encoding instructions properly. Misapplication of encoding instructions has undefined results. For example, X.693 does not generally allow `ATTRIBUTE` to be applied to a sequence type (there are a few cases where it can be); such an application produces malformed XML.

  In particular, when applying `ATTRIBUTE` to a restricted character string type, the type should be restricted to exclude the control characters listed in X.680 15.15.5, since these control characters are encoded as empty elements. (Another solution would be to use `ATTRIBUTE` and `BASE64` together, except that ASN1C does not currently support `BASE64` for restricted character strings.) ASN1C will not enforce this rule, but you will get malformed XML if you try to encode a string having control characters as an attribute.

- XSD Generation: The `-xsd` switch does not currently generate XSD that can be used to validate EXTENDED-XER encodings. (Actually, in the worst cases, it is not possible to produce XSD that validates precisely the set of valid EXTENDED-XER encodings; the closest approximations would either fail to reject some invalid encodings or fail to accept some valid encodings. This is a result of the encoder's options, which can produce mixed content models and XML Schema's limited abilities to constrain mixed content models.)

Working with generated EXTENDED-XER code

As noted above, when generating code for EXTENDED-XER, you will be working with the "XML" runtime, as oppposed to the "XER" runtime. This makes coding for EXTENDED-XER slightly different from coding for XER:

- You will use Asn1XmlEncoder instead of Asn1XerEncoder

- You will supply the name of the element when encoding a value. Typically, this will be the name of the ASN.1 PDU type.

Finally, there is a sample reader and writer program in `java/sample_xer/EmployeeEXER`, should you need to see an example.

Generated Encode Methods

The generation of methods to encode data in XML is similar to how methods were generated in the BER/DER and PER cases discussed previously. For each ASN.1 production defined in the ASN.1 source file, a Java encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the encoding of the base class method. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method:

```
X ::= INTEGER
```

In this case, the generated Java class extends the *Asn1Integer* base class and the default encode method within this class is sufficient to encode a value of the generated type.

A custom encode method is only generated if:

1. The ASN.1 type is constructed (SEQUENCE, SET, SEQUENCE OF, SET OF, or CHOICE).

2. The ASN.1 type contains a testable constraint (for example, INTEGER (1..100))

3. The ASN.1 type is enumerated. This includes an INTEGER type with named numbers, a BIT STRING with named bit constants, or the ENUMERATED built-in type.

Note

Two variations are discussed here: "XER" and "XML". The "XML" variant applies to EXTENDED-XER. See the Overview section above for more information.

Generated Java Method Format and Calling Parameters

The signature for a Java XER encode method is as follows:

```
public void encode (Asn1XerEncoder buffer, String elemName)
    throws Asn1Exception, java.io.IOException
```

The signature for a Java XML encode method is similar:

```
public void encode (Asn1XmlEncoder buffer, String elemName)
    throws Asn1Exception, java.io.IOException
```

The *buffer* argument is a reference to an *Asn1XerEncoder* or *Asn1XmlEncoder* derived object that describes the buffer or output stream into which a message is to be encoded. *Asn1XerEncoder* is a base interface for the *Asn1XerEncodeBuffer* and *Asn1XerOutputStream* classes. Similarly, *Asn1XmlEncoder* is an interface to a pure XML version of these base classes. There is no difference which encode method is used: output stream or message buffer. The generated logic is the same, the difference is only in the first parameter of the encode method. This must be created and initialized before calling any encode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used.

The *elemName* argument is a reference to a string containing the element name text. This text is used to form the standard XML angle-bracketed wrapper that is applied to each element in a message. Note the name passed must not contain the angle-brackets (i.e. the $<>$ characters). These will be added by the encode method.

The *elemName* can be passed in different ways to control how the name is applied. The normal way is to pass a name that is applied as the element name of the element. If null is passed, then the default element name for the referenced ASN.1 built-in type is used. For example, <BOOLEAN> is the default element name for the ASN.1 BOOLEAN type. The complete list of default element names can be found in the X.693 standard. If an empty string is passed (i.e. ""), this tells the encode method to omit the element name string all together and just encode the value (this is similar to implicit tagging in the BER case).

The XER or XML encode methods do not return a value. This is different than the C/C++ version that returns a negative status value to indicate an encoding failure. For Java, errors are reported via the exception mechanism. All ASN1C Java exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown. If I/O error occurs then the *java.io.IOException* is thrown.

Procedure for Calling Java XER Encode Methods

The Java class variables corresponding to each of the ASN.1 types and method of population are the same as they were in the BER encoding case. See the section *Populating Generated Variables* for Encoding for instructions on how to populate the variables prior to encoding.

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode message buffer or output stream object into which the value will be encoded

2. Invoke encode methods. These include the *encodeStartDocument* and *encodeEndDocument* methods from the *Asn1XerEncodeBuffer* class and the encode method from the ASN1C generated class.

3. If the encode message buffer is used: invoke encode message buffer methods to access the encoded message component. If the output stream is used: close the stream.

The first step is the creation of an encode message buffer object. For XER encoding, this is an object of the *Asn1XerEncodeBuffer* class. The following constructors are available for creating an XER encode buffer object:

```
public Asn1XerEncodeBuffer ();

public Asn1XerEncodeBuffer (boolean canonical, int sizeIncrement);
```

The default constructor sets all internal buffer control variables to default values. Canonical XER is set to false and size increment is set to 1024. The other forms of the constructor allow these variables to be changed. Canonical XER specifies that the canonical form of XER encoding (CXER as specified in X.693) should be used. Size increment specifies the amount by which the dynamic encode buffer should be expanded when it fills up. This should be set lower for small, memory-constrained environments and higher if large messages are being encoded.

If the output stream method is used then the first step is the creation of an output stream. For XER encoding, this is an object of the *Asn1XerOutputStream* class. The following constructors are available for creating an XER encode buffer object:

```
public Asn1XerOutputStream (OutputStream os);

public Asn1XerOutputStream (OutputStream os, boolean canonical, int bufSize);
```

The first constructor creates a buffered XER output stream with default size of an internal buffer. Canonical XER is set to false. The other form of the constructor allows these variables to be changed. Canonical XER specifies that the canonical form of XER encoding (CXER as specified in X.693) should be used. The buffer size argument specifies the size of the internal buffer of the stream. Larger buffer sizes typically provide better performance at the expense of increased memory consumption.

Similar classes exist for XML encode buffer and streams:

```
public Asn1XmlEncodeBuffer ()
public Asn1XmlEncodeBuffer (int sizeIncrement)
public Asn1XmlOutputStream (OutputStream os)
public Asn1XmlOutputStream (OutputStream os, int bufSize)
```

The main difference is the XML classes to not have a canonical XML option; therefore, there is not cxer or *canonical* boolean argument.

The second step is the invocation of the encode methods. The calling arguments were described earlier. As per the Java standard, this method must be invoked from within a try/catch block to catch the possible *Asn1Exception* or *java.io.IOException* that may be thrown. Alternatively, the method from which the encode method is called can declare that it throws *Asn1Exception* and *java.io.IOException* leaving it to be dealt with at a higher level.

Finally, if a message buffer is used, encode buffer methods can be called to access the encoded message component. The Java API provides an object called a *ByteArrayInputStream* that provides a way to look at the encoded component as a stream. The encode buffer object provides a method called *getInputStream* that returns a byte array input stream representing the message component. This is the preferred way to access the encoded component.

In addition to *getInputStream*, there is a *getMsgCopy* method that will retrieve a copy of the generated message into a byte array object. This is somewhat slower because a copy needs to be done. Another option that is available when doing XER encoding is the *getBuffer* method. This returns a reference to the actual message buffer into which the message was encoded. Since an XER message is encoded front-to-back (unlike the back-to-front used in BER/DER encoding), the buffer reference returned will point to the start of the encoded message. The *getMsgLength* method can then be used to get the message length (in bytes). Note that the byte count may not correspond to the actual character count as UTF-8 encoding is used and some characters may be multiple bytes in length.

If an output stream is used, the stream should be closed when encoding is complete to ensure all buffered data is flushed to the output device.

The *Asn1XerEncodeBuffer* encode buffer class also contains other methods for operating directly on the encoded component (for example, the write method can be used to write it to a file or other medium). A user could also derive their own special encode buffer class from this class to add more functionality. See the description of the *Asn1XerEncodeBuffer* class in the run-time section for a full description of the available methods.

A complete example showing how to invoke an XER encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

// Step 1: Create a message buffer object. This object uses
// standard XER (non-canonical) and the default size increment
// for buffer expansion..

Asn1XerEncodeBuffer encodeBuffer = new Asn1XerEncodeBuffer();

// Step 2: Invoke the encode methods. These include
// encodeStartDocument to encode the XML document header,
// the generated Java encode method to encode the document body,
// and the encodeEndDocument method to complete the message.
// Note that these methods must be invoked from within a
// try/catch block..

try {
    encodeBuffer.encodeStartDocument ();

    personnelRecord.encode (encodeBuffer, null);

    encodeBuffer.encodeEndDocument ();

    if (trace) {
        System.out.println ("Encoding was successful");
```

```
        encodeBuffer.write (System.out);
    }

    // Step 3: Access the encoded message component. In this
    // case, we use methods in the class to write the encoded
    // XML document to a file..

    encodeBuffer.write (new FileOutputStream (filename));

    // We can also directly access the buffer as follows:

    byte[] buffer = encodeBuffer.getBuffer();
    int msglen = encodeBuffer.getMsgByteCnt();
}
catch (Exception e) {
    System.out.println (e.getMessage());
    e.printStackTrace();
    return;
}
```

An example showing stream-based encoding is as follows:

```
// Note: personnelRecord object was previously populated with data

Asn1XerOutputSteram out = null;

try {

    // Step 1: Create an output stream object. This object
    // uses standard XER (non-canonical) and the default
    // internal buffer's size.

    out = new Asn1OutputStream(new FileOutputStream (filename));

    // Step 2: Invoke the encode methods. These include
    // encodeStartDocument to encode the XML document header,
    // the generated Java encode method to encode the document body,
    // and the encodeEndDocument method to complete the message.
    // Note that these methods must be invoked from within a
    // try/catch block..

    out.encodeStartDocument ();

    personnelRecord.encode (out, null);

    out.encodeEndDocument ();

    if (trace) {
        System.out.println ("Encoding was successful");
        encodeBuffer.write (System.out);
    }
}
catch (Exception e) {
    System.out.println (e.getMessage());
```

```
        e.printStackTrace();
        return;
    }
    finally {
        // Step 3: Close the stream.

        try {
            if (out != null)
                out.close ();
        }
        catch (Exception e) {}
    }
```

If you compare these examples with the other encoding examples, you will see the procedures are similar. This makes it very easy to switch encoding methods should the need arise.

In the case of XML encode, the procedure is very similar. The only difference is that it is not necessary to call the *encodeStartDocument* and *encodeEndDocument* methods. The are built into the generated *encode* method for PDU data types.

The resulting XML document from running the program above is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<PersonnelRecord>
    <name>
        <givenName>John</givenName>
        <initial>P</initial>
        <familyName>Smith</familyName>
    </name>
    <number>51</number>
    <title>Director</title>
    <dateOfHire>19710917</dateOfHire>
    <nameOfSpouse>
        <givenName>Mary</givenName>
        <initial>T</initial>
        <familyName>Smith</familyName>
    </nameOfSpouse>
    <children>
        <ChildInformation>
            <name>
                <givenName>Ralph</givenName>
                <initial>T</initial>
                <familyName>Smith</familyName>
            </name>
            <dateOfBirth>19571111</dateOfBirth>
        </ChildInformation>
        <ChildInformation>
            <name>
                <givenName>Susan</givenName>
                <initial>B</initial>
                <familyName>Jones</familyName>
            </name>
            <dateOfBirth>19590717</dateOfBirth>
        </ChildInformation>
    </children>
```

```
        </PersonnelRecord>
```

Generated Decode Methods

ASN1C decoders now default to using the XmlPull API to parse XML. The kXML implementation of this API is provided with ASN1C in the `java/xmlpull` folder and is a standard API on Android. You may alternatively use `-stax` on the ASN1C command line to use the StAX API. The StAX API is a standard part of Java SE 1.6+, and is available for Java ME under JSR-280. A final alternative, for the time being, is to specify `-sax` on the ASN1C command line to use the SAX API, which was previously the only parser API supported by ASN1C.

Generated Decode Methods using the XmlPull API

Generated classes will have two methods for decoding:

```
public void decodeDocument(org.xmlpull.v1.XmlPullParser reader)

public void decode(org.xmlpull.v1.XmlPullParser reader, boolean asGroup)
```

The `decodeDocument` method should be used when the parser is positioned at the start of an XML document, making it necessary to move to the root element before reaching the data to be decoded. This is the method you will most likely use.

The `decode` method should be used when the parser is already positioned on the element to be decoded. If the current element contains the content to be decoded, invoke `decode` with `asGroup = false`. If, however, the current element is itself part of the content to be decoded (and, in some cases, its siblings also), then invoke `decode` with `asGroup = true`. Most likely, you will pass `false`.

To illustrate, suppose that we have a `SEQUENCE` type that defines `elem1` and `elem2`. If we have an XML snippet to decode, `<root><elem1/><elem2/></root>`, and we are positioned on the `root` element, we would use `asGroup = false`, because `root` contains the content to be decoded and is not a part of it. If, however, we were positioned on the `elem1` element, we would use `group = true`, because `elem1` is a part of the content to be decoded; it is the beginning of the content itself.

The general procedure for decoding is as follows:

1. Create an `XmlPullParserFactory`.

2. Get a new `XmlPullParser` from the parser factory.

3. Configure the new parser by setting it to process namespaces and giving it the input stream to be parsed.

4. Instantiate an instance of the generated class you want to decode.

5. Invoke the `decodeDocument` method on the generated class.

6. Access the decoded data through the public members of the generated class.

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

```
    public class Reader {
```

```
    public static void main (String args[]) {
        String filename = "message.xml";

        java.io.InputStream inputFile = null;

        try {
            // Create an XML reader object

            inputFile = new java.io.BufferedInputStream(
                        new java.io.FileInputStream(filename) );

            org.xmlpull.v1.XmlPullParserFactory xmlInputFactory =
                        org.xmlpull.v1.XmlPullParserFactory.newInstance();

            org.xmlpull.v1.XmlPullParser reader =
                        xmlInputFactory.newPullParser();

            reader.setFeature(
                        org.xmlpull.v1.XmlPullParser.FEATURE_PROCESS_NAMESPACES, true);

            reader.setInput(inputFile, null);


            // Read and decode the message

            PersonnelRecord personnelRecord = new PersonnelRecord ();
            personnelRecord.decodeDocument(reader);
            if (trace) {
                System.out.println ("Decode was successful");
                personnelRecord.print (System.out, "personnelRecord", 0);
            }
        }
        catch (Exception e) {
            System.out.println (e.getMessage());
            e.printStackTrace();
            System.exit(-1);
        }
        finally {
            if ( inputFile != null )
                try { inputFile.close(); } catch (java.io.IOException e) {}
        }
    }
}
```

Generated Decode Methods using the StAX API

Generated classes will have two methods for decoding:

```
public void decodeDocument(javax.xml.stream.XMLStreamReader reader)

public void decode(javax.xml.stream.XMLStreamReader reader, boolean asGroup)
```

The `decodeDocument` method should be used when the parser is positioned at the start of an XML document, making it necessary to move to the root element before reaching the data to be decoded. This is the method you will most likely use.

The `decode` method should be used when the parser is already positioned on the element to be decoded. If the current element contains the content to be decoded, invoke `decode` with `asGroup = false`. If, however, the current element is itself part of the content to be decoded (and, in some cases, its siblings also), then invoke `decode` with `asGroup = true`. Most likely, you will pass `false`.

To illustrate, suppose that we have a `SEQUENCE` type that defines `elem1` and `elem2`. If we have an XML snippet to decode, `<root><elem1/><elem2/></root>`, and we are positioned on the `root` element, we would use `asGroup = false`, because `root` contains the content to be decoded and is not a part of it. If, however, we were positioned on the `elem1` element, we would use `group = true`, because `elem1` is a part of the content to be decoded; it is the beginning of the content itself.

The general procedure for decoding is as follows:

1. Create an `XMLInputFactory`.

2. Configure the factory to do coalescing.

3. Get a new `XMLStreamReader` from the factory.

4. Instantiate an instance of the generated class you want to decode.

5. Invoke the `decodeDocument` method on the generated class.

6. Access the decoded data through the public members of the generated class.

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

```
public class Reader {
    public static void main (String args[]) {
        String filename = "message.xml";

        java.io.InputStream inputFile = null;

        try {
            // Create an XML reader object

            inputFile = new java.io.BufferedInputStream(
                        new java.io.FileInputStream(filename) );

            javax.xml.stream.XMLInputFactory xmlInputFactory =
                javax.xml.stream.XMLInputFactory.newInstance();

            xmlInputFactory.setProperty(
                javax.xml.stream.XMLInputFactory.IS_COALESCING, Boolean.TRUE);

            javax.xml.stream.XMLStreamReader reader =
                xmlInputFactory.createXMLStreamReader(inputFile);


            // Read and decode the message
```

```
        PersonnelRecord personnelRecord = new PersonnelRecord ();
        personnelRecord.decodeDocument(reader);
        if (trace) {
            System.out.println ("Decode was successful");
            personnelRecord.print (System.out, "personnelRecord", 0);
        }
    }
    catch (Exception e) {
        System.out.println (e.getMessage());
        e.printStackTrace();
        System.exit(-1);
    }
    finally {
        if ( inputFile != null )
            try { inputFile.close(); } catch (java.io.IOException e) {}
    }
  }
}
```

Generated Decode Methods when Using the SAX parser

As a SAX parser parses XML, it invokes callbacks on a content handler interface to notify any interested parties of the data it has parsed. ASN1C generates an implementation of the content handler interface as an inner class of each class generated for an ASN.1 production. This content handler populates the fields of the data class as the XML is parsed.

The default SAX parser used is the XERCES parser developed by the Apache Software Foundation (http:// xml.apache.org). Since SAX is a de-facto standard, it should be relatively straightforward to use the generated handlers with any other implementation of SAX parser.

A diagram showing the components used in the XML decode process is as follows:

Step 1: Generate code:
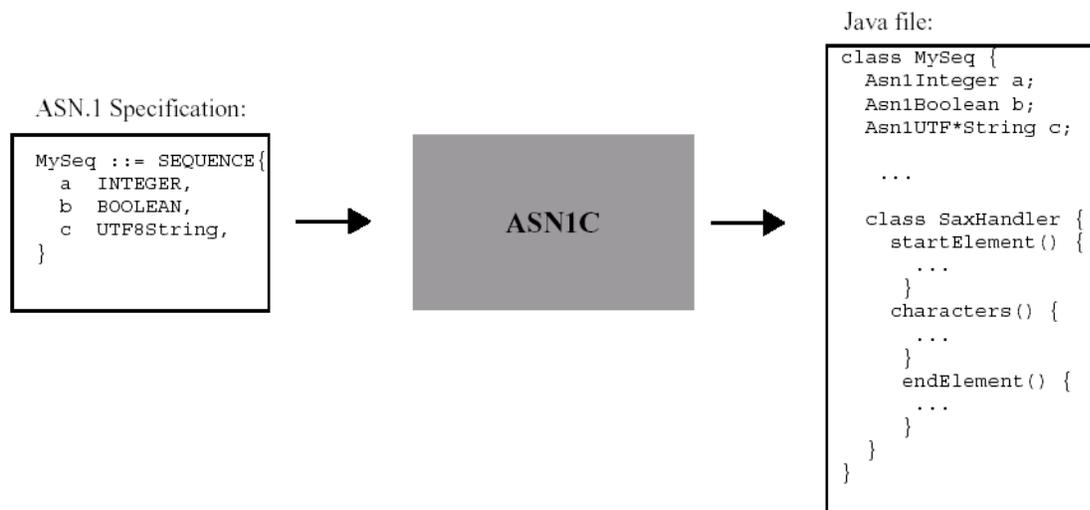
Step 2: Build Application:
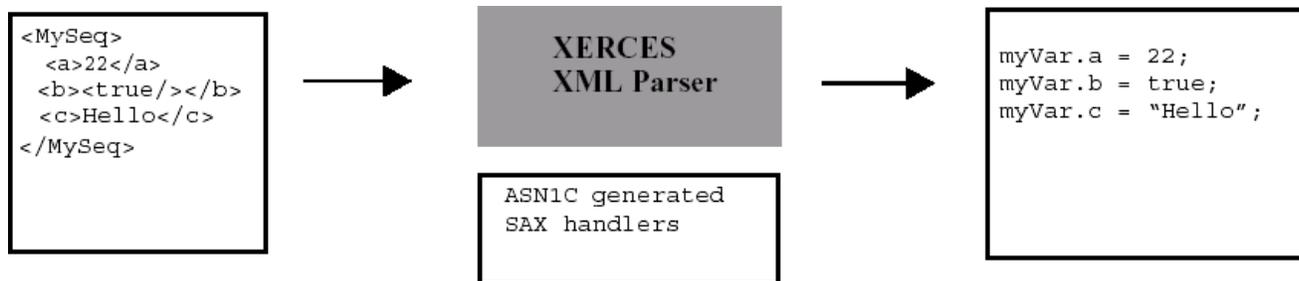
ASN1C generates code to implement the following methods defined in SAX content handler interface:

```
startElement

        characters

        endElement
```

The interface defines other methods that can be implemented as well, but these are sufficient to decode XER encoded data. These methods are added to an inner SAX handler class generated for each ASN.1 production.

The procedure to invoke the generated decode method is similar to that for the other encoding rules. It is as follows:

1. Instantiate an *XMLReader* object. The XML parser interface should provide a factory method for creating an object of this type for any vendor-specific XML parser implementation.

2. Instantiate a generated Java <ProdName> object to hold the decoded message data.

3. Invoke the <ProdName> object decode method passing the reader created in step 1 and the URI of the XML document to be parsed. This method initiates and invokes the XML parser's *parse* method to parse the document. This, in turn, invokes the generated SAX handler methods.

4. Methods within the <ProdName> object can now be used to access the decoded data. The member variables that were declared to be public can be accessed directly.

5. Error handling is accomplished using a try-catch block to catch SAX exceptions.

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

```
public class Reader {
   public static void main (String args[]) {
      String filename = "employee.xml";
      String vendorParserClass =
      "org.apache.xerces.parsers.SAXParser";

      try {
         // Create an XML reader object

         XMLReader reader =
         XMLReaderFactory.createXMLReader (vendorParserClass);

         // Read and decode the message

         PersonnelRecord personnelRecord = new PersonnelRecord ();
         personnelRecord.decode (reader, filename);
         if (trace) {
```

```
                System.out.println ("Decode was successful");
                personnelRecord.print (System.out, "personnelRecord", 0);
            }
        }
        catch (Exception e) {
            System.out.println (e.getMessage());
            e.printStackTrace();
            return;
        }
    }
}
```

Chapter 11. Generated OER Encode Methods

The generation of methods to encode data in accordance with the Octet Encoding Rules (OER) is similar to how methods were generated in the BER/DER case discussed previously. For each ASN.1 production defined in the ASN.1 source file, a Java encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the encoding of the base class method. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For simple assignments, the generation of an encode method is not necessary. For example, the following production will not result in the generation of an encode method:

```
X ::= INTEGER
```

In this case, the generated Java class extends the *Asn1Integer* base class and the default encode method within this class is sufficient to encode a value of the generated type.

An encode method is typically only generated when a testable constraint is present.

For example, the following declaration will cause a custom encode method to be generated because the value range constraint is OER-visible and alters the encoding:

```
X ::= INTEGER (1..255)
```

In this case, special logic is necessary to apply the value range constraint.

Constructed types (SEQUENCE, SET, SEQUENCE/SET OF, and CHOICE) will always have a generated encode method.

Generated Java Method Format and Calling Parameters

The signature for a Java OER encode method is as follows:

```
public void encode (Asn1OerEncodeBuffer buffer)
throws java.io.IOException
```

The *buffer* argument is a reference of an *Asn1OerEncodeBuffer* object that describes the buffer into which a message is to be encoded. This must be created and initialized before calling any encode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used.

The OER encode methods do not return a value. Errors are reported via the exception mechanism. All ASN1C Java exceptions are derived from the *Asn1Exception* base class. See the section on exceptions for a complete list and description of the various exceptions that can be thrown.

Procedure for Calling Java OER Encode Methods

The Java class variables corresponding to each of the ASN.1 types and method of population are the same as they were in the BER encoding case. See the section on BER encoding for instructions on how to populate the variables prior to encoding.

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode message buffer object into which the value will be encoded

2. Invoke the encode method

3. Invoke encode message buffer methods to access the encoded message component

The first step is the creation of an encode message buffer object. For OER encoding, this is an object of the *Asn1OerEncodeBuffer* class. The following constructors are available for creating an OER encode buffer object:

```
public Asn1OerEncodeBuffer ();

public Asn1OerEncodeBuffer (int size);
```

The second form of the constructor includes a size argument for the initial size of the buffer. The default constructor sets the value to a reasonable mid-range value (see INITIAL_SIZE in *Asn1EncodeBuffer.java*, as of this writing the value was set to 1024).

The second step is the invocation of the encode method, described earlier. As the method throws *java.io.IOException*, your code must either handle that exception in a try/catch block or else declare that it throws it.

Finally, encode buffer methods can be called to access the encoded message component. The Java API provides an object called a *ByteArrayInputStream* that provides a way to look at the encoded component as a stream. The encode buffer object provides a method called *getInputStream* that returns a byte array input stream representing the message component. This is the preferred way to access the encoded component.

In addition to getInputStream there is a *getMsgCopy* function that will retrieve a copy of the generated message into a byte array object. This is somewhat slower because a copy needs to be done. Another option is the *getBuffer* method. This returns a reference to the actual message buffer into which the message was encoded. The *getMsgByteCnt* method can then be used to get the message length in bytes or the *getMsgBitCnt* method can be called to get the length in bits.

The encode buffer class also contains other methods for operating directly on the encoded component (for example, the *write* method can be used to write it to a file or other medium). And of course, one could derive their own special encode buffer class from this class to add more functionality. See the description of the *Asn1OerEncodeBuffer* class in the runtime section for a full description of the available methods.

A complete example showing how to invoke an OER encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

// Step 1: Create a message buffer object. This object uses the
// default size.

Asn1OerEncodeBuffer encodeBuffer = new Asn1OerEncodeBuffer();
```

```
    // Step 2: Invoke the encode method. Note that it must be done
    // from within a try/catch block..

    try {
        personnelRecord.encode (encodeBuffer);

        if (trace) {
            System.out.println ("Encoding was successful");
            System.out.println ("Hex dump of encoded record:");
            encodeBuffer.hexDump ();
        }

        // Step 3: Access the encoded message component. In this
        // case, we use methods in the class to write the component
        // to a file and output a formatted dump to the message.dmp
        // file.

        // Write the encoded record to a file

        encodeBuffer.write (new FileOutputStream (filename));

        // Generate a dump file for comparisons

        encodeBuffer.hexDump
            (new PrintStream (new FileOutputStream ("message.dmp")));

        // We can also directly access the buffer as follows:

        byte[] buffer = encodeBuffer.getBuffer();
        int msglen = encodeBuffer.getMsgByteCnt();
    }
    catch (Exception e) {
        System.out.println (e.getMessage());
        e.printStackTrace();
        return;
    }
```

If you compare this example with the BER encoding example in Figure 2, you will see the encoding procedure is almost identical. This makes it very easy to switch encoding methods should the need arise. All you need to do is change the buffer class and slightly alter the invocation of the encode method.

Canonical OER

If you want to produce a Canonical-OER encoding, you can choose to do so at runtime by invoking `setCanonicalMode(true)` on the encode buffer. This is different from the approach used with some other encoding rules, where the canonical variety of encoding is chosen at compile-time using an ASN1C command line option.

Reuse of Java Encoding Objects

The concept of reusing Java objects for OER encoding is the same as was described previously for BER encoding. Basically, all that needs to be done is the creation of a single OER encode buffer object and an object corresponding to the ASN.1 data type to be encoded outside of the processing loop. These objects can then be reused to encode

each instance of the messages to be sent. After each message is encoded, the OER buffer must be reset for the next message by using the reset method. See the section on reuse of objects in the BER encoding section for a more thorough discussion and sample code on using this capability.

Chapter 12. Generated OER Decode Methods

For each ASN.1 production defined in the ASN.1 source file, a Java decode method *may* be generated. This method will decode an ASN.1 message into public member variables within the Java object.

As was the case for encode methods, a decode method is only generated if it is required to alter the default method in the base class. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

For primitive types, a custom OER decode method is only generated if one or more of the following is true:

1. The type contains a OER-visible constraint

2. The generation of event handlers was specified

The exception to this rule is the ENUMERATED primitive type (or likewise, INTEGER type with a named number list) that will always cause a decode method to be generated.

Constructed types will always cause custom OER decode methods to be generated.

Generated Java Method Format and Calling Parameters

The signature for a Java OER decode method is as follows:

```
public void decode (Asn1OerDecodeBuffer buffer)
    throws java.io.Exception
```

The *buffer* argument is a reference of an *Asn1OerDecodeBuffer* object that describes the message that is being decoded. This must be created and initialized before calling any decode method. See the description of this class in the *Java Run- Time Classes* section for details on how this class is used.

The decode method returns no result. Failure is handled by the exception mechanism. All ASN1C Java exceptions are derived from the Asn1Exception base class. See the section on exceptions for a complete list and description of the various ASN.1 exceptions that can be thrown.

Procedure for Calling Java OER Decode Methods

The general procedure to decode an ASN.1 OER message involves the following three steps:

1. Create a decode message buffer object to describe the message to be decoded

2. Invoke the decode method

3. Process the decoded data values

The first step is the creation of a decode message buffer object. The *Asn1OerDecodeBuffer* object contains constructors that can either accept a message as a byte array or as an I/O input stream. The input stream option makes it possible to decode messages directly from other mediums other than a memory buffer (for example, a message can be decoded directly from a file or a socket).

Unlike some other encoding rules, OER encodings do not indicate the ASN.1 type that was encoded. The ASN.1 type must be known prior to decoding in order to successfully decode it. A common approach in specifications is to have a specific outer level type known as a "Protocol Data Unit" (PDU) that encompasses all of the different message types that might be received. This is typically a CHOICE construct with each option representing a different type of message.

Whatever the message type is, the generated decode method for that message is invoked to decode the message. The calling arguments were described earlier. As the method throws *java.io.IOException*, your code must either handle that exception in a try/catch block or else declare that it throws it.

The final step is to process the data. All data is contained within public member variables so access is quite easy. All of the primitive data type classes contain a public member variable called *value* that contains decoded data. This can be accessed in nested structures by prefixing *value* with each of the element names from the top down. For example, the given name element in the Name type shown earlier would be accessed as follows: *name.givenName.value* (this assumes an instance of the Name class was created using the variable name *name*).

A complete example showing how to invoke a decode method is as follows:

```
try {

    // Step 1: create a decode message buffer object to describe the
    // message to be decoded. This example will use a file input
    // stream to decode a message directly from a binary file..

    // Create an input file stream object

    FileInputStream in = new FileInputStream (filename);

    // Create a decode buffer object

    Asn1OerDecodeBuffer decodeBuffer = new Asn1OerDecodeBuffer (in);

    // Step 2: create an object of the generated type and invoke the
    // decode method..

    PersonnelRecord personnelRecord = new PersonnelRecord ();
    personnelRecord.decode (decodeBuffer);

    // Step 3: process the data

    if (trace) {
        System.out.println ("Decode was successful");
        personnelRecord.print (System.out, "personnelRecord", 0);
    }
}
catch (Exception e) {
    System.out.println (e.getMessage());
    e.printStackTrace();
    return;
    }
```

Canonical OER

Many of the rules for Canonical-OER can be enforced during decoding, such that exceptions are thrown if the decoder detects that the encoding was not canonical. By default, this behavior is disabled. You can enable this behavior by invoking `setCanonicalMode(true)` on the decode buffer.

Reuse of Java Decoding Objects

Java objects can be reused for decoding OER messages in the same way they were for BER messages. The decode buffer and message type objects are created outside of the main decoding loop. Then in the main loop these objects are reused to process each input message. Data must be saved from the message type object after each iteration because the contents of the object will be overwritten on each consecutive loop iteration. Nothing special needs to be done at the bottom of the loop to ready the decoder for the next message. All necessary initialization will be handled internally.

Chapter 13. Generated JSON Encode Methods

Encoding to JSON is similar to other encoding rules. For each ASN.1 production defined in an ASN.1 source file, an encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the behavior of the base class. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types. The generated classes for all constructed types (SEQUENCE, SEQUENCE OF, and CHOICE) will include generated encode methods.

> ## Note
>
> Rules for encoding values of ASN.1 types to JSON have not be standardized. We follow the specification [http://www.obj-sys.com/docs/JSON%20Encoding%20Rules.pdf] available on our webiste at http://www.obj-sys.com/docs/JSON%20Encoding%20Rules.pdf

Generated Method Format and Calling Parameters

The signature for the JSON encode method is shown below:

```
public void encode(Asn1JsonOutputStream buffer )
    throws java.io.IOException
```

The *buffer* argument is an *Asn1JsonOutputStream* object which receives the encoded message. This must be created and initialized before calling any encode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used. Since JSON is a character-based encoding, *Asn1JsonOutputStream* are constructed on java.io.Writer instances. There is a subclass, *Asn1JsonOutputBuffer*, that can be used to output to a byte array.

As you can see, the encode methods return void; an exception is thrown if an error occurs. All ASN1C Java exceptions are derived from *Asn1Exception*. See the section on exceptions for a complete list and description of the various exceptions that can be thrown.

Populating Generated Variables for Encoding

Populating generated types for encoding can be done in most cases either through the constructors or by assigning object references to public member variables.

Constructors are provided for most generated types to allow direct population of the encapsulated member variable(s) on initialization. The exception to this is for classes generated for SEQUENCE OF types. In that case, the constructors only allow the size of an array to be specified – population of the array elements must be done manually.

All of the base run-time classes except *Asn1Null* contain public member variables. In practically all cases there is a single variable called *value* that is of the base type that needs to be populated. For example, the *Asn1Integer* base class contains the following item:

```
public long value;
```

So, for the following assignment:

```
X ::= INTEGER(0..255)
```

you may populate a variable of this type either using the constructor:

```
X x = new X (25);
```

or via direct access of the member variable:

```
X x = new X ();
x.value = 25;
```

The only primitive type that does not have a single member called *value* to represent its value is BIT STRING. The *Asn1BitString* class also contains a second variable called *numbits* to specify the number of bits in the string.

Procedure for Calling JSON Encode Methods

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode output stream into which the value will be encoded.

2. Invoke the encode method.

3. Close the output stream.

The first step is the creation of an encode output stream, an *Asn1JsonOutputStream*. There is a single constructor which accepts a java.io.Writer. As usual, you may use a buffered writer, file writer, char array writer etc., or some combination thereof.

The second step is the invocation of the encode method. The calling arguments were described earlier.

Finally, close the output stream.

A complete example showing how to invoke an encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

Asn1JsonOutputStream encodeStream = null;

try {
    // Step 1: Create an encode output stream, with UTF-8 character encoding
    encodeStream = new Asn1JsonOutputStream (
        new java.io.OutputStreamWriter( new java.io.FileOutputStream(filename), "UTF-8")

    // Step 2: Invoke the encode method.

    personnelRecord.encode (encodeStream);

    if (trace) {
        System.out.println ("Encoding was successful");
    }
}
catch (Exception e) {
    System.out.println(e.getMessage());
```

```
        e.printStackTrace();
    }
    finally {
        try {
            if (encodeStream != null) encodeStream.close ();
        }
        catch (Exception e) {}
    }
```

Reuse of Java Encoding Objects

The simple example above showed the procedure to encode a single record. But what if you had to encode a series of the same type of record over and over again?

In such cases, you can avoid some object creation and garbage collection by reusing objects you have already created. The generated classes and the ASN1C runtime classes can often be viewed as reusable containers into which you can simply assign new data.

To show an example of object reuse, suppose we were going to encode a series of octet strings. The ASN.1 type for our data might be:

```
    Data ::= SEQUENCE {
        first OCTET STRING,
        second OCTET STRING,
        third OCTET STRING
    }
```

The generated Java class would contain public member variables for each of the octet strings:

```
    public Asn1OctetString first;
    public Asn1OctetString second;
    public Asn1OctetString third;
```

The most efficient way to repopulate these variables within a loop would be simply to assign the data to be encoded to the public `value` field of the *Asn1OctetString* objects. You do not need to create new *Asn1OctetString* or *Data* objects each time.

A code snippet showing how this could be done is as follows:

```
    // Step 1: Create Data and Asn1JsonOutputStream objects for use in
    // the loop..

    Data data = new Data(null, null, null); // creates empty octet string objects
    Asn1JsonOutputStream encodeStream = new Asn1JsonOutputStream (yourWriter);

    for (;;) {

        // logic here to read name components from a DB or other medium

        ...

        // populate octet strings (assume first, second, third are byte arrays
        // populated by the above logic)

        data.first.value = first;
```

```
      data.second.value = second;
      data.third.value = third;

      // encode

      try {
         name.encode (encodeStream);

         // perhaps write some non-ASN.1 data to the stream?
         ...
      }
      catch (Asn1Exception e) {
         // handle error ..
      }
   }
}
```

Chapter 14. Generated JSON Decode Methods

For each ASN.1 production defined in the ASN.1 source file, a decode method *may* be generated. This method will decode an ASN.1 message into public member variables within the Java object.

As was the case for encode methods, a decode method is only generated if it is required to alter the behavior of the base class. The Java model is built on inheritance from a set of common run-time base classes. These run-time classes contain default implementations of encode/decode methods that handle the encoding/decoding of the basic types.

Note

Rules for encoding values of ASN.1 types to JSON have not be standardized. We follow the specification [http://www.obj-sys.com/docs/JSON%20Encoding%20Rules.pdf] available on our webiste at http://www.obj-sys.com/docs/JSON%20Encoding%20Rules.pdf

Generated Method Format and Calling Parameters

The signature for a JSON decode method is as follows:

```
public void decode(Asn1JsonDecodeBuffer buffer)
    throws java.io.IOException
```

The *buffer* argument is an *Asn1JsonDecodeBuffer* object which provides the message to be decoded. This must be created and initialized before calling any decode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used.

As you can see, the decode method returns void; the data is decoded into the instance on which decode is invoked and an exception is thrown if an error occurs. All ASN1C exceptions are derived from *Asn1Exception*. See the section on exceptions for a complete list and description of the various ASN1C exceptions that can be thrown.

Procedure for Calling Java JSON Decode Methods

The general procedure to decode an ASN.1 JSON message involves the following three steps:

1. Create a decode buffer on the message to be decoded

2. Invoke the decode method

3. Process the decoded data values

The first step is the creation of a decode buffer. An *Asn1JsonDecodeBuffer* can be constructed on either a java.io.Reader or a java.io.InputStream. In the latter case, the character encoding can be detected (except on Java ME, where it is assumed to be UTF-8). Thus, messages can easily be sourced from streams, files, or arrays.

The second step is to invoke the generated decode method. The calling arguments were described earlier.

The final step is to apply your application-specific processing to the data. All data is contained within public member variables so access is quite easy.

A complete example showing how to invoke a decode method is as follows:

```
try {
    // Step 1: create a decode buffer for the message to be decoded.
    // This example will use a file input stream to decode a message
    // in a binary file.

    // Create an input file stream object
    FileInputStream ins = new FileInputStream (filename);

    // Create a decode buffer object
    Asn1JsonDecodeBuffer decodeBuffer = new Asn1JsonDecodeBuffer (ins);

    // Step 2: create an object of the generated type and invoke the
    // decode method..
    PersonnelRecord personnelRecord = new PersonnelRecord ();
    personnelRecord.decode (decodeBuffer);

    // Step 3: process the data
    if (trace) {
        System.out.println ("Decode was successful");
        personnelRecord.print (System.out, "personnelRecord", 0);
    }
}
catch (Exception e) {
    System.out.println (e.getMessage());
    e.printStackTrace();
    return;
}
```

Reuse of Java Decoding Objects

The sample above showed the JSON decoding of a single message. In a typical application, a loop would be involved to decode a series of messages.

A single decode buffer can be used to process a stream of messages (assuming all messages are using the same character encoding). If the decode buffer is created using an input stream that contains a series of messages (for example, a file containing multiple records, or a communications device), you can repeatedly invoke the JSON decode method on the given message type.

Note that you can also use the same instance of your message type for repeated decoding, rather than creating a new object and leaving the old one to be garbage collected. Nothing special needs to be done to do this. The generated decode method will automatically call the internal *init()* method before decoding to make sure all items are reset to their starting state.

In the example above, all that would need to be done to decode a series of personnel records is the inclusion of a loop after the PersonnelRecord object was created in step 2:

```
for (;;) {
    personnelRecord.decode (decodeBuffer);

    if (trace) {
        System.out.println ("Decode was successful");
        personnelRecord.print (System.out, "personnelRecord", 0);
```

```
    }
}
```

Chapter 15. Generated MDER Encode Methods

Unlike what is done for other encoding rules, for MDER we provide stream-based encoding only. In order to encode to a memory buffer, you simply use a java.io.ByteArrayOutputStream.

For each ASN.1 production defined in an ASN.1 source file, an encode method *may* be generated. This function will convert a populated variable of the given type into an encoded ASN.1 message.

An encode method is only generated if it is required to alter the behavior of the base class. The Java model is built on inheritance from a set of common run-time base classes. MDER supports only a few primitive types, and most of these are subsets of ASN.1 built-in types. Therefore, in most cases, there is not an applicable MDER encode method in the common run-time base class and so an encode method will be generated.

The generated classes for all constructed types (SEQUENCE, SEQUENCE OF, and CHOICE) will include generated encode methods.

Generated Method Format and Calling Parameters

The signature for the MDER encode method comes in two forms, shown below:

```
public void encode(Asn1MderOutputStream buffer, boolean useCachedLength )
   throws Asn1Exception, java.io.IOException

public void encode(Asn1MderOutputStream buffer )
   throws Asn1Exception, java.io.IOException
```

The *buffer* argument is an *Asn1MderOutputStream* object which receives the encoded message. This must be created and initialized before calling any encode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used.

The *useCachedLength* argument indicates whether the encode method can rely on cached length information. In some cases, MDER requires pre-calculation of the length of nested structures and this piece of information is needed twice during encoding (once when encoding the containing structure and once when encoding the nested structure). The generated types cache this information during encoding. User code should normally pass `false` for this argument. The generated encoding methods will pass `true` to other encoding methods when appropriate.

As you can see, the encode methods return void; an exception is thrown if an error occurs. All ASN1C Java exceptions are derived from *Asn1Exception*. See the section on exceptions for a complete list and description of the various exceptions that can be thrown.

Populating Generated Variables for Encoding

Populating generated types for encoding can be done in most cases either through the constructors or by assigning object references to public member variables.

Constructors are provided for most generated types to allow direct population of the encapsulated member variable(s) on initialization. The exception to this is for classes generated for SEQUENCE OF types. In that case, the constructors only allow the size of an array to be specified – population of the array elements must be done manually.

All of the base run-time classes except *Asn1Null* contain public member variables. In practically all cases there is a single variable called *value* that is of the base type that needs to be populated. For example, the *Asn1Integer* base class contains the following item:

```
public long value;
```

So, for the following assignment:

```
X ::= INTEGER(0..255)
```

you may populate a variable of this type either using the constructor:

```
X x = new X (25);
```

or via direct access of the member variable:

```
X x = new X ();
x.value = 25;
```

The only primitive type that does not have a single member called *value* to represent its value is BIT STRING. The *Asn1BitString* class also contains a second variable called *numbits* to specify the number of bits in the string.

Procedure for Calling MDER Encode Methods

Once an object's member variables have been populated, the object's encode method can be invoked to encode the value. The general procedure to do this involves the following three steps:

1. Create an encode output stream into which the value will be encoded.

2. Invoke the encode method.

3. Close the output stream.

The first step is the creation of an encode output stream, an *Asn1MderOutputStream*. There is a single constructor which accepts a java.io.OutputStream. As usual, you may use a buffered output stream, file output stream, byte array output stream, etc., or some combination thereof.

The second step is the invocation of the encode method. The calling arguments were described earlier.

Finally, close the output stream.

A complete example showing how to invoke an encode method is as follows:

```
// Note: personnelRecord object was previously populated with data

Asn1MderOutputStream encodeStream = null;

try {
    // Step 1: Create an encode output stream.
     encodeStream = new Asn1MderOutputStream (new FileOutputStream (filename));

    // Step 2: Invoke the encode method. Note that it must be done
    // from within a try/catch block..

     personnelRecord.encode (encodeStream, /*useCachedLength=*/false);
```

```
       if (trace) {
            System.out.println ("Encoding was successful");
       }
   }
   catch (Exception e) {
       System.out.println(e.getMessage());
       e.printStackTrace();
   }
   finally {
       try {
            if (encodeStream != null) encodeStream.close ();
       }
       catch (Exception e) {}
   }
```

Reuse of Java Encoding Objects

The simple example above showed the procedure to encode a single record. But what if you had to encode a series of the same type of record over and over again?

In such cases, you can avoid some object creation and garbage collection by reusing objects you have already created. The generated classes and the ASN1C runtime classes can often be viewed as reusable containers into which you can simply assign new data.

To show an example of object reuse, suppose we were going to encode a series of octet strings. The ASN.1 type for our data might be:

```
Data ::= SEQUENCE {
    first OCTET STRING,
    second OCTET STRING,
    third OCTET STRING
}
```

The generated Java class would contain public member variables for each of the octet strings:

```
public Asn1OctetString first;
public Asn1OctetString second;
public Asn1OctetString third;
```

The most efficient way to repopulate these variables within a loop would be simply to assign the data to be encoded to the public `value` field of the *Asn1OctetString* objects. You do not need to create new *Asn1OctetString* or *Data* objects each time.

A code snippet showing how this could be done is as follows:

```
// Step 1: Create Data and Asn1MderOutputStream objects for use in
// the loop..

Data data = new Data(null, null, null); // creates empty octet string objects
Asn1MderOutputStream encodeStream = new Asn1MderOutputStream (outputStream);

for (;;) {

    // logic here to read name components from a DB or other medium
```

```
    ...

    // populate octet strings (assume first, second, third are byte arrays
    // populated by the above logic)

    data.first.value = first;
    data.second.value = second;
    data.third.value = third;

    // encode

    try {
        name.encode (encodeStream, /*useCachedLength=*/false);

        // perhaps write some non-ASN.1 data to the stream?
        ...
    }
    catch (Asn1Exception e) {
        // handle error ..
    }
}
```

Chapter 16. Generated MDER Decode Methods

For each ASN.1 production defined in the ASN.1 source file, a decode method *may* be generated. This method will decode an ASN.1 message into public member variables within the Java object.

As was the case for encode methods, a decode method is only generated if it is required to alter the behavior of the base class. The Java model is built on inheritance from a set of common run-time base classes. MDER supports only a few primitive types, and most of these are subsets of ASN.1 built-in types. Therefore, in most cases, there is not an applicable MDER decode method in the common run-time base class and so a decode method will be generated.

Generated Method Format and Calling Parameters

The signature for an MDER decode method is as follows:

```
public void decode(Asn1MderDecodeBuffer buffer)
    throws Asn1Exception, java.io.IOException
```

The *buffer* argument is an *Asn1MderDecodeBuffer* object which provides the message to be decoded. This must be created and initialized before calling any decode method. See the description of this class in the *Java Run-Time Classes* section for details on how this class is used.

As you can see, the decode method returns void; the data is decoded into the instance on which decode is invoked and an exception is thrown if an error occurs. All ASN1C exceptions are derived from *Asn1Exception*. See the section on exceptions for a complete list and description of the various ASN1C exceptions that can be thrown.

Procedure for Calling Java MDER Decode Methods

The general procedure to decode an ASN.1 MDER message involves the following three steps:

1. Create a decode buffer on the message to be decoded

2. Invoke the decode method

3. Process the decoded data values

The first step is the creation of a decode buffer. *Asn1MderDecodeBuffer* has a constructor that accepts messages stored in a byte array. It also has a constructor that accepts a java.io.InputStream so that messages may be streamed from various sources, such as from a file.

The second step is to invoke the generated decode method. The calling arguments were described earlier.

The final step is to apply your application-specific processing to the data. All data is contained within public member variables so access is quite easy.

A complete example showing how to invoke a decode method is as follows:

```
try {
```

```
        // Step 1: create a decode buffer for the message to be decoded.
        // This example will use a file input stream to decode a message
        // in a binary file.

        // Create an input file stream object
        FileInputStream ins = new FileInputStream (filename);

        // Create a decode buffer object
        Asn1MderDecodeBuffer decodeBuffer = new Asn1MderDecodeBuffer (ins);

        // Step 2: create an object of the generated type and invoke the
        // decode method..
        PersonnelRecord personnelRecord = new PersonnelRecord ();
        personnelRecord.decode (decodeBuffer);

        // Step 3: process the data
        if (trace) {
            System.out.println ("Decode was successful");
            personnelRecord.print (System.out, "personnelRecord", 0);
        }
    }
    catch (Exception e) {
        System.out.println (e.getMessage());
        e.printStackTrace();
        return;
    }
```

Reuse of Java Decoding Objects

The sample above showed the MDER decoding of a single message. In a typical application, a loop would be involved to decode a series of messages.

A single decode buffer can be used to process a stream of messages. If the decode buffer is created using an input stream that contains a series of messages (for example, a file containing multiple records, or a communications device), you can repeatedly invoke the MDER decode method on the given message type.

Note that you can also use the same instance of your message type for repeated decoding, rather than creating a new object and leaving the old one to be garbage collected. Nothing special needs to be done to do this. The generated decode method will automatically call the internal *init()* method before decoding to make sure all items are reset to their starting state.

In the example above, all that would need to be done to decode a series of personnel records is the inclusion of a loop after the PersonnelRecord object was created in step 2:

```
    for (;;) {
        personnelRecord.decode (decodeBuffer);

        if (trace) {
            System.out.println ("Decode was successful");
            personnelRecord.print (System.out, "personnelRecord", 0);
        }
    }
```

Chapter 17. Table Constraint Processing

The ASN1C Java code generator can generate code to process ASN.1 table constraints as specified in the X.681 and X.682 ASN.1 standards. This code is generated through the use of the **-tables** option. This instructs the compiler to generate additional methods and tables to allow multi-level message types specified using table constraints to be encoded or decoded with a single method call.

Special code is generated for the CLASS, Information Object, and Information Object Set items to create the table necessary to for table constraint processing. Then additional encode and decode methods are generated that use these tables to branch to the multiple message levels.

CLASS specification

NOTE: Class code generation is done only when -tables is specified.

This additional code is generated to support the processing required to verify table constraints, which is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

The Java class generated to model an ASN.1 class contains member variables for each of the fields within the class. To create an instance of this class, an information object is required to populate these variables with the values defined in the ASN.1 information object specification.

Java code will be generated for each ASN.1 CLASS definition in a separate Java source file containing a Java class corresponding to the ASN.1 CLASS definition. The name of the source file and class is of the following format:

```
<ClassName>.java
```

In this definition, *<ClassName>* would be replaced with the name of the ASN.1 CLASS for which this file is generated.

Data Member Generation

For each of the following ASN.1 CLASS fields, a corresponding member variable is generated in the Java class definition:

For a value field:

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

For a type field:

```
public Asn1Type <FieldName>;
```

For an information object field:

```
public <ClassName> <FieldName>;
```

For an information object set field:

```
public <ClassName> <FieldName>;
```

where:

<FieldName> is replaced with the name of the field.

<TypeName> is replaced with the generated runtime Java classname for the ASN.1 Type.

<ClassName> is replaced with the name of the information object class.

For a type field definition, an element with type *Asn1Type* is generated which is the base class for all types in the Java runtime package. A type field can hold a value of any type.

Method and Constructor Generation

Each generated Java class will have two constructors. The first constructor will be the default constructor. This will initialize each member variable value to *null*. The second constructor will accept values for all the data members.

Example

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

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

A file named ATTRIBUTE.java is generated with following definition:

```
public class ATTRIBUTE {
  public Asn1Type Type;
  public Asn1ObjectIdentifier id;

  public ATTRIBUTE() {
     Type = null;
     id = null;
  }

  public ATTRIBUTE(
     Asn1Type Type_,
     Asn1ObjectIdentifier id_
     ) {
     Type = Type_;
     id = id_;
  }
}
```

NOTE: If the ASN.1 type name is same as the ASN.1 class name (ignoring case) in a single module definition, then the ASN.1 class name will be changed to following:

```
<ClassName>_CLASS
```

In this definition, <ClassName> would be replaced with the name of the ASN.1 CLASS and the literal token "_CLASS" would be appended.

For example:

```
Test DEFINITION ::= BEGIN
   Attribute ::= INTEGER
   ATTRIBUTE ::= ABSTRACT-SYNTAX
END
```

ASN1C will change the ATTRIBUTE class name to ATTRIBUTE_CLASS to avoid conflicts with the Attribute type.

This automated feature will help users to successfully compile the generated code without having to manually change the name via a configuration file setting.

Additional Java classes are generated to create types for fields within the class definitions as follows:

1. New type assignments are created for TypeField type definitions as follows:

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

Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of the field. *Type* is the type definition in the ASN.1 CLASS's TypeField.

This type is used as a defined type in the information object definition for absent values of the TypeField. It is also useful for the user to generate a value for a related OpenType definition in a table constraint.

2. New type assignments are created for ValueField or ValueSetField type definitions if the type is with a constraint definition and/or the type is Sequence / Set / Choice / Sequenceof / SetOf definition.

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

Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of the ValueField or ValueSetField. *Type* is the type definition in The ASN.1 CLASS's ValueField or ValueSetField. This type will appear as a defined type in the ASN.1 CLASS's ValueField or ValueSetField.

This new type assignment is used for compiler internal code generation purpose. It is not designed for use by the end user.

3. New value assignments are created for ValueField default value definitions as follows:

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

Here *ClassName* is replaced with name of the Class Assignment and *FieldName* is replaced with name of the ValueField. *Value* is the default value in the ASN.1 CLASS's ValueField & *Type* is the type in the ASN.1 CLASS's ValueField.

This value is used as a defined value in the information object definition for an absent value of the field. This new value assignment is used for compiler internal code generation purpose. It is not designed for use by the end user.

ABSTRACT-SYNTAX

The ASN.1 ABSTRACT-SYNTAX class is a useful class definition used to declare the top-level protocol data units (PDU's) defined within a specification. The class is described using the following ASN.1 definition:

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

ASN1C is used to create a meta-definition for this structure. The definition is stored in the file *Asn1AbstractSyntax.java* (or *Asn1XerAbstractSyntax.java* for XER). An object created from the resulting Java class is populated just like any other compiler-generated structure for working with ASN.1 data.

TYPE-IDENTIFIER

The ASN.1 TYPE-IDENTIFIER class is a useful class definition for uniquely identifying typed data at runtime. The class is described using the following ASN.1 definition:

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

The ASN.1 compiler is used to create a meta-definition for this structure. The definition is stored in the file *Asn1TypeIdentifier.java* (or *Asn1XerTypeIdentifier.java* for XER). An object created from the resulting Java class is populated just like any other compiler-generated structure for working with ASN.1 data.

Information Object

NOTE: Information Object code generation is only done when the -tables option is selected.

This additional code is generated to support the processing required to verify table constraints, which is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

Information Object code will be generated in a Java source file with a special class to hold the values. The name of the source file and class is of the following format:

```
_<ModuleName>Values.java
```

In this definition, *<ModuleName>* would be replaced with the name of the ASN.1 module in which the values are defined.

For each Information Object defined within a specification, a Java constant is generated which is an instance of the ASN.1 CLASS definition for the object. Each Information Object constant calls the Class constructor with the field value specified in the ASN.1 information object definition.

If the ASN.1 CLASS field is optional and the field value is absent in the Information Object definition, then its corresponding member variable will be initialized to "*null*". If the ASN.1 CLASS field has a default value and its field value is absent in the Information Object, then the generated code for the Information Object will set the Class field's value to the default value.

ASN.1 definition:

```
<name> <ClassName> ::= <InfoObject>
```

Generated Java constants:

```
public final static <ClassName> <name> =
    new <ClassName> (<InfoObject values>>);
```

For example, consider the following Information Object declaration for the above ATTRIBUTE class:

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

This would result in the following Java constant being generated:

```
public static final ATTRIBUTE name =
    new ATTRIBUTE (
        new Asn1VisibleString(),
        new Asn1ObjectIdentifier(new int[]{0, 1, 1}));
```

NOTE: The following new Type Assignment is created for each TypeField's type definition if the type is one of the following ASN.1 built-in types: Sequence / Set / SequenceOf / SetOf / Choice / Constrained Type / Enumerated Type / NamedInteger Type / NamedBitList Type / ParameterizedType:

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

Here *ObjectName* is replaced with name of the Object Assignment. If Object is defined in ObjectSet, then *ObjectName* is replaced with the name of the ObjectSet Assignment. *FieldName* is replaced with name of this type field. *Type* is the type definition in Object's typefield.

This type is used as Defined Type in the information object definition for type field. It is also useful for the user to generate value for related OpenType definition in table constraint.

Information Object Set

NOTE: Information Object Set code generation is only done when the -tables option is selected.

This additional code is generated to support the processing required to verify table constraints which is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

Information Object code will be generated in a Java source file with a special class to hold the values. The name of the source file and class is of the following format:

```
_<ModuleName>Values.java
```

In this definition, *<ModuleName>* would be replaced with the name of the ASN.1 module in which the Information Object Sets are defined.

Each Information Object Set specification causes a Java constant to be generated containing an array of Information Object values. Each object in the array is an instance of the equivalent Java class representing the corresponding ASN.1 information object

As of this writing, a static array is used to hold the objects, but this could be changed to something like a linked list or hash.

ASN.1 definition:

```
<name> <ClassName> ::= { <Information Object1> | <Information Object2> }
```

Generated Java constants:

```
public static final <ClassName> <name> =
    new <ClassName> {<Information Object1>, <Information Object1> };
```

For example, consider the following Information Object Set declaration for above ATTRIBUTE definition:

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

This would result in the following Java constant being generated:

```
public static final ATTRIBUTE[] SupportedAttributes =
    new ATTRIBUTE[] {
        _TestValues.name,
        _TestValues.commonName
};
```

Generated Information Object Table Structure

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 array for storing information on an encoded message fragment for processing at the next level.

The ASN1C compiler is capable of generating code in one of two forms for information in an object specification:

1. **Simple form**: in this form, references to variable type fields within standard types are simply treated as open types and an open type placeholder is inserted.

2. **Table form:** in this form, all of the classes, objects, and object sets within a specification result in the generation of code for parsing and formatting the information field references within standard type structures.

The second form is selected by specifying the –tables command line option.

To better understand the support in this area, the individual components of Information Object specifications are examined. We begin with the "CLASS" specification that provides a schema for Information Object definitions. A sample class specification is as follows:

```
OPERATION ::= CLASS {
    &operationCode CHOICE { local INTEGER,
                    global OBJECT IDENTIFIER }
    &ArgumentType,
    &ResultType,
    &Errors ERROR OPTIONAL
}
```

Users familiar with ASN.1 will recognize this as a simplified definition of the ROSE OPERATION MACRO using the Information Object format. When a class specification such as this is parsed, information on its fields is maintained in memory for later reference. In the simple form of code generation, the class definition itself does not result in the generation of any corresponding Java code. It is only an abstract template that will be used to define new items later on in the specification. In the table form, a Java container class is generated to hold the Information Object instances of the ASN.1 CLASS.

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,
    argumentOPERATION.&ArgumentType
}
```

This is a very simple case that purposely omits a lot of additional information such as Information Object Set constraints that are typically part of definitions such as this. The reason this information is not present is because we are just interested in showing the items that the compiler is concerned with. We will use this type to demonstrate the simple form of code generation. We will then add table constraints and discuss what changes when the –tables command line options is used.

The opcode field within this definition is an example of a *fixed type* field reference. It is known as this because if you go back to the original class specification, you will see that operationCode is defined to be of a specific type (namely a choice between a local and global value). The generated typedef for this field will contain a reference to the type from the class definition.

The argument field is an example of a *variable type* field. In this case, if you refer back to the class definition, you will see that no type is provided. This means that this field can contain an instance of any encoded type (note: in practice, table constraints can be used with Information Object Sets to limit the message types that can be placed in this field). The generated typedef for this field contains an "open type" (Java *Asn1OpenType* class) reference to hold a previously encoded component to be specified in the final message.

Simple Form Code Generation

In the simple form of information object code generation, the Invoke type above would result in the following Java typedefs being generated:

```
public class Invoke extends Asn1Type {
    public Asn1Integer invokeID;
    public OPERATION_operationCode opcode;
    public Asn1OpenType argument;


    ...
}
```

The following would be the procedure to add the Invoke header type to an ASN.1 message body:

1. Encode the body type

2. Get the message bytes and length of the encoded body

3. Plug the bytes into the "data" argument of the open type constructor in the Invoke type variable.

4. Populate the remaining Invoke type fields.

5. Encode the Invoke type to produce the final message.

In this case, the amount of code generated to support the information object references is minimal. The amount of coding required by a user to encode or decode the variable type field elements, however, can be rather large. This is a trade-off that exists between using the compiler generated table constraints solution (as we will see below) and using the simple form.

Table Form Code Generation

If we now add table constraints to our original type definition, it might look as follows:

```
Invoke ::= SEQUENCE {
    invokeID INTEGER,
    opcode OPERATION.&operationCode ({My-ops}),
    argument OPERATION.&ArgumentType ({My-ops}{@opcode})
}
```

The "{My-ops}" constraint on the opcode element specifies an information object set (not shown) that constrains the element value to one of the values in the object set. The {My-ops}{@opcode} constraint on the argument element goes a step further – it ties the type of the field to the type specified in the row that matches the given opcode value. ASN1C generates an in-memory table for each of the items in the information object sets defined in a specification. In the example above, a table would be generated for the My-ops information object set. The code generated for the type would then use this table to verify that the given items in a structure that reference this table match the constraints. The Java type generated for the SEQUENCE above when –tables is specified would be as follows:

```
public class Invoke extends Asn1Type {
  public Asn1Integer invokeID;
  public OPERATION_operationCode opcode;
  public Asn1Type argument;


  ...
}
```

This is almost identical to the type generated in the simple case. The difference is that *ASN1Type* is used instead for the argument element instead of *ASN1OpenType*. This type is defined as the base class for all the generated ASN.1 types. It holds the value to be encoded or decoded. The way a user Would use this to encode a value of this type is as follows:

1. Populate a variable of the type to be used as the argument to the invoke type.

2. Assign it to the argument member variable in the structure above.

3. Populate the remaining Invoke type fields.

4. Encode the Invoke type to produce the final message.

Note that in this case, the intermediate type does not need to be manually encoded by the user. The generated encoder has logic built-in to encode the complete message using the information in the generated tables.

Additional Code Generated for the -tables Option

Following additional code is generated for type definition when the -tables command line option is used. The code generated to support table constraints is intended for use only in compiler-generated code. Therefore, it is not necessary for the average user to understand the mappings in order to use the product. The information presented here is informative only to provide a better understanding of how the compiler handles table constraints.

Additional equals() method will be generated for Sequence, Set, Sequence Of, Set Of or Choice types if required for table constraint processing. This method will be an implementation of Asn1Type.equals() virtual method. These methods are used by the generated code to verify that data in a generated structure to be encoded (or data that has just been decoded) matches the table constraint values.

An additional table constraint check method is also generated for each type that contains table constraints. These functions have the following prototypes:

BER/DER:

```
void checkTC (boolean decode);
```

PER:

```
void checkTC (boolean decode, boolean aligned);
```

The *decode* argument is used to decide if this method is to used for encoding or decoding. The *aligned* argument is for PER and specified whether aligned or unaligned encoding/decoding is in effect.

The purpose of these methods is to verify that the fixed values within the table constraints are what they should be and to encode or decode the open type fields using the encoder or decoder methods from the *Asn1Type* objects assigned to the given table row. Calls to these functions are automatically built into the standard encode or decode functions for the given type. They should be considered hidden functions not for use within an application that uses the API.

The *checkTC* method will have different logic for relative and simple table constraints. The logic to invoke this method is as follows:

On the encode side:

Relative Table Constraint:

1. The table constraint key is searched in the object set array to find the class object for the data in the populated type variable to be encoded.

2. If the key element value is NOT found and the table constraint object set is extensible, the *checkTC* method will do no further processing (i.e. a value field match will not be performed). The user will have had to populate the type field using an *Asn1OpenType* object in order for it to be decoded because the generated table contains no information on how to encode the value.

3. If the key element value is found, the method will verify all fixed type values match what is defined in the key row of the object set and will also verify that the type of any variable type fields matches the expected type.

4. If the key element value is not found in the table (or object set) and the objectset is NOT extensible, then a table constraint violation exception will be thrown.

Simple Table Constraint:

1. The *checkTC* method will verify that all of the fixed type values match what is defined in the table constraint object set. If the element value does not exist in the table (or object set) and the object set is NOT extensible, then a table constraint violation exception will be thrown.

After the *checkTC* method call, the normal encode logic is performed.

For decoding, the logic is reversed:

The normal decode logic is performed first to populate the standard and open type fields in the generated structure. After that, the *checkTC* method is invoked to perform following table constraint checks:

Relative Table Constraint:

1. The table constraint key is searched in the object set array to find the class object for the data in the populated type variable to be encoded.

2. If the key element value is NOT found and the table constraint object set is extensible, the *checkTC* method will do no further processing (i.e. a value field match will not be performed) and the variable type fields will be stored as open types (i.e. as instances of Java *Asn1OpenType* classes). The user will be responsible for further decoding of the open type value.

3. If the key element value is found, the *checkTC* method will verify all fixed type values match what is defined in the key row of the object set and will fully decode all type fields according to the key row type and store the resulting decoded type in the *ASN1Type* fields.

4. If the key element value is NOT found in the table (or object set) and the object set is NOT extensible, then a table constraint violation exception will be thrown.

Simple Table Constraint:

1. This function will verify all the fixed type values match what is defined in the table constraint object set. If an element value does not exist in the table (or object set) and the object set is NOT extensible, then a table constraint violation exception will be thrown.

Populating OpenType Variables for Encoding

When -tables option is used, open type fields are generated as *Asn1Type* fields. The general procedure to populate the value for these fields is as follows:

1. Check the possible Type in ObjectSet from index element value.

2. Populate the value for this type and assign it to the open type member variable.

3. Follow the common encode procedure.

A complete example showing how to assign open type values when table constraint code is generated is as follows:

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

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

commonName ATTRIBUTE ::= {
    WITH SYNTAX   INTEGER
    ID            { 0 1 2 } }

SupportedAttributes ATTRIBUTE ::= { name | commonName }

Invoke ::= SEQUENCE {
    opcode ATTRIBUTE.&id ({SupportedAttributes}),
    argument ATTRIBUTE.&Type ({SupportedAttributes}{@opcode})
}
```

In the above example, the *Invoke* type contains a relative table constraint. Its element *opcode* refers to the ATTRIBUTE class's *id* field and the *argument* element refers to ATTRIBUTE class's *Type* field. The *opcode* element is the index element into the *{SupportedAttributes}* information object set. The *argument* element is an open type but its type must match that specified at the location in the *{SupportedAttributes}* information object set indexed by *opcode*.

In this example, *opcode* can have only two possible values { 0 1 1 } or { 0 1 2 }. If the *opcode* value is { 0 1 1} then *argument* must be a value of type *VisibleString*. If the *opcode* value is { 0 1 2 } then *argument* will have an INTEGER value. Any other value of the *opcode* element will be a violation of the Table Constraint.

If the *SupportedAttributes* object set was extensible (in this example, it is not), then the *argument* element can be a value of any type. In this case, if the user is using an index element value outside the object set, then the user will have to encode the *argument* element as an *Asn1OpenType*.

The following sample code populates the open type value:

```
// Step 1: populate the "Invoke" type with data
Invoke pdu = new Invoke();
```

```
    pdu.opcode = new Asn1ObjectIdentifier(new int[]{0, 1, 1});
    pdu.argument = new Asn1VisibleString("objsys");
    // note: opcode value is {0 1 1 }, so argument must be
    // Asn1VisibleString type

    // note: the rest of the encode method will be same as general
    // PER/DER/BER encoding rules

    // Step 2: Create a message buffer object.
    Asn1PerEncodeBuffer encodeBuffer = new Asn1PerEncodeBuffer();

    // Step 3: Invoke the encode method. Note that it must be done
    // from within a try/catch block..
    try {
        pdu.encode (encodeBuffer);
        if (trace) {
            System.out.println ("Encoding was successful");
            System.out.println ("Hex dump of encoded record:");
            encodeBuffer.hexDump ();
            System.out.println ("Binary dump:");
            encodeBuffer.binDump ("Invoke");
        }
        // Step 3: Access the encoded message component. In this
        // case, we use methods in the class to write the component
        // to a file and output a formatted dump to the message.dmp
        // file..
        // Write the encoded record to a file
        encodeBuffer.write (new FileOutputStream (filename));
        // Generate a dump file for comparisons
        encodeBuffer.hexDump
            (new PrintStream (new FileOutputStream ("message.dmp")));
        // We can also directly access the buffer as follows:
        byte[] buffer = encodeBuffer.getBuffer();
        int msglen = encodeBuffer.getMsgByteCnt();
    }
    catch (Exception e) {
        System.out.println (e.getMessage());
        e.printStackTrace();
        return;
    }
```

The important thing to note is that not much changes from the normal procedure. The only significant difference is that now the argument field can be directly populated with an instance of its target type. Without table constraint checking logic, this value would have to have been first encoded and then placed in an *Asn1OpenType* container object.

Decoding Types with Table Constraints

The general procedure to decode an ASN.1 message with table constraints is the same as without table constraints. The only difference is that after decoding, variable type fields will be replaced with instances of the actual types they are specified to contain in the associated object set instead of with generic *Asn1OpenType* fields.

Chapter 18. Generated Print Methods

The *-print* option causes print methods to be generated. These functions can be used to print the contents of variables of generated types. A print method is generated in each of the generated Java source files.

The *-prtToStr* option causes ASN1C to generate print methods whose primary argument is a StringBuilder object. This provides a somewhat higher-performance means for users to write data to a string (instead of using StringWriter or another stream-based method).

Generated Java Print Method Format and Calling Parameters

The print method is provided in two forms with the following signatures:

```
public void print (PrintWriter out, String varName, int level)
```

and

```
public void print (PrintStream out, String varName, int level)
```

The fundamental difference between them is the *out* argument. In the first method, *out* is a *PrintWriter* object, and in the second method, it is a *PrintStream* object. In both cases, *out* specifies a stream to which output should be written. For text output (such as this method performs), *PrintWriter* is preferred over *PrintStream* and will usually perform better. However, the *PrintStream* form is provided for backwards compatibility and for writing to standard output (i.e. *System.out*).

The *varName* argument is used to specify the top-level variable name of the item being printed. Normally, this would be set to the same name as the variable declared in your program that holds the object being printed. For example, if you declared a variable called *personnelRecord* to hold a *PersonnelRecord* object, the *varName* object would be set to "personnelRecord".

The *level* argument is used to specify the indentation level for printing nested types. The user would always want to set this to zero at the outer-level.

For example, the call to print the personnelRecord from the previous examples would be as follows:

```
personnelRecord.print (System.out, "personnelRecord", 0);
```

The output would be formatted as follows:

```
personnelRecord {
    name {
        givenName = 'John'
        initial = 'P'
        familyName = 'Smith'
    }
    number = 51
    title = 'Director'
    dateOfHire = '19710917'
    nameOfSpouse {
        givenName = 'Mary'
        initial = 'T'
        familyName = 'Smith'
```

```
    }
    children[0] {
        name {
            givenName = 'Ralph'
            initial = 'T'
            familyName = 'Smith'
        }
        dateOfBirth = '19571111'
    }
    children[1] {
        name {
            givenName = 'Susan'
            initial = 'B'
            familyName = 'Jones'
        }
        dateOfBirth = '19590717'
    }
}
```

Generated Java Print-To-String Method Format and Calling Parameters

The print method is provided with the following signature:

```
public void print (StringBuilder sb, String varName, int level)
```

The method calls are made in much the same way as the calls detailed in the previous section, and users are recommended to read them if questions persist. The output format is exactly the same as well.

Users should note that StringBuilder is unsynchronized, and so, too, are the generated print-to-string methods. Access to the StringBuilder should be locked properly (or one StringBuilder per thread should be allocated) to ensure that output is in order.

Chapter 19. Generated Copy Methods

When `-copy` is specified on the command line, ASN1C will generate overridden `clone` methods in Java code. These methods perform a deep copy.

It is noted by many that writing a good `clone()` method is non-trivial, and we found this to be true. In general, users can expect `clone` to act as it is specified in the Java documentation. The sole exception to this is that `Asn1Enumerated` instances are immutable and will return themselves rather than a new instance.

The `clone` method is provided in the following form:

```
@Override
public Object clone()
    throws CloneNotSupportedException
{
    /* ... */
}
```

When users generate code compatible with the CLDC or Java Micro Edition, the `@Override` annotation is removed.

Chapter 20. Generated Compare Methods

The *-compare* command line option causes an equals method to be added to each generated class. The signature of this method is as follows:

```
public boolean equals (ClassName rhs);
```

where *ClassName* is the name of the generated class to which the member function belongs. The method returns a boolean result of true if the object instances are equal and false if not.

Note that for classes extended from the `Asn1Choice` class, no equals member function is generated. This is because the *Asn1Choice* class already has an *equals* member function, which is inherited by classes extending the *Asn1Choice* class.

Chapter 21. Generated Metadata Methods

In version 6.8, ASN1C adds the ability to generate methods to inspect the elements in a `SEQUENCE` or `SET`. Using the `-genmetadata` option, users can generate the following methods:

- `public static boolean isRequired (String elemName)`
    `throws Asn1InvalidElemException`

- `public static boolean getValueRange (String elemName, Asn1ValueRange range)`

These methods provide access to syntactic information present in the schema used during code generation: whether the elements in a `SEQUENCE` or `SET` are required or optional, and what, if any, value range may be applied to the elements. The following sections describe these methods in detail.

isRequired

The `isRequired` method has the following signature:

```
public static boolean isRequired (String elemName)
    throws Asn1InvalidElemException
```

The method returns `true` when the element given by `elemName` is required to be present in the given `SEQUENCE` or `SET` and `false` when it is not. In the event that the named element is not a member of the structure, the `Asn1InvalidElemException` is thrown.

The following ASN.1, taken from a slightly-modified version of the Employee sample program, provides a good example:

```
PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
    name                Name,
    title           [0]     IA5String,
    number              EmployeeNumber,
    dateOfHire      [1]     Date,
    nameOfSpouse    [2]     Name,
    children        [3]     IMPLICIT SEQUENCE OF ChildInformation,
    salary          INTEGER (0..10000) OPTIONAL
}
```

There are seven total elements: one is optional and the others are required. The generated code is straightforward:

```
public static boolean isRequired (String elemName)
    throws Asn1InvalidElemException
{
    if (elemName.equals("salary")) {
        return false;
    }
    else if (elemName.equals("name")) {
        return true;
    }
    else if (elemName.equals("number")) {
        return true;
    }
    else if (elemName.equals("title")) {
        return true;
```

```
      }
      else if (elemName.equals("dateOfHire")) {
         return true;
      }
      else if (elemName.equals("nameOfSpouse")) {
         return true;
      }
      else if (elemName.equals("children")) {
         return true;
      }

      throw new Asn1InvalidElemException(elemName);
   }
```

getValueRange

The getValueRange method has the following signature:

```
   public static boolean getValueRange (String elemName, Asn1ValueRange range)
```

The method returns true when the value range range has been set by the method and false when it has not been set. No exceptions are thrown by this method.

The following ASN.1, taken from a slightly-modified version of the Employee sample program, provides a good example:

```
   PersonnelRecord ::= [APPLICATION 0] IMPLICIT SET {
      name                Name,
      title       [0]     IA5String,
      number              EmployeeNumber,
      dateOfHire  [1]     Date,
      nameOfSpouse [2]    Name,
      children    [3]     IMPLICIT SEQUENCE OF ChildInformation,
      salary      INTEGER (0..10000) OPTIONAL
   }
```

The generated code is straightforward:

```
   public static boolean getValueRange (String elemName, Asn1ValueRange range)
   {
      if (elemName.equals("salary")) {
         range.min = 0;
         range.max = 10000;
         return true;
      }
      return false;
   }
```

Chapter 22. Generated Sample Programs

The *–writer* and *-reader* options cause writer and reader sample programs to be generated.

The writer program contains sample code to populate and encode an instance of ASN.1 data. The main purpose is to provide a code template to users for writing code to populate objects. This is quite useful to users because generated classes can become very complex as the ASN.1 schemas become more complex. The writer code also shows users how to instantiate an encode buffer object and how to use encode functions. The writer program writes the encoded data to a file. If the writer program is generated by using both *-writer* and *-getset* options, then the generated writer program uses getset functions to populate data.

The reader program on the other hand reads the encoded data from a file. It shows users how to use a decode buffer object to decode data and populate the corresponding class object. On successful decode, it prints the decoded data to standard output.

Chapter 23. Generated Build Script

The *-genbuild* option causes a build script to be generated. This script can be used to Java compile the generated source files.

For Windows, the *-w32* command line option should be specified along with *-genbuild* to generate a DOS batch file (.bat). This file is named *build.bat*.

For Linux/UNIX, a shell script is generated. The name of this file is *build.sh*.

When a build script is generated, it is assumed that the ASN1C project exists within the ASN1C installation directory tree. The generation logic tries to determine the root directory of the installation by traversing upward from the project directory in an attempt to locate the *java* subdirectory which is assumed to be the installation root directory. If the project is located outside of the ASN1C hierarchy, the user can set the *OSROOTDIR* environment variable to point at the root directory.

If the root directory is located successfully, the generated build script will use that directory; however, if the compiler fails to find the installation root directory, it will use @ROOT_DIR@ instead and print an error message. Users will have to manually replace @ROOT_DIR@ with the actual compiler installation root directory. Also, for the *-xer* or *-xml* option along with *-genbuild*, an XML parser is required. The compiler will try to locate a parser and use it if found. However, if a parser is not found, then the compiler will use @XERCES_ROOT@ instead of the parser root directory. An error message will be printed and the user will update the file accordingly.

Chapter 24. Event Handler Interface

The *–events* command line switch causes hooks for user-defined event handlers to be inserted into the generated Java decode methods. What these event handlers do is up to the user. They fire when key message-processing events occur during the course of parsing an ASN.1 message. They are similar in functionality to the Simple API for XML (SAX) that was described earlier 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:
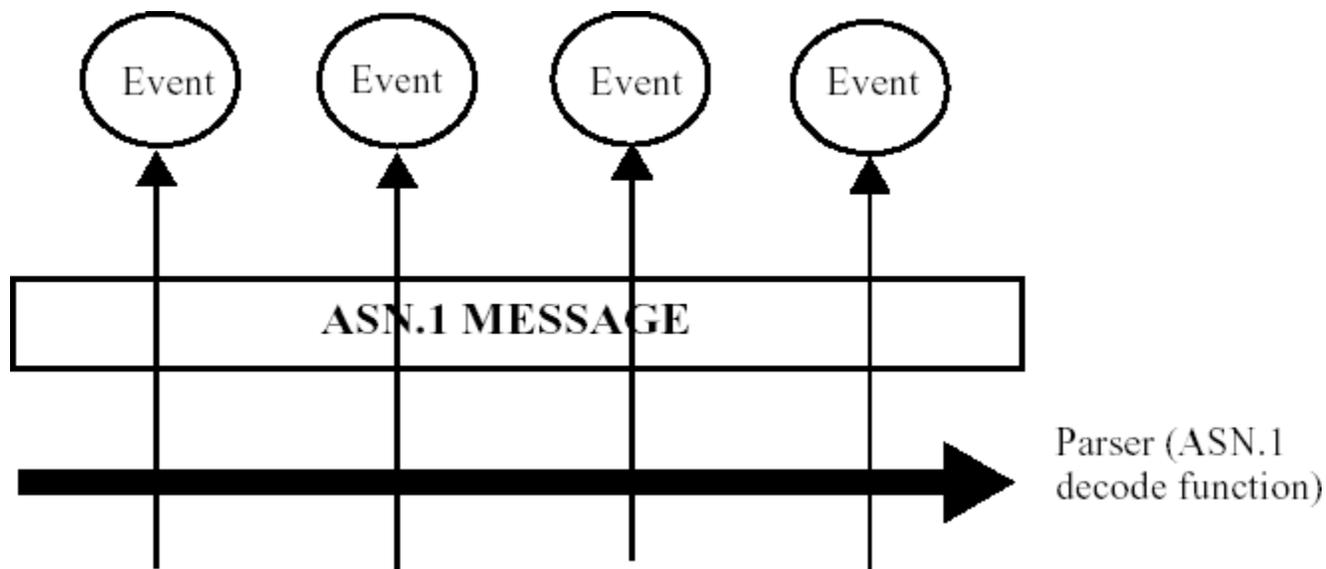
The events are defined to be significant actions that occur during the parsing process. We will define the following events that will be passed to the user when an ASN.1 message is parsed:

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. **characters method** – This method is defined to pass all of the different types of primitive values that are encountered when parsing a message. The primitive values are passed out in a stringified form.

The methods corresponding to these events are defined in *Asn1NamedEventHandler* interface.

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 or DER message. The end method is invoked after the contents of the field are processed. The signature of these methods is as follows:

```
void startElement (String name, int index);
```

```
void endElement (String name, int index);
```

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.

The *characters* method is used to pass out ASN.1 primitive data. This is a departure from the C++ event handler methodology in which separate methods are defined for all of the different data types. This implementation is more closely aligned with the standard SAX implementation for XML. The reason it is done this way in Java and not C++ is because it is much easier to stringify values. Since memory management is built-in to Java, it is easy to create a string and pass it out. This is a problem in C++ because it becomes a performance issue if too many malloc's are done and it also places a burden on the user to free the memory for the allocated strings.

The *signature* for the characters method is as follows:

```
void characters (String svalue, short typeCode);
```

The *svalue* argument contains the stringified value. The format of this value is ASN.1 value notation for the value as defined in the X.680 standard. The *typeCode* argument contains an identifier that specifies the ASN.1 type of the value. The identifier corresponds to the universal identifier values (the ID number in the universal tags) for each of the primitive data types. The only exception to this rule is that the identifier 99 was added to represent an Open Type construct. Constants for all of the identifier values are provided in the *Asn1Type* class. See the javadoc documentation for this class for a list of the constants.

How to Use It

To define event handlers, two things must be done:

1. One or more new classes must implement the *Asn1NamedEventHandler* interface.

2. Objects of these classes must be created and registered prior to calling the generated decode method for a particular type.

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 BER message. Then we will show a simple XML converter class that will convert the data in a BER message to XML.

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 *java/sample_ber/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.

We will first create a class called *PrintHandler* that implements the *Asn1NamedEventHandler* interface and handles the formatted printing of the data. The rule for the implementation of interfaces is that you must provide an implementation for each of the methods listed. That is it. You can add as many additional methods, member variables, etc., that you like.

The *PrintHandler* implementation that we created is as follows:

```
class PrintHandler implements Asn1NamedEventHandler {
```

```
    protected String mVarName;
    protected int mIndentSpaces = 0;

    public PrintHandler (String varName) {
        mVarName = varName;
        System.out.println (mVarName + " = {");
        mIndentSpaces += 3;
    }

    public void startElement (String name, int index) {
        indent();
        System.out.print (name);
        if (index >= 0)
            System.out.print ("[" + index + "]");
        System.out.println (" = {");
        mIndentSpaces += 3;
    }

    public void endElement (String name, int index) {
        mIndentSpaces -= 3;
        indent ();
        System.out.println ("}");
    }

    public void characters (String svalue, short typeCode) {
        indent ();
        System.out.println (svalue);
    }

    private void indent () {
        for (int i = 0; i < mIndentSpaces; i++)
            System.out.print (" ");
    }
}
```

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 class is the implementation of the methods defined in the interface.

We implement these methods as follows:

```
public void startElement (String name, int index) {
    indent();
    System.out.print (name);
    if (index >= 0)
        System.out.print ("[" + index + "]");
    System.out.println (" = {");
    mIndentSpaces += 3;
}
```

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. Logic is also present to check the index value to see if it is zero or greater. If it is, an array subscript is added to the element name.

In *endElement*, we simply terminate our brace block as follows:

```
public void endElement (String name, int index) {
   mIndentSpaces -= 3;
   indent ();
   System.out.println ("}");
}
```

The *characters* method simply indents and prints the stringified value:

```
public void characters (String svalue, short typeCode) {
   indent ();
   System.out.println (svalue);
}
```

That completes the *PrintHandler* class implementation.

Next, we need to create an object of the class and register it prior to invoking the decode method. In the Reader.java program, the following lines do this:

```
// Register event handler object

PrintHandler printHandler = new PrintHandler ("personnelRecord");
decodeBuffer.addNamedEventHandler (printHandler);
```

The *addEventHandler* method defined in the *Asn1DecodeBuffer* 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 XML Converter Class

The ASN1C XML Encoding Rules (XER) encode and decode capabilities were presented in an earlier section of this document. An alternate way to create an XML document from ASN.1 data is through the event handler interface.

It turns out that with event handlers, this conversion is fairly easy. As the handler events fire, all of the required symbolic data is passed out to generate an XML document. The programmer is free to massage this data any way he or she wants to comply with whatever DTD or XML Schema is in use.

The *ToXML* sample program demonstrates the conversion of ASN.1 data to XML using event handlers. The sample is not intended to be a robust implementation – it is merely designed to provide guidance in how one would go about doing this transformation.

The sample program can be found in the *java/sample_ber/ToXML* subdirectory within the ASN1C installation. The complete class definition for the *XMLHandler* class is as follows:

```
class XMLHandler implements Asn1NamedEventHandler {
   protected String mVarName;
   protected int mIndentSpaces = 0;

public XMLHandler (String varName) {
   mVarName = varName;
   System.out.println ("<" + mVarName + ">");
   mIndentSpaces += 3;
}

public void startElement (String name, int index) {
   indent();
   System.out.println ("<" + name + ">");
   mIndentSpaces += 3;
}

public void endElement (String name, int index) {
   mIndentSpaces -= 3;
   indent ();
   System.out.println ("</" + name + ">");
}

public void characters (String svalue, short typeCode) {
   indent ();
   String typeName = new String (Asn1Type.getTypeName(typeCode));
   typeName.replace (' ', '_');
   System.out.print ("<" + typeName + ">");
   System.out.print (svalue);
   System.out.println ("</" + typeName + ">");
}

public void finished () {
   System.out.println ("</" + mVarName + ">");
}

private void indent () {
   for (int i = 0; i < mIndentSpaces; i++)
      System.out.print (" ");
   }
}
```

This is very similar to the *PrintHandler* class defined earlier. The *startElement* method simply opens an XML element block:

```
public void startElement (String name, int index) {
   indent();
   System.out.println ("<" + name + ">");
   mIndentSpaces += 3;
```

```
    }
```

The *endElement* method closes it:

```
public void endElement (String name, int index) {
    mIndentSpaces -= 3;
    indent ();
    System.out.println ("</" + name + ">");
}
```

The *characters* method outputs the data with a type wrapper:

```
public void characters (String svalue, short typeCode) {
    indent ();
    String typeName = new String (Asn1Type.getTypeName(typeCode));
    typeName.replace (' ', '_');
    System.out.print ("<:" + typeName + ">");
    System.out.print (svalue);
    System.out.println ("</:" + typeName + ">");
}
```

This illustrates the use of the *typeCode* argument for obtaining information on the ASN.1 type of the data. Note that this is a simplified version of an XER formatting method. A true implementation would need to do some massaging of the stringified data to fit the XER rules which, in general, do not follow the ASN.1 value formatting rules. The implementation would also need some logic to check if the type wrapper should be output or not; it is not always done in certain cases.

Finally note the constructor and *finished* method. The constructor prints out the outer-level wrapper tag. Since Java does not have destructors, a finished method is defined to terminate this tag. This method must be called manually from within the application program after the Java decode method. See the *Reader.java* program to see how this is done.

Object registration is done as before in the *PrintHandler* example. The only difference is that an object of the *XMLHandler* class is created instead of the *PrintHandler* class.

When compiled and executed, the output from the Reader program looks like this:

```
<PersonnelRecord>
    <name>
        <givenName>
            <IA5String>'John'</IA5String>
        </givenName>
        <initial>
            <IA5String>'P'</IA5String>
        </initial>
        <familyName>
            <IA5String>'Smith'</IA5String>
        </familyName>
    </name>
    <number>
        <INTEGER>51</INTEGER>
    </number>
    <title>
        <IA5String>'Director'</IA5String>
    </title>
    <dateOfHire>
```

```
            <IA5String>'19710917'</IA5String>
        </dateOfHire>
        <nameOfSpouse>
            <givenName>
                <IA5String>'Mary'</IA5String>
            </givenName>
            <initial>
                <IA5String>'T'</IA5String>
            </initial>
            <familyName>
                <IA5String>'Smith'</IA5String>
            </familyName>
        </nameOfSpouse>
        <children>
            <element>
                <name>
                    <givenName>
                        <IA5String>'Ralph'</IA5String>
                    </givenName>
                    <initial>
                        <IA5String>'T'</IA5String>
                    </initial>
                    <familyName>
                        <IA5String>'Smith'</IA5String>
                    </familyName>
                </name>
                <dateOfBirth>
                    <IA5String>'19571111'</IA5String>
                </dateOfBirth>
            </element>
            <element>
                <name>
                    <givenName>
                        <IA5String>'Susan'</IA5String>
                    </givenName>
                    <initial>
                        <IA5String>'B'</IA5String>
                    </initial>
                    <familyName>
                        <IA5String>'Jones'</IA5String>
                    </familyName>
                </name>
                <dateOfBirth>
                    <IA5String>'19590717'</IA5String>
                </dateOfBirth>
            </element>
        </children>
    </PersonnelRecord>
```

Add an XML document header and you should be able to display this data in XML-enabled browser.

Chapter 25. 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.

.

Chapter 26. Compact Code Generation

The *-compact* command line switch can be used to reduce the amount of source code generated for a given ASN.1 specification. This is done by generating the code for simple definitions inline within structured type definitions instead of creating separate classes.

For example, consider the following definition:

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

Y ::= [APPLICATION 2] OCTET STRING (SIZE (1..32))

Z ::= [APPLICATION 3] SEQUENCE {
      x      [0] X,
      y      [1] Y
}
```

In normal mode, the compiler would generate three classes for these productions: one corresponding to X, Y, and Z respectively. But in compact mode, it is recognized that a user would normally not be interested in encoding or decoding X and Y on their own. They would primarily be interested in encoding or decoding the more complex structured types (i.e. the PDU's) that make up fully formed messages. Taking this into account, when *–compact* is specified, the compiler will not generate separate classes for X and Y in the above definition. Instead, it will include only the base types for X and Y in the generated code for the SEQUENCE Z. All logic to handle the tags and constraints will be built directly into the Z encode and decode methods.

So the result will be only a single class generated (Z) that will contain an *Asn1Integer* object to represent X and an *Asn1OctetString* object to represent Y. The logic to process the application tags and the size constraint on the octet string will be generated inline in the encode and decode methods in Z.

.

Chapter 27. ROSE and SNMP Macro Support

The ASN1C compiler has a special processing mode that contains extensions to handle items in the older 1990 version of ASN.1 (i.e. the now deprecated X.208 and X.209 standards). This mode is activated by using the -asnstd x208 commandline option.

Although the X.208 and X.209 standards are no longer supported by the ITU-T, they are still in use today. This version of ASN1C contains logic to parse some common MACRO definitions that are still in widespread use despite the fact that MACRO syntax was retired with this version of the standard. The types of MACRO definitions that are supported are ROSE OPERATION and ERROR and SNMP OBJECT-TYPE.

ROSE OPERATION and ERROR

ROSE stands for "Remote Operations Service Element" and defines a request/response transaction protocol in which requests to a conforming entity must be answered with the result or errors defined in operation definitions Variations of this are used in a number of protocols in use today including CSTA and TCAP.

The definition of the ROSE OPERATION MACRO that is built into the 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 va
                                 | type            -- shall reference an operation ty
                                                   -- if no operation value is specif
    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 ASN1C compiler generates two types of items for the OPERATION:

1. It extracts the type definitions from within the OPERATION definitions and generates equivalent Java classes 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 open type object used to hold the encoded parameter value from step 2 is populated by creating an Asn1OpenType object using the length of the encoded component.

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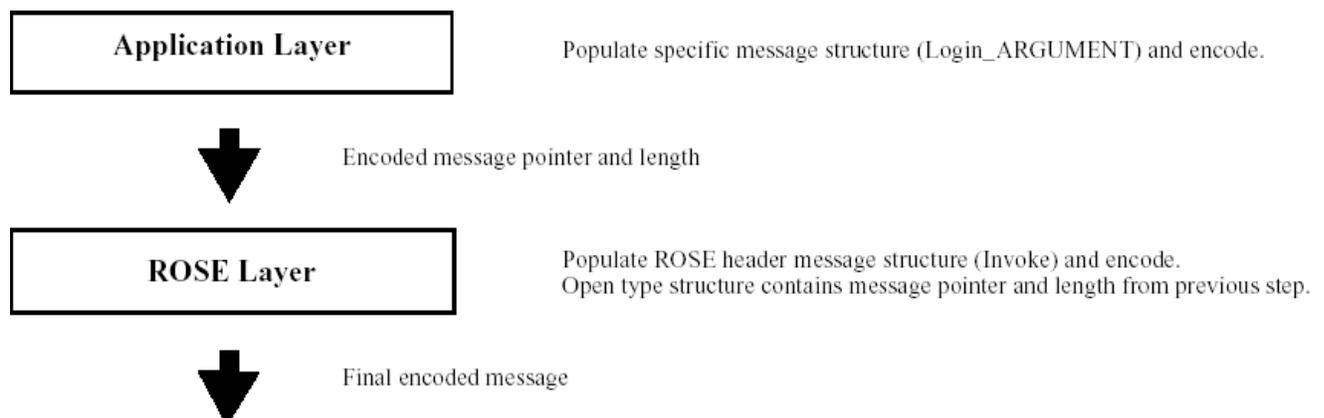
On the decode side, the process would be reversed with the message flowing up the stack:

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:
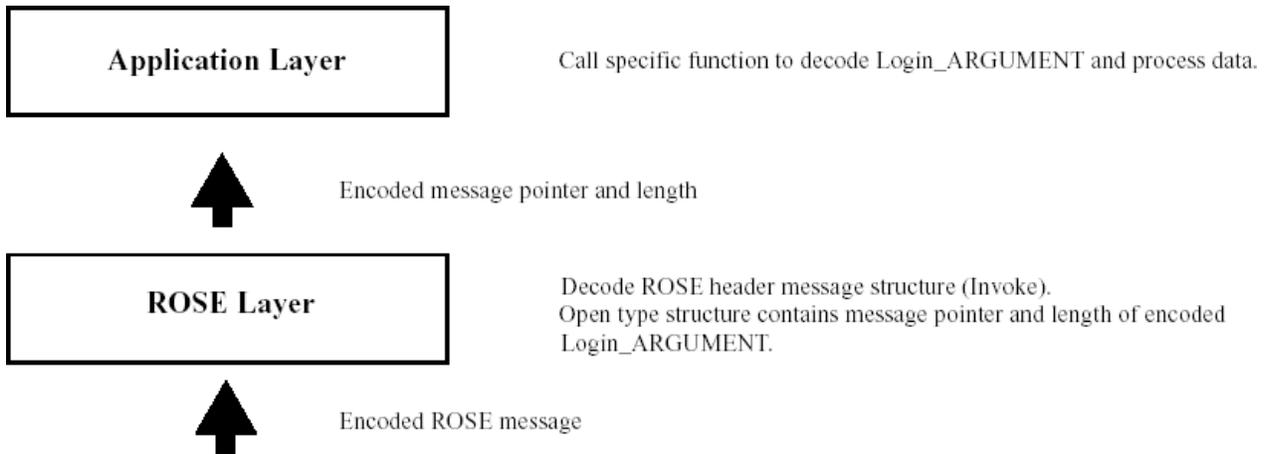
The login OPERATION also contains references to ERROR definitions. These are defined using a separate MACRO that is built into the compiler. The definition of this MACRO is as follows:

```
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 OBJECTTYPE name. So for the above definitions, the following two Java items would be generated:

1. A "udpInDatagrams_SYNTAX.java" file. This would contain the udpInDatagrams_SYNTAX class definition, and

2. A udpInDatagrams value definition in the _UDP_MIBValues class.

Chapter 28. Other Java-based Platforms

Java Micro Edition Support

If your ASN1C license purchase includes support for BER or PER encoding, it may also include support for Java Micro Edition. You may find the following additional JAR files in your installation's java folder:

- asn1rt-jme-ber.jar: ASN1C Java ME runtime with support for BER

- asn1rt-jme-per.jar: ASN1C Java ME runtime with support for PER

- asn1rt-jme-px.jar: ASN1C Java ME runtime with support for PER and XER/XML

When using ASN1C generated code on a Java Micro Edition platform, observe the following:

- Use option -cldc instead of -java. The generated code should be compatible with CLDC 1.1.

- Do NOT use any of the following ASN1C compiler options: -print -events -trace. These are not supported with CLDC 1.1.

- For XER or XML, the optional package for JSR-280 is required. This provides the SAX or StAX XML parser used by the generated code.

- For XER or XML the following tasks are more limited or not supported in the Java Micro Edition environment:

  - Working with XSD pattern constants.

  - Working with Base 64 values.

  - Working with real numbers (e.g., 3.14). For example, the encoding of most real values is not supported because the Java infrastructure needed to do such encodings simply isn't present in the Java Micro Edition environment.

- Your client code may use either one of two approaches for referring to classes such as ArrayList, which are not present in CLDC 1.1:

  1. explicitly import objsys.asn1j.runtime.ArrayList. Your code will only be compatible with our Java ME support jar.

  2. import java.util.* and objsys.asn1j.runtime.*. You code may be used in Java SE or Java ME environment. You must make sure that only one class named ArrayList is available on your classpath when compiling. If compiling for Java SE, do not include the ASN1C Java ME runtime JAR. If compiling for Java ME, use the CLDC 1.1 JAR (from the Java ME SDK) in your bootclasspath and include the ASN1C Java ME runtime JAR in your classpath.

- Asn1BigInteger and Asn1BigDecimal are not supported.

- Using the same generated code on both Java SE and Java ME CLDC 1.1 is not supported. Compile your ASN.1 specification once for Java SE and once for Java ME.

- Note that we have not tested using Java ME Connected Device Configuration (CDC) rather than CLDC.

Android

ASN1C supports the Android platform. The ASN1C runtime must be deployed on Android just as it must be deployed in a normal JSE environment. However, the runtime's Java bytecode must first be converted into Android (Dalvik VM)

bytecode. The easiest method is to use the Android ADT Eclipse Plug-in and reference the ASN1C Java runtime in the same way as you would for a normal Java application. The Android developer tools provide the ability to convert the Java bytecode that is located in any referenced JAR files. The ADT plug-in handles this automatically for you. If you choose not to use the plug-in, Android's command-line tools can provide the same functionality.

For use with Android, the ASN1C Java runtime should be either an unexpired, evaluation runtime or else an unlimited runtime.

Your installation contains a sample Android project in java/sample_ber/AndroidSample. Note that while this sample is located in the sample_ber folder, it is really a catch-all project and it is not strictly a BER-only sample.

Problems with Dalvik VM Verifier?

Our Java runtime is obfuscated, which means it can have some unusual, but legal, bytecode produced by the obfuscator. Sometimes, a particular JVM (or, in this case, the Dalvik VM) doesn't like bytecode that an obfuscator produces. We first found this was a problem with our unlimited runtimes and so we changed our obfuscation settings. If you have version 6.4.2 (or later) of ASN1C, and are using the unlimited runtime, you should not encounter verification errors.

Using the unlimited runtime would be the typical case for use with Android (per-host-licensing not really being an option). If, however, you are working with the evaluation runtime, you should be sure to have ASN1C version 6.4.3 (or later). Note that we just discovered that the verification errors persisted in our evaluation runtime while in the midst of packaging and releasing our 6.4.3 kits. As a result, we repackaged those kits. If you have an ASN1C 6.4.3 evaluation kit, you should not encounter verification errors on Android. If you do, and you were an early adopter of 6.4.3, you may just need to re-download the evaluation kit. This is likely the case if your runtime JAR file is dated before August 12, 2011.