

ASN1C

ASN.1 Compiler User's Guide for Go

Version 7.8

Objective Systems, Inc.

December 2023

ASN1C: ASN.1 Compiler User's Guide for Go

Copyright © 2020-2023 Objective Systems, Inc.

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

Author's Contact Information. Comments, suggestions, and inquiries regarding ASN1C or this document may be sent by electronic mail to <info@obj-sys.com>.

Table of Contents

1. Overview of ASN1C for Go	1
2. ASN1C Command Line Interface (CLI)	2
Running ASN1C	2
ASN1C Go Command Line Options	2
Compiler Configuration File	5
Compiler Error Reporting	8
3. ASN1C GUI Users Guide	9
Quick Start	9
Activating a License Key	9
Creating a New Project	11
.....	13
.....	13
.....	13
Interface	13
Editor	14
Project Window	15
ASN.1 Tree Window	18
Error Log Window	18
Project Settings	19
4. Generated Go Source Code	30
General Hierarchy of Generated Go Source Files	30
General Form of a Generated Go Source File	30
5. Generated Go Main File (main.go)	31
6. ASN.1 To Go Type Mappings	32
Type Mappings	32
BOOLEAN	32
INTEGER	32
BIT STRING	32
OCTET STRING	33
ENUMERATED	34
NULL	35
OBJECT IDENTIFIER	35
RELATIVE-OID	35
REAL	36
SEQUENCE/SET	36
SEQUENCE OF/SET OF	39
SET OF	40
CHOICE	40
Open Type	41
Character String Types	41
Time String Types	42
EXTERNAL	42
EMBEDDED PDV	42
Parameterized Types	43
Value Mappings	44
BOOLEAN Value	45
INTEGER Value	45
REAL Value	45
ENUMERATED Value	45
BIT STRING Value	46
OCTET STRING Value	46

Character String Value	46
Object Identifier Value Specification	46
Constructed Type Values	47
Table Constraint Related Structures	47
Unions Table Constraint Model	47
7. Generated BER/DER Functions	52
Generated BER/DER Encode Functions	52
Generated Go Function Format and Calling Parameters	52
Example of Calling the Go Marshal Function	52
How Marshal Works And Alternatives	53
Generated BER/DER Decode Functions	54
DER Restriction Violations	54
Generated Go Function Format and Calling Parameters	54
Example of Calling the Go Unmarshal Function	55
How Unmarshal Works And Alternatives	55
8. Generated PER Functions	57
Generated PER Encode Functions	57
Generated Go Function Format and Calling Parameters	57
Example of Calling the Go Marshal Function	57
How Marshal Works And Alternatives	58
Generated PER Decode Functions	59
Generated Go Function Format and Calling Parameters	59
Example of Calling the Go Unmarshal Function	59
How Unmarshal Works And Alternatives	60
PER Bit Tracing	61
9. Generated JSON Functions	62
Custom JSON Attributes	62
Custom JSON Marshal/Unmarshal Functions	62
Invoking JSON Marshal/Unmarshal Functions	62
10. Generated Print Functions	63
Generated Print Functions	63
Print to Standard Output	63
Print to String	63
11. IMPORT/EXPORT of Types	64

Chapter 1. Overview of ASN1C for Go

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#, Java, Python, or Go. This manual discusses the Go 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
- *ASN1C Java Compiler User's Manual* : Java code generation
- *ASN1C Python Compiler User's Manual* : Python code generation

Each ASN.1 module that is encountered in an ASN.1 schema source file results in the generation of a Go source file. The name of the file is the ASN.1 module name, with hyphens removed and changed to camel-case, and an extension of '.go'.

There is also a set of functions that form the run-time component of the Go package. These functions provide 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.

This release of the ASN1C Compiler for Go 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:

- Packed Encoding Rules (PER) as published in the ITU-T X.691 and ISO/IEC 8825-2 standards. Includes both the aligned and unaligned variants (note: the unaligned variant is commonly known as UPER).
- JSON Encoding Rules (JER) as published in the ITU-T X.697 and ISO/IEC 8825-8 standards.

ASN1C for Go 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 ITU-T X.681 as well as Parameterized Types as defined in ITU-T X.683.

Chapter 2. ASN1C Command Line Interface (CLI)

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 Go.

ASN1C Go Command Line Options

The following table shows a summary of the command line options that have meaning when generating Go code:

Option	Argument	Description
-ber	None	This option is used to generate encode/decode functions that implement the Basic Encoding Rules (BER) as specified in the ITU-T X.690 ASN.1 standard.
-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 section called "Compiler Configuration File".
-depends	None	Without this option, the compiler only generates Go code for the files that were explicitly given as input, and not for any files it may automatically load on account of IMPORTS. This option tells the compiler to also generate code for files it automatically loads, but only that code which is necessary to satisfy dependencies that originate in the main file set.
-der	None	This option is used to generate encode/decode functions that implement the Distinguished Encoding Rules (DER) as specified in the ITU-T X.690 ASN.1 standard.
-genPrint -print	None	Generate code to print object contents. The open source go-spew package (https://github.com/davecgh/go-spew) is used for this purpose.
-genTest -test	None	Generate code to populate a PDU object with random test data. This code

Option	Argument	Description
		will be embedded in the body of a generated main.go program.
-go-rt-inc	<module(s)>	Force inclusion of the given comma-separated list of run-time modules in the final compilation. Normally, run-time modules are included only if needed by the generated code. This option is used if a specific run-time module is required in user developed code. For example, the TS32297 module in a main.go program that requires this package to encode and decode headers.
-I	<directory>	This option specifies a directory where the compiler should search for ASN.1 source files for IMPORT items. Multiple -I qualifiers can be used to specify multiple directories to search.
-json or -jer	None	This option is used to generate encode/decode functions that implement the Javascript Object Notation (JSON) Encoding Rules (JER) as specified in the X.697 ASN.1 standard.
-lax	None	Suppress generation of code to check constraints.
-list	None	Generate listing. This will dump the source ASN.1 to the standard output device as it is parsed. This can be useful for finding parse errors.
-noacomment	None	This option suppresses the generation of the comment that shows the asn1c command that was used to generate the code.
-nodecode	None	Suppress generation of decode functions.
-noencode	None	Suppresses generation of encode functions.
-noOpenExt	None	Suppress the generation of open extension elements 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 and increase performance.

Option	Argument	Description
-noPLMN	None	If not specified, if the ASN.1 contains a production named PLMNidentity (any case), that production will be treated as an OCTET STRING, regardless of how it may be defined in the ASN.1. But if -noPLMN is specified, this special recognition of PLMNidentity will be disabled, and it will be treated according to its definition in the ASN.1 (frequently TBCD-STRING, although in most cases that is incorrect).
-noUniqueNames	None	Turn off the capability 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).
-no-go-main	None	Suppress the generation of a main.go file. By default, this file is generated with options to encode (-writer) and decode (-reader) a message of a PDU type.
-o	<directory>	Specify the name of a directory to which all of the generated files will be written.
-oid-as-string	None	Use string to represent OBJECT IDENTIFIER and RELATIVE-OID values. The strings must be in dotted-number format (e.g. "0.5.1234").
-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.

Option	Argument	Description
-per	None	This option is used to generate encode/decode functions that implement the Packed Encoding Rules (PER - aligned variant) as specified in the ASN.1 standards.
-shortnames	None	Change the names generated by compiler for embedded types in constructed types.
-tables	None	Generate additional code for the handling of table constraints as defined in the X.682 standard.
-uper	None	This option is used to generate encode/decode functions that implement the Packed Encoding Rules (PER - unaligned variant) as specified in the ASN.1 standards.
-warnings	None	Output information on compiler generated warnings.

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.

The basic structure of a configuration file is described in the C/C++ User's Guide. Configuration items that are applicable to Go code generation are described in the following sections.

Global Level

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

Name	Values	Description
<code><includedir></includedir></code>	<code><Include directory></code>	This configuration item is used to specify a directory that will be searched for IMPORT files. It is equivalent to the <code>-I</code> command-line option.

Module Level

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

Name	Values	Description
<code><name> </name></code>	module name	This attribute identifies the module to which this section applies. Either this or the <code><oid></code> element/attribute is required.
<code><oid></code>	module OID (object identifier)	This attribute provides for an alternate form of module identification for the case when module name is not unique. For example, a given ASN.1 module may have multiple versions. A unique

Name	Values	Description
		version of the module can be identified using the OID value.
<code><include types="names" values="names"/></code>	ASN.1 type or values names are specified as an attribute list	<p>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 by selecting only a subset of the types/values in a specification for compilation.</p> <p>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. However, if an element is marked <code><notUsed/></code> or <code><notUsedSkip/></code>, the referenced type is not automatically included; if an element is marked <code><notUsedDecode/></code> or <code><notUsedDecodeSkip/></code>, the referenced type is included for encoding only; if an element is marked <code><notUsedEncode/></code>, the referenced type is included for decoding only.</p>
<code><exclude types="names" values="names"/></code>	ASN.1 type or values names are specified as an attribute list	<p>This item allows a list of ASN.1 types and/or values to be excluded from the generated code. By default, the compiler generates code for all types and values within a specification. This is generally not as useful as the <i>include</i> directive because most types in a specification are referenced by other types. If an attempt is made to exclude a type or value referenced by another item, the directive will be ignored.</p>
<code><sourceFile></code> <code></sourceFile></code>	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.
<code><valuePrefix></code> <code></valuePrefix></code>	prefix text	This is used to specify a prefix that will be applied to all generated value constants within a module. This can be used to prevent name clashes if multiple modules are involved that use a common name for two or more different value declarations.

Production Level

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

Name	Values	Description
<name> </name>	production name	This attribute identifies the production (type) to which this section applies. The name may also be specified as an attribute. In either case, it is required.
<isPLMNidentity/>	n/a	This flag identifies the production as a PLMNidentity. Any production so identified will be treated as an OCTET STRING, regardless of how the production might be defined in the ASN.1.
<typePrefix> </typePrefix>	prefix text	This is used to specify a prefix that will be applied to generated Go type names for this production. This can be used to prevent name clashes if multiple modules are involved in a compilation and they contain common names.

Element Level

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

Name	Values	Description
<name> </name>	element name	This element identifies the element within a SEQUENCE, SET, or CHOICE construct to which this section applies. It may also be specified as an attribute. In either case, it is required.
<notUsed/>	n/a	This flag variable specifies that this element will not be used at all in the generated code. It can only be applied to optional elements within a SEQUENCE or SET, or to elements within a CHOICE. Its purpose is for production of more compact code by allowing users to configure out items that are of no interest to them.
<notUsedSkip/>	n/a	This is the same as <notUsed/> except that it tells the compiler to generate skip functions when necessary. Skip functions are used for skipping over values of a type during decoding, rather than reporting an error if that type should happen to be encountered in the encoding.
<notUsedDecode/>	n/a	Similar to the notUsed flag, except that rather than indicating the element

Name	Values	Description
		is not used at all, it indicates the element is not used for decoding (i.e. it is still used for encoding). This can be useful for reducing the amount of generated code, when used in conjunction with a production level <code><include/></code> (which see). It signals that the element's type does not require a decode function for the sake of this element.
<code><notUsedDecodeSkip/></code>	n/a	This is the same as <code><notUsedDecode/></code> except that it tells the compiler to generate skip functions when necessary. Skip functions are used for skipping over values of a type during decoding, rather than reporting an error if that type should happen to be encountered in the encoding.
<code><notUsedEncode/></code>	n/a	The encoding complement to <code><notUsedDecode/></code>

Compiler Error Reporting

Errors that can occur when generating source code from ASN.1 take two forms: syntax errors and semantic 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 names must begin with an uppercase letter, production (type) names 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, Go does not allow hyphens in identifiers and prefers names be in camel-case form. ASN1C will attempt to transform the ASN.1 names into the Go preferred format.

Semantic 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.

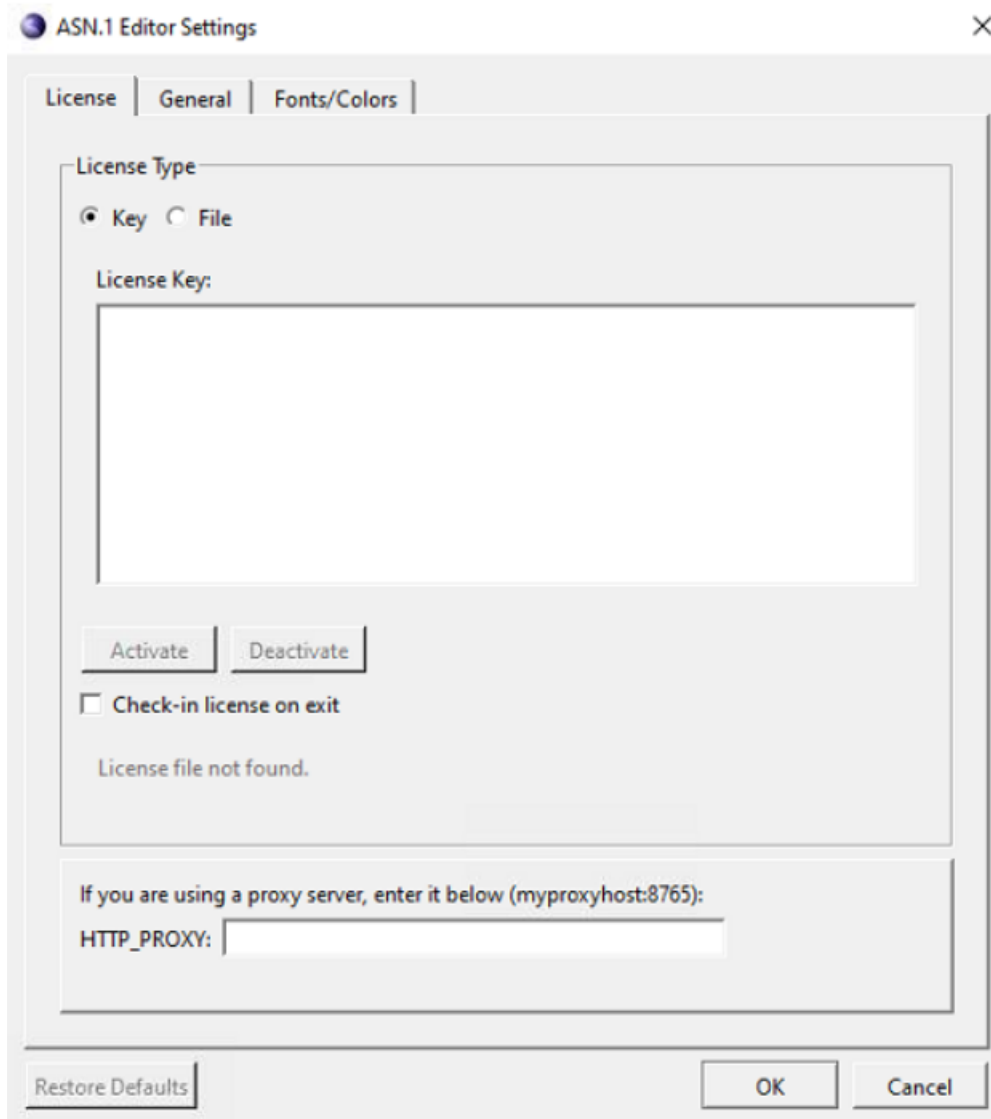
Chapter 3. ASN1C GUI Users Guide

Quick Start

This section demonstrates running ACGUI, activating a license key, creating a new ASN.1 schema, and compiling it to C for BER data. The process is similar for other languages.

Activating a License Key

First, start ACGUI. The Enter License Key window may be displayed to activate a license key.



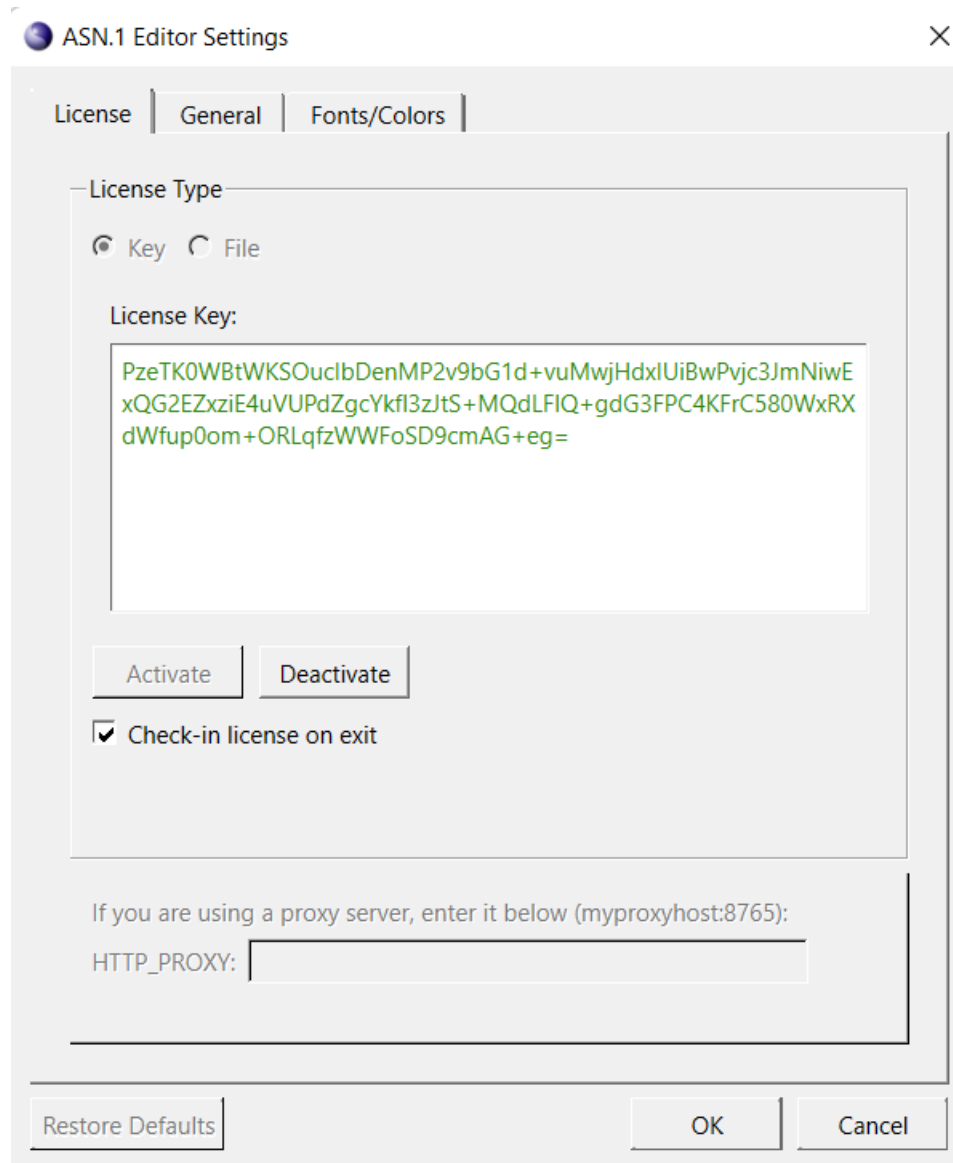
If the Enter License Key window is displayed and it is not showing a current license key, right-click in the text box and paste the accurate license key. Then click Activate to unlock ASN1C.

In some cases, the Enter License Key window is displayed and showing a current license key. In these cases, it is likely that the key being shown is expired. First deactivate the current key by clicking Deactivate. Then, right-click in the text box and paste the current license key, and click Activate to unlock ASN1C.

If an osyslic.txt license file is being used instead of a key, and the osyslic.txt is in a location where the GUI does not look, click Import to find the file and use it to unlock ASN1C.

If Skip is clicked within the Enter License Key window, the features of the GUI can be explored, but code cannot be generated.

If a new license key must be activated and the current one is still valid (for example, if ASN1C is purchased before the evaluation key expires), the existing license must be deactivated first. This deactivation can be done from the GUI by navigating to Tools > Options and then selecting the License tab. The ASN.1 Editor Settings window is displayed.



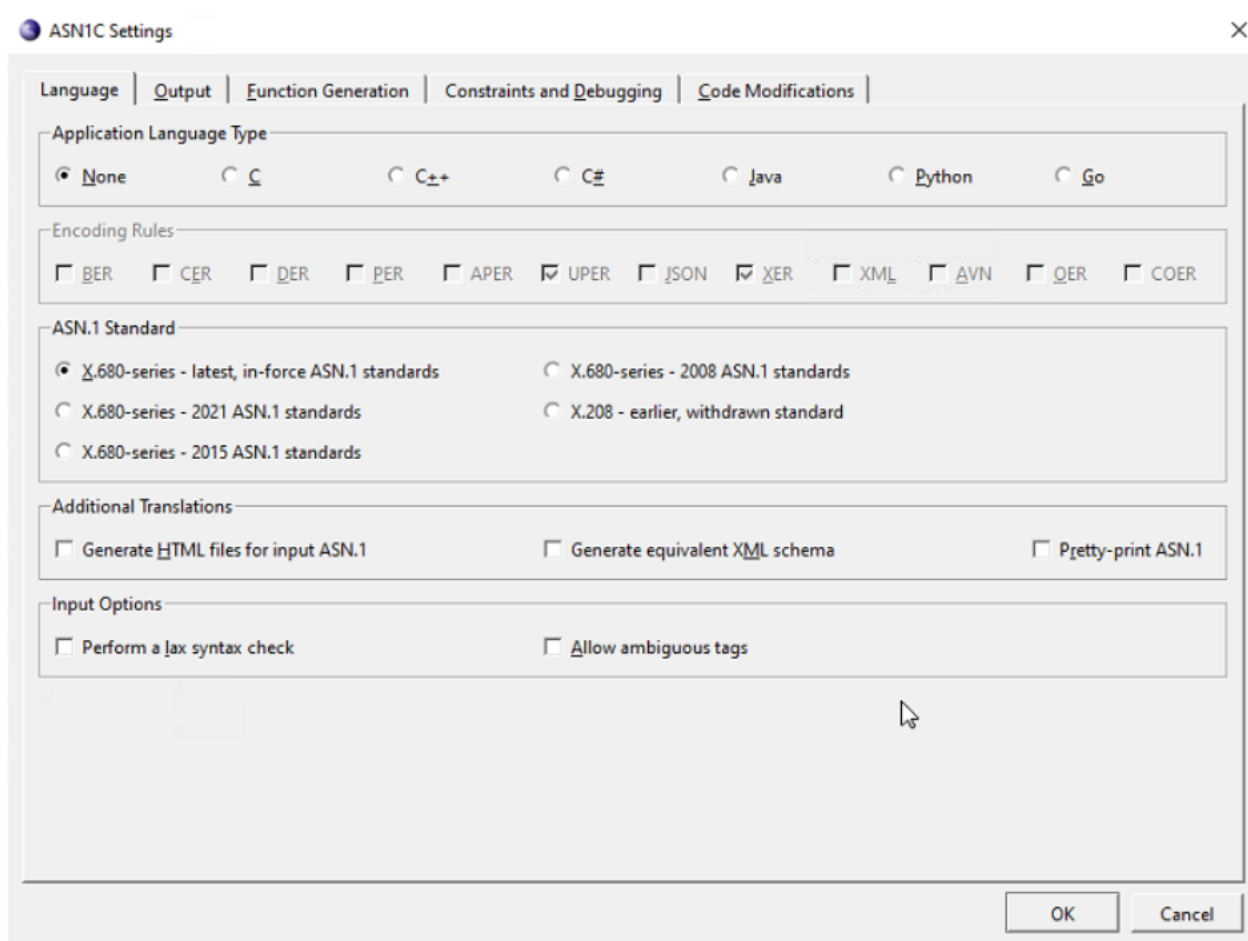
Click Deactivate to deactivate the existing license. Then click OK. Next, navigate to Tools > Options > License tab again to activate the new key.

The "Check-in license on exit" check box is used to indicate that the license should be immediately returned to the license pool upon exit making it available for other users on different machines. If not checked, the machine on which it is being used will continue to hold it until it times out (typically in 24 hours). If you will only be using ASNIC on a single machine most of the time, it is better to keep it unchecked as it will lead to faster startups since the Internet check will not need to be done each time.

The HTTP PROXY box can be used if you are using ASNIC on a machine that requires Internet requests go through a proxy server.

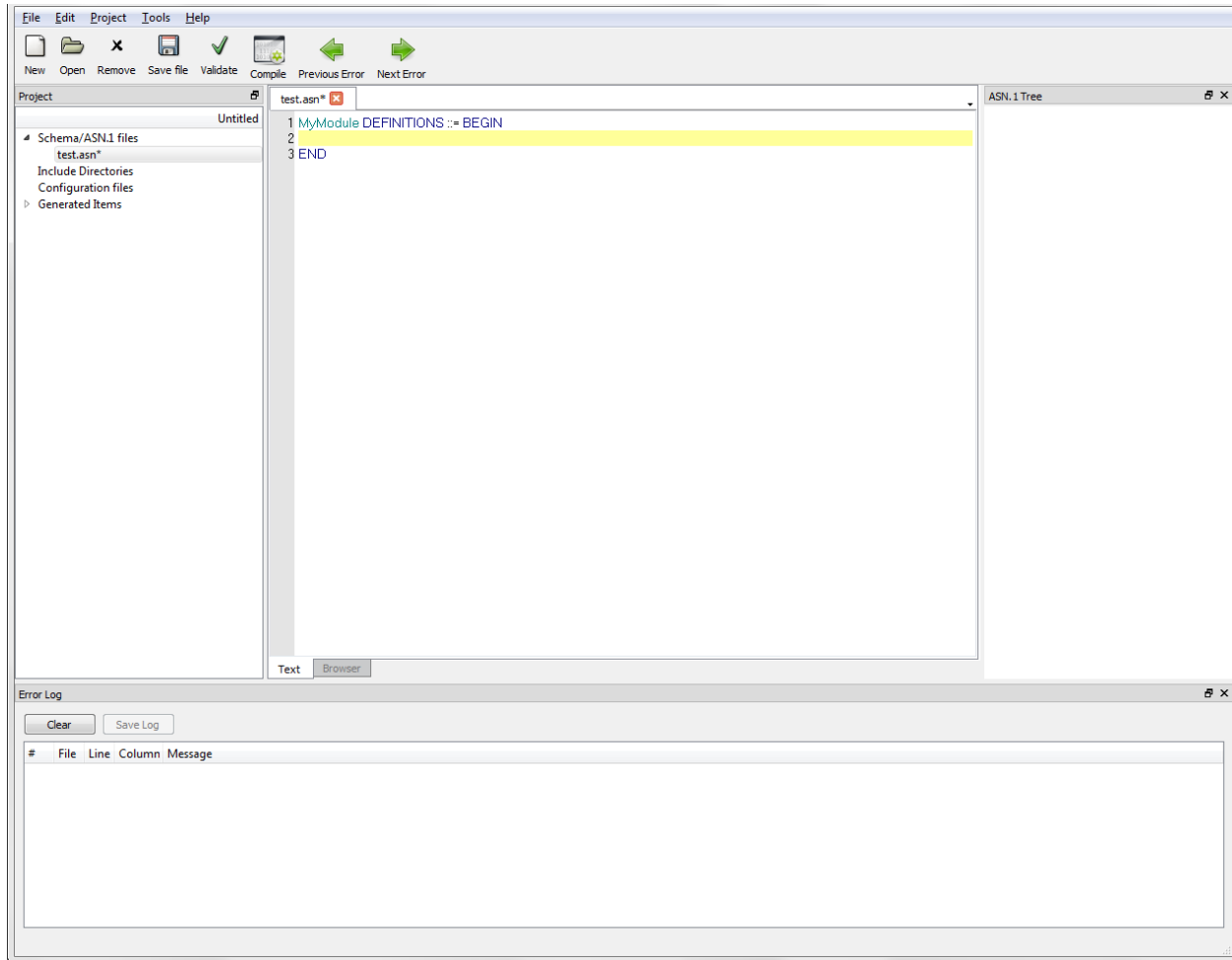
Creating a New Project

Once the ASNIC program has been activated, a new project can be created to store all of the settings. To do this, select Project > New Project from the menu. The ASNIC Settings window is displayed.



The Language tab is displayed by default. In the Application Language Type section, select C. In the Encoding Rules section, select BER. Finally, on the Output tab, in the Output Directory text box, enter or browse to the location where the generated files should be saved. When finished defining the project settings, click OK. These settings can be changed at any time by selecting Project > Project Settings from the menu.

Next, a new schema file is created for the project. Click New in the toolbar or navigate to File > New Schema File. A dialog box is displayed to define a name for the new file. Once entered, the file is added to the project window under the "Schema/ASN.1 files" heading and its empty contents are shown in the editor.



The schema is then written between the "DEFINITIONS ::= BEGIN" and "END" statements in the file. For this example, the following can be entered:

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

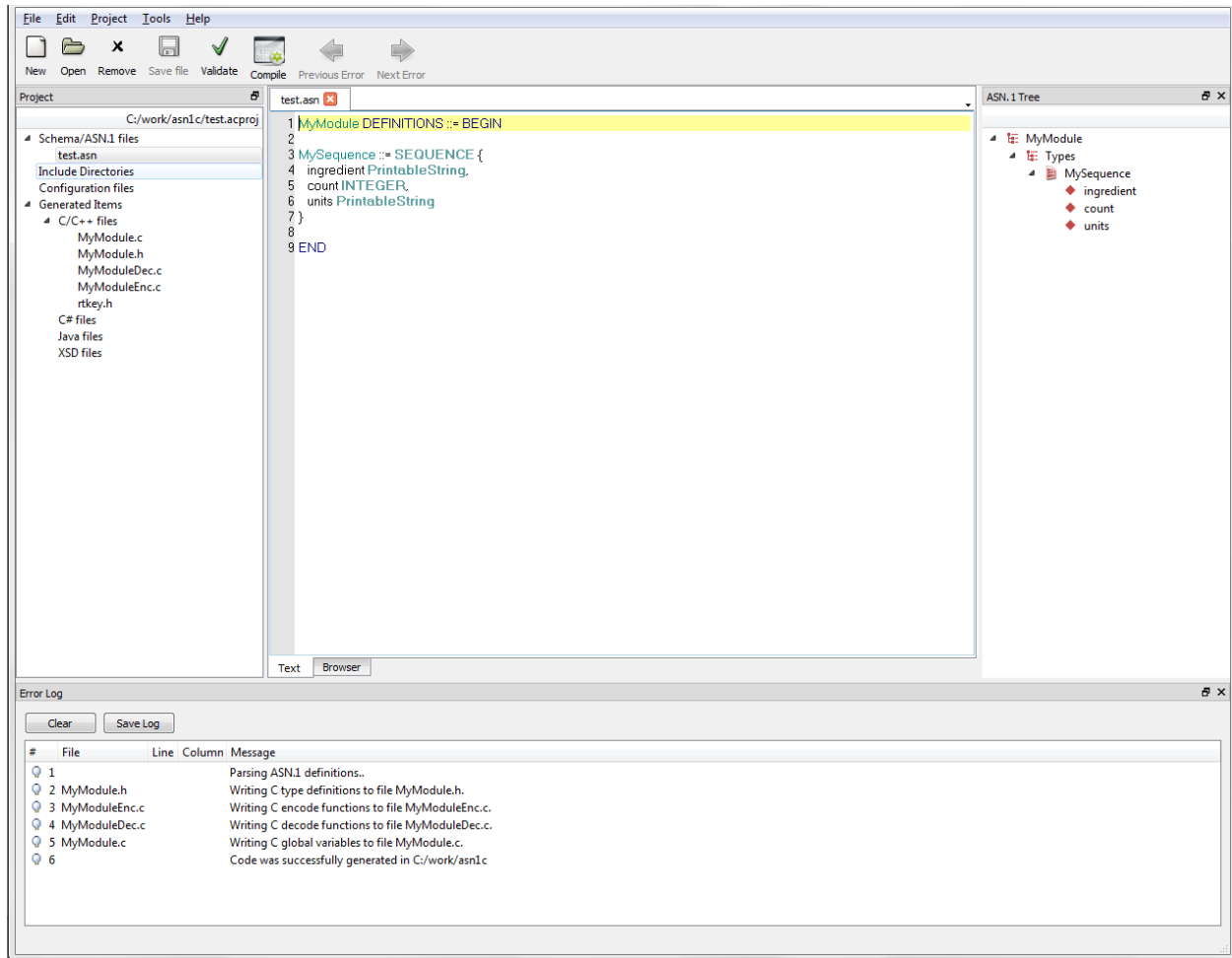
Once the schema is created, click Validate to perform a check for errors.

Note

If the new schema file has not yet been saved, ACGUI asks if it should be. Once saved, ACGUI validates the file.

If the schema has errors, they are displayed in the log at the bottom of the ACGUI window.

Once the project has been configured, click Compile to generate code according to the project settings. If compilation is successful, a list of generated files, sorted according to the selected language, is displayed under the Generated Items heading in the Project pane. If compilation is not successful, any errors are displayed in the Error Log.



At this time, project settings can be changed and schema files can be edited as needed.

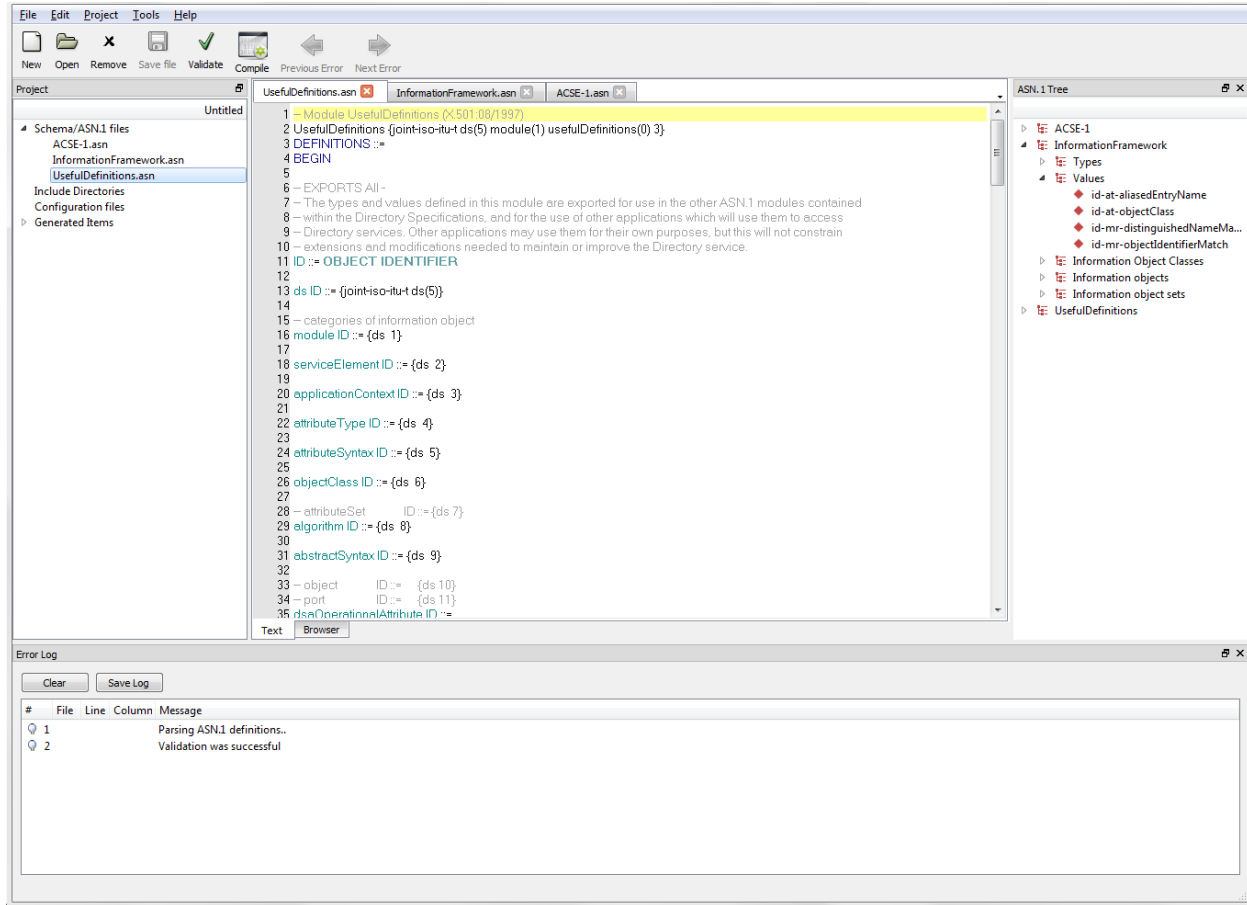
<xi:include></xi:include>

<xi:include></xi:include>

<xi:include></xi:include>

Interface

The ACGUI interface is comprised of five parts: Editor, Project window, ASN.1 Tree window, Error Log window, and Project Settings.



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 or open a schema file. A new schema file can be opened by clicking New in the toolbar or by navigating to File > New Schema File. An existing schema file can be accessed via the Open button in the toolbar or navigating to File > Open File. The file is added to the current project and shown in the Editor.

```

1 - Module UsefulDefinitions (X.501:08/1997)
2 UsefulDefinitions {joint-iso-itu-t ds(5) module(1) usefulDefinitions(0) 3}
3 DEFINITIONS ::=
4 BEGIN
5
6 - EXPORTS All -
7 - The types and values defined in this module are exported for use in the other ASN.1 modules contained
8 - within the Directory Specifications, and for the use of other applications which will use them to access
9 - Directory services. Other applications may use them for their own purposes, but this will not constrain
10 - extensions and modifications needed to maintain or improve the Directory service.
11 ID ::= OBJECT IDENTIFIER
12
13 ds ID ::= {joint-iso-itu-t ds(5)}
14
15 - categories of information object
16 module ID ::= {ds 1}
17
18 serviceElement ID ::= {ds 2}
19
20 applicationContext ID ::= {ds 3}
21
22 attributeType ID ::= {ds 4}
23
24 attributeSyntax ID ::= {ds 5}
25
26 objectClass ID ::= {ds 6}
27
28 - attributeSet      ID ::= {ds 7}
29 algorithm ID ::= {ds 8}
30
31 abstractSyntax ID ::= {ds 9}
32
33 - object          ID ::= {ds 10}
34 - port            ID ::= {ds 11}
35 dsaOperationalAttribute ID ::=

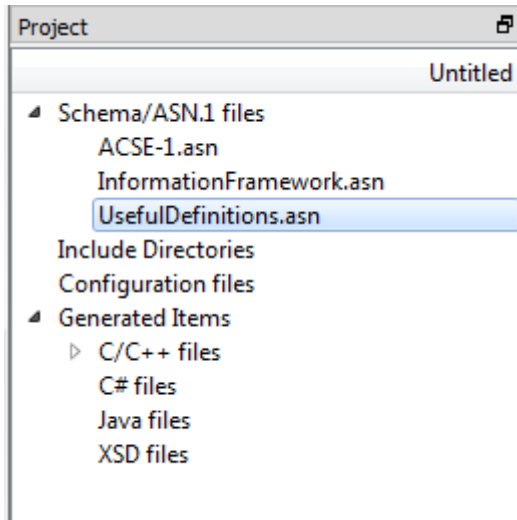
```

The Editor window is also used to display a schema browser for navigating within a validated schema. To display the browser after validating a schema, click on an item in the ASN.1 Tree window. The browser displays a hyperlinked version of the schema, centered on the definition of the selected item. Click the names of other defined types in the browser to show their definitions.

By default, schema documents are displayed as tabs in the Editor. The Text and Browser tabs at the bottom of the window are for schema editing and hyperlinked schema browsing, respectively. Schema files currently open for edit are displayed as tabs at the top of the Text tab.

Project Window

The Project window allows the user to interact with project assets, and contains the following components: Schema/ASN.1 files, Include Directories, Configuration files and Generated Items.



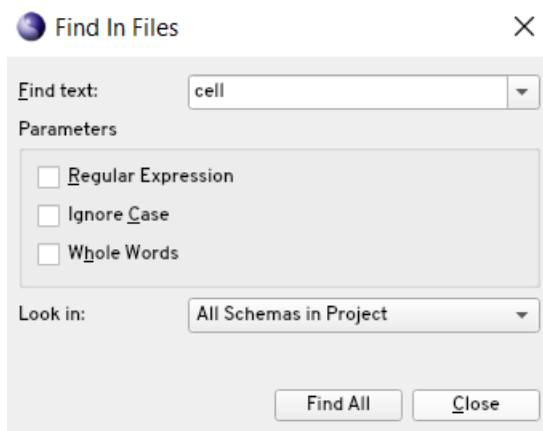
The Project window contains the following sections:

- | | |
|---------------------|--|
| Schema/ASN.1 files | This section lists the files that contain the current project's ASN.1 schema definitions. |
| Include Directories | This section lists the directories containing auxiliary ASN.1 schema files. The current project's schema may import definitions from modules defined in an included directory. |
| Configuration file | This section lists the ASN1C compiler configuration file currently in use. |
| Generated Items | This section lists the files generated by the compiler, separated by target language. |

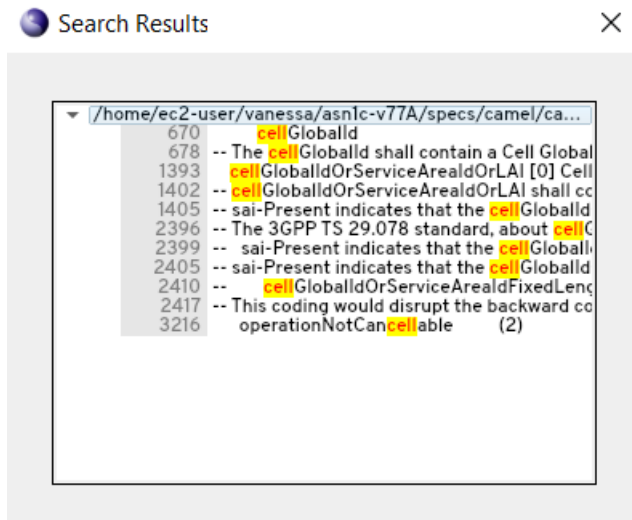
Click on a schema or configuration file in the Project window to open that file in the Editor. Right-click on any schema file, include directory, or configuration file to easily add another or remove that particular asset from the project.

Right-clicking on the Schema/ASN.1 files heading also displays options for 'Find in Files' and 'Replace in Files.'

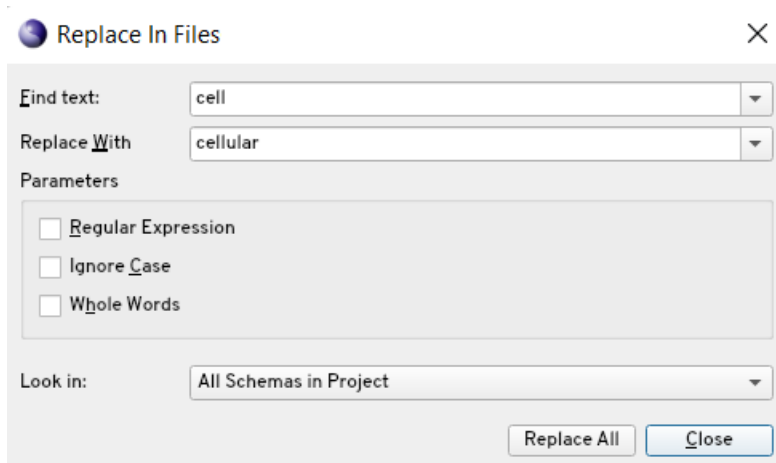
Selecting 'Find in Files' displays the Find in Files window, which provides users with the ability to enter any text to be located in either all schemas in the project, or only the schemas currently open in the editor. Once the Find parameters have been defined, click Find All to be presented with a list of files that contain the search text.



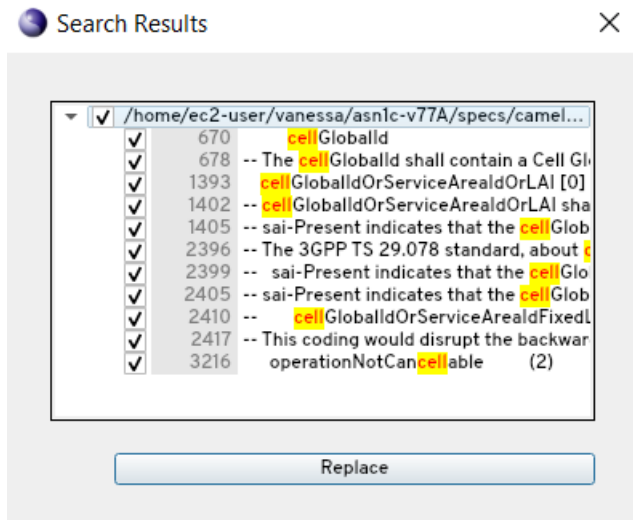
Click an entry within the results list to display the text in the editor.



Selecting 'Replace in Files' displays the Replace in Files window, which provides users with the ability to enter any text to be located and replaced in either all schemas in the project, or only the schemas currently open in the editor. Once the Find and Replace parameters have been defined, click Replace All to be presented with a list of files that include the word to be changed.

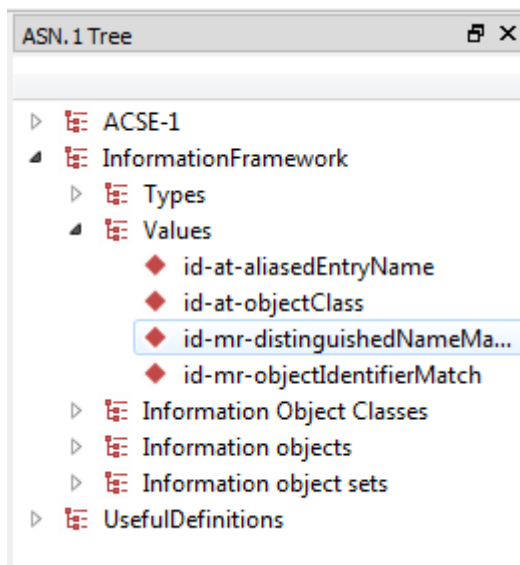


Click an entry within the results list to display the text in the editor, and if desired, deselect any check box next to an entry in the results list to prevent the replacement from happening on that result. Click Replace to perform the word replacement.



ASN.1 Tree Window

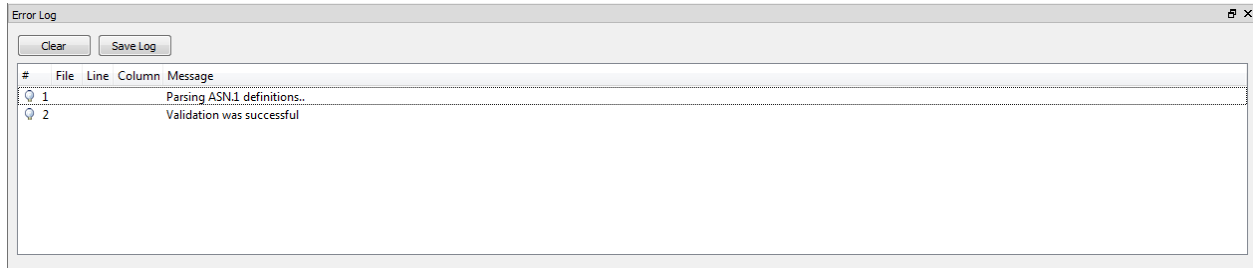
Once a schema has been validated or compiled in ACGUI, the ASN.1 Tree window provides an interactive view of the ASN.1 types defined in it.



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 within. Click on any node of the tree to 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 reports a success. If an error occurs, an error message is displayed.



In many cases, an error is associated with a particular portion of the schema being compiled. Click on an error to open the schema editor at the location in which the error occurred. If more than one error is reported, clicking Next Error and Previous Error in the toolbar moves the Editor window to the part of the schema where the next or previous error occurred.

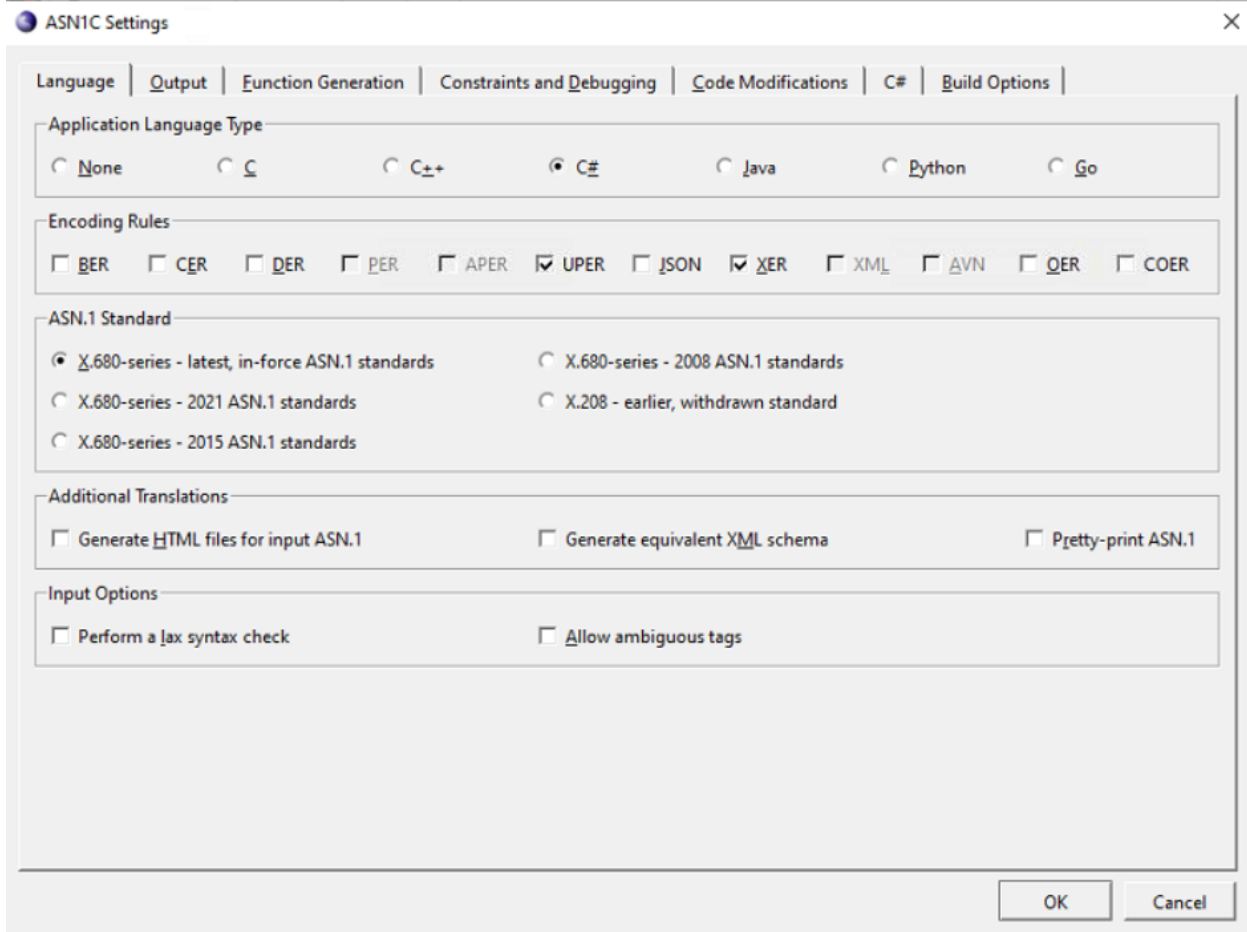
When the reported errors are no longer needed, click Clear in the Error Log window to remove them from display.

Project Settings

The Project Settings window is where details regarding the project, such as encoding rules, target language, and code features to generate are modified. The window contains the following tabs: Language, Output, Function Generation, Constraints and Debugging, Code Modifications, language-specific code modifications, and Build Options.

Language tab

The Language tab contains options for selecting a target language, encoding rules, ASN.1 Standards, translation, and input options.



The Language tab contains the following sections:

Application Language Type

This section provides users with the ability to define the target language for the project. A target language must be selected in order to compile a schema.

Depending on the target language selected, additional options are displayed within the Output tab.

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

C/C++ Output Options

Output code to .c/.h files based on module names

Output all code to a single .c/.h file

Output each generated function to its own source file

Default integer type: int32

Max lines per file:

Object directory:

Libraries Directory:

Binaries Directory:

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

C# Code Organization

Output code to directories based on module names

Code generation option: One .cs file per type File name:

Binaries Directory:

Libraries Directory:

For Java, the Java Code Organization section controls how generated code should be organized into directories based on the ASN.1 module for which they were generated. Alternatively, generated files are placed directly into the output directory.

Java Code Organization

Output code to directories based on module names

Class directory:

Encoding Rules

This section provides users with the ability to define one or more encoding rule sets to be selected for the generated code.

ASN.1 Standard

This section provides users with the ability to apply current or previously-defined ASN.1 standards to the generated code.

Additional Translations

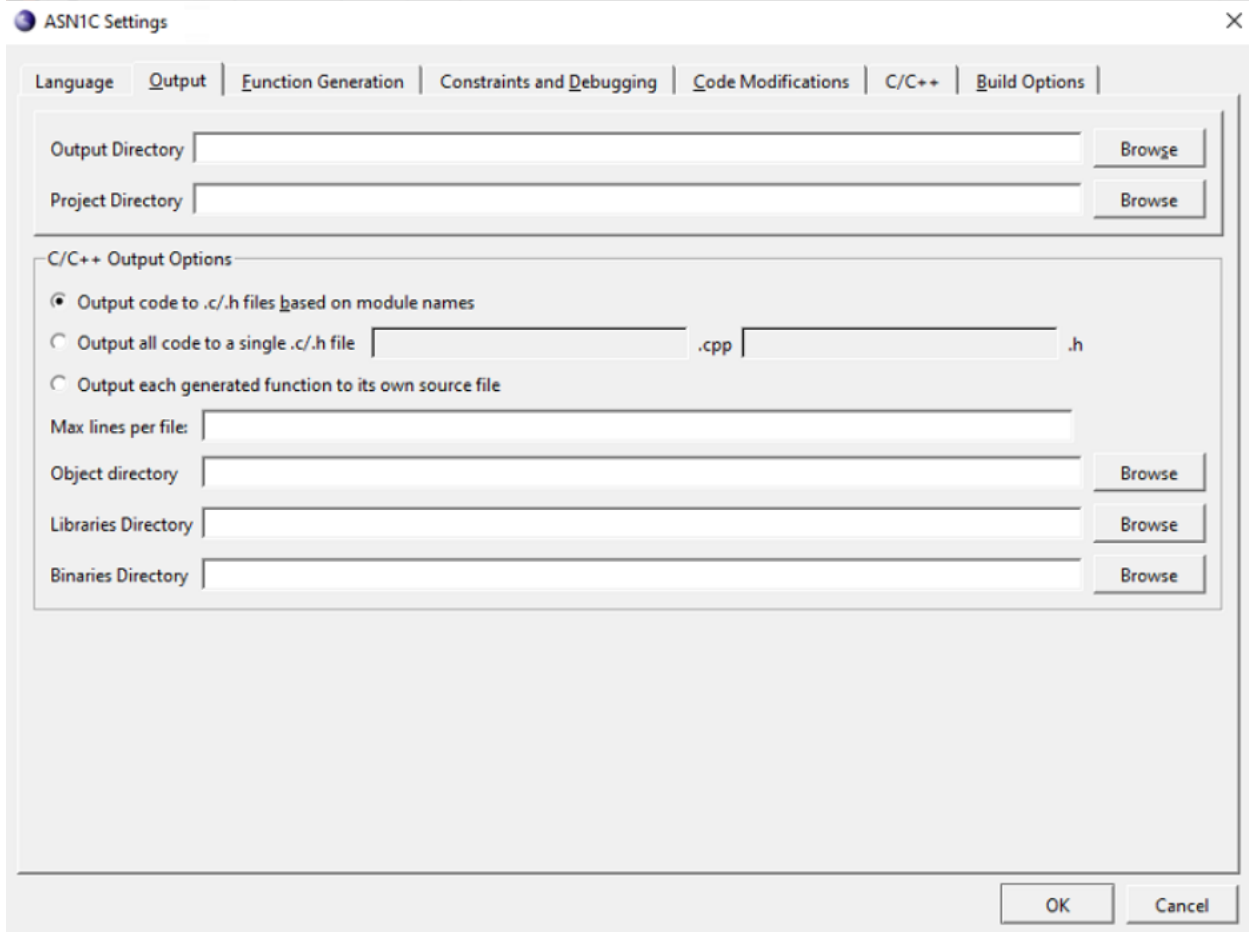
This section provides users with the ability to define the options for generating transformed versions of the input schema, such as HTML or pretty-printed.

Input Options

This section provides users with the ability to define how strict the compiler is when parsing ASN.1 schema.

Output tab

The Output tab contains options for selecting encoding rules, as well as output and project directories.

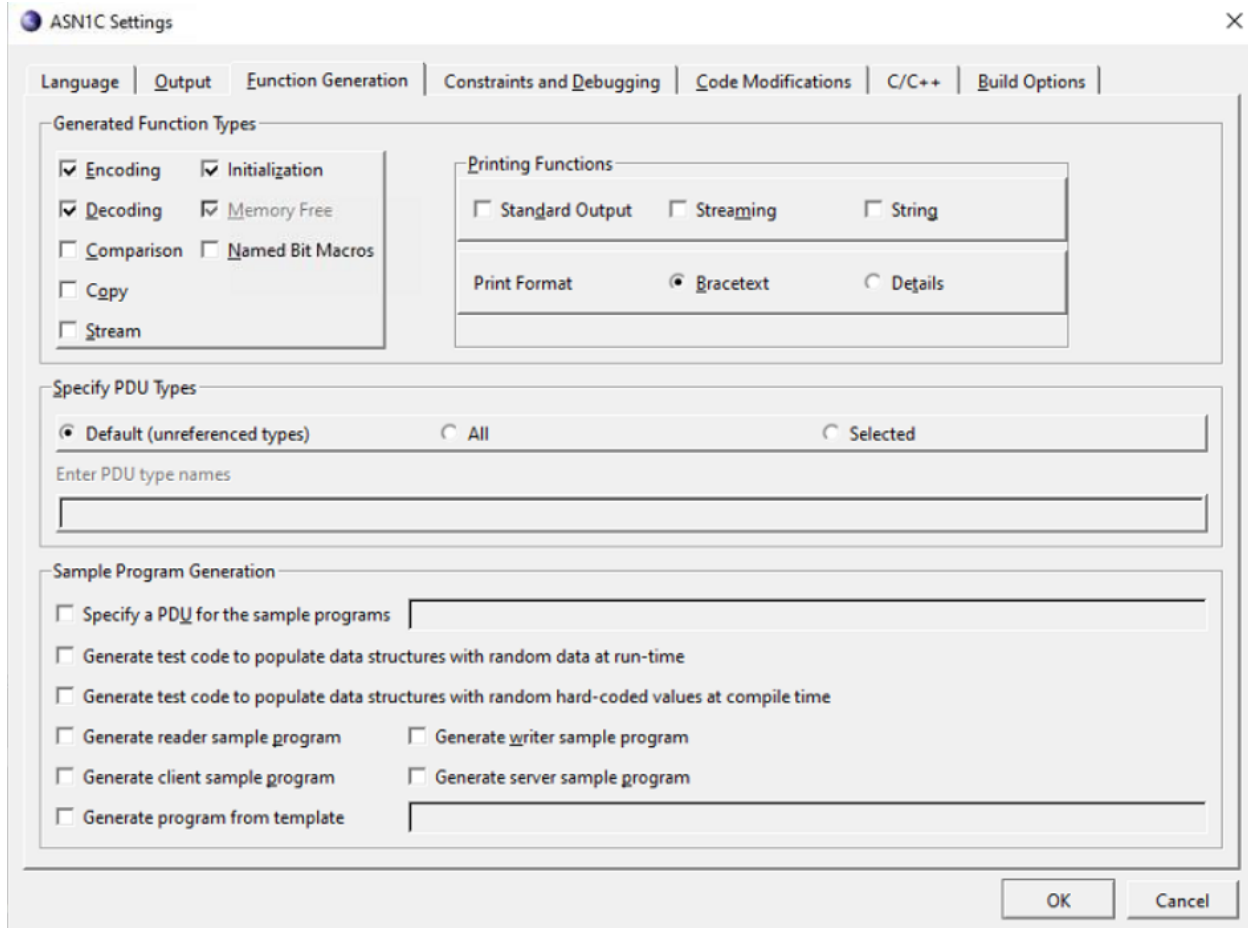


The Output tab contains the following sections:

- | | |
|--|---|
| Output Directory and Project Directory | This section provides users with the ability to define in which directory the code output and/or project should be stored. |
| Language-specific Output Options | This section provides users with the ability to define additional output options specific to the language type chosen on the Language tab, as well as the ability to define additional directories. |

Function Generation tab

The Function Generation tab provides settings for the functionality to include in generated code.



The Function Generation tab contains the following sections:

Generated Function Types	This section provides users with the ability to define granular control of which 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.
Specify PDU Types	This section provides users with the ability to define which productions to select as PDUs.
Sample Program Generation	This section provides users with the ability to define the generation of simple encoding and decoding programs, which demonstrate using the generated code. Additionally, the sample writer program can optionally encode randomly-generated test data.
Language-specific Functions	Depending on the target language selected, additional options may be displayed within this section of the Function Generation tab. 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, but may be turned off.

C/C++ Generated Functions

- Generate Initialization Functions
- Generate Memory Free Functions
- Generate Named Bit Macros

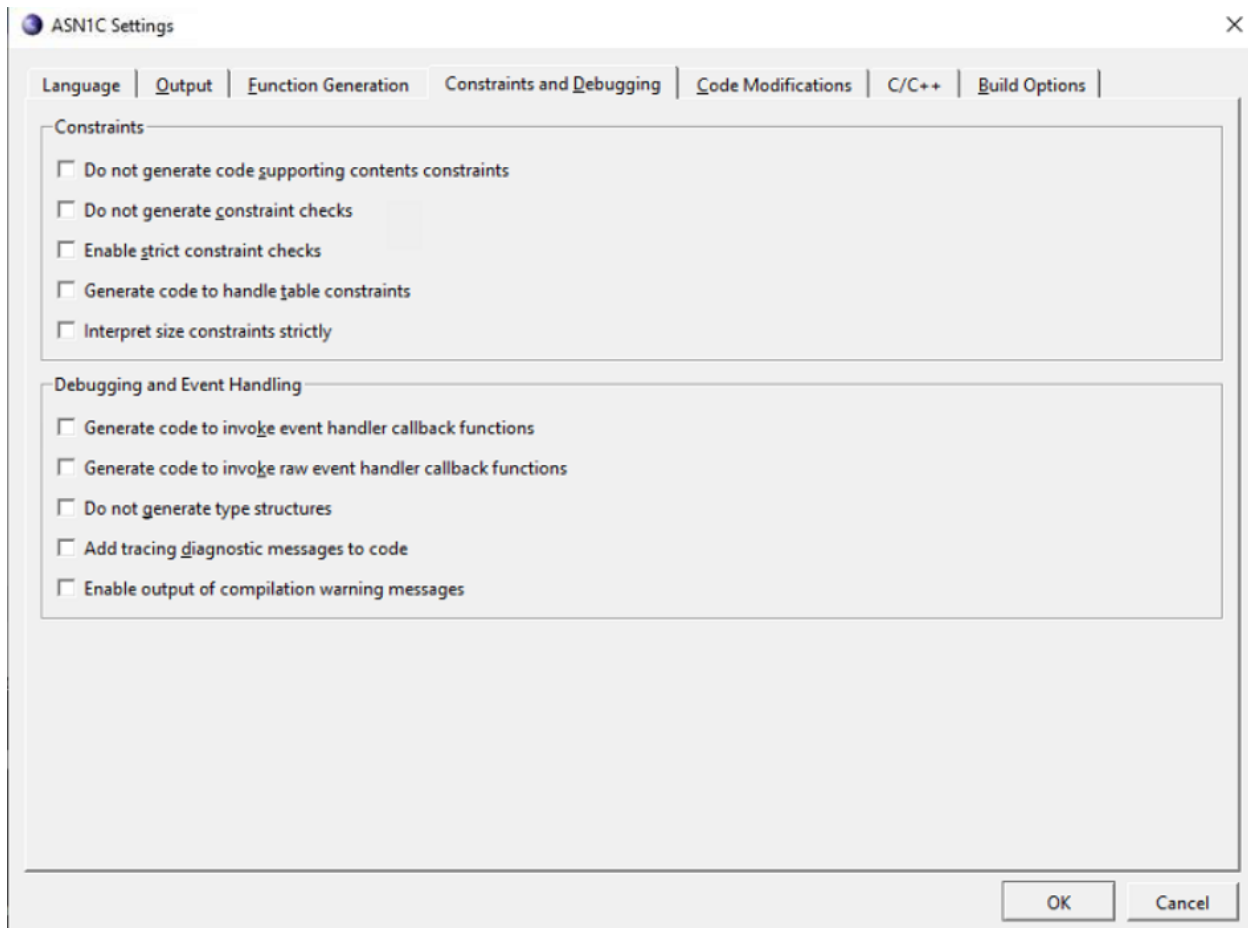
For Java, get and set methods can be generated for members of generated classes. It is also possible to generate methods that can fetch certain types of metadata (for example, if an element is optional). A similar option exists for C#.

Java Function Options

- Generate getter and setter methods
- Generate metadata methods

Constraints and Debugging tab

The Constraints and Debugging tab holds settings related to constraint handling, event handling, and logging in generated code.



The Constraints and Debugging tab contains the following sections:

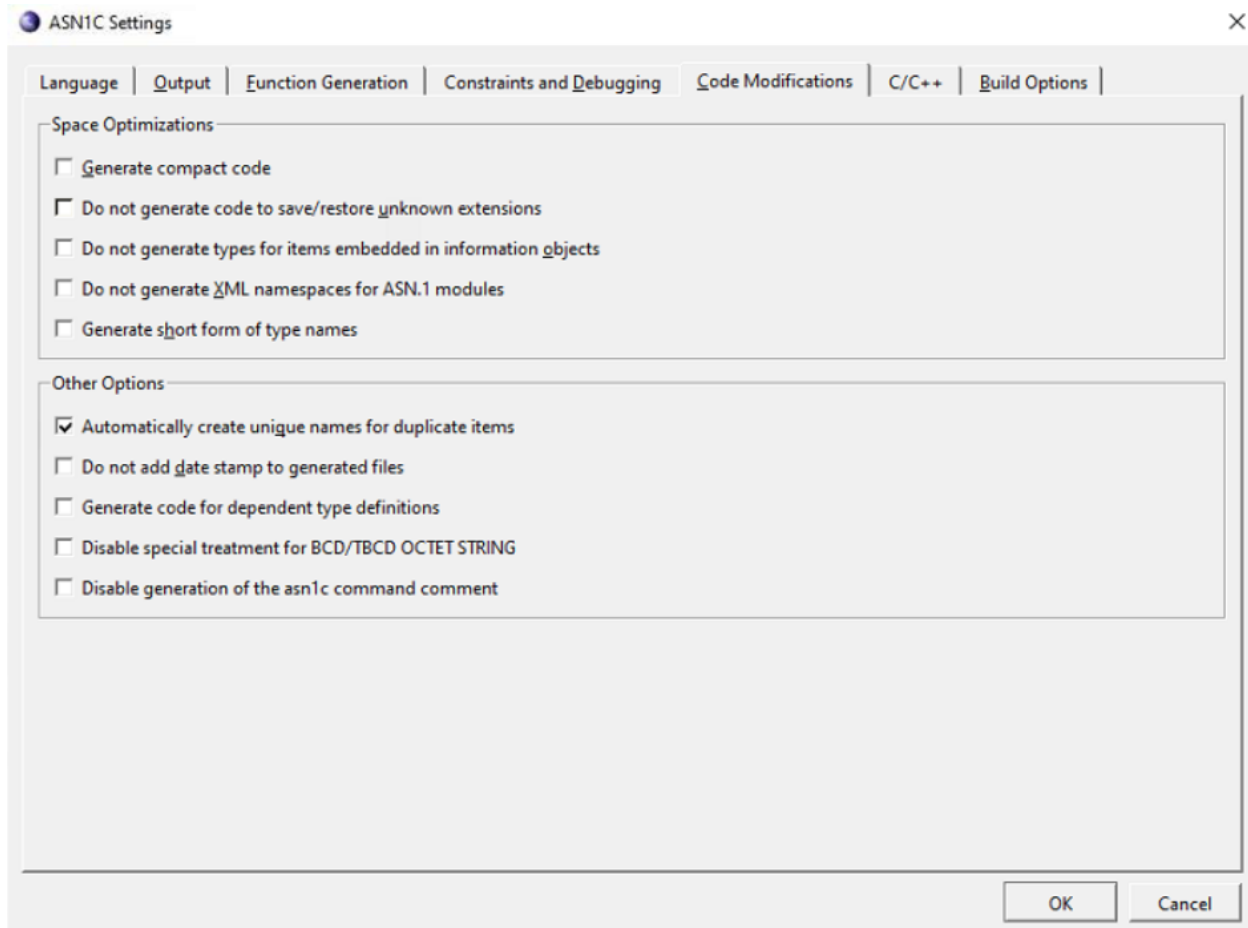
Constraints

This section provides users with the ability to add or remove various types of restriction checks from the generated code.

Debugging and Event Handling This section provides users with the ability to add debug tracing and event hooks. In addition to enabling event callbacks, generation of type structures can also be disabled, in which case generated decode functionality simply calls user-created event handlers and does not perform its own decoding operation.

Code Modifications tab

The Code Modifications tab contains a number of options for generating simplified code.



The Code Modifications tab contains the following sections:

Space Optimizations This section provides users with the ability to remove unwanted or unneeded functionality and shorten the names of generated types.

Other Options This section provides users with the ability to define several miscellaneous settings, including the option to generate code for types that have been imported into the current schema.

Language-specific tab

Additional code modification options that are language-specific are shown in a separate tab next to the Code Modifications tab. The label and contents of this tab changes based on the language selected within the Output tab.

For C/C++, the tab is displayed as follows, and includes several settings for adjusting how ASN.1 types are mapped to native C/C++ types:

C/C++ Code Modifications

Add a header guard prefix

Add a C++ namespace

SEQUENCE OF/SET OF use: default linked list dynamic array static array

Generate static member variables in choice constructs

Use enumerated types instead of integers

Generate fully qualified enumerated constants

Disable fixed sizing for small bit strings

Support indefinite PER lengths

Pad PER BIT STRING-contained values

Generate code compatible with 64-bit architectures

Generate code that uses C++11 features including STL

For C#, the tab is displayed as follows, and includes settings to allow for manipulating the namespace into which code is generated:

Note

The Java tab contains similar options.

C# Code Modifications

Specify namespace

Specify namespace prefix

Build Options tab

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

For C or C++, the tab is as follows:

The screenshot shows a GUI with several sections of options:

- Generate Visual Studio Project** (checkbox):
 - VS 2019 (radio)
 - VS 2017 (radio, selected)
 - VS 2015 (radio)
 - VS 2013 (radio)
 - VS 2012 (radio)
 - VS 2010 (radio)
 - VS 2008 (radio)
 - VS 2005 (radio)
 - VS 2003 (radio)
 - VS 6.0 (1998) (radio)
- Generate Makefiles** (checkbox):
 - Windows (nmake) (radio)
 - GNU (radio, selected)
- Build Libraries** (checkbox):
 - Generate static libraries (radio)
 - Generate shared libraries (radio)
 - Generate multi-threaded libraries (checkbox)
- C/C++ Compile Optimization** (checkbox):
 - Default (radio, selected)
 - Space Optimization (radio)
 - Speed (Time) Optimization (radio)
- Link applications using shared libraries (checkbox)
- Link applications against 64-bit libraries (checkbox)

A makefile can be generated in either Windows or GNU format. For Windows, a Visual Studio project can also be generated. Under the Build Libraries section, which generates the build script to build a library rather than an executable, the desired variety of library can be selected.

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

For C#, the tab is as follows:

Build Options

Generate a list of .cs files (in <modulename>.mk)

Generate a makefile

Generate a Visual Studio Project

- VS 2019
- VS 2017
- VS 2015
- VS 2013
- VS 2012
- VS 2010
- VS 2008
- VS 2005

Strongly named key file:

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 is also available.

For Java, the tab is as follows:

Build Options

Generate a list of .java files (in <modulename>.mk)

Generate an Ant build script

Generate a batch file or shell script

Like C#, Java can also provide a *.mk generated file list, as well as an Ant build script and a batch or shell script.

For Python, the tab is as follows:

Build Options

Create a batch file or shell script to generate Python code

For Python, ASNIC can create a batch file (Windows) or shell script (non-Windows) that generates the Python code as set up by the GUI settings.

For Go, the tab is as follows:

Build Options

- Create a makefile to generate Go code
- Create a JSON file with random test data
- Don't create a main.go file

The Go code generator can create a makefile to generate and build Go code. The generator can also create a JSON file with random test data. Additionally, an option for the generator not to create a main.go file (for instance, if there already is one that has been modified) is available.

Chapter 4. Generated Go Source Code

A Go source file with extension '.go' will be generated for most ASN.1 modules defined in the ASN.1 source file. If a module does not contain any ASN.1 types that would trigger generating a Go type or value, then no Go file is generated.

General Hierarchy of Generated Go Source Files

When Go code is generated, the output directory (which defaults to the current working directory) is the root of the Go module. ASN1C will use the last part of the directory name as the Go module path, e.g. if the output directory is `.../some/project1`, the Go module path will be `"project1"`. If a `go.mod` file does not exist in this location, one will be created with this module path. If a `main.go` file is being generated, it will also be generated here.

ASN1C will generate a Go package, in this module path, named `asn1gen`. This package name can be changed using the `-pkgname` option. A corresponding subdirectory will be created using this name. Files generated in this directory will contain the corresponding package clause of `package asn1gen`. Within this directory, subdirectory `asn1rt` will be created. The directory `asn1gen/asn1rt` contains the `asn1rt` package, which is the package for the base run-time.

Separate files are generated for each ASN.1 module in the `asn1gen` package, but everything is in that one package (aside from the run-time layer).

Executing `"go build"` will build the full package.

General Form of a Generated Go Source File

A generated Go file begins with `package asn1gen`. All generated Go code is included in the `asn1gen` package.

After that is an import statement, which will import the base run-time package, `asn1rt`, and any required built-in Go packages. This is followed by type and constant definitions. See the `"ASN.1 To Go Type Mappings"` section for details on the mapping of ASN.1 types to Go types.

For PER, type-specific encode and decode functions are generated. For JSON, custom marshal and unmarshal functions are generated for some types. The built-in Go marshaling logic is used wherever possible to do the JSON conversion.

Chapter 5. Generated Go Main File (main.go)

Go code is normally generated from within the `golang` subdirectory in an ASN1C installation. In the output directory (which defaults to the working directory) `main.go` and `go.mod` files are generated. The `main.go` file is a template file for using the files in the `asn1gen` package (see Chapter 4, *Generated Go Source Code* for general information on the overall structure of the generated code). The `main.go` file contains a sample writer section to encode a data record of the chosen Protocol Data Unit (PDU) type and to a file. The reader section reads decodes from the file and prints the results. Each is invoked by running the main program with an argument of 'writer' or 'reader', respectively.

By default, the `main.go` writer section contains a TODO comment to remind the user that a data variable of the PDU type needs to be populated prior to encoding. If the `-test` command-line option is specified, code will be added to populate the PDU variable with random test data.

The procedure to generate and build Go source code is rather straightforward:

1. Invoke the `asn1c` compiler on your ASN.1 source file(s) to generate Go source code. For example: `asn1c test.asn -go -per`
2. Invoke 'go build' to run the Go compiler to build the generated package.

That is it. If successful, an executable file should have been created which can be run with the writer or reader option.

Note that it is possible to use the `-make` command-line option to generate a makefile for future compilations. The user should also be careful to use the `-no-go-main` option on future compilations so as to not regenerate the `main.go` file in the event it was edited by the user.

Chapter 6. ASN.1 To Go Type Mappings

Type Mappings

BOOLEAN

A Go type is generated for the ASN.1 BOOLEAN type which equates the ASN.1 type assignment to the built-in Go *bool* type. For example:

ASN.1:

```
MyBool ::= BOOLEAN
```

Generated Go code:

```
type MyBool bool
```

If a BOOLEAN type is referenced directly in an element within a constructed type, the Go *bool* type is referenced directly.

INTEGER

A Go type is generated for the ASN.1 INTEGER type which equates the ASN.1 type assignment to a built-in Go 64-bit integer type. For an unconstrained integer type, the Go type used is a signed 64-bit integer (*int64*). For example:

ASN.1:

```
MyInt ::= INTEGER
```

Generated Go code:

```
type MyInt int64
```

If a constraint constrains the value to a nonnegative range, an unsigned 64-bit integer type is used (*uint64*):

ASN.1:

```
MyUInt ::= INTEGER (0..9)
```

Generated Go code:

```
type MyInt uint64
```

Note that no attempt is made to use smaller integer types for smaller ranges.

If an INTEGER type is referenced directly in an element within a constructed type, the Go integer type is referenced directly.

If the INTEGER type contains a list of named numbers, Go constants are generated for each number, the same as is done for the ASN.1 ENUMERATED type.

BIT STRING

The ASN.1 BIT STRING type is mapped to the Go *BitString* type defined within the ASN.1 run-time (*asn1rt.BitString*). This is in turn mapped to the *encoding/asn1.BitString* from the Go standard library. For example:

ASN.1:

```
MyBitStr ::= BIT STRING
```

Generated Go code:

```
type MyBitStr asn1rt.BitString
```

Bits are specified in the Bytes field in most-significant bit order. For example, the bit string '1'B (consisting of a single bit which is set) would be represented using BitLength = 1 and Bytes being a slice of 1 byte, set to 0x80.

Named Bits

In the ASN.1 standard, it is possible to define a named bit string that specifies names for different bit positions. ASN1C provides support for this type by generating Go constants and run-time functions that can be used to set, clear, or test these named bits. These constants equate the bit name to the bit number defined in the specification. They can be used with the *SetBit*, *ClearBit*, and *TestBit* run-time functions to set, clear, and test the named bits.

Contents Constraint

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

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

ASN1C will generate a type definition that references the type that is within the containing constraint. In this case, that would be INTEGER; therefore, the Go type for INTEGER would be used, which is int64. This direct use of the containing type can be suppressed through the use of the *-noContaining* command-line argument. In that case, a normal BIT STRING type will be used and it will be the user's responsibility to do the necessary packing and unpacking operations to encode and decode the variable correctly.

OCTET STRING

The ASN.1 OCTET STRING type is mapped to the Go OctetString type defined within the ASN.1 run-time (asn1rt.OctetString). This is in turn mapped to a byte slice. For example:

ASN.1:

```
MyOctStr ::= OCTET STRING
```

Generated Go code:

```
type MyOctStr asn1rt.OctetString
```

Contents Constraint

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

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

ASN1C will generate a type definition that references the type that is within the containing constraint. In this case, that would be INTEGER; therefore, the Go type for INTEGER would be used, which is int64. This direct use of

the containing type can be suppressed through the use of the *-noContaining* command-line argument. In that case, a normal OCTET STRING type will be used and it will be the user's responsibility to do the necessary packing and unpacking operations to encode and decode the variable correctly.

ENUMERATED

The Go mapping for an ASN.1 ENUMERATED type is the same as for INTEGER. By default, uint64 is used. If the ENUMERATED type contains negative-value identifiers, then int64 is used. Go constants are generated for each of the enumerated identifiers. The constant names consist of the type name concatenated with the enumerated identifier name in order to provide unique names. Note that for Go, camel-case is used in the generated identifiers and underscores are not used.

If the enumerated type is extensible, a special UNKNOWN identifier constant is added to the enumerated constant list.

If an enumerated type is referenced directly in an element within a constructed type, a compiler-generated type is generated for it consisting of the constructed type name concatenated with the element name to form a camel-cased identifier. This identifier is then used as the prefix for generated enumerated identifiers.

Example - Enumerated Type Assignment

ASN.1:

```
MyEnum ::= ENUMERATED { red(0), green(1), blue(2), ... }
```

Generated Go code:

```
type MyEnum uint64
const (
    MyEnumRed = 0
    MyEnumGreen = 1
    MyEnumBlue = 2
    MyEnumUNKNOWN = 3
)
```

Example - Enumerated Type Referenced in Element

ASN.1:

```
MySeq ::= SEQUENCE {
    enum ENUMERATED { a, b, c }
}
```

Generated Go code:

```
type MySeqEnum uint64
const (
    MySeqEnumA = 0
    MySeqEnumB = 1
    MySeqEnumC = 2
)

type MySeq struct {
    Enum MySeqEnum
}
```

NULL

A Go type is generated for the ASN.1 NULL type which equates the ASN.1 type assignment to the built-in Go *bool* type. The purpose of having a type for it is so that it can act as a placeholder to denote the presence or absence of this type within a constructed type. For example, if it used in an optional element or within a choice construct.

If a NULL type is referenced directly in an element within a constructed type, the Go bool type is referenced directly.

Example

ASN.1:

```
MyNull ::= NULL
```

Generated Go code:

```
type MyNull bool
```

OBJECT IDENTIFIER

The Go representation for OBJECT IDENTIFIER depends on whether the `-oid-as-astring` option is used. Without this option, the representation is a slice of integers; with it, the representation is a string.

Standard case (slice of integers)

By default, the ASN.1 OBJECT IDENTIFIER type is mapped to the Go `ObjectIdentifier` type defined within the ASN.1 run-time (`asn1rt.ObjectIdentifier`). This in turn is mapped to a `uint64` slice (`[]uint64`), with each `uint64` representing a single subidentifier. For example:

ASN.1:

```
MyOID ::= OBJECT IDENTIFIER
```

Generated Go code:

```
type MyOID asn1rt.ObjectIdentifier
```

String case

When `-oid-as-string` is used, OBJECT IDENTIFIER is mapped to `asn1rt.ObjectIdentifierStr`, defined in the ASN.1 run-time as `string`. The advantage of this representation is that it supports arbitrarily large subidentifier values. The string must be in dotted-number format (e.g. "0.5.1234").

For example:

ASN.1:

```
MyOID ::= OBJECT IDENTIFIER
```

Generated Go code:

```
type MyOID asn1rt.ObjectIdentifierStr
```

RELATIVE-OID

The ASN.1 RELATIVE-OID type mapping is the same as described above for OBJECT IDENTIFIER.

REAL

The ASN.1 REAL type is mapped to the Go Real type defined within the ASN.1 run-time (`asn1rt.Real`). This in turn is mapped to the Go built-in `float64` type.

If the REAL type is constrained to base 10 values, it is mapped to the Go `RealBase10` type defined within the ASN.1 run-time (`asn1rt.RealBase10`). This in turn is mapped to the Go built-in `string` type.

Example

ASN.1:

```
MyReal ::= REAL
MyReal10 ::= REAL(0 | WITH COMPONENTS { ..., base (10) })
```

Generated Go code:

```
type MyReal asn1rt.Real
type MyReal10 asn1rt.RealBase10
```

SEQUENCE/SET

An ASN.1 SEQUENCE or SET type is a constructed type consisting of a series of element definitions; each element has an assigned ASN.1 type. An ASN.1 SEQUENCE or SET is mapped to a Go struct type with each ASN.1 element having an equivalent Go element definition. The general form is as follows:

ASN.1 production:

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

Generated Go code:

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

The `<type1>` and `<type2>` placeholders represent the Go types for the ASN.1 types `<element1-type>` and `<element2-type>`, respectively.

Here is an example:

ASN.1:

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


Generated Go code:

```

type AX struct {
    A1 int64
    A2 bool
}

type A struct {
    X AX
    Y asn1rt.OctetString
}

```

First, note that the Go field names in the struct begin with uppercase letters. This means the names are exported for use outside of the package in which they are defined.

Second, notice that we generated type AX, even though no such type was defined in the ASN.1. This is the result of using a nested, constructed type in the ASN.1. The elements in a SEQUENCE or SET can be of any ASN.1 type, including other constructed types. Sometimes, one constructed type definition is nested inside another, rather than defining a separate type and referencing it by name. For example, it is possible to nest a SEQUENCE definition within another SEQUENCE definition (see element x in the above example).

The ASN1C compiler first recursively pulls all nested constructed elements out of the SEQUENCE and forms new internal types. The internal ASN.1 names of these types are of the form <name>_<element-name1>_<elementname2>_ . . . <element-nameN>. In the above example, the compiler internally created a type for element x: A_x. The compiler then generates code as if the ASN.1 had been written without nesting, using the type definitions it created internally. Transforming the ASN.1 name to a Go name produced AX.

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 compiler-generated types results in the following equivalent type definitions:

```

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

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

X ::= SEQUENCE { X-a a }

```

Note that the name for the aa element type is X-a-aa. It contains both the name for a (at level 1) and aa (at level 2). The concatenation of all of the intermediate element names can lead to very long names in some cases. To get around the problem, the *-shortnames* command-line option can be used to form shorter names. In this case, only the type name and the last element name are used. In the example above, this would lead to an element name of X-aa. The disadvantage of this is that the names may not always be unique. If using this option results in non-unique names, an *_n* suffix is added where n is a sequential number to make the names unique.

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

```

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

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

```

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

OPTIONAL keyword

Elements within a sequence can be declared to be optional using the `OPTIONAL` keyword. This indicates that the element is not required. Optional elements are handled in Go using pointer types. If the pointer to the element value is nil, it means the element is not present.

For example, the elements in the following production are marked as optional:

```

Aseq ::= SEQUENCE {
    x    INTEGER OPTIONAL,
    y    BOOLEAN OPTIONAL
}

```

In this case, the following Go type is generated:

```

type Aseq struct {
    X *int64
    Y *bool
}

```

When this structure is populated for encoding, if the element is present, the developer must set the pointer value to point at a value. This can be done using `new` or by assigning the value to a local variable and then storing the address of the variable in the generated structure. Conversely, when a message is decoded into this structure, the developer must test to see if the elements are nil to see if they were present in the message.

DEFAULT keyword

The `DEFAULT` keyword allows a default value to be specified for elements within the `SEQUENCE`. ASN1C will generate a non-pointer type for elements using `DEFAULT` if the type is one of several simple types (`BOOLEAN`, `INTEGER`, `ENUMERATED`, `REAL`, `BIT STRING`, `OCTET STRING`, and character string types). In other cases, ASN1C will generate a pointer type, the same as it does for an element with `OPTIONAL`; a nil pointer is interpreted as representing the default value.

For `PER`, elements set to their default value **MUST** be encoded as `ABSENT`. In the cases where ASN1C generates a non-pointer type, the compiler automatically generates code to meet this requirement. In the cases where ASN1C generates a pointer type, the programmer must set the pointer to nil rather than populate the field with the default value.

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 `[][]byte` and have the name `ExtElem1`. This holds an array of encoded components.

The *-noOpenExt* command line option can be used to alter this default behavior. If this option is specified, the *ExtElem1* element is not included in the generated structure; any extension data that may be present in a decoded message is skipped.

If the SEQUENCE type contains an extension marker and extension elements, then the actual extension elements will be present in addition to the *ExtElem1* element. These elements will be treated as optional elements whether they were declared that way or not. The reason is because a version 1 message could be received that does not contain the elements.

If version brackets are present, a struct is generated for each extension element group. This struct has type name of the form *<container>ExtGrpV<n>* where *<container>* is the name of the container type and *<n>* is the version number of the group. This type is then referenced as an optional element in the container type.

An example of a type with version brackets is as follows:

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

In this case, a structure named *TestSequenceExtGrpV3* is generated for the items in the version brackets. The number 3 is used because 2 would be the version extension number of the urgency field. The generated Go types for this construct would be as follows:

```
type TestSequenceExtGrpV3 struct {
    AlternateItemCode uint64
    AlternateItemName *string
}

type TestSequence struct {
    ItemCode uint64
    ItemName *string
    Urgency *uint64
    ExtGrpV3 *TestSequenceExtGrpV3
    ExtElem1 [][]byte
}
```

The presence or absence of the entire version block would be determined by whether the *ExtGrpV3* element pointer is set.

SEQUENCE OF/SET OF

An ASN.1 SEQUENCE OF or SET OF type is mapped to a Go slice. The general form is *[]<element type>* where *<element type>* would be replaced with the Go type of the SEQUENCE OF element.

SEQUENCE or SET OF Element in a Constructed Type

If an element in a SEQUENCE, SET, or CHOICE directly references a SEQUENCE OF or SET OF type, a slice type is generated inline. A separate type is not generated to be referenced by the element. For example, consider the following type definitions:

```
SeqOfInt ::= SEQUENCE OF INTEGER

Seq ::= SEQUENCE {
    a SeqOfInt,
    b SEQUENCE OF BOOLEAN
}
```

This results in the following Go types being generated:

```
type SeqOfInt []int64

type Seq struct {
    A SeqOfInt
    B []bool
}
```

Note that a separate type is not generated for the inline SEQUENCE OF BOOLEAN, the bool slice is generated inline.

SEQUENCE OF Constructed Type

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

In such cases, the compiler internally creates an ASN.1 type for the nested, constructed type. The format of the name for this type is as follows:

```
<ProdName>Element
```

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

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

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

In this case, a type is generated for the element of the SEQUENCE OF production. Internally, this is transformed into the equivalent ASN.1:

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

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

These types are then converted into the equivalent Go types using the standard mapping that was previously described.

SET OF

The ASN.1 SET OF type is mapped to a slice type, just the same as described previously for SEQUENCE OF.

CHOICE

The ASN.1 CHOICE type is converted into a Go struct containing an integer for the choice tag value (T) followed by a struct (U) with a field for each of the CHOICE elements. This differs from C/C++, where a union type is used for the set of alternatives. There is no union type in Go, so a struct containing pointer values for all of the alternatives is used. The T field indicates which of the alternatives is chosen and which of the fields in the U struct shall be non-nil; exactly one field shall be non-nil at a time.

The tag value is simply a sequential number starting at one for each alternative in the CHOICE. A Go constant is generated for each of these values. The format of this constant is <name><element-name>TAG where <name> is the name of the ASN.1 production and <element-name> is the name of the CHOICE alternative.

The fields of the U struct, corresponding to the alternatives for the CHOICE type, are generated much the same as for the elements of a SEQUENCE type (described earlier), except that pointer types are used. The compiler internally pulls out any nested constructed types and creates an ASN.1 type for it, and that is reflected in the generated code.

ASN.1 production:

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

Generated Go code:

```
const (
    <name>_<element1-name>TAG = 1
    <name>_<element2-name>TAG = 2
    ...
)

type <name> struct {
    T uint64
    U struct {
        <type1> *<element1-name>
        <type2> *<element2-name>
        ...
    }
}
```

Open Type

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

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

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

The *Open Type* is converted into a Go byte slice ([]byte). The slice is assumed to contain previously encoded ASN.1 data. When a message containing an open type is decoded, a copy of the encoded data is made.

If the *-tables* or *-table-unions* command line option is selected and the ASN.1 type definition references a table constraint, the code generated is different. In this case, instead of a byte slice, a union-type structure is used, similar to what is done for a CHOICE type. Each information object in the information object set referenced by the table constraint specifies an actual ASN.1 type that can be used for the open type; taken together, these objects represent all of the possible alternatives for the open type.

Character String Types

All ASN.1 character string types, including multi-byte types such as BMPString and UniversalString, are mapped to the Go string type. The Go string will be interpreted by the generated code as UTF-8.

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

```

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

```

Time String Types

The ASN.1 *GeneralizedTime* and *UTCTime* types are mapped to the Go string type.

EXTERNAL

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

```

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

```

The ASN.1 EXTERNAL type is mapped to a Go struct, generated from the above SEQUENCE definition.

EMBEDDED PDV

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

```

EmbeddedPDV ::= [UNIVERSAL 11] IMPLICIT SEQUENCE {
  identification CHOICE {
    syntaxes SEQUENCE {
      abstract OBJECT IDENTIFIER,
      transfer OBJECT IDENTIFIER
    },
    syntax OBJECT IDENTIFIER,
    presentation-context-id INTEGER,
    context-negotiation SEQUENCE {

```

```

        presentation-context-id INTEGER,
        transfer-syntax OBJECT IDENTIFIER
    },
    transfer-syntax OBJECT IDENTIFIER,
    fixed NULL
},
data-value OCTET STRING
}

```

The ASN.1 EMBEDDED PDV type is mapped to a Go struct, generated from the above SEQUENCE definition.

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 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 just as if the original type was declared to be an octet string of size 1 to 32. That is, it will map the type to a go byte slice (`[]byte`) and do constraint checks before encoding and after decoding to ensure the value is within the specified range.

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

```

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

```

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

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

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

The compiler performs the substitution to create the proper Go type for `SignedName`:

```

type SignedName struct {
    ToBeSigned Name
    AlgorithmID []uint64
    ParamS Params
    Signature asn1.BitString
}

```

}

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

For example, consider the following oft-repeated pattern from the UMTS 3GPP specifications:

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

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

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

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

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

Value Mappings

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

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

ASN.1 value specifications are mapped to Go constant declarations of various types with the name transformed to start with prefix 'Asn1v' followed by the value name transformed as per Go naming rules.

BOOLEAN Value

A BOOLEAN value is mapped to a Go true or false constant declaration:

ASN.1 production:

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

Generated code:

```
const Asn1v<name> bool = <value>;
```

ASN.1 BOOLEAN values are the uppercase names TRUE and FALSE. The equivalent Go names are true and false.

INTEGER Value

An INTEGER type is mapped to a Go integer value constant:

ASN.1 production:

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

Generated code:

```
const Asn1v<name> int64 = <value>
```

If the integer type is constrained to be nonnegative, than uint64 is used as the Go type for the constant.

REAL Value

A REAL type value is mapped to a Go floating point constant:

ASN.1 production:

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

Generated code:

```
const Asn1v<name> float64 = <value>
```

ENUMERATED Value

The mapping of an ASN.1 enumerated value declaration to Go is to an integer constant.

ASN.1 production:

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

Generated code:

```
const <name> = <int value>
```

Where <int value> is the integer value associated with the ASN.1 enumerated value.

BIT STRING Value

ASN.1 values of type BIT STRING are mapped to Go variables of type `asn1.BitString`. The ASN.1 values may be defined using binary strings, hexadecimal strings, or using sets of named bits:

ASN.1 production:

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

Generated code :

```
var Asn1v<name> asn1.BitString = asn1.BitString{ BitLength: <len>, Bytes: []byte{<data>}}
```

where `<len>` would be the length in bits and `<data>` would be a comma-separated list of hexadecimal byte constants.

OCTET STRING Value

ASN.1 values of type OCTET STRING are mapped to Go variables of type `[]byte`. The ASN.1 values may be defined using binary or hexadecimal strings. If a binary string is used and the bit count is not evenly divisible by eight, the last byte is zero-padded on the right with zero bits to form a full byte.

ASN.1 production:

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

Generated code :

```
var Asn1v<name> []byte = []byte{<data>}}
```

where `<data>` would be a comma-separated list of hexadecimal byte constants.

Character String Value

All ASN.1 character string type values are mapped to Go string constants.

ASN.1 production:

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

Generated code:

```
const Asn1v<name> string = <value>
```

Object Identifier Value Specification

ASN.1 values of type OCTET STRING are mapped to Go variables of type `[]uint64`.

ASN.1 production:

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

Generated code:

```
var Asn1v<name> []uint64 = []uint64{<arcs>};
```

Where `<arcs>` is a comma-separated list of the OID integer arc values.

For example, consider the following declaration:

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

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

```
var AsnlvOid []uint64 = []uint64{ 0, 5, 10 }
}
```

Constructed Type Values

Generation of Go variables or constants for constructed type value specifications is not supported in this release.

Table Constraint Related Structures

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

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

For Go, the table constraint code generation model is roughly equivalent to C when the *-table-unions* option is used. That is, the generated structures are very similar to what would be generated for CHOICE constructs.

Unions Table Constraint Model

The unions table constraint model originated from common patterns used in a series of ASN.1 specifications in-use in 3rd Generation Partnership Project (3GPP) standards. These standards include Node Application Part B (NBAP) and S1AP and X2AP protocols in 4G network (LTE) standards. This pattern is repeated in 5G standards such as NGAP.

Generated Go Type Definitions for Message Types

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

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

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

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

```
InitiatingMessage ::= SEQUENCE {
```

```

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

```

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

The generated Go structure is similar to what is used to represent a CHOICE type. The structure consists of a choice selector value (T) followed by an inner structure (U) containing pointers to all of the alternative types that can appear in the field. This is the general form:

```

type <Type> struct {
    <element1> <Element1Type>
    <element2> <Element2Type>

    /**
     * information object selector
     */
    T uint64

    /**
     * <ObjectSet> information objects
     */
    U struct {
        /**
         * <element1> : <object1-element1-value>
         * <element2> : <object1-element2-value>
         */
        <object1-name> *<object1-element3-type>

        /**
         * <element1> : <object2-element1-value>
         * <element2> : <object2-element2-value>
         */
        <object2-name>; *<object2-element3-type>

        ...
    }
}

```

In this definition, the first two elements of the sequence would use the equivalent Go type as defined in the fixed-type field in the information object. The open type field (element3) would be expanded into the union structure as is shown. The T value would hold a sequential number which is generated to represent each of the choices in the referenced information object set. The union struct then contains an entry for each of the possible types as defined in the object set that can be used in the open type field. Comments are used to list the fixed-type fields corresponding to each open type field.

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

```

const (
    S1APDUDescriptionsS1APELEMENTARYPROCEDURESUNDEFTAG = 0

```

```

SlAPPDUDescriptionsSlAPELEMENTARYPROCEDURESHandoverPreparationTAG = 1
SlAPPDUDescriptionsSlAPELEMENTARYPROCEDURESHandoverResourceAllocationTAG = 2
SlAPPDUDescriptionsSlAPELEMENTARYPROCEDURESPathSwitchRequestTAG = 3
    etc..
)

type InitiatingMessage struct {
    ProcedureCode uint64
    Criticality uint64
    Value struct {
        /**
         * information object selector
         */
        T uint64

        /**
         * SlAP-ELEMENTARY-PROCEDURES information objects
         */
        U struct {
            /**
             * procedureCode: id-HandoverPreparation
             * criticality: CriticalityReject
             */
            // T = 1
            HandoverPreparation *HandoverRequired

            /**
             * procedureCode: id-HandoverResourceAllocation
             * criticality: CriticalityReject
             */
            // T = 2
            HandoverResourceAllocation *HandoverRequest

            /**
             * procedureCode: id-PathSwitchRequest
             * criticality: CriticalityReject
             */
            // T = 3
            PathSwitchRequest *PathSwitchRequest

            etc..
        }
    }
}

```

Note that the long names generated for the TAG constants can be reduced by using the <alias> configuration element.

Generated Go Type Definitions for Information Element (IE) Types

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

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

```

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

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

```

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

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

```

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

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

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

```

In this case, standard parameterized type instantiation is used to create a type definition for the protocolIEs element. This results in the following element declaration in the HandoverRequired type:

```
ProtocolIEs []HandoverRequiredProtocolIEsElement
```

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

```

type HandoverRequiredProtocolIEsElement struct {
  Id uint64
  Criticality uint64
  Value struct {
    /**
     * information object selector
     */
    T uint64

    /**
     * HandoverRequiredIEs information objects
     */
    U struct {
      /**
       * id: id-MME-UE-S1AP-ID
       * criticality: CriticalityReject
       * presence: PresenceMandatory

```

```

*/
// T = 1
HandoverRequiredIEsIdMMEUES1APID *uint64

/**
 * id: id-eNB-UE-S1AP-ID
 * criticality: CriticalityReject
 * presence: PresenceMandatory
 */
// T = 2
HandoverRequiredIEsIdENBUES1APID *uint64

...
}
}
}

```

In this case, the protocol IE id field and criticality are generated as usual using the fixed-type field type definitions. The open type field once again results in the generation of a union structure of all possible type fields that can be used.

Note in this case the field names are automatically generated (HandoverRequiredIEsIdMMEUES1APID, etc.). The reason for this was the use of inline information object definitions in the information object set as opposed to defined object definitions. This is a sample from that set:

```

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

```

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

Chapter 7. Generated BER/DER Functions

Generated BER/DER Encode Functions

BER/DER encode functions are generated when the *-ber* (basic encoding rules) or *-der* (distinguished encoding rules) switch is specified on the command line. Encoding for BER and DER is quite similar, with the more restrictive DER being also acceptable as a BER encoding. To produce a valid DER encoding, you must use *-der*, which will generate DER encode functions instead of BER encode functions. For the remainder of this section, unless otherwise noted, you may understand BER to refer to DER in the case that *-der* was used.

For each ASN.1 production defined in the ASN.1 source file, a BER encode function is generated. This function will convert a populated Go variable of the given type into a BER encoded ASN.1 message.

Generated Go Function Format and Calling Parameters

The primary interface to encode data in Go is through the use of the generated *Marshal* function. This function has the same signature as that found in the Go `encoding/asn1` package:

```
func Marshal(val interface{}) ([]byte, error)
```

The `val` argument holds a value of the type to be encoded. Note that the *Marshal* function only encodes types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the *-pdu* command-line option to explicitly select types as PDU types.

The *Marshal* function returns the encoded message in memory as a byte slice. If any errors occur during encoding, the error result is returned in the error return value. The encoded value byte slice will be `nil` in this case.

Example of Calling the Go Marshal Function

The following code snippet was derived from the `golang/sample_ber/employee` sample program in the ASN1C distribution:

```
package main

import (
    "employee/asn1gen"
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
    var pdu asn1gen.PersonnelRecord

    // populate PDU type variable to be encoded
    fillName(&pdu.Name, "John", "P", "Smith")

    pdu.Title = "Director"
    pdu.Number = 51
    pdu.DateOfHire = "19710917"
```



```

fillName(&pdu.NameOfSpouse, "Mary", "T", "Smith")

// fill first child record

var childInfo asn1gen.ChildInformation
fillName(&childInfo.Name, "Ralph", "T", "Smith")
childInfo.DateOfBirth = "19571111"
pdu.Children = append(pdu.Children, childInfo)

// fill second child record

fillName(&childInfo.Name, "Susan", "B", "Jones")
childInfo.DateOfBirth = "19590717"
pdu.Children = append(pdu.Children, childInfo)

// Marshal

encodedData, err := asn1gen.Marshal(pdu)

if err == nil {
    fmt.Printf("PersonnelRecord encoded successfully\n")
    fmt.Printf("%s\n", hex.Dump(encodedData))
    err = ioutil.WriteFile("message.per", encodedData, 0777)
} else {
    fmt.Printf("Encoding failed: %s\n", err)
}

```

How Marshal Works And Alternatives

The Marshal function is the most convenient way to encode, but taking a look at how it works also provides alternatives, should you want them.

First, we'll look at the other BER encode functions that are generated. There are two general forms:

```

func BerEncode<type name>(pctxt *asn1rt.OSRTContext, value <go type>) (err error)

func (pvalue *<type name>) BerEncode(pctxt *asn1rt.OSRTContext) (err error)

```

Here are two examples from the employee sample:

```

func BerEncodeEmployeeNumber(pctxt *asn1rt.OSRTContext, value int64) (err error)

func (pvalue *Name) BerEncode(pctxt *asn1rt.OSRTContext) (err error)

```

The first form is used when the Go type for the ASN.1 type is a built-in Go type - so basically it is used for the simple, nonconstructed ASN.1 types. The second form is used when the Go type is a generated type.

Now we can look at the Marshal function to see what it does and how these functions are used:

```

// Create context object to manage encoding
pctxt := new(asn1rt.OSRTContext)

```

```
pctxt.InitBerEncode(1024) // Initialize for encoding
...
err = v.BerEncode(pctxt) // Invoke the encode function; v is a generated type
...
pctxt.FlushBuffer()

return pctxt.GetBerEncData(), err // access the encoded data
```

Generated BER/DER Decode Functions

BER or DER decode functions are generated when the *-ber* (basic encoding rules) or the *-der* (distinguished encoding rules) switch is specified on the command line. BER and DER decode functions are similar. The only difference is that the DER functions contain additional checks to make sure the encoding is in the required canonical form. For the remainder of this section, the acronym BER will be used to mean either BER or DER.

For each ASN.1 production defined in the ASN.1 source file, a BER decode function is generated. This function decodes an encoded ASN.1 message contained in a message buffer (byte slice) and stores the results in a data variable of a Go type corresponding to the ASN.1 production type.

DER Restriction Violations

The DER decode functions validate that the encoding follows the restrictions imposed by DER (e.g. that SET OF components are sorted). If a restriction was violated by the encoder, the DER decode function will add a `CanonicalError` to the context's warnings (`asn1rt.OSRTContext.Warnings`). The warnings are held in an `asn1rt.OSRTWarnings` object. Note the following properties of `asn1rt.OSRTWarnings`:

- It implements interface `error`. As noted below, `Unmarshal` will return it as an error.
- Its `Remove` function may be used to remove all `CanonicalError` objects from the list, should you wish to ignore those warnings and treat anything that remains as an error.
- Its `RemoveAllExcept` function may be used to remove all objects except for `CanonicalError` objects, should you wish to treat canonical violations, and not other warnings, as an error.

Generated Go Function Format and Calling Parameters

The primary interface to decode data in Go is through the use of the generated `Unmarshal` function. This function has the same signature as that found in the Go encoding/asn1 package:

```
func Unmarshal(b []byte, val interface{}) (rest []byte, err error)
```

The `b` argument is used to pass a byte slice into the function containing the message to be decoded. The `val` argument holds a pointer to a variable of the type into which the message is to be decoded. Note that the `Unmarshal` function only supports decoding of types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the *-pdu* command-line option to explicitly select types as PDU types>

The `Unmarshal` function returns any unused bytes after the message is decoded in the `rest` argument. This makes it possible to decode a buffer containing multiple messages by looping to decode a message and then assigning the rest result to the message buffer. When the buffer is empty, decoding would be terminated. If any errors occur during decoding, the error result is returned in the `err` error return value.

In the case of DER, if the `Unmarshal` function does not encounter any errors but does encounter warnings (such as for DER violations), it will return those warnings as an error, in the form of an `asn1rt.OSRTWarnings` object. If you do not want to treat those warnings as an error, you will need to ignore the returned error object when it is of that type.

Example of Calling the Go Unmarshal Function

The following code snippet was derived from the `golang/sample_ber/employee` sample program in the ASN1C distribution:

```
package main

import (
    "employee/asn1gen"
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
    var pdu asn1gen.PersonnelRecord
    encoding, err := ioutil.ReadFile("message.ber")
    if err != nil {
        fmt.Printf("Read message.ber failed: %s\n", err)
    }
    return
}

_, err = asn1gen.Unmarshal(encoding, &pdu)

if err == nil {
    fmt.Printf("Decoding was successful:\n")
} else {
    fmt.Printf("Decoding failed: %s\n", err)
}
```

How Unmarshal Works And Alternatives

The `Unmarshal` function is the most convenient way to decode, but taking a look at how it works also provides alternatives, should you want them.

First, we'll look at the other BER decode functions that are generated. There are two general forms:

```
func BerDecode<type name>(pctxt *asn1rt.OSRContext, impltag bool, impllen *asn1rt.Length) (pvalue *<type name>)
func (pvalue *<type name>) BerDecode(pctxt *asn1rt.OSRContext, impltag bool, impllen *asn1rt.Length)
```

Here are two examples from the `employee` sample:

```
func BerDecodeDate(pctxt *asn1rt.OSRContext, impltag bool, impllen *asn1rt.Length) (pvalue *Date)
func (pvalue *Name) BerDecode(pctxt *asn1rt.OSRContext, impltag bool, impllen *asn1rt.Length)
```

The first form is used when the Go type for the ASN.1 type is a built-in Go type - so basically it's used for the simple, nonconstructed ASN.1 types. The second form is used when the Go type is a generated type. The 'impltag' and 'impllen'

arguments are for dealing with implicit tags. A user calling a function of this form at the top level would almost always set these arguments to 'false' and 'nil' respectively.

Now we can look at the Unmarshal function to see what it does and how these functions are used:

```
// Create context object to manage encoding
pctx := new(asn1rt.OSRContext)
pctx.InitDecodeBytes(b) // Initialize for decoding; b holds the encoded data
...
err = v.BerDecode(pctx, false, nil) // Invoke the decode function; v is a generated
...
return pctx.BufferRemBytes(), err
```

Chapter 8. Generated PER Functions

Generated PER Encode Functions

PER encode functions are generated when the *-per* (packed encoding rules), *-aper* (aligned packed encoding rules), or *-uper* (unaligned packed encoding rules), switch is specified on the command line. For Go, *-per* and *-aper* have the same meaning - aligned PER encoders will be generated. The *-uper* option is used to generate unaligned decoders. There is no option to switch between aligned and unaligned at run-time as there is for some other languages. For the remainder of this section, the acronym PER will be used to mean either aligned or unaligned PER.

For each ASN.1 production defined in the ASN.1 source file, a PER encode function is generated. This function will convert a populated Go variable of the given type into a PER encoded ASN.1 message.

Generated Go Function Format and Calling Parameters

The primary interface to encode data in Go is through the use of the generated *Marshal* function. This function has the same signature as that found in the Go `encoding/asn1` package:

```
func Marshal(val interface{}) ([]byte, error)
```

The `val` argument holds a value of the type to be encoded. Note that the *Marshal* function only encodes types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the *-pdu* command-line option to explicitly select types as PDU types.

The *Marshal* function returns the encoded message in memory as a byte slice. If any errors occur during encoding, the error result is returned in the error return value. The encoded value byte slice will be `nil` in this case.

Example of Calling the Go Marshal Function

The following code snippet was derived from the `golang/sample_per/employee` sample program in the ASN1C distribution:

```
package main

import (
    "employee/asn1gen"
    "encoding/hex"
    "fmt"
    "io/ioutil"
    "os"
)

func main() {
    var pdu asn1gen.PersonnelRecord

    // populate PDU type variable to be encoded
    fillName(&pdu.Name, "John", "P", "Smith")

    pdu.Title = "Director"
    pdu.Number = 51
    pdu.DateOfHire = "19710917"
```

```

fillName(&pdu.NameOfSpouse, "Mary", "T", "Smith")

// fill first child record

var childInfo asnlgen.ChildInformation
fillName(&childInfo.Name, "Ralph", "T", "Smith")
childInfo.DateOfBirth = "19571111"
pdu.Children = append(pdu.Children, childInfo)

// fill second child record

fillName(&childInfo.Name, "Susan", "B", "Jones")
childInfo.DateOfBirth = "19590717"
pdu.Children = append(pdu.Children, childInfo)

// Marshal

encodedData, err := asnlgen.Marshal(pdu)

if err == nil {
    fmt.Printf("PersonnelRecord encoded successfully\n")
    fmt.Printf("%s\n", hex.Dump(encodedData))
    err = ioutil.WriteFile("message.per", encodedData, 0777)
} else {
    fmt.Printf("Encoding failed: %s\n", err)
}

```

How Marshal Works And Alternatives

The Marshal function is the most convenient way to encode, but taking a look at how it works also provides alternatives, should you want them.

First, we'll look at the other PER encode functions that are generated. There are two general forms:

```

func PerEncode<type name>(pctxt *asn1rt.OSRTContext, value <go type>) (err error)

func (pvalue *<type name>) PerEncode(pctxt *asn1rt.OSRTContext) (err error)

```

Here are two examples from the employee sample:

```

func PerEncodeEmployeeNumber(pctxt *asn1rt.OSRTContext, value int64) (err error)

func (pvalue *Name) PerEncode(pctxt *asn1rt.OSRTContext) (err error)

```

The first form is used when the Go type for the ASN.1 type is a built-in Go type - so basically it's used for the simple, nonconstructed ASN.1 types. The second form is used when the Go type is a generated type.

Now we can look at the Marshal function to see what it does and how these functions are used:

```

// Create context object to manage encoding

```

```

pctxt := new(asn1rt.OSRTContext)
pctxt.InitEncode() // Initialize for encoding
pctxt.NewBitFieldList() // Optional: enable bit tracing
...
err = v.PerEncode(pctxt) // Invoke the encode function; v is a generated type
...
pctxt.FlushBuffer()
pctxt.PrintBitFieldList("pdu") // Optional: print bit trace

return pctxt.BufferData(), err // access the encoded data

```

Generated PER Decode Functions

PER decode functions are generated when the *-per* (packed encoding rules), *-aper* (aligned packed encoding rules), or *-uper* (unaligned packed encoding rules), switch is specified on the command line. For Go, *-per* and *-aper* have the same meaning - aligned PER decoders will be generated. The *-uper* option is used to generate unaligned encoders. There is no option to switch between aligned and unaligned at run-time as there is for some other languages. For the remainder of this section, the acronym PER will be used to mean either aligned or unaligned PER.

For each ASN.1 production defined in the ASN.1 source file, a PER decode function is generated. This function decodes an encoded ASN.1 message contained in a message buffer (byte slice) and stores the results in a data variable of a Go type corresponding to the ASN.1 production type.

Generated Go Function Format and Calling Parameters

The primary interface to decode data in Go is through the use of the generated *Unmarshal* function. This function has the same signature as that found in the Go `encoding/asn1` package:

```
func Unmarshal(b []byte, val interface{}) (rest []byte, err error)
```

The `b` argument is used to pass a byte slice into the function containing the message to be decoded. The `val` argument holds a pointer to a variable of the type into which the message is to be decoded. Note that the `Unmarshal` function only supports decoding of types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the *-pdu* command-line option to explicitly select types as PDU types>

The `Unmarshal` function returns any unused bytes after the message is decoded in the `rest` argument. This makes it possible to decode a buffer containing multiple messages by looping to decode a message and then assigning the rest result to the message buffer. When the buffer is empty, decoding would be terminated. If any errors occur during decoding, the error result is returned in the `err error` return value.

Example of Calling the Go Unmarshal Function

The following code snippet was derived from the `golang/sample_per/employee` sample program in the ASN1C distribution:

```

package main

import (
    "employee/asn1gen"
    "encoding/hex"
    "fmt"
    "io/ioutil"

```

```

"os"
)

func main() {
    var pdu asnlgen.PersonnelRecord
    encoding, err := ioutil.ReadFile("message.per")
    if err != nil {
        fmt.Printf("Read message.per failed: %s\n", err)
    }
    return
}

_, err = asnlgen.Unmarshal(encoding, &pdu)

if err == nil {
    fmt.Printf("Decoding was successful:\n")
} else {
    fmt.Printf("Decoding failed: %s\n", err)
}
}

```

How Unmarshal Works And Alternatives

The Unmarshal function is the most convenient way to decode, but taking a look at how it works also provides alternatives, should you want them.

First, we'll look at the other PER decode functions that are generated. There are two general forms:

```

func PerDecode<type name>(pctxt *asnlrt.OSRtContext) (value <go type>, err error)

func (pvalue *<type name>) PerDecode(pctxt *asnlrt.OSRtContext) (err error)

```

Here are two examples from the employee sample:

```

func PerDecodeEmployeeNumber(pctxt *asnlrt.OSRtContext) (value int64, err error)

func (pvalue *Name) PerDecode(pctxt *asnlrt.OSRtContext) (err error)

```

The first form is used when the Go type for the ASN.1 type is a built-in Go type - so basically it's used for the simple, nonconstructed ASN.1 types. The second form is used when the Go type is a generated type.

Now we can look at the Unmarshal function to see what it does and how these functions are used:

```

// Create context object to manage encoding
pctxt := new(asnlrt.OSRtContext)
pctxt.InitDecodeBytes(b) // Initialize for decoding; b holds the encoded data
pctxt.NewBitFieldList() // Optional: enable bit tracing
...
err = v.PerDecode(pctxt) // Invoke the decode function; v is a generated type
...
pctxt.AlignBuffer() // Optional: prepare to decode a second message
pctxt.PrintBitFieldList("pdu") // Optional: print bit trace

```


PER Bit Tracing

To print out a bit trace using ASN1C, invoke the appropriate functions after encoding or decoding a message and a marked-up binary dump of the PER encoded data is produced.

To enable bit tracing logic to be embedded in the generated code, the *-trace* option needs to be specified on the ASN1C command-line. If using the GUI, the *Add tracing diagnostic messages to code* box must be checked under the 'Constraints and Debugging' tab.

If enabled, a display similar to the following will be printed out from within the Marshal or Unmarshal function:

```
Dump of decoded bit fields:
employee childrenPresent
1xxxxxxx ----- 80----- .---

employee.name.givenName length
----- 00000100 ----- --04---- .--

employee.name.givenName data
----- 01001010 01101111 ---4a6f --Jo
01101000 01101110 ----- 686e---- hn--
```

Here, a description can be seen of the field that was decoded, followed by 4 columns of a base 2 representation of the relevant input data, followed by a column with a hexadecimal representation of the same data, followed by a column with an ASCII representation of the same data. The hexadecimal and ASCII representations are only printed after all bits for a full byte have been processed, so these representations might not be printed on every line, and when they are printed, might correspond to binary bits on several preceding lines.

The x values indicate padding bits used in aligned PER.

Chapter 9. Generated JSON Functions

The Go language contains built-in support for Marshaling/Unmarshaling data to and from JSON format. However, the default representation of the data is not always compliant with the JSON encoding rules (JER) as defined in ITU-T X.697.

Go provides mechanisms to customize the default marshalling behavior so that alternate JSON formats, including JER, can be supported. This is done through two code alterations:

- Setting custom JSON attributes
- Creating custom JSON Marshal/Unmarshal functions for specific types.

The following sections describe what is done in each of these areas and also general procedures for encoding and decoding JSON data.

Custom JSON Attributes

Custom JSON attributes are mainly used to alter the marshaling and unmarshaling of elements within a SEQUENCE or SET. They allow the following alterations to be performed:

- Change element name. This allows the ASN.1 name to be substituted for the Go element name.
- Omit empty items from the encoding (omitempty). This allows optional elements to be omitted.
- Remove elements (-). This allows elements to be removed that cannot be represented in JER (for example, an unknown extension).

Custom JSON Marshal/Unmarshal Functions

Custom JSON marshal and unmarshal functions are used to alter the encoded content for certain types. For example, the content of a BIT STRING in JER does not match what would be encoded by default. The JER encoding has a 'length' and 'value' member. A customized Marshal and Unmarshal method are therefore generated in the run-time to modify the encoded value to be in the JER format.

Invoking JSON Marshal/Unmarshal Functions

JSON marshal and unmarshal functions are invoked using the standard Go JSON calling conventions defined in encoding/json. For Marshal the format is:

```
func Marshal(v interface{}) ([]byte, error)
```

where *v* is the value to be encoded. The encode result is returned as a byte slice.

For Unmarshal, the format is:

```
func Unmarshal(data []byte, v interface{}) error
```

where *data* is a byte buffer containing the encoded JSON data to be unmarshaled (decoded) and *v* is the target variable into which the decoded data will be stored.

Chapter 10. Generated Print Functions

Generated Print Functions

The following options are available for generating code to print the contents of variables of generated types:

-print - This is the standard print option that causes print functions to be generated that output data to the standard output device (*stdout*).

-genPrtToStr - This option causes print functions to be generated that write their output to a string variable.

Print to Standard Output

The **-print** option causes functions to be generated that print the contents of variables of generated types to the standard output device. In this case, a function named *Print* is generated in the *Marshal.go* file with the following signature:

```
func Print(val interface{})
```

The *val* argument holds a value of the type to be printed. Note that the *Print* function only prints types determined to be Protocol Data Unit (PDU) types. By default, a PDU type in a schema is a type not found to be referenced by any other type. This behavior can be overridden by using the **-pdu** command-line option to explicitly select types as PDU types.

Print to String

The **-prttostr** option causes functions to be generated that print the contents of variables of generated types to a string variable. In this case, a function named *PrintToString* is generated in the *Marshal.go* file with the following signature:

```
func PrintToString(val interface{}) string
```

The *val* argument holds a value of the type to be printed. As was the case for the *Print* function, only variables of PDU types may be passed to the function for printing.

Chapter 11. 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.

To avoid having ASN1C search for additional source files, simply provide the definition of each imported module in one of the source files specified as input to ASN1C. The imported module may be defined in its own source file, or multiple module definitions can be located in a single source file. Here is an example of combining modules into a single source file:

```
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.