User Guide for TITAN Standalone ASN.1 Encoder

Botond Baranyi

Version 11.1.0, 2025-05-28

Table of Contents

Abstract

This document describes detailed information on using the TITAN Standalone ASN.1 Encoder.

Chapter 1. About the Document

1.1. Purpose

The purpose of this document is to provide detailed information on using the Standalone ASN.1 Encoder from the TITAN toolset, that is, encoding and decoding values of types defined in ASN.1 modules.

1.2. Target Groups

This document is intended for users of the TITAN Standalone ASN.1 Encoder.

1.3. Typographical Conventions

This document uses the following typographical conventions:

- **Bold** is used to represent graphical user interface (GUI) components such as buttons, menus, menu items, dialog box options, fields and keywords, as well as menu commands. Bold is also used with '+' to represent key combinations. For example, **Ctrl+Click**
- The character '/' is used to denote a menu and sub-menu sequence. For example, **File / Open**.
- `Monospaced` font is used represent system elements such as command and parameter names, program names, path names, URLs, directory names and code examples.
- **`Bold monospaced`** font is used for commands that must be entered at the Command Line Interface (CLI).

Chapter 2. Overview

The Standalone ASN.1 Encoder is a subset of the TITAN TTCN-3 Test Executor. It only has the ability to compile ASN.1 modules, and it provides a C++ interface for encoding and decoding ASN.1 values. It cannot be executed on its own. It is expected to be integrated into an existing C++ project.

The TTCN-3 Test Executor is an implementation of the TTCN–3 Core Language standard with support of ASN.1. There are limitations to supported ASN.1 language constructs in the TTCN-3 Test Executor which also apply to the Standalone ASN.1 Encoder. Information on these limitations and some clarifications on how the ASN.1 standard has been implemented in TITAN can be found in the TITAN programmer's technical reference guide.

2.1. Components

The main components of the Standalone ASN.1 Encoder are the following:

- The **Compiler**, which translates ASN.1 modules into C++ program code.
- The **Base Library**, written in C++, which contains important supplementary functions for the generated code, but does not contain a `main` function.

The generated C++ modules should be compiled to binary object code and linked together with the Base Library (and with the rest of the user's project) using a traditional C++ compiler.

Chapter 3. Generating Code and Building

This section describes the steps needed to integrate ASN.1 modules into an existing C++ project using the TITAN Standalone ASN.1 Encoder.

NOTE	The instructions and examples in this section assume that the environment variable `$TTCN3_DIR` contains the installation path of TITAN.

3.1. Compiling ASN.1 Modules and Generating Code

The Standalone ASN.1 Encoder uses the same `compiler` for ASN.1 modules as the TTCN-3 Toolset.

The command line option `--asn1enc` instructs the ASN.1 compiler to only generate the C++ code necessary for ASN.1 encoding and decoding. It is followed by the list of files containing ASN.1 modules.

Example:

```
$TTCN3_DIR/bin/compiler --asn1enc MyModule1.asn MyModule2.asn
```

This command analyses the ASN.1 modules `MyModule1.asn` and `MyModule2.asn`, and generates the C++ headers `MyModule1.hh` and `MyModule2.hh`, and the C++ source files `MyModule1.cc` and `MyModule2.cc` (if no syntax or semantic errors were found).

There are also command line options for disabling the generation of encoder/decoder functions for each codec. These are: `-b` for `BER`, `-j` for `JSON`, `-o` for `OER`, and `-p` for `PER`. These can only be added before the list of ASN.1 modules.

Example:

```
$TTCN3_DIR/bin/compiler --asn1enc -j MyModule1.asn MyModule2.asn
```

Similar to the previous example, except no `JSON` encoder/decoder functions are generated in this case.

3.2. C++ Compilation and Linking

When compiling the generated C++ code, and any user source files that include headers generated by the ASN.1 compiler, the following pre-processor flags must be set:

- `-DTITAN_ASN1ENC`: this is needed by TITAN runtime headers;
- `-DSOLARIS`, `-DSOLARIS8`, `-DLINUX`, `-DFREEBSD`, `-DWIN32` or `-DINTERIX`: this indicates which platform the project is built on;
- `-I$TTCN3_DIR/include`: this sets the path to the TITAN runtime headers.

The following flags must be used when linking the final project:

- `-lasn1enc` or `-lasn1enc-dynamic`: links the static or dynamic Standalone ASN.1 Encoder base library;
- `-lcrypto`: links an OpenSSL library needed by TITAN;
- `-L$TTCN3_DIR/lib`: this sets the path to the TITAN base libraries.

Example:

```
g++ -c -DTITAN_ASN1ENC -DLINUX -I$TTCN3_DIR/include -o MyModule1.o MyModule1.cc
g++ -c -DTITAN_ASN1ENC -DLINUX -I$TTCN3_DIR/include -o MyModule2.o MyModule2.cc
g++ -c -DTITAN_ASN1ENC -DLINUX -I$TTCN3_DIR/include -o OtherUserCode.o
OtherUserCode.cc

g++ -o MyExecutable MyModule1.o MyModule2.o OtherUserCode.o -lasn1enc -lcrypto
-L$TTCN3_DIR/lib
```

Chapter 4. The API

This section describes the TITAN Standalone ASN.1 Encoder's API on C++ level.

4.1. Mapping ASN.1 Data Types to C++ Constructs

The mapping of ASN.1 data types to C++ takes place in two steps.

First, ASN.1 data types are mapped to TTCN-3. This is described in section 8 of the 'Using ASN.1 with TTCN-3' part of the TTCN-3 standard.

Afterward, TTCN-3 data types are mapped to C++ classes. This is described in sections 6.1 through 6.4 of the TITAN API guide.

4.2. Contents of the Generated Code

The generated C++ header and source files contain the following:

- a class for each structured type defined in the ASN.1 module, or a `typedef` of an existing class for each type alias or subtype defined in the ASN.1 module (subtype constraints are ignored in both cases);
- an encoder and a decoder helper function for each type defined in the ASN.1 module, which can be used to encode or decode values of that type;
- an `init_module` function, which initializes all data related to the ASN.1 module's generated code;
- a type descriptor object for each type defined in the ASN.1 module, which contains data needed for encoding;
- a constant for each value defined in the ASN.1 module;
- all of the above are placed into a namespace, whose name is generated from the ASN.1 module's name.

4.3. Usage Tips and Example

In order to use the types or values defined in an ASN.1 module from C++, the header file generated from that module must be included, and the `init_module` function generated for that module must be called. If an ASN.1 module imports from another module, then its generated header will automatically include the imported module's header, and its `init_module` function will automatically call the imported module's `init_module` function.

NOTE	Everything defined in a module's generated code is inside a namespace created for that module. A namespace prefix, or a `using` declaration, will be required when referencing these definitions.

4.3.1. Initializing and Using Values

All values declared in ASN.1 are included in the generated code and can be accessed from C++,

including the default values of SEQUENCE/SET fields.

New values of ASN.1 types can also be defined in C++. They can be initialized using the constructor of the corresponding class and various class methods.

Instances of classes corresponding to built-in ASN.1 types can be initialized with basic C++ values through their constructors. For example INTEGER instances can be initialized from an int, and string types can be initialized from a const char*.

Classes of structured types can be initialized one field or element at a time. SEQUENCE, SET and CHOICE classes have methods for accessing or setting each field, named after the field. SEQUENCE OF and SET OF classes have indexing operators for accessing or setting their elements. An optional field can be omitted in C++ by setting it to the special value OMIT_VALUE.

Classes of structured types also have a set_implicit_omit method, which sets all optional fields to the OMIT_VALUE and all default fields to their default values.

WARNING	Performing most operations on an unbound (i.e. uninitialized) object of a class mapped from ASN.1 will result in a runtime error. The object's is_bound method can be used to safely check whether it is initialized or not. An unbound optional field is not the same as an omitted optional field.

4.3.2. Encoding and Decoding

The Standalone ASN.1 Encoder supports the BER, PER, OER and JSON codecs for ASN.1 values. The XER codec is currently not supported.

The generated code provides a global encoder and decoder function for each type defined in the ASN.1 module. The names of these functions contain the C++ version of the type name, followed by _encoder and _decoder.

For example, the encoder and decoder functions for a type called MyType would have the following prototypes:

```
void MyType_encoder(const MyType& input_value, OCTETSTRING& output_stream,
TTCN_EncDec::coding_t coding_type, unsigned int extra_options);
INTEGER MyType_decoder(OCTETSTRING& input_stream, MyType& output_value,
TTCN_EncDec::coding_t coding_type, unsigned int extra_options);
```

The OCTETSTRING class (i.e. OCTET STRING in ASN.1) is used as the buffer that stores encoded ASN.1 values.

The coding_type parameter indicates which codec to use, and extra_options sets additional options for the specified codec. These are detailed in section 5 of the TITAN API guide.

The encoder function overwrites the initial value of output_stream with the encoded value. The value of input_value is left untouched.

The decoder function overwrites the initial value of output_value with the decoded value. The

encoding of `output_value` is extracted from the beginning of `input_stream`, if decoding was successful. The return value can be one of the following:

- `0`, if decoding was successful;
- `1`, if the buffer contains invalid data;
- `2`, if the buffer contains insufficient, but otherwise valid data.

Alternatively, the `encode` and `decode` member functions of each C++ class mapped from an ASN.1 type can be used directly. This requires using the `TTCN_Buffer` class (or its alias `ASN_Buffer`) and type descriptors.

The coders can be instructed about what to do when certain error situations occur. This is done through the `TTCN_EncDec` class (or its alias `ASN_EncDec`), which is described in section 5.1.1 of the TITAN API guide.

NOTE	By default, the generated decoder functions throw exceptions for most error situations (instead of returning `1` or `2`). If these return values are desired instead of an exception, then the error situations need to be set to warnings.

4.3.3. Logging and Error Handling

The Standalone ASN.1 Encoder does not have a built-in logging tool. Instead, the `log` member functions of C++ class mapped from ASN.1 types return a `CHARSTRING` value.

```
CHARSTRING MyType::log() const
{
  ...
}
```

The returned `CHARSTRING` can be converted on demand to `const char *` using its overloaded casting operator, like in the following example:

```
INTEGER a = INTEGER(5);
printf("Value of a: %s\n", (const char*) a.log());
```

Errors that occur while using the generated code or the base library throw an exception of type `TTCN_Error` (or its alias `ASN_Error`). This has a method called `get_message`, that returns the error message as a `char *`.

Warnings that occur while using the generated code or the base library are printed onto the standard error.

NOTE	Most error and warning messages will refer to TTCN-3 language elements instead of ASN.1 ones (e.g. a `SEQUENCE` will be referred to as a `record`, a `CHOICE` will be referred to as a `union`, etc.). The string returned by the `log` methods contains the TTCN-3 representation of the value.

4.3.4. Example

This section contains an example of using a type defined in ASN.1 for PER encoding and decoding in C++.

MyModule.asn:

```
MyModule
DEFINITIONS
AUTOMATIC TAGS ::= BEGIN
IMPORTS ;

MySequence ::= SEQUENCE {
  f1 [1] OCTET STRING,
  f2 [0] OCTET STRING,
  f3 [16] OCTET STRING OPTIONAL,
  ...,
  e1 OCTET STRING,
  e2 [2] OCTET STRING,
  [[
    g1 [3] OCTET STRING,
    g2 [4] OCTET STRING OPTIONAL
  ]],
  e3 [7] INTEGER (2),
  ...,
  f4 [15] OCTET STRING DEFAULT ''H
}

END
```

PerTest.cc:

```
#include "MyModule.hh"

int main() {
  try {
    // initializing the ASN.1 module
    MyModule::init_module();

    // initializing a value of type MySequence
    MyModule::MySequence myValue;
    myValue.f1() = OCTETSTRING("12");
    myValue.f2() = OCTETSTRING("34");
    myValue.f3() = OCTETSTRING("56");
    myValue.g1() = OCTETSTRING("AB");
    myValue.e3() = INTEGER(2);
    myValue.f4() = OCTETSTRING("78");
    myValue.set_implicit_omit();

    // PER encoding
    OCTETSTRING myBuffer;
    MyModule::MySequence_encoder(myValue, myBuffer,
      ASN_EncDec::CT_PER, PER_ALIGNED);
    printf("Encoding: %s\n", (const char*) myBuffer.log());

    // PER decoding
    MyModule::MySequence myDecodedValue;
    INTEGER myResult = MyModule::MySequence_decoder(myBuffer, myDecodedValue,
      ASN_EncDec::CT_PER, PER_ALIGNED);
    printf("Decoding: %s (returned: %s, remaining bytes: %s)\n",
      (const char*) myDecodedValue.log(),
      (const char*) myResult.log(),
      (const char*) myBuffer.log());

    return 0;
  }
  catch (ASN_Error e) {
    printf("%s\n", e.get_message());
    return 1;
  }
}
```

Result:

```
Encoding: 'E00112013401560178066003000 1AB0100'O
Decoding: { f1 := '12'O, f2 := '34'O, f3 := '56'O, e1 := <unbound>, e2 := <unbound>,
g1 := 'AB'O, g2 := omit, e3 := 2, f4 := '78'O } (returned: 0, remaining bytes: ''O)
```

Chapter 5. References

- [1] Programmers Technical Reference for TITAN TTCN-3 Test Executor
- [2] Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3. Part 7: Using ASN.1 with TTCN-3; European Telecommunications Standards Institute. ES 201 873-7 Version 4.10.1, April 2022
- [3] API Technical Reference for TITAN TTCN-3 Test Executor