



# **TSN.1 Compiler User's Manual**

(For Version 5.4)

## Table of Contents:

1	Overview .....	6
1.1	Limitations .....	7
1.2	System Requirements.....	7
1.2.1	Windows.....	7
1.2.2	Unix .....	7
1.3	Installation.....	8
1.3.1	Windows.....	8
1.3.2	Unix .....	8
2	Using the Compiler.....	9
2.1	Compile Options .....	9
2.1.1	Define Compile Time Constants .....	10
2.1.2	64-bit Integer Support.....	11
2.1.3	Imported TSN.1 File Paths .....	11
2.1.4	Automatics Storage vs. Dynamic Storage .....	11
2.1.5	Optimize Byte Aligned Messages .....	11
2.2	Examples.....	12
2.2.1	Configuration File.....	12
3	Generated Code .....	13
3.1	Generated C Code .....	13
3.1.1	Output Files .....	13
3.1.2	Integers .....	13
3.1.3	Strings.....	13
3.1.4	Enumerations.....	13
3.1.5	Messages.....	15
3.1.6	Namespaces .....	15
3.1.7	Message Fields .....	16
3.2	Generated C++ Code .....	17
3.2.1	Output Files .....	17
3.2.2	Integers .....	17
3.2.3	Strings.....	17
3.2.4	Enumerations.....	17
3.2.5	Messages.....	19
3.2.6	Namespaces .....	20
3.2.7	Field Declarations.....	20
3.3	Generated Java Code.....	20
3.3.1	Output Files .....	20
3.3.2	Integers .....	20
3.3.3	Strings.....	21
3.3.4	Enumerations.....	21
3.3.5	Messages.....	22
3.3.6	Namespaces .....	23
3.3.7	Field Declarations.....	23
3.4	Auxiliary Fields .....	24
3.5	Byte Order.....	24

3.6	Enumerated Bit Fields (C/C++) .....	25
3.7	Unpacking Signed Bit Fields .....	25
3.8	String Field.....	25
3.9	Unbounded Arrays (C/C++) .....	26
3.10	Optional Fields .....	26
3.11	Field Conflicts .....	26
3.12	Field References and Duplicate Fields.....	26
4	Using the Runtime Library .....	28
4.1	Using the C/C++ Runtime Library .....	28
4.1.1	Overview .....	28
4.1.2	C/C++ Compile Flags.....	29
4.1.3	Integration.....	29
4.1.4	Error Status Code.....	30
4.1.5	Message Settings .....	32
4.2	Using the Java Runtime Library .....	34
4.2.1	Runtime Exception .....	34
5	Using the C Runtime Library API.....	36
5.1	Initialize and Finalize.....	37
5.1.1	Using the Default Scheme .....	38
5.1.2	Using an Allocator.....	39
5.2	Configure .....	42
5.3	Reference & Unreference .....	43
5.4	Clone .....	43
5.5	Equal .....	44
5.6	SizeOf .....	44
5.7	SetLength .....	45
5.8	Pack & Unpack .....	45
5.9	Print & Scan.....	46
5.10	FPrint & FScan.....	47
5.11	Get Error Code .....	48
6	Using the C++ Runtime Library API .....	50
6.1	Initialize and Finalize.....	52
6.1.1	Using the Default Scheme .....	53
6.1.2	Using an Allocator.....	54
6.2	Configure .....	56
6.3	Reference & Unreference .....	56
6.4	Clone .....	57
6.5	Equal .....	58
6.6	SizeOf .....	58
6.7	SetLength .....	59
6.8	Pack & Unpack .....	59
6.9	Print & Scan.....	60
6.10	FPrint & FScan.....	61
6.11	Get Error Code .....	62
7	Using the Java Runtime Library API .....	64
7.1	Clone .....	64

7.2	Equals.....	64
7.3	SizeOf .....	65
7.4	SetLength .....	65
7.5	Pack.....	65
7.6	Unpack .....	65
7.7	Print.....	65
7.8	Scan.....	66
7.9	PrintXML.....	66
7.10	ScanXML .....	67
8	Using the Generated Wireshark Code .....	69
9	Using the Generated XML Code .....	73
9.1	Output Files.....	73
9.2	Integers.....	73
9.3	Strings .....	73
9.4	Enumerations .....	74
9.5	Messages .....	75
9.6	Field Declarations .....	76
10	Extensions.....	79
10.1	Base .....	79
10.1.1	Example .....	79
10.2	Bit Field.....	79
10.2.1	Example .....	80
10.3	Byte Alignment .....	80
10.3.1	Example .....	80
10.4	Code .....	80
10.4.1	Example .....	81
10.5	Display .....	81
10.5.1	Example .....	81
10.6	Dissect.....	82
10.6.1	Example .....	84
10.7	Doc .....	85
10.7.1	Example .....	86
10.8	Endianness.....	87
10.8.1	Example .....	87
10.9	Finalize.....	87
10.9.1	Example .....	88
10.10	Initialize.....	88
10.10.1	Example .....	89
10.11	Literal .....	89
10.11.1	Example .....	89
10.12	Options .....	89
10.12.1	Example .....	90
10.13	Pack.....	90
10.13.1	Example .....	91
10.14	Print .....	92
10.14.1	Example .....	93

10.15	Scan .....	93
10.15.1	Example .....	94
10.16	Storage.....	94
10.16.1	Example .....	94
10.17	Type.....	95
10.17.1	Example .....	97
10.18	Unpack .....	98
10.18.1	Example .....	99
References	.....	101

**List of Tables:**

Table 2-1	Compile Options.....	9
Table 3-1	TSN.1 and C/C++ Data Mapping.....	16
Table 3-2	TSN.1 and Java Data Mapping.....	23
Table 4-1	C/C++ Message Operations and Required Compile Flags.....	28
Table 4-2	The C/C++ Runtime Library Sources .....	28
Table 4-3	C/C++ Platform Adaptation Data Types .....	29
Table 4-4	C/C++ Platform Adaptation Functions.....	30
Table 4-5	C/C++ Message Operation Error Status Code .....	30
Table 4-7	Endianness Settings .....	32
Table 4-6	Print Settings .....	33
Table 4-8	Scan Settings .....	33
Table 4-9	Java Message Operation Runtime Exception .....	34
Table 9-1	Field Declaration to XML Element Mapping .....	76

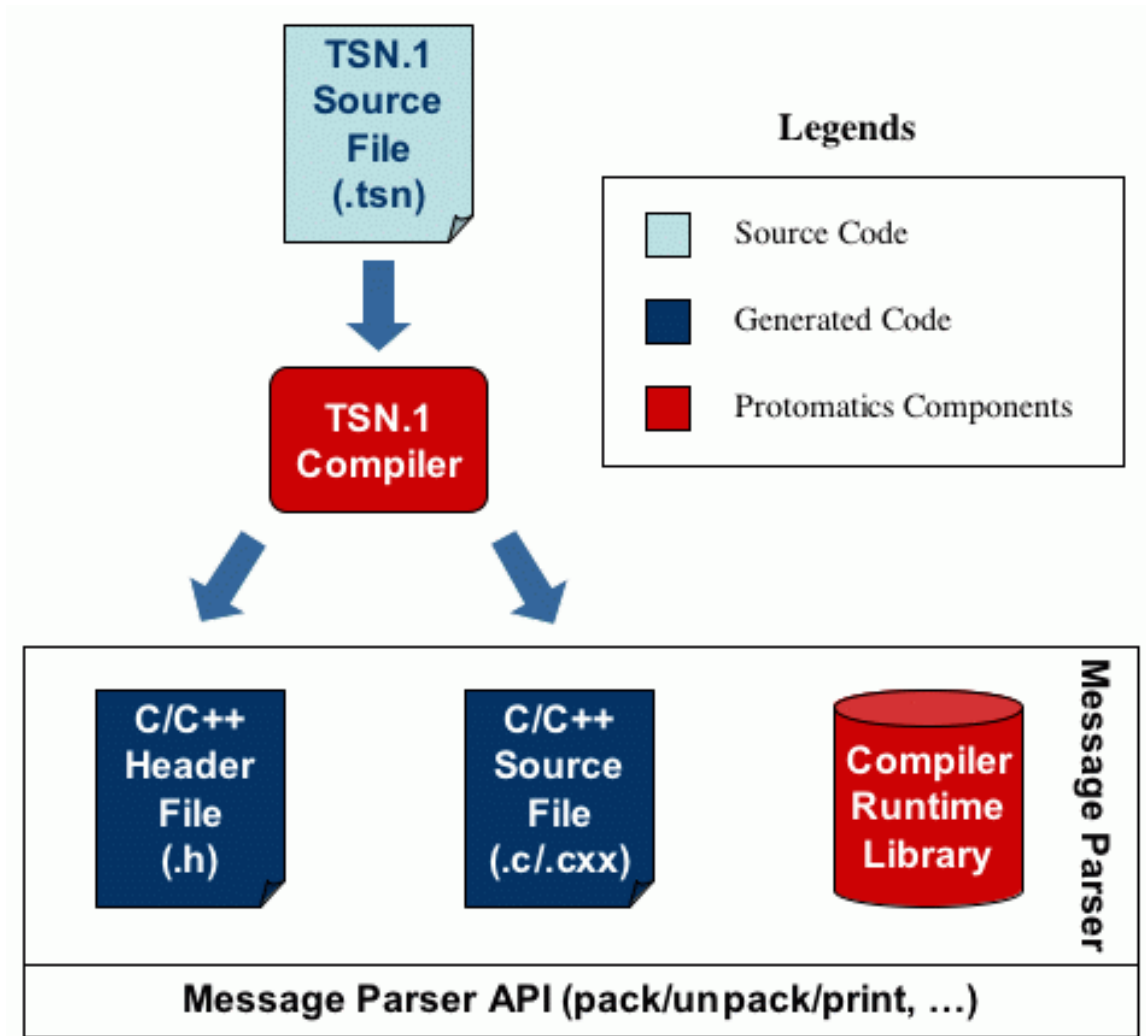
**List of Figures:**

Figure 1-1	The Transfer Syntax Notation One Compiler .....	6
------------	-------------------------------------------------	---

# 1 Overview

The TSN.1 Compiler takes TSN.1 specifications as input and produces C/C++/Java/Wireshark/XML code as output.

The Compiler is accompanied with the TSN.1 Compiler Runtime Library, which implements the common message operations such as pack, unpack, print, etc. The generated files, when compiled and linked with the Runtime Library, provide the complete solution for working with the TSN.1 messages.



**Figure 1-1 The Transfer Syntax Notation One Compiler**

The Compiler is capable of compiling all TSN.1 syntax as specified by the Transfer Syntax Notation One Specification [1]. The Compiler generates C/C++ code that is compliant to the ISO C99/ANSI C Standard [2].

## 1.1 Limitations

The following are some limitations imposed by the Compiler. Most of them are for practical reasons.

- The maximum size of an array is limited to 2147483647. Note that your C/C++/Java compiler may impose a limit smaller than this value.
- The maximum size of a bit field is 2147483647 bits or 256 Mbytes.
- The Compiler does not check for integer expression overflow at compile-time, nor does the generated C/C++/Java code check for it at run-time.
- If the shift count of a left or right shift operation evaluates to a constant at compile time and the count is less than zero or exceeds the width of the value being shifted, the Compiler issues a warning. However, this check is not performed in the generated C/C++/Java code at run-time.
- When passing an argument to a message by value, the Compiler and the generated C/C++/Java code do not check if the value exceeds the range permitted by the formal argument. Similarly this check is also not performed when assigning values to a variable in an assignment statement.
- The Compiler sometimes needs to determine the maximum size of an array or a bit field. It does that by performing interval arithmetic on the integer expression specifying the size. However, this is rather difficult for certain expressions such as those involving bitwise operators. In these instances, the Compiler issues a warning and user shall check the generated code to make sure the correct size is generated.
- When enum type is generated for a bit field and the bit field is used in an integer expression, there is no consistent way of predicting the integer type used by the C/C++ compiler to represent the enumerated values. The ISO C99/ANSI C [2] leaves this choice to the C/C++ compiler implementation. This can be problematic in integer expressions that use the enum typed bit field.
- When an enum typed bit field is used in a case of expression, the Compiler does not check if all enum values are listed in the case labels, although some C/C++ compilers check for it.
- The size of a field can be so large that it causes the “offset” to overflow in the pack, unpack, or sizeof function. The generated C/C++/Java does not check for this overflow condition.

## 1.2 System Requirements

### 1.2.1 Windows

- 32-bit Intel® Pentium® Processor 800 MHz or higher
- Microsoft® Windows 2000, Microsoft® Windows XP
- 256MB of RAM
- 60MB of available hard-disk space

### 1.2.2 Unix

Any Unix system supporting J2SE Runtime (1.5.0 or above) such as Linux, SUN Solaris, HP-UX, IBM AIX, etc

## **1.3 Installation**

Please refer to the online Installation Guides for instructions on how to install the Compiler onto your system.

### **1.3.1 Windows**

Installation Guide URL: [http://www.protomatics.com/tsnc\\_install\\_win32.html](http://www.protomatics.com/tsnc_install_win32.html).

### **1.3.2 Unix**

Installation Guide URL: [http://www.protomatics.com/tsnc\\_install\\_unix.html](http://www.protomatics.com/tsnc_install_unix.html).

## 2 Using the Compiler

The TSN.1 Compiler is a command-line application. A valid command is of the form:

```
tsnc [options] file.tsn ...
```

At least one TSN.1 file must be provided. The source file can be any valid pathname in the file system. If directory information is not provided, the current working directory is assumed. The source file must have the extension “.tsn”. The Compiler validates the syntax and semantics of the TSN.1 file before generating any code. If an error is detected, an error message is printed to the standard error.

### 2.1 Compile Options

Table 2-1 lists all the supported options and their intended usage:

**Table 2-1 Compile Options**

Option	Argument	Description
@<config_file>	None	Read compile options from the configuration file
-C	None	Check syntax and semantics only
-Did[=value]	None	Define compile-time constant
-M	None	Generate makefile dependencies to the standard output
-all	None	Short for -pack, -unpack, -sizeof, -equal, -clone, -print, -scan, -fprint, and fscan (C/C++)
-built-in	None	Add “packet-“ prefix to the generated filenames (Wireshark)
-byte-aligned	None	Message starts at a fixed byte boundary (C/C++)
-c	None	Generate C code
-c++	None	Generate C++ code
-cext	<extension>	C/C++ source file extension (default is .c/.cxx) (C/C++/Wireshark)
-check-range	None	Check field value range (C/C++/Wireshark)
-clean	None	Remove the generated Java files (Java)
-clone	None	Generate clone function (C/C++)
-copiable	None	Generate copy constructor and assignment operator (C++)
-d	<directory>	Specify the output directory for header and source files
-dc	<directory>	Specify the output directory for source files (C/C++/Wireshark)
-dh	<directory>	Specify the output directory for header files (C/C++/Wireshark)
-dissect-text	None	Dissect packets using text input (Wireshark)
-doxygen	None	Generate Doxygen style comments (C/C++)
-dynamic-endian	None	Allow changing endianness at runtime (C/C++)
-equal	None	Generate equal function (C/C++)

-fprint	None	Generate fprint function (C/C++)
-fscan	None	Generate fscan function (C/C++)
-h	None	Display help information
-hext	<extension>	Specify the C/C++ header file extension (default is .h) (C/C++/Wireshark)
-int64	None	Support 64-bit integers
-java	None	Generate Java code
-jdk1.5	None	Generate Java code that uses Generics
-license	<license_file>	Specify the license file to use
-little-endian	None	Pack/unpack messages in little endian format. The default is big endian.
-namespace	None	Generate namespaces
-omit-msg-name	None	Omit nested message names from being displayed (Wireshark)
-pack	None	Generate pack function (C/C++)
-packed-struct	None	Generate C bit field struct only (C)
-print	None	Generate print function (C/C++)
-scan	None	Generate scan function (C/C++)
-sizeof	None	Generate sizeof function (C/C++)
-sourcepath	<directories>	Specify the directories where to find imported files. The directories are separate by “;”.
-static	None	Use automatic storage for bit buffers and nested messages (C/C++)
-static-union	None	Use automatic storage for union members (C)
-unpack	None	Generate unpack function (C/C++)
-v	None	Display version information
-verbose	None	Print out the functions supported by the generated code
-w	None	Suppress all warning messages
-wireshark	None	Generate Wireshark dissector code (Professional Edition)
-xml	None	Generate XML code (Professional Edition)
-xsl	<xsl_file>	Specify the XSL stylesheet to use for the XML output (XML)

Some compile options can be specified inside a TSN.1 file. Please read section 10.12 for details.

If no target language is specified, the Compiler generates C code by default.

### 2.1.1 Define Compile Time Constants

Use the “-D” option to pass compile time constants to the Compiler. This is typically used to conditionally compile the TSN.1 file. If a value is not specified, it defaults to 0, i.e. “-Did” is equivalent to “-Did=0”. If the constant is also defined in the TSN.1 file, the value supplied at the command line overrides the definition in the TSN.1 file.

### 2.1.2 64-bit Integer Support

By default, only integers up to 32-bit can be used in the TSN.1 file and integer types up to 32-bit are generated. To extend this to 64-bit, use the “-int64” option.

### 2.1.3 Imported TSN.1 File Paths

Use the “-sourcepath” option to specify the list of directories in which the Compiler looks for the imported TSN.1 files. When multiple “-sourcepath” options are specified, the directories are appended to the search list. The directories are searched in the order they are specified. The current working directory is always searched as the last resort.

### 2.1.4 Automatics Storage vs. Dynamic Storage

By default, the Compiler generates pointer types for bit buffers and nested messages. To generate automatic storage types instead, use the “-static” option. Note that this option has no effect on union members. They are always generated as pointers. However, if your target language is C you can generate automatic union members using the “-static-union” flag. Bit buffers with a maximum size greater than 4K bytes will always be generated as pointers.

### 2.1.5 Optimize Byte Aligned Messages

By default, the generated code does that require messages to be byte-aligned. It can pack and unpack messages starting from an arbitrary bit offset. If your message always starts at byte boundary, turn on the “-byte-aligned” flag. This will generate more efficient code using the C bit-wise operators. When this flag is used, the Compiler assumes that each message starts at byte boundary and calculates the byte alignment of every field in that message. If a message is nested inside another message and the byte alignment for the nested message is not at a byte boundary, the Compiler automatically adjusts the nested message’s byte alignment to this new alignment. For example,

```
N() ::=
{
  ...
}

M() ::=
{
  A  3;
  n  : N;
}
```

Since the nested message “n” starts at the byte boundary plus 3 bits, the Compiler automatically adjust the byte alignment of message “N” to 3. If “N” is nested inside another message and the alignment for that occurrence is not 3, the Compiler sets the byte alignment of “N” to -1 and the generated code for “N” is as if “-byte-aligned” is not specified. However, if “N” and “M” are not defined in the same TSN.1 file, then the Compiler will issue an error message complaining that “n” is miss aligned because the Compiler cannot change the alignment of a message defined in another file. To resolve this error, use the byte alignment extension (See 10.3) to manually specify the byte

alignment for “N”. It should be clear from this discussion that the “-byte-aligned” flag does not require every single message to be byte aligned and neither does it require every field of a message to be byte aligned. The Compiler optimizes as much as possible.

## 2.2 Examples

To check the version of the Compiler installed on your system:

```
tsnc -v
```

To check syntax and semantics only on “is856\_ami.tsn”:

```
tsnc -C is856_ami.tsn
```

To generate C++ code that supports all runtime functions and uses the “.cpp” for the generated C++ source file extension:

```
tsnc -c++ -cext cpp -all is856_ami.tsn
```

To generate C code with all compile warnings turned off and search imported files in directories /projects/is856/tsn and /projects/common/tsn and place the output files under directory /projects/is856/src:

```
tsnc -w -sourcepath "/projects/is856/tsn;/projects/common/tsn" -d  
/projects/is856/src is856_ami.tsn
```

To generate pack, unpack, and sizeof functions only:

```
tsnc -pack -unpack -sizeof is856_ami.tsn
```

### 2.2.1 Configuration File

Compile options, except those that specify a target language, can be stored in a configuration file, which can be referred from the command line. For example, create a file called “project\_tsnc.cfg” and put the following line in the file:

```
-w -sourcepath "/projects/is856/tsn;/projects/common/tsn" -d  
/projects/is856/src is856_ami.tsn
```

You can then invoke the Compiler as illustrated below assuming “project\_tsnc.cfg” is in your current working directory:

```
tsnc -c++ @project_tsnc.cfg
```

## 3 Generated Code

The TSN.1 Compiler translates TSN.1 definitions into their corresponding data and type definitions in the target language. Your application uses the generated types to instantiate messages and access message fields. The Compiler also generates code that are used by the TSN.1 Compiler Runtime Library to perform operations on the messages.

### 3.1 Generated C Code

#### 3.1.1 Output Files

If the target language is C, the Compiler generates a “foo.h” and a “foo.c” for a TSN.1 file named “foo.tsn”. The generated header file contains the constant, enumeration, and message structure definitions for the corresponding definitions in the TSN.1 file while the generated source file contains the functions and descriptors for the messages.

#### 3.1.2 Integers

Integer constants are generated as #define's. For example, the constant definition

```
N_ADMPDefault ::= 0;
```

is translated into

```
#define N_ADMPDefault 0
```

#### 3.1.3 Strings

String constants are generated as #define's. For example, the constant definition

```
GREETING ::= "Hello, World!";
```

is translated into

```
#define GREETING "Hello, World!"
```

#### 3.1.4 Enumerations

Enumerations are translated into typedef'ed enums. For example, the enumeration definition

```
is856_admp_AirMessageId ::= enumerated
{
    IS856_ADMP_UATI_REQUEST,
    IS856_ADMP_UATI_ASSIGNMENT,
    IS856_ADMP_UATI_COMPLETE,
    IS856_ADMP_HARDWARE_ID_REQUEST,
    IS856_ADMP_HARDWARE_ID_RESPONSE,

    IS856_ADMP_CONFIGURATION_REQUEST (0x50),
    IS856_ADMP_CONFIGURATION_RESPONSE
}
```

is translated into

```
typedef enum
{
    IS856_ADMP_UATI_REQUEST = 0,
    IS856_ADMP_UATI_ASSIGNMENT = 1,
    IS856_ADMP_UATI_COMPLETE = 2,
    IS856_ADMP_HARDWARE_ID_REQUEST = 3,
    IS856_ADMP_HARDWARE_ID_RESPONSE = 4,

    IS856_ADMP_CONFIGURATION_REQUEST = 80,
    IS856_ADMP_CONFIGURATION_RESPONSE = 81
} is856_admp_AirMessageId;
```

If any of the “-print”, “-scan”, “-fprint”, or “-fscan” option is specified, a function that maps an enum value to a string is also generated:

```
extern const char* is856_admp_AirMessageId_v21
(
    is856_admp_AirMessageId value
);
```

### 3.1.4.1 Nested Enumerations

The type name and the enum literals for a nested enumeration is prefixed with the name of the parent message(s). For example,

```
is856_admp() ::=
{
    MessageId 8 enumerated AirMessageId
    {
        UATI_REQUEST,
        UATI_ASSIGNMENT,
        UATI_COMPLETE,
        HARDWARE_ID_REQUEST,
        HARDWARE_ID_RESPONSE,

        CONFIGURATION_REQUEST (0x50),
        CONFIGURATION_RESPONSE
    };
}
```

is translated into

```
typedef enum
{
    is856_admp_UATI_REQUEST = 0,
    is856_admp_UATI_ASSIGNMENT = 1,
    is856_admp_UATI_COMPLETE = 2,
    is856_admp_HARDWARE_ID_REQUEST = 3,
    is856_admp_HARDWARE_ID_RESPONSE = 4,

    is856_admp_CONFIGURATION_REQUEST = 80,
    is856_admp_CONFIGURATION_RESPONSE = 81
} is856_admp_AirMessageId;
```

### 3.1.5 Messages

Messages are translated into typedef'ed structs. For example, the message definition

```
is856_admp_UATIDelete() ::=
{
    ...
}
```

is translated into

```
typedef struct
{
    ...
} is856_admp_UATIDelete;
```

#### 3.1.5.1 Nested Messages

The type name for a nested message is prefixed with the name of the parent message(s). For example,

```
is2000_rdsch_PSMM() ::=
{
    REF_PN          9;
    PILOT_STRENGTH  6;
    KEEP            1;

    PILOTS [] : Pilot
    {
        PILOT_PN_PHASE  15;
        PILOT_STRENGTH  6;
        KEEP            1;
    }
}
```

The type name for the nested message “Pilot” is “is2000\_rdsch\_PSMM\_Pilot”.

### 3.1.6 Namespaces

If the “-namespace” option is specified, the type names for constants, enumerations, and messages will be prefixed with the name of the package they belong. For example, if the message “PSMM” is defined as part of the “is2000.rdsch” package, then the type name for the message is “is2000\_rdsch\_PSMM”.

### 3.1.7 Message Fields

Field declarations in a TSN.1 message body are translated into field members of the message struct according to the mapping in Table 3-1.

**Table 3-1 TSN.1 and C/C++ Data Mapping**

Field Declaration	Storage	C/C++ Data Members	Description
<code>foo 8;</code>	N/A	<code>tsnc_uint8 foo;</code>	Bit field with a constant size <= 8
<code>foo 8 signed;</code>	N/A	<code>tsnc_int8 foo;</code>	Signed bit field with a constant size <= 8
<code>foo 11;</code>	N/A	<code>tsnc_uint16 foo;</code>	Bit field with a constant size > 8 and <= 16
<code>foo 11 signed;</code>	N/A	<code>tsnc_int16 foo;</code>	Signed bit field with a constant size > 8 and <= 16
<code>foo 30;</code>	N/A	<code>tsnc_uint32 foo;</code>	Bit field with a constant size > 16 and <= 32
<code>foo 30 signed;</code>	N/A	<code>tsnc_int32 foo;</code>	Signed bit field with a constant size > 16 and <= 32
<code>foo 64;</code>	N/A	<code>tsnc_uint64 foo;</code>	Bit field with a constant size > 32 and <= 64 (if <code>-int64</code> option is on)
<code>foo 64 signed;</code>	N/A	<code>tsnc_int64 foo;</code>	Signed bit field with a constant size > 32 and <= 64 (if <code>-int64</code> option is on)
<code>foo 128;</code>	N/A	<code>tsnc_uint8 foo[16];</code>	Bit field with a constant size > 32 (or 64 if <code>-int64</code> option is on)
<code>length 2;</code> <code>foo length * 8;</code>	N/A	<code>tsnc_uint8 length;</code> <code>tsnc_uint32 foo;</code>	Bit field with a maximum size <= 32
<code>length 3;</code> <code>foo length * 8;</code>	N/A	<code>tsnc_uint8 length;</code> <code>tsnc_uint64 foo;</code>	Bit field with a maximum size > 32 and <= 64 (if <code>-int64</code> option is on)
<code>length 4;</code> <code>foo length * 8;</code>	Dynamic	<code>tsnc_uint8 length;</code> <code>tsnc_uint8 *foo;</code>	Bit field with a maximum size > 32 (or 64 if <code>-int64</code> option is on)
<code>length 4;</code> <code>foo length * 8;</code>	Static	<code>tsnc_uint8 length;</code> <code>tsnc_uint8 foo[15];</code>	Bit field with a maximum size > 32 (or 64 if <code>-int64</code> option is on)
<code>foo 8 string;</code>	Dynamic	<code>tsnc_uint8 *foo;</code>	String field with a character size <= 8 and the default maximum size 64
<code>foo 8 string;</code>	Static	<code>tsnc_uint8 foo[65];</code>	String field with a character size <= 8 and the default maximum size 64
<code>foo 16 string;</code>	Dynamic	<code>tsnc_uint16 *foo;</code>	String field with a character size <= 16 and the default maximum size 64
<code>foo 16 string;</code>	Static	<code>tsnc_uint16 foo[65];</code>	String field with a character size <= 16 and the default maximum size 64
<code>foo 32 string;</code>	Dynamic	<code>tsnc_uint32 *foo;</code>	String field with a character size <= 32 and the default maximum size 64
<code>foo 32 string;</code>	Static	<code>tsnc_uint32 foo[65];</code>	String field with a character size <= 32 and the default maximum size 64
<code>foo 8 string[32];</code>	Static	<code>tsnc_uint8 foo[33];</code>	String field with a user specified maximum size
<code>msg : Type;</code>	Dynamic	<code>Type *msg;</code>	Nested message
<code>msg : Type;</code>	Static	<code>Type msg;</code>	Nested message
<code>msgs : case Id of</code> <code>{</code> <code>  0 =&gt;</code> <code>  foo 8;</code> <code>  1 =&gt;</code> <code>  msg1 : Type1;</code> <code>  2 =&gt;</code> <code>  msg2 : Type2;</code> <code>  _ =&gt;</code> <code>  msg3 : Type3;</code> <code>};</code>	N/A	<code>union</code> <code>{</code> <code>  tsnc_uint8 foo;</code> <code>  Type1 *msg1;</code> <code>  Type2 *msg2;</code> <code>  Type3 *msg3;</code> <code>} msgs;</code>	Case of field

<code>foo[6] 8;</code>	N/A	<code>tsnc_uint8 foo[6];</code>	Fixed size array
<code>size 8;</code> <code>foo[size] 15;</code>	N/A	<code>tsnc_uint8 size;</code> <code>tsnc_uint16 foo[255];</code>	Variable size array
<code>msgs[] : Type;</code>	Dynamic	<code>tsnc_uint16 _msgs_size_;</code> Type <code>*msgs[16];</code>	Unbounded array with the default maximum size 16. The auxiliary field “_msg_size_” indicates the actual size of the array
<code>msgs[] : Type;</code>	Static	<code>tsnc_uint16 _msgs_size_;</code> Type <code>msgs[16];</code>	Unbounded array with the default maximum size 16
<code>msgs[(0..32)] : Type;</code>	Dynamic	<code>tsnc_uint16 _msgs_size_;</code> Type <code>*msgs[32];</code>	Unbounded array with a user specified maximum size
<code>msgs[(0..32)] : Type;</code>	Static	<code>tsnc_uint16 _msgs_size_;</code> Type <code>msgs[32];</code>	Unbounded array with a user specified maximum size
<code>reserve 7;</code>	N/A	None	Reserved field
<code>align(8);</code>	N/A	None	Alignment
<code>optional tag :</code> { ... }	N/A	<code>tsnc_uint8 _tag_optional_;</code> ...	Optional fields. The auxiliary field “_tag_optional_” indicates if the optional fields are present.

## 3.2 Generated C++ Code

### 3.2.1 Output Files

If the target language is C++, the Compiler generates a “foo.h” and a “foo.cxx” for a TSN.1 file named “foo.tsn”. The header file contains the constant, enumeration, and class definitions for the corresponding definitions in the TSN.1 file while the generated source file contains the functions and descriptors for the messages.

### 3.2.2 Integers

Integer constants are generated as “const” integer constants. For example, the constant definition

```
N_ADMPDefault ::= 0;
```

is translated into

```
const tsnc_int32 N_ADMPDefault = 0;
```

### 3.2.3 Strings

String constants are generated as “const char\*”. For example, the constant definition

```
GREETING ::= "Hello, World!";
```

is translated into

```
const char *GREETING = "Hello, World!";
```

### 3.2.4 Enumerations

Enumerations are translated into typedef’ed enums. For example, the enumeration definition

```
is856_admp_AirMessageId ::= enumerated
{
    IS856_ADMP_UATI_REQUEST,
    IS856_ADMP_UATI_ASSIGNMENT,
    IS856_ADMP_UATI_COMPLETE,
    IS856_ADMP_HARDWARE_ID_REQUEST,
    IS856_ADMP_HARDWARE_ID_RESPONSE,

    IS856_ADMP_CONFIGURATION_REQUEST (0x50),
    IS856_ADMP_CONFIGURATION_RESPONSE
}
```

is translated into

```
typedef enum
{
    IS856_ADMP_UATI_REQUEST = 0,
    IS856_ADMP_UATI_ASSIGNMENT = 1,
    IS856_ADMP_UATI_COMPLETE = 2,
    IS856_ADMP_HARDWARE_ID_REQUEST = 3,
    IS856_ADMP_HARDWARE_ID_RESPONSE = 4,

    IS856_ADMP_CONFIGURATION_REQUEST = 80,
    IS856_ADMP_CONFIGURATION_RESPONSE = 81
} is856_admp_AirMessageId;
```

If any of the “-print”, “-scan”, “-fprint”, or “-fscan” option is specified, a function that maps an enum value to a string is also generated:

```
extern const char* is856_admp_AirMessageId_v21
(
    is856_admp_AirMessageId value
);
```

### 3.2.4.1 Nested Enumerations

A nested enumeration is generated as a nested enum of the parent message class. For example, the nested enumeration

```
is856_admp() ::=
{
    MessageId 8 enumerated AirMessageId
    {
        UATI_REQUEST,
        UATI_ASSIGNMENT,
        UATI_COMPLETE,
        HARDWARE_ID_REQUEST,
        HARDWARE_ID_RESPONSE,

        CONFIGURATION_REQUEST (0x50),
        CONFIGURATION_RESPONSE
    };
}
```

is translated into

```
class is856_admp : public tsncxx_Message
{
public:
    typedef enum
    {
        UATI_REQUEST = 0,
        UATI_ASSIGNMENT = 1,
        UATI_COMPLETE = 2,
        HARDWARE_ID_REQUEST = 3,
        HARDWARE_ID_RESPONSE = 4,

        CONFIGURATION_REQUEST = 80,
        CONFIGURATION_RESPONSE = 81
    } AirMessageId;

    ...
};
```

If the name of the enumeration is omitted, then the field name is used. However, in order to avoid name conflict with the field name, the name is prefixed and postfixed with an underscore, “is856\_admp::\_MessageId\_.”

### 3.2.5 Messages

Messages are translated into classes. For example, the message definition

```
is856_admp_UATIDelete() ::=
{
    ...
}
```

is translated into

```
class is856_admp_UATIDelete : public tsncxx_Message
{
    ...
};
```

where “tsncxx\_Message” is the base class for all messages.

#### 3.2.5.1 Nested Messages

Nested messages are generated as nested classes. For example,

```

is2000_rdsch_PSM() ::=
{
    REF_PN          9;
    PILOT_STRENGTH  6;
    KEEP            1;

    PILOTS [] : Pilot
    {
        PILOT_PN_PHASE  15;
        PILOT_STRENGTH  6;
        KEEP            1;
    }
}

```

The class for nested message “Pilot” is generated as “is2000\_rdsch\_PSM::Pilot.” If the name of the message is omitted, then the field name is used. However, in order to avoid name conflict with the field name, the name is prefixed and postfixed with an underscore, “is2000\_rdsch\_PSM::\_PILOTS\_.”

### 3.2.6 Namespaces

If the “-namespace” option is specified, the type names for constants, enumerations, and messages are generated in the namespace according to the package they belong. For example, if the message “PSMM” is defined as part of the “is2000.rdsch” package, then the type name for the message is “is2000::rdsch::PSMM”.

### 3.2.7 Field Declarations

Field declarations in a TSN.1 message body are translated into data members of the message class according to the mapping in Table 3-1.

## 3.3 Generated Java Code

### 3.3.1 Output Files

If the target language is Java, the Compiler generates a “fooConstants.java” for a TSN.1 file named “foo.tsn” and a separate java file for each top-level enumeration and message in “foo.tsn”. The “fooConstants.java” defines the public “fooConstants” class, which contains the global constants in the TSN.1 file. The other .java files defines the classes for the enumerations and messages.

### 3.3.2 Integers

Integer constants are generated as public static final integer constants. For example, the constant definition

```
N_ADMPDefault ::= 0;
```

is translated into

```
public static final int N_ADMPDefault = 0;
```

### 3.3.3 Strings

String constants are generated as public static final String constants. For example, the constant definition

```
GREETING ::= "Hello, World!";
```

is translated into

```
public static final String GREETING = "Hello, World!";
```

### 3.3.4 Enumerations

Enumerations are translated into public static final integer constants. For example, the enumeration definition

```
is856_admp_AirMessageId ::= enumerated  
{  
    UATI_REQUEST,  
    UATI_ASSIGNMENT,  
    UATI_COMPLETE,  
    HARDWARE_ID_REQUEST,  
    HARDWARE_ID_RESPONSE,  
  
    CONFIGURATION_REQUEST (0x50),  
    CONFIGURATION_RESPONSE  
}
```

is translated into

```
public class is856_admp_AirMessageId  
{  
    public static final byte UATI_REQUEST = 0;  
    public static final byte UATI_ASSIGNMENT = 1;  
    public static final byte UATI_COMPLETE = 2;  
    public static final byte HARDWARE_ID_REQUEST = 3;  
    public static final byte HARDWARE_ID_RESPONSE = 4;  
    public static final byte CONFIGURATION_REQUEST = 80;  
    public static final byte CONFIGURATION_RESPONSE = 81;  
}
```

#### 3.3.4.1 Nested Enumerations

A nested enumeration is generated as a nested class of the parent message class. For example, the nested enumeration

```
is856_admp() ::=
{
  MessageId 8 enumerated AirMessageId
  {
    UATI_REQUEST,
    UATI_ASSIGNMENT,
    UATI_COMPLETE,
    HARDWARE_ID_REQUEST,
    HARDWARE_ID_RESPONSE,

    CONFIGURATION_REQUEST (0x50),
    CONFIGURATION_RESPONSE
  };
}
```

is translated into

```
public class is856_admp extends TSNMessage
{
  public static class
  {
    public static final byte UATI_REQUEST = 0;
    public static final byte UATI_ASSIGNMENT = 1;
    public static final byte UATI_COMPLETE = 2;
    public static final byte HARDWARE_ID_REQUEST = 3;
    public static final byte HARDWARE_ID_RESPONSE = 4;
    public static final byte CONFIGURATION_REQUEST = 80;
    public static final byte CONFIGURATION_RESPONSE = 81;
  }
};
```

### 3.3.5 Messages

Messages are translated into public classes. For example, the message definition

```
is856_admp_UATISComplete() ::=
{
  ...
}
```

is translated into

```
public class is856_admp_UATISComplete extends TSNMessage
{
  ...
};
```

All message classes are derived from “TSNMessage”.

#### 3.3.5.1 Nested Messages

Nested messages are generated as nested classes. For example,

```

is2000_rdsch_PSMM() ::=
{
    REF_PN          9;
    PILOT_STRENGTH  6;
    KEEP            1;

    PILOTS [] :
    {
        PILOT_PN_PHASE  15;
        PILOT_STRENGTH  6;
        KEEP            1;
    }
}

```

is translated into

```

public class is2000_rdsch_PSMM extends TSNMessage
{
    ...

    public static class _PILOTS_
    {
        ...
    }
}

```

The class name is prefixed and postfixed with an underscore to avoid name conflict with the field name.

### 3.3.6 Namespaces

The TSN.1 package specification is mapped directly into the Java package specification.

### 3.3.7 Field Declarations

Field declarations in a TSN.1 message body are translated into data members of the message class according to the mapping in Table 3-2.

**Table 3-2 TSN.1 and Java Data Mapping**

Field Declaration	Java Data Members	Description
foo 7;	byte foo;	Bit field with a constant size < 8
foo 8 signed;	byte foo;	Signed bit field with a constant size <= 8
foo 11;	short foo;	Bit field with a constant size >= 8 and < 16
foo 16 signed;	short foo;	Signed bit field with a constant size > 8 and <= 16
foo 30;	int foo;	Bit field with a constant size >= 16 and < 32
foo 30 signed;	int foo;	Signed bit field with a constant size > 16 and <= 32
foo 64;	long foo;	Bit field with a constant size > 32 and <= 64
foo 64 signed;	long foo;	Signed bit field with a constant size > 32 and <= 64
foo 128;	byte foo[] = new byte[16];	Bit field with a constant size > 64
length 2;	byte length;	Bit field with a maximum size < 32
foo length * 8;	int foo;	

<code>length 3; foo length * 8;</code>	<code>byte length; long foo;</code>	Bit field with a maximum size $\geq 32$ and $\leq 64$
<code>length 4; foo length * 8;</code>	<code>byte length; byte foo[];</code>	Bit field with a maximum size $> 32$ (or 64 if <code>-int64</code> option is on)
<code>foo 8 string;</code>	<code>int foo[] = new int[65];</code>	String field with the default maximum size 64
<code>msg : Type;</code>	<code>Type msg;</code>	Nested message
<code>msgs : case Id of {   0 =&gt;     foo 8;   1 =&gt;     msg1 : Type1;   2 =&gt;     msg2 : Type2;   - =&gt;     msg3 : Type3; };</code>	<code>public class _msgs_ {     short foo;     Type1 msg1;     Type2 msg2;     Type3 msg3; } msgs;</code>	Case of field
<code>foo[6] 8;</code>	<code>short foo = new short[6];</code>	Fixed size array
<code>size 8; foo[size] 15;</code>	<code>short size; short foo = new short[255];</code>	Variable size array
<code>msgs[] : Type;</code>	<code>Vector msgs;</code>	Unbounded array
<code>msgs[] : Type;</code>	<code>Vector&lt;Type&gt; msgs;</code>	Unbounded array (if <code>-jdk1.5</code> option is on)
<code>reserve 7;</code>	<code>None</code>	Reserved field
<code>align(8);</code>	<code>None</code>	Alignment
<code>optional tag : {   ... }</code>	<code>short _tag_optional; ...</code>	Optional fields. The auxiliary field “_tag_optional_” indicates if the optional fields are present.

By default, integer bit fields are represented using the primitive types. However, if the object version of the primitive type is desired, for example, “Short” for “short”, use the type extension “<type language=“java” object=“true”/>” to instruct the Compiler to generate the object type instead.

## 3.4 Auxiliary Fields

In the case of unbounded arrays and optional fields, extra data members may be generated. These members start and end with a ‘\_’ character to avoid possible conflict with other members. Please see section 3.9 and 3.10 for details.

## 3.5 Byte Order

For bit fields with a maximum size less than 32 bits (64 if `-int64` option is used), the values are represented using the platform native byte order (little endian or big endian). For bit field with a constant size that is greater than 32 (64 if `-int64` option is used), the values are represented such that the most significant byte is at lowest address (big endian). For example, the following bit field declaration,

```
Value 33;
```

is translated by the Compiler to:

```
tsnc_uint8 Value[5];
```

The bit sequence 100100011010001010110011110001001, where the left most bit is the most significant bit, is represented as:

Value[0]	Value[1]	Value[2]	Value[3]	Value[4]
00000001	00100011	01000101	01100111	10001001
0x1	0x23	0x45	0x67	0x89

However, if the bit field is of variable size and the maximum size is greater than 32 (64 if `-int64` options is used), then the value is represented such that the most significant bit of the field is the most significant bit of the first byte in the buffer without any leading padding bits. For example,

```
Length 8;
Value 3 * Length;
```

is translated by the Compiler to:

```
tsnc_uint8 Length;
tsnc_uint8 *Value;
```

Assuming `Length` is 11, the same bit sequence 100100011010001010110011110001001 is represented as:

Value[0]	Value[1]	Value[2]	Value[3]	Value[4]
10010001	10100010	10110011	11000100	10000000
0x91	0xa2	0xb3	0xc4	0x80

The examples are in C/C++, but the same rule also applies to Java.

### 3.6 Enumerated Bit Fields (C/C++)

For an enumerated bit field, the field is generated using the enum type.

### 3.7 Unpacking Signed Bit Fields

When unpacking a signed bit field with a size that is less than the underlying integer type, the value is automatically sign extended.

### 3.8 String Field

Although a string field is not terminated by the null character in the encoded form when the length is equal to the maximum length, it should always be null terminated in the generated integer array. The maximum size of the array should be one bigger than the maximum length.

### 3.9 Unbounded Arrays (C/C++)

Unbounded arrays in TSN.1 are denoted using a pair of empty brackets “[ ]”. They can only appear at the end of a message. Generally speaking, when unpacking such a message, the size of the array cannot be determined until the data buffer is exhausted. This is why unbounded arrays are not generated as pointers. Instead the Compiler generates a fixed size array with a maximum size of 16. You can override this default maximum value by specifying an interval within the brackets, for example, “[ (0 .. 32) ]”. The Compiler also generates an additional field “\_<array\_name>\_size\_” to indicate the actual size of the array. This field is populated by the user before packing a message, and set by the Unpack function during unpacking.

If you want to use your own data structures for arrays, you can specify that using the Type extension. Please see section 10.17.

### 3.10 Optional Fields

For optional fields, the Compiler generates an extra field “\_<tag\_name>\_optional\_”, which indicates if the optional fields are present. This field is populated by the user before packing a message, and set by the Unpack function during unpacking.

### 3.11 Field Conflicts

Fields with the same name may appear in different branches of a conditional without causing any conflict. In this case, the Compiler tries to generate the *maximum* compatible type for the field. For example, if a field is mapped to “tsnc\_uint8” in one branch, and “tsnc\_uint32” in another, then “tsnc\_uint32” is generated for the field. If no compatible type is possible, the Compiler issues an error message.

### 3.12 Field References and Duplicate Fields

Semantic check for field references and duplicate fields can be problematic in TSN.1 when fields are defined inside conditionals. The following example illustrates the challenge.

```

FieldReference() ::=
{
    A 8;

    if(A == 3)
    {
        B 8;
    }

    if(A > 2 && A < 4)
    {
        C B * 8;
    }
}

```

In this example, field “B” is not always present in the message. The Compiler, therefore, needs to check that any subsequent reference to “B” in an expression only happens when “B” is present. The reference to “B” in the expression “B \* 8” is obviously valid in this example since the guard condition “A > 2 && A < 4” implies “A == 3”. However, this is not always computable at compile-time because the guard condition can be arbitrarily complex. When such computation is not possible, the Compiler generates an error, which can be manually overridden by appending a ‘?’ character after the field name. For example “C B? \* 8”. When the error is overridden, extra code is generated to perform a run-time check. The check for duplicate fields is similarly handled.

## 4 Using the Runtime Library

This section gives an overview of the Runtime Library and describes how to integrate the library with your application. The Runtime Library provides an API that allows your application to perform various operations on the generated messages. Please refer to the subsequent Chapters for details about this API.

### 4.1 Using the C/C++ Runtime Library

Some operations require support from the generated code, which depends on the TSN.1 Compiler flag. The table below lists the operations available in the Runtime Library and their compile flag requirements,

**Table 4-1 C/C++ Message Operations and Required Compile Flags**

Operation	Description	Compile Flag Requirements
Initialize	Initialize a message	None
Finalize	Finalize a message	None
Reference	Increment the reference count	None
Unreference	Decrement the reference count	None
Clone	Make a deep copy	-pack, -unpack, and -sizeof
Equal	Compare for equality	-pack and -sizeof
Pack	Pack into a buffer	-pack
Unpack	Unpack from a buffer	-unpack
SizeOf	Compute the size in bits	-sizeof
Print	Print to a string	-print
Scan	Scan from a string	-scan
Fprint	Print to a file	-fprint
FScan	Scan from a file	-fscan

#### 4.1.1 Overview

**Table 4-2 The C/C++ Runtime Library Sources**

File	Description
tsnc.h	Common definitions
tsnc_msg.h	Message operation C interface
tsncxx_msg.h	Message operation C++ interface
tsnc_custom.h	Runtime library customization (Section 4.1.2)
tsnc_buf.h	Bit buffer utility functions interface
tsnc_rte.h	Runtime environment interface
tsnc_buf.c	Bit buffer utility functions implementation
tsnc_buf_int64.c	Bit buffer utility functions for 64-bit integers
tsnc_msg.c	Basic message operation implementation
tsnc_msg_clone.c	The Clone operation implementation
tsnc_msg_equal.c	The Equal operation implementation
tsnc_msg_print.c	The Print operation implementation

<code>tsnc_msg_scan.c</code>	The Scan operation implementation
<code>tsnc_msg_fprint.c</code>	The FPrint operation implementation
<code>tsnc_msg_fscan.c</code>	The FScan operation implementation

Note that your application should only include “`tsnc.h`”, “`tsnc_msg.h`” (“`tsncxx_msg.h`” for C++). All other header files are intended for internal use by the Runtime Library and the generated code.

If your application does not require some of the message operations, you can omit the corresponding “.c” files from your build to minimize the Runtime Library size. For example, if you never use the FPrint and FScan functions, do not include “`tsnc_msg_fprint.c`” and “`tsnc_msg_fscan.c`” in your build.

## 4.1.2 C/C++ Compile Flags

### 4.1.2.1 Flags for the Runtime Library

There is no compile flag for the Runtime Library.

### 4.1.2.2 Flags for the Generated Code and the Application

There is no compile flag for the generated code.

## 4.1.3 Integration

The Runtime Library may require some platform adaptation before it can be used. All the platform specific customization is defined in “`tsnc_custom.h`”. This file captures all the external dependencies of the Runtime Library and the generated code. This is the only file you need to modify for platform adaptation. The default “`tsnc_custom.h`” works on many development platforms without any modification required.

### 4.1.3.1 Macros

When an error is detected during a message operation, the “`TSNC_MSG_ERROR`” macro is used to report the error. By default, the macro simply returns the error status code. During debugging you can redefine this macro to an “`assert`” to help you pinpoint exactly where the error is reported.

Different platforms express 64-bit integer constants differently and use different format modifiers for 64-bit integer values. The “`TSNC_INT64_CONSTANT`” macro and the “`TSNC_INT64_MODIFIER`” macro are designed to shield the Runtime Library and the generated code from this platform dependent feature.

### 4.1.3.2 Types

The following platform independent integer types must be defined to match your environment. If you already defined them in another header file, simply include that header file.

**Table 4-3 C/C++ Platform Adaptation Data Types**

Type	Description
<code>tsnc_int8</code>	Signed 8-bit integer

tsnc_int16	Signed 16-bit integer
tsnc_int32	Signed 32-bit integer
tsnc_int64	Signed 64-bit integer (if your platform supports it)
tsnc_uint8	Unsigned 8-bit integer
tsnc_uint16	Unsigned 16-bit integer
tsnc_uint32	Unsigned 32-bit integer
tsnc_uint64	Unsigned 64-bit integer (if your platform supports it)
tsnc_float32	32-bit floating pointer number.

### 4.1.3.3 Functions

The following functions need to be implemented according to your environment. They can be implemented using macros.

**Table 4-4 C/C++ Platform Adaptation Functions**

Function	Description
tsnc_assert	Assertion
tsnc_new	Allocate a block of memory in bytes. This is the default memory allocation function.
tsnc_delete	Deallocate a block memory. This is the default memory deallocation function.
tsnc_strlen	String length
tsnc_strcpy	String copy
tsnc_memset	Memory set
tsnc_memcpy	Memory copy

### 4.1.4 Error Status Code

The message operation functions return a status code. The status codes are described in Table 4-5.

**Table 4-5 C/C++ Message Operation Error Status Code**

Status Code	Description
TSNC_STATUS_OK	The function is completed successfully.
TSNC_STATUS_FUNCTION_NOT_SUPPORTED	The function is not supported by the generated code due to compile flags.
TSNC_STATUS_INVALID_PARAMETER	Invalid function argument
TSNC_STATUS_INVALID_FIELD_VALUE	Invalid field value. Not currently used.
TSNC_STATUS_DIVIDE_BY_ZERO	Divide by zero while evaluating an expression
TSNC_STATUS_NEGATIVE_FIELD_LENGTH	A field length evaluates to a negative number.
TSNC_STATUS_ILLEGAL_ALIGNMENT	An alignment expression evaluates to an illegal value, for example, zero alignment.
TSNC_STATUS_STACK_OVERFLOW	Runtime Library internal error. This should never occur.
TSNC_STATUS_STACK_UNDERFLOW	Runtime Library internal error. This

	should never occur.
TSNC_STATUS_BUFFER_OVERFLOW	The pack and print function may return this status code to indicate that the buffer is too small.
TSNC_STATUS_TOO_FEW_BITS	The unpack function may return this to indicate that there is not enough bits in the buffer.
TSNC_STATUS_SIZE_TOO_SMALL	The pack or unpack function may return this status code to indicate the size of a sized block is too small. See section <b>Error! Reference source not found.</b>
TSNC_STATUS_SIZE_TOO_BIG	The pack function may return this status code to indicate the size of a sized block is too big. See section <b>Error! Reference source not found.</b>
TSNC_STATUS_FILE_IO_ERROR	The fprintf function may return this to indicate file I/O error.
TSNC_STATUS_NOT_EQUAL	The equal function returns this when two messages are not equal.
TSNC_STATUS_FIELD_NOT_FOUND	Expression references to a field that is not present in the message due to conditionals in the message definition.
TSNC_STATUS_DUPLICATE_FIELD	A field is being visited twice due to conditionals in the message definition.
TSNC_STATUS_ARRAY_INDEX_OUT_OF_BOUND	Array index exceeds the maximum size of the array.
TSNC_STATUS_FIELD_VALUE_OUT_OF_RANGE	A message field value is out of range.
TSNC_STATUS_OUT_OF_MEMORY	Out of memory while allocating memory for internal pointer fields
TSNC_STATUS_MESSAGE_ERROR_CODE	A message error is reported. Use the get error code function to retrieve the error code.
TSNC_STATUS_USER_START	The user defined status starts here. The custom status can be returned from user-defined functions.

#### 4.1.4.1 Fields with a Size Constraint

TSN.1 allows a size constraint be specified for a nested message, a case of field, or a block of fields. An example of this is the following TLV definition.

```
wimax_tlv_Trigger() ::=
{
    Type      8;
    Length    8;

    Value     Length * 8 : case Type of
    {
        1 =>
            Trigger :
            {
                Type      2;
                Function   3;
                Action     3;
            }
        2 =>
            TriggerValue  8;
        3 =>
            TriggerAveragingDuration  8;
    }
}
```

It is imperative for the user to set the Length field to the correct size. Failure to do so may cause your message parser not being backward compatible with earlier versions of the message or not being forward compatible with future versions of the message. To ensure this does not happen, the generated parser checks for these errors at runtime. When the size is set too small or too big while packing a message, the pack function returns the error status “TSNC\_STATUS\_SIZE\_TOO\_SMALL” or “TSNC\_STATUS\_SIZE\_TOO\_BIG” respectively. If the size is too small during unpack, the unpack function returns “TSNC\_STATUS\_SIZE\_TOO\_SMALL”. It is legal for the size to be bigger than expected during unpack. This usually happens when a new version of the message adds additional fields at the end of the TLV, which should be skipped by the parser. This is why the message parser should always honor the Length field when unpacking a message.

When the size constraint only involves simple arithmetic and the length field appears only once, for example, “Length \* 8” or “(Length – 1) \* 8”, the length field is automatically set to the correct value when you call the SizeOf or the SetLength function on the top level message provided the user has not set the length field yet.

### 4.1.5 Message Settings

The behavior of the runtime functions can be customized by changing the message settings of a message instance. The settings are set using the configure function.

#### 4.1.5.1 TSNC\_MSG\_SETTING\_ENDIANNESS

If “-dynamic-endian” option is used, the generated code allow users to change the endianness for pack and unpack functions at runtime.

**Table 4-6 Endianness Settings**

Value	Description
-------	-------------

TSNC_MSG_DEFAULT_ENDIAN	Use endianness as specified in the TSN.1 file
TSNC_MSG_BIG_ENDIAN	Use big endian format to pack or unpack data
TSNC_MSG_LITTLE_ENDIAN	Use little endian format to pack or unpack data

#### 4.1.5.2 TSNC\_MSG\_SETTING\_PRINT

The print setting is used to configure the output format of the print and fprintf functions. The following table describes the different values of this setting.

**Table 4-7 Print Settings**

Value	Description
TSNC_MSG_PRINT_ALL	The default print format
TSNC_MSG_PRINT_FLAT	Print fully qualified field name and value pairs
TSNC_MSG_PRINT_XML	Print XML
TSNC_MSG_PRINT OMIT NONE	Print all fields except the ones marked “none” in the display extension. This setting only applies to TSNC_MSG_PRINT_ALL.
TSNC_MSG_PRINT OMIT BRACES	Omit curly braces in the output. This setting only applies to TSNC_MSG_PRINT_ALL and TSNC_MSG_PRINT_FLAT.
TSNC_MSG_PRINT OMIT MSG_NAME	Omit message names for the nested messages in the output. This setting only applies to TSNC_MSG_PRINT_ALL.

#### 4.1.5.3 TSNC\_MSG\_SETTING\_SCAN

The scan setting is used to configure how message data is parsed by the scan and fscanf functions. The following table describes the different values of this setting.

**Table 4-8 Scan Settings**

Value	Description
TSNC_MSG_SCAN_ALL	Scan output printed using the TSNC_MSG_PRINT_ALL setting
TSNC_MSG_SCAN_FLAT	Scan output printed using the TSNC_MSG_PRINT_FLAT setting
TSNC_MSG_SCAN_XML	Scan output printed using the TSNC_MSG_PRINT_XML setting
TSNC_MSG_SCAN_IGNORE_UNEXPECTED_FIELD	Ignore unexpected field in the scan input. This setting only

	applies to TSNC_MSG_SCAN_ALL.
--	----------------------------------

## 4.2 Using the Java Runtime Library

The TSN.1 Compiler uses the TSN.1 Server [5] as its Java Runtime Library. The Library is contained in a single jar file called “tsns.jar”. Store the jar file on your CLASSPATH where your Java program can access. Please refer to your Java software development toolkit documentation for detailed information on how to set the CLASSPATH.

### 4.2.1 Runtime Exception

The message operations may throw exceptions. The exceptions are part of the “com.protomatics.tsn.runtime” package and they have a common base class “TSNException”, which is derived from “RuntimeException”. The exceptions are described in Table 4-9.

**Table 4-9 Java Message Operation Runtime Exception**

Exception	Description
TSNInvalidParameterException	Invalid function argument
TSNInvalidFieldValueException	Invalid field value
TSNDivideByZeroException	Divide by zero while evaluating an expression
TSNNegativeFieldLengthException	A field length evaluates to a negative number.
TSNIllegalAlignmentException	An alignment expression evaluates to an illegal value, for example, zero alignment.
TSNBufferOverflowException	The pack function may return this status code to indicate that the buffer is too small.
TSNTooFewBitsException	The unpack function may return this to indicate that there is not enough bits in the buffer.
TSNSizeTooSmallException	The pack or unpack function may return this status code to indicate the size of a sized block is too small. See section <b>Error! Reference source not found.</b>
TSNSizeTooBigException	The pack function may return this status code to indicate the size of a sized block is too big. See section <b>Error! Reference source not found.</b>
TSNFieldNotFoundException	Expression references to a field that is not present in the message due to conditionals in the message definition.
TSNDuplicateFieldException	A field is being visited twice due to conditionals in the message definition.
TSNArrayIndexOutOfBoundsException	Array index exceeds the maximum size of

	the array.
<code>TSNFieldValueOutOfRangeException</code>	A message field value is out of range.
<code>TSNMessageErrorCodeException</code>	A message error is reported. Use the get error code function to retrieve the error code.

## 5 Using the C Runtime Library API

The C Runtime Library API provides the function interface for your application to perform operations on TSN.1 messages. The details of these functions are described in the subsequent sections of this Chapter as well as in the “tsnc\_msg.h” header file.

The examples in this section use the following TSN.1 definitions:

```

is856_admp_UATIRequest() ::=
{
    TransactionID 8;
}

is856_admp_UATIComplete() ::=
{
    MessageSequence      8;
    reserve               4;
    UpperOldUATILength  4;
    UpperOldUATI         8 * UpperOldUATILength;
}

is856_admp_ATAirMessage() ::=
{
    MessageID 8;

    messages : case MessageID of
    {
        0 =>
            UATIRequest : is856_admp_UATIRequest;
        2 =>
            UATIComplete : is856_admp_UATIComplete;
    }
}

```

The TSN.1 Compiler translates the above definitions to the following C structures:

```

extern tsnc_msg_Descriptor is856_admp_UATIRrequest_descriptor;

typedef struct _is856_admp_UATIRrequest
{
    /* Message Base */
    tsnc_Message _tsnc_msg_;

    /* Fields */
    tsnc_uint8    TransactionID;

} is856_admp_UATIRrequest;

extern tsnc_msg_Descriptor is856_admp_UATIComplete_descriptor;

typedef struct _is856_admp_UATIComplete
{
    /* Message Base */
    tsnc_Message _tsnc_msg_;

    /* Fields */
    tsnc_uint8    MessageSequence;
    tsnc_uint8    UpperOldUATILength;
    tsnc_uint8    *UpperOldUATI;

} is856_admp_UATIComplete;

extern tsnc_msg_Descriptor is856_admp_ATAirMessage_descriptor;

typedef struct _is856_admp_ATAirMessage
{
    /* Message Base */
    tsnc_Message _tsnc_msg_;

    /* Fields */
    tsnc_uint8    MessageID;

    union
    {
        is856_admp_UATIRrequest    *UATIRrequest;
        is856_admp_UATIComplete    *UATIComplete;
    } messages;

} is856_admp_ATAirMessage;

```

## 5.1 Initialize and Finalize

A message *must* be properly created and initialized before it is used and properly finalized and destroyed after you are done with it. Some internal fields, bit buffers and nested messages, are generated as pointers. They must be populated with special care.

The Runtime Library offers flexible memory management schemes. The default scheme uses the global memory management functions `tsnc_new` and `tsnc_delete` defined in `tsnc_custom.h`. These two functions can be customized when you integrate the Runtime Library onto your platform (Section 4.1.3.3).

The second scheme is to use an allocator object. Each allocator object has two functions, an allocate function and a deallocate function. They are defined and supplied by the user. By associating an allocator object with a message, this scheme gives you precise control over how memory is managed for the message.

If the message has arguments, they can be set just like any bit field in the message after the message is initialized.

## 5.1.1 Using the Default Scheme

### 5.1.1.1 Automatic or static messages

Use the `tsnc_msg_initialize` function to initialize an automatic or static message variable and use the `tsnc_msg_finalize` function to finalize it. The initialize function sets the internal pointer fields to NULL. If the message has nested messages that use automatic storage, they will be initialized automatically. The finalize function deallocates memory recursively for the internal pointer fields, bit buffers and nested messages.

If you use the “-static-union” flag to generate union members as automatic, it is important that you also initialize the union member that will be present in the message before packing or printing the message. This cannot be done automatically during the parent message initialization, like the normal nested messages, because the parent message does not know in advance which union member will be present. You do not need to worry about this for unpack or scan.

```
{
    is856_admp_UATISComplete  msg;

    tsnc_msg_initialize(&msg, is856_admp_UATISComplete);

    ...

    tsnc_msg_finalize(&msg);
}
```

### 5.1.1.2 Dynamically created messages

Use the `tsnc_msg_new` function to create a message dynamically and use the `tsnc_msg_delete` function to delete the message. After the memory is allocated for the message, the `tsnc_msg_new` function also initializes the message; and before the memory is deallocated, the `tsnc_msg_delete` function finalizes the message.

```

{
    is856_admp_UATIDelete *msg_ptr;

    msg_ptr = tsnc_msg_new(is856_admp_UATIDelete);

    ...

    tsnc_msg_delete(msg_ptr);
}

```

### 5.1.1.3 Populating pointer fields

**While using the default scheme, memory for bit buffers must be allocated using `tsnc_new`. Nested message must be created using `tsnc_msg_new`. The `finalize` function depends on this fact.** If you decide to manage the memory for these pointers differently, remember to set the pointers back to NULL before the message is finalized.

```

{
    is856_admp_ATAirMessage *msg_ptr;

    /* Create and initialize the message.*/
    msg_ptr = tsnc_msg_new(is856_admp_ATAirMessage);

    /* Populate the message fields. */
    msg_ptr->MessageID = 2;

    msg_ptr->messages.UATIDelete = tsnc_msg_new
    (
        is856_admp_UATIDelete
    );
    msg_ptr->messages.UATIDelete->MessageSequence = 1;
    msg_ptr->messages.UATIDelete->UpperOldUATILength = 4;
    msg_ptr->messages.UATIDelete->UpperOldUATI = tsnc_new(4);

    ...

    /* This frees msg_ptr->messages.UATIDelete and
       msg_ptr->messages.UATIDelete->UpperOldUATI as well. */
    tsnc_msg_delete(msg_ptr);
}

```

### 5.1.2 Using an Allocator

The allocator scheme allows you to specify an allocator object when you instantiate a message. The Runtime Library will use this allocator for memory allocation and deallocation for the pointer fields in the message. The default scheme is equivalent to the allocator scheme with a NULL allocator.

The examples in this section assume that you have the following allocator object defined.

```

void* mem_pool_allocate(tsnc_uint32 size)
{
    /* Define your own memory allocation method */
    ...
}

void mem_pool_deallocate(void *p)
{
    /* Define your own memory deallocation method */
    /* Make sure p == NULL is allowed */
    ...
}

tsnc_msg_Allocator MemPoolAllocator =
{
    mem_pool_allocate,
    mem_pool_deallocate
};

```

### 5.1.2.1 Automatic or static messages

Use the `tsnc_msg_initialize_allocator` function to initialize an automatic or static message variable and use the `tsnc_msg_finalize` function to finalize it. The initialize function sets the internal pointer fields to NULL. If nested messages are automatic types, they will also be initialized. The finalize function deallocates memory recursively for the internal pointer fields, bit buffers and nested messages.

```

{
    is856_admp_UATIComplete msg;

    tsnc_msg_initialize_allocator
    (
        &msg,
        is856_admp_UATIComplete,
        &MemPoolAllocator
    );

    ...

    tsnc_msg_finalize(&msg);
}

```

### 5.1.2.2 Dynamically created messages

Use the `tsnc_msg_new_allocator` function to create a message dynamically and use the `tsnc_msg_delete` function to delete the message. After the memory is allocated for the message, the `tsnc_msg_new_allocator` function also initializes the message; and before the memory is deallocated, the `tsnc_msg_delete` function finalizes the message.

```
{
    is856_admp_UATIComplete *msg_ptr;

    msg_ptr = tsnc_msg_new_allocator
        (
            is856_admp_UATIComplete
            &MemPoolAllocator
        );

    ...

    tsnc_msg_delete(msg_ptr);
}
```

Note that the memory for the `msg_ptr` itself is also allocated using the allocator object.

### 5.1.2.3 Populating pointer fields

When using the allocator scheme, memory for bit buffers must be allocated using the same allocator. Nested messages must also be created using `tsnc_msg_new_allocator` with the same allocator object. The finalize function depends on this fact. If you decide to manage the memory for these pointers differently, remember to set them back to NULL before the message is finalized.

```

{
    is856_admp_ATAirMessage *msg_ptr;

    /* Create and initialize the message.*/
    msg_ptr = tsnc_msg_new_allocator
        (
            is856_admp_ATAirMessage,
            &MemPoolAllocator
        );

    /* Populate the message fields. */
    msg_ptr->MessageID = 2;

    msg_ptr->messages.UATIDelete = tsnc_msg_new_allocator
        (
            is856_admp_UATIDelete,
            &MemPoolAllocator
        );
    msg_ptr->messages.UATIDelete->MessageSequence = 1;
    msg_ptr->messages.UATIDelete->UpperOldUATIDLength = 4;
    msg_ptr->messages.UATIDelete->UpperOldUATI =
        MemPoolAllocator.allocate(4);

    ...

    /* This frees msg_ptr->messages.UATIDelete and
       msg_ptr->messages.UATIDelete->UpperOldUATI as well. */
    tsnc_msg_delete(msg_ptr);
}

```

## 5.2 Configure

The configure function is used to configure the various settings of a message. The settings are designed to customize the behavior of runtime functions. For a list of available settings, please refer to section 4.1.5.

```

{
    is856_admp_UATIDelete *msg_ptr;

    msg_ptr = tsnc_msg_new(is856_admp_UATIDelete);

    tsnc_msg_configure
        (
            msg_ptr,
            TSNC_MSG_SETTINGS_PRINT_FORMAT,
            TSNC_MSG_OMIT_NONE | TSNC_MSG_OMIT_MSG_NAME
        );

    ...
}

```

## 5.3 Reference & Unreference

Each message maintains a reference count, which is initialized to one when the message is created for the first time. To make a shallow copy of the message, call the reference function to increase the reference count by one. When you are done with a copy, call the unreference function to decrease the count by one. Delete the message only when the reference count reaches zero.

```

{
    is856_admp_UATIComplete *msg_ptr, msg_ptr2;

    msg_ptr = tsnc_msg_new(is856_admp_UATIComplete);
    /* Reference count is 1. */

    msg_ptr2 = (is856_admp_UATI_Complete *)
        tsnc_msg_reference(msg_ptr);
    /*
       Reference count becomes 2.
       msg_ptr2 and msg_ptr points to the same copy.
    */
    */

    ...

    if(tsnc_msg_unreference(msg_ptr) == 0)
    {
        tsnc_msg_delete(msg_ptr);
    }
    /* Reference count is back to 1; message is not deleted. */

    if(tsnc_msg_unreference(msg_ptr2) == 0)
    {
        tsnc_msg_delete(msg_ptr2);
    }
    /* Reference count is 0; message is deleted. */
}

```

## 5.4 Clone

This function makes a deep copy of a message. New memory blocks are allocated for bit buffers and nested messages. Nested messages are copied recursively. The clone function makes the copy by packing the source message into a buffer and then unpacks the buffer into the destination message.

```

{
    is856_admp_UATIComplete msg, msg2;
    tsnc_Status status;

    tsnc_msg_initialize(&msg, is856_admp_UATIComplete);
    msg.TransactionID = 1;

    tsnc_msg_initialize(&msg2, is856_admp_UATIComplete);

    /* Clone msg into msg2 */
    status = tsnc_msg_clone(&msg, &msg2);
    if(status != TSNC_STATUS_OK)
    {
        /* Report error */
    }
}

```

## 5.5 Equal

The equal function compares two messages for equality. The function first makes sure the two messages are of the same type, then packs the two messages into separate buffers, and finally compares the bits in the buffers for equality. It does not do a member-wise comparison.

```

{
    is856_admp_UATIComplete msg, msg2;

    /* Initialize and populate msg and msg2 */
    ...

    if(tsnc_msg_equal(&msg, &msg2) == TSNC_STATUS_OK)
    {
        /* Messages are equal. */
        ...
    }
}

```

## 5.6 SizeOf

This function computes the size of a message in unit of bits. This function also automatically sets the length field in a size constraint. See **Error! Reference source not found.** for details.

```
{
    is856_admp_UATIRrequest  msg;
    tsnc_uint32              nbit;
    tsnc_Status              status;

    tsnc_msg_initialize(&msg, is856_admp_UATIRrequest);

    msg.TransactionID = 0;

    status = tsnc_msg_sizeof(&msg, &nbit);
    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    tsnc_msg_finalize(&msg);
}
```

## 5.7 SetLength

This function automatically sets the length field in a size constraint. See **Error! Reference source not found.** for details.

## 5.8 Pack & Unpack

The pack function packs a properly populated message structure into a byte array while the unpack function unpacks the bits in a byte array into a message structure. When unpacking a message, you only need to instantiate the top-level message. You should not allocate memory for the internal pointer fields. They are allocated automatically by the unpack function using the memory allocation scheme specified for the message. Reserved and alignment fields in a message are packed with 0's and they are skipped by the unpack function.

```

{
    is856_admp_UATIRequest  msg;
    tsnc_uint8              buf[128];
    tsnc_uint32             nbit_packed, nbit_unpacked;
    tsnc_Status             status;

    tsnc_msg_initialize(&msg, is856_admp_UATIRequest);

    msg.TransactionID = 0;

    status = tsnc_msg_pack
    (
        &msg,                /* Message to be packed. */
        buf,                 /* Buffer to pack into. */
        0,                   /* Bit offset to start packing. */
        sizeof(buf) * 8,     /* Buffer length in bits. */
        &nbit_packed         /* Returns number of bits packed. */
    );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    status = tsnc_msg_unpack
    (
        &msg,                /* Message to be unpacked. */
        buf,                 /* Buffer to unpack from. */
        0,                   /* Bit offset to start unpacking. */
        nbit_packed,        /* Buffer length in bits. */
        &nbit_unpacked       /* Returns number of bits unpacked. */
    );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    tsnc_msg_finalize(&msg);
}

```

Note that the fourth argument to the pack and unpack function indicates the number of *available* bits in the buffer, not the total size. For example, if the total size of the buffer is “BUF\_SIZE” bytes and offset is “3”, then buffer length should be set to “BUF\_SIZE \* 8 – 3”.

## 5.9 Print & Scan

The print function prints a properly populated message structure into a human readable ASCII string while the scan function parses the string into a message structure. The scan function can only parse strings printed by the print function using the default “TSNC\_MSG\_PRINT\_ALL” format. When scanning a message, you only need to instantiate the top-level message. You should not allocate memory for the internal pointer

fields. They are allocated automatically by the scan function using the memory allocation scheme specified for the message.

You *must* also compile your TSN.1 files with the “-print” and “-scan” options in order for these two functions to work properly.

```

{
    is856_admp_UATIRrequest  msg;
    char                     text[1024];
    tsnc_uint32              nbyte_scanned;
    tsnc_Status              status;

    tsnc_msg_initialize(&msg, is856_admp_UATIRrequest);

    msg.TransactionID = 1;

    status = tsnc_msg_print
        (
            &msg,          /* Message to be printed. */
            text,          /* String buffer to print into. */
            sizeof(text) /* Size of the string buffer. */
        );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    status = tsnc_msg_scan
        (
            &msg,          /* Message to be printed. */
            text,          /* String buffer to scan from. */
            &nbyte_scanned /* Number of character scanned. */
        );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    tsnc_msg_finalize(&msg);
}

```

## 5.10FPrint & FScan

The fprint function behaves exactly like the print function except it prints to a file instead of a string, similarly for fscan. The fscan function can only parse strings printed by the fprint function using the default “TSNC\_MSG\_PRINT\_ALL” format. When scanning a message, you only need to instantiate the top-level message. You should not allocate memory for the internal pointer fields. They are allocated automatically by the scan function using the memory allocation scheme specified for the message.

You *must* also compile your TSN.1 files with the “-fprint” and “-fscan” options in order for these two functions to work properly.

```

{
  is856_admp_UATIRrequest  msg;
  tsnc_Status              status;
  FILE                    *file;

  tsnc_msg_initialize(&msg, is856_admp_UATIRrequest);

  msg.TransactionID = 1;

  file = fopen("msgs.txt", "w");
  status = tsnc_msg_fprint
    (
      &msg, /* Message to be printed. */
      file /* Pointer to file to print into. */
    );
  fclose(file);

  if(status != TSNC_STATUS_OK)
  {
    /* Report error. */
  }

  file = fopen("msgs.txt", "r");
  status = tsnc_msg_fscan
    (
      &msg, /* Message to be printed. */
      file /* Pointer to file to scan from. */
    );
  fclose(file);

  if(status != TSNC_STATUS_OK)
  {
    /* Report error. */
  }

  tsnc_msg_finalize(&msg);
}

```

## 5.11 Get Error Code

When a message error code is detected while performing a message operation, the operation returns `TSNC_STATUS_MESSAGE_ERROR_CODE`. To retrieve the error code as specified in the TSN.1 file using the “error” keyword, use the get error code function. After this function is called, the error code is reset back to zero.

```
{
    is856_admp_UATIRrequest  msg;
    tsnc_uint8               buf[128];
    tsnc_uint32              nbit_packed;
    tsnc_Status              status;
    tsnc_int32               error_code;

    tsnc_msg_initialize(&msg, is856_admp_UATIRrequest);

    msg.TransactionID = 0;

    status = tsnc_msg_pack
        (
            &msg,          /* Message to be packed. */
            buf,           /* Buffer to pack into. */
            0,             /* Bit offset to start packing. */
            sizeof(buf) * 8, /* Buffer length in bits. */
            &nbit_packed   /* Returns number of bits packed. */
        );

    if(status == TSNC_STATUS_MESSAGE_ERROR_CODE)
    {
        error_code = tsnc_msg_get_error_code(&msg);
        /* Report error */
    }
}
```

## 6 Using the C++ Runtime Library API

The C++ Runtime Library API provides an interface for your C++ application to perform operations on TSN.1 messages. The details of these operations are described in the subsequent sections of this Chapter as well as the “tsncxx\_msg.h”. These operations are declared as member functions of the “tsncxx\_Message.h” class, which serves as the base class for all generated message classes.

The examples in this section use the following TSN.1 definitions:

```

is856_admp_UATIRequest() ::=
{
    TransactionID 8;
}

is856_admp_UATIComplete() ::=
{
    MessageSequence      8;
    reserve               4;
    UpperOldUATILength  4;
    UpperOldUATI         8 * UpperOldUATILength;
}

is856_admp_ATAirMessage() ::=
{
    MessageID 8;

    messages : case MessageID of
    {
        0 =>
            UATIRequest : is856_admp_UATIRequest;
        1 =>
            UATIComplete : is856_admp_UATIComplete;
    }
}

```

The TSN.1 Compiler translates the above definitions to the following C++ classes:

```

extern tsnc_msg_Descriptor is856_admp_UATIRrequest_descriptor;

class is856_admp_UATIRrequest: public tsncxx_Message
{
public:
    is856_admp_UATIRrequest
    (
        const tsnc_msg_Allocator *allocator = 0
    )
    {
        _tsnc_msg_initialize
        (
            this,
            &is856_admp_UATIRrequest_descriptor,
            allocator
        );
    }

    virtual ~is856_admp_UATIRrequest()
    {
        _tsnc_msg_finalize(this);
    }

    /* Fields */
    tsnc_uint8    TransactionID;
};

extern tsnc_msg_Descriptor is856_admp_UATIRcomplete_descriptor;

class is856_admp_UATIRcomplete: public tsncxx_Message
{
public:
    is856_admp_UATIRcomplete
    (
        const tsnc_msg_Allocator *allocator = 0
    )
    {
        _tsnc_msg_initialize
        (
            this,
            &is856_admp_UATIRcomplete_descriptor,
            allocator
        );
    }

    virtual ~is856_admp_UATIRcomplete()
    {
        _tsnc_msg_finalize(this);
    }

    /* Fields */
    tsnc_uint8    MessageSequence;
    tsnc_uint8    UpperOldUATIRlength;
};

```

```

    tsnc_uint8    *UpperOldUATI;
};

extern tsnc_msg_Descriptor is856_admp_ATAirMessage_descriptor;

class is856_admp_ATAirMessage: public tsncxx_Message
{
public:
    is856_admp_ATAirMessage
    (
        const tsnc_msg_Allocator *allocator = 0
    )
    {
        _tsnc_msg_initialize
        (
            this,
            &is856_admp_ATAirMessage_descriptor,
            allocator
        );
    }

    virtual ~is856_admp_ATAirMessage()
    {
        _tsnc_msg_finalize(this);
    }

    /* Fields */
    tsnc_uint8    MessageID;

    union
    {
        is856_admp_UATIRequest    *UATIRequest;
        is856_admp_UATIComplete  *UATIComplete;
    } messages;
};

```

## 6.1 Initialize and Finalize

A message *must* be properly created and initialized before it is used and properly finalized and destroyed after you are done with it. Some internal fields, bit buffers and nested messages, are generated as pointers. They must be populated with special care.

The Runtime Library offers flexible memory management schemes. If you use the `new` and `delete` operators directly to allocate and deallocate memory for your message, then you are using the default scheme. The `new` and `delete` operators are overloaded in the message base class `tsncxx_Message` to invoke `tsnc_new` and `tsnc_delete` respectively, which are defined in `tsnc_custom.h`. These two functions can be customized when you integrate the Runtime Library to your platform (Section 4.1.3.3).

The second scheme is to use an allocator object. Each allocator object has two functions, an allocate function and a deallocate function. They are defined and supplied by the user. By associating an allocator object with a message, this scheme gives you precise control over how memory is managed for the message.

If the message has arguments, they can be passed in as parameters to the message constructor.

## 6.1.1 Using the Default Scheme

### 6.1.1.1 Automatic or static messages

C++ messages are automatically initialized by the default constructor and finalized by the default destructor. The constructor sets the internal pointer fields to NULL. The destructor deallocates memory recursively for the internal pointer fields, bit buffers and nested messages.

```
{
    is856_admp_UATIComplete  msg;
    ...
}
```

### 6.1.1.2 Dynamically created messages

Use the overloaded `new` operator to create a message dynamically and use the `delete` operator to delete the message.

```
{
    is856_admp_UATIComplete *msg_ptr;

    msg_ptr = new is856_admp_UATIComplete();

    ...

    delete msg_ptr;
}
```

### 6.1.1.3 Populating pointer fields

**When using the default scheme, memory for bit buffers must be allocated using `tsnc_new`. Nested message must be created using the overloaded `new`. The destructor depends on this fact.** If you decide to manage the memory for these pointers differently, remember to set them back to NULL before the message is destructed.

```

{
    is856_admp_ATAirMessage *msg_ptr;

    /* Create and initialize the message.*/
    msg_ptr = new is856_admp_ATAirMessage();

    /* Populate the message fields. */
    msg_ptr->MessageID = 2;

    msg_ptr->messages.UATIDelete =
        new is856_admp_UATIDelete();
    msg_ptr->messages.UATIDelete->MessageSequence = 1;
    msg_ptr->messages.UATIDelete->UpperOldUATILength = 4;
    msg_ptr->messages.UATIDelete->UpperOldUATI = tsnc_new(4);

    ...

    /* This frees msg_ptr->messages.UATIDelete and
       msg_ptr->messages.UATIDelete->UpperOldUATI as well. */
    delete msg_ptr;
}

```

## 6.1.2 Using an Allocator

The allocator scheme allows you to specify an allocator object when you instantiate a message. The Runtime Library will use this allocator for memory allocation and deallocation for the pointer fields in the message. The default scheme is equivalent to the allocator scheme with a NULL allocator.

The examples in this section assume that you have the following allocator object defined.

```

void* mem_pool_allocate(tsnc_uint32 size)
{
    /* Define your own memory allocation method */
    ...
}

void mem_pool_deallocate(void *p)
{
    /* Define your own memory deallocation method */
    /* Make sure p == NULL is allowed */
    ...
}

tsnc_msg_Allocator MemPoolAllocator =
{
    mem_pool_allocate,
    mem_pool_deallocate
};

```

### 6.1.2.1 Automatic or static messages

C++ messages are automatically initialized by the default constructor and finalized by the default destructor. The constructor sets the internal pointer fields to NULL. The

destructor uses the allocator to deallocate memory recursively for the internal pointer fields, bit buffers and nested messages.

```
{
    is856_admp_UATIComplete  msg(&MemPoolAllocator);
    ...
}
```

### 6.1.2.2 Dynamically created messages

Use the overloaded `new` operator to create a message dynamically. Pass the allocator object to both the `new` operator and the constructor.

```
{
    is856_admp_UATIComplete  *msg_ptr;

    msg_ptr = new (&MemPoolAllocator) is856_admp_UATIComplete
        (
            &MemPoolAllocator
        );
    ...
    delete msg_ptr;
}
```

Note that the memory for the `msg_ptr` itself is also allocated using the allocator object.

### 6.1.2.3 Populating pointer fields

When using the allocator scheme, memory for bit buffers must be allocated using the same allocator. Nested messages must also be created using the same allocator object. The destructor depends on this fact. If you decide to manage the memory for these pointers differently, remember to set them back to NULL before the message is destructed.

```

{
    is856_admp_ATAirMessage *msg_ptr;

    /* Create and initialize the message.*/
    msg_ptr = new (&MemPoolAllocator) is856_admp_ATAirMessage
        (
            &MemPoolAllocator
        );

    /* Populate the message fields. */
    msg_ptr->MessageID = 2;

    msg_ptr->messages.UATIDelete =
        new (&MemPoolAllocator) is856_admp_UATIDelete
            (
                &MemPoolAllocator
            );
    msg_ptr->messages.UATIDelete->MessageSequence = 1;
    msg_ptr->messages.UATIDelete->UpperOldUATIDeleteLength = 4;
    msg_ptr->messages.UATIDelete->UpperOldUATI =
        MemPoolAllocator.Allocate(4);

    ...

    /* This frees msg_ptr->messages.UATIDelete and
       msg_ptr->messages.UATIDelete->UpperOldUATI as well. */
    delete msg_ptr;
}

```

## 6.2 Configure

The configure function is used to configure the various settings of a message. The settings are designed to customize the behavior of runtime functions. For a list of available settings, please refer to section 4.1.5.

```

{
    is856_admp_UATIDelete *msg_ptr;

    msg_ptr = new is856_admp_UATIDelete();

    msg_ptr->Configure
        (
            TSNC_MSG_SETTINGS_PRINT_FORMAT,
            TSNC_MSG_OMIT_NONE | TSNC_MSG_OMIT_MSG_NAME
        );

    ...
}

```

## 6.3 Reference & Unreference

Each message maintains a reference count, which is initialized to one when the message is created for the first time. To make a shallow copy of the message, call the reference function to increase the reference count by one. When you are done with a copy, call the

unreference function to decrease the count by one. Delete the message only when the reference count reaches zero.

```

{
    is856_admp_UATIDelete *msg_ptr, msg_ptr2;

    msg_ptr = new is856_admp_UATIDelete();
    /* Reference count is 1. */

    msg_ptr2 = (is856_admp_UATI_Delete *) msg_ptr->Reference();
    /*
       Reference count becomes 2.
       msg_ptr2 and msg_ptr points to the same copy.
    */
    */

    ...

    if(msg_ptr->Unreference() == 0)
    {
        delete msg_ptr;
    }
    /* Reference count is back to 1; message is not deleted. */

    if(msg_ptr2->Unreference() == 0)
    {
        delete msg_ptr2;
    }
    /* Reference count is 0; message is deleted. */
}

```

## 6.4 Clone

Unless the “-copiable” is flag is used, the default C++ copy constructor and assignment operator are declared as private in the message base class `tsncxx_Message` to avoid accidental copying of a message. If you intend to make a deep copy of a message, use the Clone function. New memory blocks are allocated for bit buffers and nested messages. Nested messages are copied recursively. The clone function makes the copy by packing the source message into a buffer and then unpacks the buffer into the destination message.

```

{
    is856_admp_UATIComplete msg, msg2;
    tsnc_Status status;

    msg.TransactionID = 1;

    /* Clone msg into msg2 */
    status = msg.Clone(&msg2);
    if(status != TSNC_STATUS_OK)
    {
        /* Report error */
    }
}

```

## 6.5 Equal

The equal function compares two messages for equality. For C++, the comparison is done using the overloaded equal or not equal operator. The equal function first makes sure the two messages are of the same type, then packs the two messages into separate buffers, and finally compares the bits in the buffers for equality. It does not do a member-wise comparison.

```

{
    is856_admp_UATIComplete msg, msg2;

    /* Populate msg and msg2 */
    ...

    if(msg == msg2)
    {
        ...
    }

    if(msg != msg2)
    {
        ...
    }
}

```

## 6.6 SizeOf

This function computes the size of a message in unit of bits. This function also automatically sets the length field in a size constraint. See **Error! Reference source not found.** for details.

```
{
    is856_admp_UATIRrequest  msg;
    tsnc_uint32              nbit;
    tsnc_Status              status;

    msg.TransactionID = 0;

    status = msg.SizeOf(&nbit);
    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }
}
```

## 6.7 SetLength

This function automatically sets the length field in a size constraint. See **Error! Reference source not found.** for details.

## 6.8 Pack & Unpack

The pack function packs a properly populated message structure into a byte array while the unpack function unpacks the bits in a byte array into a message structure. When unpacking a message, you only need to instantiate the top-level message. You should not allocate memory for the internal pointer fields. They are allocated automatically by the unpack function using the memory allocation scheme specified for the message. Reserved and alignment fields in a message are packed with 0's and they are skipped by the unpack function.

```

{
    is856_admp_UATIRequest  msg;
    tsnc_uint8              buf[128];
    tsnc_uint32             nbit_packed, nbit_unpacked;
    tsnc_Status             status;

    msg.TransactionID = 0;

    status = msg.Pack
    (
        buf,                /* Buffer to pack into. */
        0,                  /* Bit offset to start packing. */
        sizeof(buf) * 8,    /* Buffer length in bits. */
        &nbit_packed        /* Returns number of bits packed. */
    );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    status = msg.Unpack
    (
        buf,                /* Buffer to unpack from. */
        0,                  /* Bit offset to start unpacking. */
        nbit_packed,        /* Buffer length in bits. */
        &nbit_unpacked      /* Returns number of bit unpacked. */
    );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }
}

```

Note that the third argument to the pack and unpack function indicates the number of *available* bits in the buffer, not the total size. For example, if the total size of the buffer is “BUF\_SIZE” bytes and offset is “3”, then buffer length should be set to “BUF\_SIZE \* 8 – 3”.

## 6.9 Print & Scan

The print function prints a properly populated message structure into a human readable ASCII string while the scan function parses the string into a message structure. The scan function can only parse strings printed by the print function using the default “TSNC\_MSG\_PRINT\_ALL” format. When scanning a message, you only need to instantiate the top-level message. You should not allocate memory for the internal pointer fields. They are allocated automatically by the scan function using the memory allocation scheme specified for the message.

You *must* also compile your TSN.1 files with the “-print” and “-scan” options in order for these two functions to work properly.

```

{
    is856_admp_UATIRrequest  msg;
    char                    text[1024];
    tsnc_uint32             nbyte_scanned;
    tsnc_Status             status;

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    status = msg.Print
    (
        text,                /* String buffer to print into.*/
        sizeof(text) /* Size of the string buffer. */
    );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    status = msg.Scan
    (
        text,                /* String buffer to scan from.*/
        &nbyte_scanned /* Number of character scanned. */
    );

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }
}

```

## 6.10 FPrint & FScan

The `fprint` function behaves exactly like the `print` function except it prints to a file instead of a string, similarly for `fscan`. The `fscan` function can only parse strings printed by the `fprint` function using the default “TSNC\_MSG\_PRINT\_ALL” format. When scanning a message, you only need to instantiate the top-level message. You should not allocate memory for the internal pointer fields. They are allocated automatically by the scan function using the memory allocation scheme specified for the message.

You *must* also compile your TSN.1 files with the “-fprint” and “-fscan” options in order for these two functions to work properly.

```

{
    is856_admp_UATIRrequest  msg;
    tsnc_Status               status;
    FILE                      *file;

    msg.TransactionID = 1;

    file = fopen("msgs.txt", "w");
    status = msg.FPrint
        (
            file /* Pointer to file to print into. */
        );
    fclose(file);

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    file = fopen("msgs.txt", "r");
    status = msg.FScan
        (
            file /* Pointer to file to scan from. */
        );
    fclose(file);

    if(status != TSNC_STATUS_OK)
    {
        /* Report error. */
    }

    tsnc_msg_finalize(&msg);
}

```

## 6.11 Get Error Code

When a message error code is detected while performing a message operation, the operation returns `TSNC_STATUS_MESSAGE_ERROR_CODE`. To retrieve the error code as specified in the TSN.1 file using the “error” keyword, use the get error code function. After this function is called, the error code is reset back to zero.

```
{
    is856_admp_UATIRrequest  msg;
    tsnc_uint8                buf[128];
    tsnc_uint32               nbit_packed;
    tsnc_Status               status;
    tsnc_int32                error_code;

    msg.TransactionID = 0;

    status = msg.Pack
        (
            buf,                /* Buffer to pack into. */
            0,                  /* Bit offset to start packing. */
            sizeof(buf) * 8,    /* Buffer length in bits. */
            &nbit_packed        /* Returns number of bits packed. */
        );

    if(status == TSNC_STATUS_MESSAGE_ERROR_CODE)
    {
        error_code = msg.GetErrorCode();
        /* Report error */
    }
}
```

## 7 Using the Java Runtime Library API

The Java Runtime Library defines the “com.protomatics.tsn.runtime.TSNMessage” class, which serves as the base class for all generated Java message classes. The message operations are defined as public member functions of the “TSNMessage” class.

The example in this section uses the following TSN.1 definition:

```

...

is856_rup_RouteUpdate() ::=
{
    MessageSequence          8;
    ReferencePilotPN         9;
    ReferencePilotStrength   6;
    ReferenceKeep             1;
    NumPilots                4;

    Pilots[NumPilots] :
    {
        PilotPNPhase        15;
        ChannelIncluded      1;

        if(ChannelIncluded == 1)
        {
            Channel : is856_ChannelRecord;
        }

        PilotStrength        6;
        Keep                  1;
    }

    align(8);
}

```

### 7.1 Clone

The clone method makes a copy of a message.

```
public Object clone();
```

### 7.2 Equals

The equal method compares two messages for equality. The method first makes sure the two messages are of the same type, then packs the two messages into separate buffers, and finally compares the bits in the buffers for equality. It does not do a member-wise comparison.

```
public boolean equals(Object obj);
```

## 7.3 SizeOf

The `sizeOf` method returns the size of a message in bits. This method also automatically sets the length fields in a size constraint. See `SetLength` (7.4) for details.

```
public int sizeOf();
```

## 7.4 SetLength

If a size constraint is specified for a nested message or a case of field, this method automatically sets the length field in the constraint if the length field is zero. For this to work, the size constraint must only involve simple arithmetic and the length field can only appear once, for example, “Length \* 8”, “(Length + 1) \* 8”, etc.

```
public void setLength();
```

## 7.5 Pack

The `pack` method packs a message into a byte buffer. The number of bits packed is returned. The “offset” argument specifies the bit offset to start packing. The “length” argument specifies the number of available bits in the buffer. For example, if “offset” is 3, then “length” should be set to “buf.length \* 8 – 3”.

```
public int pack(byte buf[], int offset, int length);
```

## 7.6 Unpack

The `unpack` method unpacks a message from a byte buffer. The number of bits unpacked is returned. The “offset” argument specifies the bit offset to start unpacking. The “length” argument specifies the number of available bits in the buffer. For example, if “offset” is 3, then “length” should be set to “buf.length \* 8 – 3”.

```
public int unpack(byte buf[], int offset, int length);
```

## 7.7 Print

The `print` method prints the content of a message to a string.

```
public String print();
```

The following example illustrates the output of an `IS856_rup_RouteUpdate` message:

```
is856_rup_RouteUpdate
{
    MessageSequence = 2
    ReferencePilotPN = 12
    ReferencePilotStrength = 26
    ReferenceKeep = 1
    NumPilots = 2

    Pilots[0] =
    {
        PilotPNPhase = 131
        ChannelIncluded = 1
        Channel =
        {
            SystemType = 0
            BandClass = 1
            ChannelNumber = 125
        }
        PilotStrength = 18
        Keep = 1
    }
    Pilots[1] =
    {
        PilotPNPhase = 178
        ChannelIncluded = 0
        PilotStrength = 3
        Keep = 0
    }
}
```

## 7.8 Scan

The scan method reads a string and parses it into a message. The scan method can parse strings output by the print method.

```
public void scan(String text);
```

## 7.9 PrintXML

The printXML method prints the content of a message to a XML string.

```
public String printXML();
```

The following example illustrates the output of an IS856\_rup\_RouteUpdate message:

```

<is856_rup_RouteUpdate>
  <MessageSequence>2</MessageSequence>
  <ReferencePilotPN>12</ReferencePilotPN>
  <ReferencePilotStrength>26</ReferencePilotStrength>
  <ReferenceKeep>1</ReferenceKeep>
  <NumPilots>2</NumPilots>
  <Pilots>
    <is856_rup_RouteUpdate_Pilots>
      <PilotPNPhase>131</PilotPNPhase>
      <ChannelIncluded>1</ChannelIncluded>
      <Channel>
        <is856_ChannelRecord>
          <SystemType>0</SystemType>
          <BandClass>1</BandClass>
          <ChannelNumber>125</ChannelNumber>
        </is856_ChannelRecord>
      </Channel>
      <PilotStrength>18</PilotStrength>
      <Keep>1</Keep>
    </is856_rup_RouteUpdate_Pilots>
    <is856_rup_RouteUpdate_Pilots>
      <PilotPNPhase>178</PilotPNPhase>
      <ChannelIncluded>0</ChannelIncluded>
      <PilotStrength>3</PilotStrength>
      <Keep>0</Keep>
    </is856_rup_RouteUpdate_Pilots>
  </Pilots>
</is856_rup_RouteUpdate>

```

## 7.10 ScanXML

The scan method reads a XML string and parses it into a message. The scanXML method can parse strings output by the printXML method.

```
public void scanXML(String text);
```

The following Java code snippet illustrates the usage of some of these APIs.

```

is856_rup_RouteUpdate rup = new is856_rup_RouteUpdate();
is856_rup_RouteUpdate rup2 = new is856_rup_RouteUpdate();

try
{
    rup.MessageSequence = 1;
    rup.ReferencePilotPN = 12;
    rup.ReferencePilotStrength = 27;
    rup.ReferenceKeep = 1;
    rup.NumPilots = 2;

    rup.Pilot[0] = rup.new Pilot();
    rup.Pilot[0].PilotPNPhase = 0x56;
    rup.Pilot[0].ChannelIncluded = 0;
    rup.Pilot[0].PilotStrength = 20;
    rup.Pilot[0].Keep = 1;

    rup.Pilot[1] = rup.new Pilot();
    rup.Pilot[1].PilotPNPhase = 0x3e;
    rup.Pilot[1].ChannelIncluded = 1;
    rup.Pilot[1].Channel = new is856_ChannelRecord();
    // populate the record;

    rup.Pilot[1].PilotStrength = 7;
    rup.Pilot[1].Keep = 0;

    int numBit = rup.sizeOf();

    byte buf[] = new byte[(numBit + 7) / 8];

    int numBitPacked = rup.pack(buf, 0, buf.length * 8);

    int numBitUnpacked = rup2.unpack(buf, 0, numBit);

    if(!rup.equals(rup2))
    {
        System.err.println("rup and rup2 are not equal.");
    }

    String text = rup.print();

    rup2.scan(text);

    if(!rup.equals(rup2))
    {
        System.err.println("rup and rup2 are not equal.");
    }
}
catch(Exception e)
{
    System.err.println(e.toString());
}

```

## 8 Using the Generated Wireshark Code

If the target is wireshark, the Compiler generates tables and functions that are used by Wireshark [4] to dissect packets. The generated dissectors are completely standalone. It does not require the Runtime Library.

For a file named “foo\_msgs.tsn”, the Compiler generates a “foo\_msgs.h” and a “foo\_msgs.c”. For built-in dissectors, Wireshark naming convention requires the filenames be prefixed with “packet-“. Use the “-built-in” flag to automatically add the “packet-“ prefix to the generated filenames. This is not required for plug-ins.

The generated header file contains the prototypes of the register and dissect functions for each message in the TSN.1 file. The generated source file contains the implementation of the register and dissect functions as well as the field info tables. The register function registers the fields of a message, and the dissect function dissects the message packets.

To link the generated code with Wireshark, you should create a file named “packet-foo.c” for a protocol named “foo”. Use the following code template to create this file and follow the instructions in the comments:

```

/* Include the generated header file */
#include "foo_msgs.h"

#ifdef HAVE_CONFIG_H
# include "config.h"
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <gmodule.h>
#include <epan/prefs.h>
#include <epan/packet.h>

/* forward reference */
void proto_register_foo();
void proto_reg_handoff_foo();
void dissect_foo(tvbuff_t *tvb, packet_info *pinfo, proto_tree
*tree);

/* Define version if we are not building Wireshark statically */
#ifdef ENABLE_STATIC
G_MODULE_EXPORT const gchar version[] = "0.0";
#endif

static int proto_foo = -1;
static int global_foo_port = 3001;
static dissector_handle_t foo_handle;

#ifdef ENABLE_STATIC
G_MODULE_EXPORT void plugin_register(void)
{
    /* register the new protocol, protocol fields, and subtrees */
    if (proto_foo == -1) { /* execute protocol initialization only
once */
        proto_register_foo();
    }
}

G_MODULE_EXPORT void plugin_reg_handoff(void){
    proto_reg_handoff_foo();
}
#endif

static int ett_foo = -1;

/* Setup protocol subtree array */
static int *ett[] = {
    &ett_foo
};

void proto_register_foo(void)
{
    module_t *module;

```

```

    if (proto_foo == -1)
    {
        proto_foo = proto_register_protocol (
            "Foo Protocol", /* name */
            "FOO",          /* short name */
            "foo"           /* abbrev */
        );

        module = prefs_register_protocol(proto_foo,
proto_reg_handoff_foo);
        proto_register_subtree_array(ett, array_length(ett));

        /* Call the generated register function on the top level */
        /* message for your protocol, "foo_msgs_Frame" in this */
        /* example. The register function recursively calls the */
        /* register functions of the nested messages. So you only */
        /* need to call the register functions on the top level */
        /* message. */
        foo_msgs_Frame_register(proto_foo);
    }
}

void proto_reg_handoff_foo(void)
{
    static int Initialized=FALSE;

    if (!Initialized) {
        foo_handle = create_dissector_handle(dissect_foo,
proto_foo);
        dissector_add("udp.port", global_foo_port, foo_handle);
    }
}

void dissect_foo
(
    tvbuff_t      *tvb,
    packet_info *pinfo,
    proto_tree *tree
)
{
    if (check_col(pinfo->cinfo, COL_PROTOCOL))
        col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
    /* Clear out stuff in the info column */
    if (check_col(pinfo->cinfo, COL_INFO)) {
        col_clear(pinfo->cinfo, COL_INFO);
    }

    if (tree) { /* we are being asked for details */
        proto_item      *ti;
        proto_tree      *foo_tree;
        is856_admp_AirMessage *msg;
        tsnc_uint32      nbit_dissected;

        ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, FALSE);

        foo_tree = proto_item_add_subtree(ti, ett_foo);
    }
}

```

```
/* Call the generated dissect function on the */
/* top-level message */
msg = ep_alloc(sizeof(foo_msgs_Frame));
nbit_dissected = foo_msgs_Frame_dissect
(
    msg,          /* The message to be dissected */
    proto_foo,   /* Protocol index */
    tvb,         /* The dissect data buffer */
    0,           /* bit offset to start dissect */
    pinfo,       /* Packet info */
    foo_tree     /* Dissect tree */
);
}
}
```

**Note that the forth argument is the start offset of the “tvb” in bits, not bytes.**

## 9 Using the Generated XML Code

When XML is selected as the target language, the Compiler translates the TSN.1 definitions into XML elements.

### 9.1 Output Files

The Compiler generates “foo.xml” for the TSN.1 file named “foo.tsn”. The root element is called “tsn1”. TSN.1 definitions are child elements of the “tsn1” element.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="tsn1_html.xsl"?>

<tsn1 file="foo.xml" xmlns=http://www.protomatics.com/TSN1>
  ...
</tsn1>
```

By default the XSLT stylesheet “tsn1\_html.xsl”, which translates the XML document into HTML, is associated with the XML output. You can override this default and supply your own stylesheet using the “-xsl” flag.

### 9.2 Integers

The integer definition

```
N_ADMPDefault ::= 0;
```

is translated into

```
<definition>
  <name>N_ADMPDefault</name>
  <integer>
    <value>0</value>
  </integer>
</definition>
```

### 9.3 Strings

The string definition

```
GREETING ::= "Hello, World!";
```

is translated into

```
<definition>
  <name>GREETING</name>
  <string>
    <value>Hello, World!</value>
  </string>
</definition>
```

## 9.4 Enumerations

The enumerated definition

```
is856_admp_AirMessageId ::= enumerated
{
  IS856_ADMP_UATI_REQUEST,
  IS856_ADMP_UATI_ASSIGNMENT,
  IS856_ADMP_UATI_COMPLETE,
  IS856_ADMP_HARDWARE_ID_REQUEST,
  IS856_ADMP_HARDWARE_ID_RESPONSE,

  IS856_ADMP_CONFIGURATION_REQUEST (0x50),
  IS856_ADMP_CONFIGURATION_RESPONSE
}
```

is translated into

```
<definition>
  <name> is856_admp_AirMessageId</name>
  <enumerated>
    <literal>
      <name>IS856_ADMP_UATI_REQUEST</name>
      <value>0</value>
    </literal>
    <literal>
      <name>IS856_ADMP_UATI_ASSIGNMENT</name>
      <value>1</value>
    </literal>
    <literal>
      <name>IS856_ADMP_UATI_COMPLETE</name>
      <value>2</value>
    </literal>
    <literal>
      <name>IS856_ADMP_HARDWARE_ID_REQUEST</name>
      <value>3</value>
    </literal>
    <literal>
      <name>IS856_ADMP_HARDWARE_ID_RESPONSE</name>
      <value>4</value>
    </literal>
    <literal>
      <name>IS856_ADMP_CONFIGURATION_REQUEST</name>
      <value>80</value>
    </literal>
    <literal>
      <name>IS856_ADMP_CONFIGURATION_RESPONSE</name>
      <value>81</value>
    </literal>
  </enumerated>
</definition>
```

## 9.5 Messages

The message definition

```
is856_admp_UATISendComplete() ::=
{
  ...
}
```

is translated into

```

<definition>
  <name>GREETING</name>
  <message>
    <argument-list>
      ...
    </argument-list>
    <field-list>
      ...
    </field-list>
  </message>
</definition>

```

The “argument-list” and “field-list” elements contain the arguments and fields of the message. Each child element of the list is a “field” element.

## 9.6 Field Declarations

Field declarations in a TSN.1 message body are translated into “field” elements in XML. Table 3-1 shows the mapping between the two:

**Table 9-1 Field Declaration to XML Element Mapping**

Field Declaration	XML Element
bits 8;	<pre> &lt;field&gt;   &lt;name&gt;bits&lt;/name&gt;   &lt;length&gt;8&lt;/length&gt;   &lt;condition&gt;&gt;true&lt;/condition&gt;   &lt;bit-field&gt;     &lt;signed&gt;&gt;false&lt;/signed&gt;     &lt;minimum-value&gt;0&lt;/minimum-value&gt;     &lt;maximum-value&gt;255&lt;/maximum-value&gt;     &lt;c-type&gt;tsnc_uint8&lt;/c-type&gt;     &lt;c++-type&gt;tsnc_uint8&lt;/c++-type&gt;   &lt;/bit-field&gt; &lt;/field&gt; </pre>
bits 8 signed;	<pre> &lt;field&gt;   &lt;name&gt;bits&lt;/name&gt;   &lt;length&gt;8&lt;/length&gt;   &lt;condition&gt;&gt;true&lt;/condition&gt;   &lt;bit-field&gt;     &lt;signed&gt;&gt;true&lt;/signed&gt;     &lt;minimum-value&gt;-128&lt;/minimum-value&gt;     &lt;maximum-value&gt;127&lt;/maximum-value&gt;     &lt;c-type&gt;tsnc_int8&lt;/c-type&gt;     &lt;c++-type&gt;tsnc_int8&lt;/c++-type&gt;   &lt;/bit-field&gt; &lt;/field&gt; </pre>
length 8; bits length * 8;	<pre> &lt;field&gt;   &lt;name&gt;bits&lt;/name&gt;   &lt;length&gt;length*8&lt;/length&gt;   &lt;condition&gt;&gt;true&lt;/condition&gt;   &lt;bit-field&gt;     &lt;signed&gt;&gt;false&lt;/signed&gt;     &lt;c-type&gt;tsnc_uint8*&lt;/c-type&gt;     &lt;c++-type&gt;tsnc_uint8*&lt;/c++-type&gt;   &lt;/bit-field&gt; &lt;/field&gt; </pre>
str 8 strings;	<pre> &lt;field&gt;   &lt;name&gt;str&lt;/name&gt; </pre>

	<pre> &lt;length&gt;8&lt;/length&gt; &lt;condition&gt;&gt;true&lt;/condition&gt; &lt;string-field&gt;   &lt;null-character&gt;0&lt;/null-character&gt;   &lt;max-num-character&gt;64&lt;/max-num-character&gt;   &lt;c-type&gt;tsnc_uint8*&lt;/c-type&gt;   &lt;c++-type&gt;tsnc_uint8*&lt;/c++-type&gt; &lt;/string-field&gt; &lt;/field&gt; </pre>
<pre> msg : Type; </pre>	<pre> &lt;field&gt;   &lt;name&gt;msg&lt;/name&gt;   &lt;length&gt;variable&lt;/length&gt;   &lt;condition&gt;&gt;true&lt;/condition&gt;   &lt;message-field&gt;     &lt;message&gt;Type&lt;/message&gt;     &lt;c-type&gt;Type*&lt;/c-type&gt;     &lt;c++-type&gt;Type*&lt;/c++-type&gt;   &lt;/message-field&gt; &lt;/field&gt; </pre>
<pre> msgs : case Id of {   0 =&gt;     bits 8;   1 =&gt;     msg1 : Type1;   2 =&gt;     msg2 : Type2;   - =&gt;     msg3 : Type3; } </pre>	<pre> &lt;field&gt;   &lt;name&gt;msgs.bits&lt;/name&gt;   &lt;length&gt;8&lt;/length&gt;   &lt;condition&gt;Id == 0&lt;/condition&gt;   &lt;bit-field&gt;     &lt;signed&gt;&gt;false&lt;/signed&gt;     &lt;minimum-value&gt;0&lt;/minimum-value&gt;     &lt;maximum-value&gt;255&lt;/maximum-value&gt;     &lt;c-type&gt;tsnc_uint8&lt;/c-type&gt;     &lt;c++-type&gt;tsnc_uint8&lt;/c++-type&gt;   &lt;/bit-field&gt; &lt;/field&gt; &lt;field&gt;   &lt;name&gt;msgs.msg1&lt;/name&gt;   &lt;length&gt;variable&lt;/length&gt;   &lt;condition&gt;Id == 1&lt;/condition&gt;   &lt;message-field&gt;     &lt;message&gt;Type1&lt;/message&gt;     &lt;c-type&gt;Type1*&lt;/c-type&gt;     &lt;c++-type&gt;Type1*&lt;/c++-type&gt;   &lt;/message-field&gt; &lt;/field&gt; &lt;field&gt;   &lt;name&gt;msgs.msg2&lt;/name&gt;   &lt;length&gt;variable&lt;/length&gt;   &lt;condition&gt;Id == 2&lt;/condition&gt;   &lt;message-field&gt;     &lt;message&gt;Type2&lt;/message&gt;     &lt;c-type&gt;Type2*&lt;/c-type&gt;     &lt;c++-type&gt;Type2*&lt;/c++-type&gt;   &lt;/message-field&gt; &lt;/field&gt; &lt;field&gt;   &lt;name&gt;msgs.msg3&lt;/name&gt;   &lt;length&gt;variable&lt;/length&gt;   &lt;condition&gt;!(Id == 0    Id == 1    Id == 2)&lt;/condition&gt;   &lt;message-field&gt;     &lt;message&gt;Type3&lt;/message&gt;     &lt;c-type&gt;Type3*&lt;/c-type&gt;     &lt;c++-type&gt;Type3*&lt;/c++-type&gt;   &lt;/message-field&gt; &lt;/field&gt; </pre>
<pre> bits[6] 8; </pre>	<pre> &lt;field&gt; </pre>

	<pre> &lt;name&gt;bits&lt;/name&gt; &lt;array&gt;   &lt;size&gt;6&lt;/size&gt; &lt;/array&gt; &lt;length&gt;8&lt;/length&gt; &lt;condition&gt;&gt;true&lt;/condition&gt; &lt;bit-field&gt;   &lt;signed&gt;&gt;false&lt;/signed&gt;   &lt;minimum-value&gt;0&lt;/minimum-value&gt;   &lt;maximum-value&gt;255&lt;/maximum-value&gt;   &lt;c-type&gt;tsnc_uint8&lt;/c-type&gt;   &lt;c++-type&gt;tsnc_uint8&lt;/c++-type&gt; &lt;/bit-field&gt; &lt;/field&gt; </pre>
<pre> size      8; bits[size] 15; </pre>	<pre> &lt;field&gt;   &lt;name&gt;size&lt;/name&gt;   &lt;length&gt;8&lt;/length&gt;   &lt;condition&gt;&gt;true&lt;/condition&gt;   &lt;bit-field&gt;     &lt;signed&gt;&gt;false&lt;/signed&gt;     &lt;minimum-value&gt;0&lt;/minimum-value&gt;     &lt;maximum-value&gt;255&lt;/maximum-value&gt;     &lt;c-type&gt;tsnc_uint8&lt;/c-type&gt;     &lt;c++-type&gt;tsnc_uint8&lt;/c++-type&gt;   &lt;/bit-field&gt; &lt;/field&gt; &lt;field&gt;   &lt;name&gt;bits&lt;/name&gt;   &lt;array&gt;     &lt;size&gt;size&lt;/size&gt;   &lt;/array&gt;   &lt;length&gt;15&lt;/length&gt;   &lt;bit-field&gt;     &lt;signed&gt;&gt;false&lt;/signed&gt;     &lt;minimum-value&gt;0&lt;/minimum-value&gt;     &lt;maximum-value&gt;32767&lt;/maximum-value&gt;     &lt;c-type&gt;tsnc_uint16&lt;/c-type&gt;     &lt;c++-type&gt;tsnc_uint16&lt;/c++-type&gt;   &lt;/bit-field&gt; &lt;/field&gt; </pre>
<pre> msgs[] : Type; </pre>	<pre> &lt;field&gt;   &lt;name&gt;msgs&lt;/name&gt;   &lt;array&gt;     &lt;size&gt;variable&lt;/size&gt;   &lt;/array&gt;   &lt;length&gt;variable&lt;/length&gt;   &lt;condition&gt;&gt;true&lt;/condition&gt;   &lt;message-field&gt;     &lt;message&gt;Type&lt;/message&gt;     &lt;c-type&gt;Type*&lt;/c-type&gt;     &lt;c++-type&gt;Type*&lt;/c++-type&gt;   &lt;/message-field&gt; &lt;/field&gt; </pre>
<pre> reserve 7; </pre>	<pre> None </pre>
<pre> align(8); </pre>	<pre> None </pre>

## 10 Extensions

Extensions are XML [3] markups in a TSN.1 file. They are designed to provide directives or hints to the Compiler as well as other TSN.1 development tools. For example, the display extension (10.5) is used to indicate the format in which a bit field is to be displayed. Extensions are not part of the TSN.1 notation.

Each extension in the TSN.1 file is specified as an XML element. Because extensions are XML elements, the usual XML escape characters apply, for example, “&lt;” for “<”, “&gt;” for “>”, and “&amp;” for “&”. They are especially relevant to the code (10.1) and doc (10.6) extensions. If you have a large amount of code and you don’t want escape each character individually, you can put the code inside a CDATA section. A CDATA section starts with “<![CDATA[“ and ends with “]]>”.

The subsequent sections of this chapter describe the extensions supported by the Compiler.

### 10.1 Base

The base extension appears at the top level of a TSN.1 file. It is used to override the default base class of the generated C++ message class, which is “tsncxx\_Message”. Multiple base classes can be specified. One and only one of them must be either “tsncxx\_Message” or derived from “tsncxx\_Message”. The base extension has two attributes. The “language” attribute, which is mandatory, specifies the name of the target language. Currently, the only valid value for this attribute is “c++”. There is an optional “def” attribute. When present, it specifies the name of the message whose base class will be overridden. If the “def” attribute is not present, all messages defined after the base extension will have their base class overridden. Subsequent base extension in the same file overrides the then current base extension starting from the point where the subsequent extension is specified.

#### 10.1.1 Example

```
is856_admp_UATIRrequest() ::=
{
  TransactionID 8;
}

<base language="c++">tsncxx_Message, MyMessageBase</base>
```

### 10.2 Bit Field

The bit field extension is intended for Wireshark. It groups together multiple consecutive bit fields and displays them using the bit decoded format in Wireshark. The extension has no attribute. The valid values for the extensions are 8, 16, and 32, which specify the width of the bit field.

## 10.2.1 Example

```

Foo() ::=
{
  A  3; <bit_field>32<bit_field>
  B  9;
  C 17;
  D  3;
}

```

The Wireshark dissected output looks like:

```

XXX. .... = A
...X XXXX XXXX .... = B
.... .... XXXX XXXX XXXX XXXX X... = C
.... .... .... .XXX = D

```

where “X” is either 0 or 1 depending on the value of the field.

## 10.3 Byte Alignment

The byte alignment extension allows user to override the default byte alignment of a message, perhaps to resolve a “miss aligned” compile error. The extension has a mandatory attribute “def”, which specifies the name of the message the alignment applies to. The value must be an integer between -1 and 7, which is the bit offset from the byte boundary with “-1” indicating the message may start from arbitrary bit offset.

### 10.3.1 Example

```

Bar() ::=
{
  A  7;
}

Foo() ::=
{
  A  3;
  bar : Bar;
}

<byte_alignment def="Bar">3</byte_alignment>

```

## 10.4 Code

The code extension appears at the top level of a TSN.1 file. It is used to insert arbitrary code into the generated files. The extension has three attributes: “language”, “file”, and “def”. The “language” attribute indicates the target language of which the code is intended for. The valid values are “c”, “c++”, “wireshark”, or any combination of the three, for example, “c/c++” or “c/c++/wireshark”. The “file” attribute specifies the type of the file into which the code is to be inserted. The valid values are “header”, “source”, and “header/source”. The “def” attribute is optional. When present, it is used to specify

the name of the message for which the code is intended. If the attribute is omitted, then the code is inserted at the top level.

### 10.4.1 Example

```
is856_admp_UATIRquest() ::=
{
    TransactionID 8;
}

<code language="c++" file="header" def="is856_admp_UATIRquest">
public:
    tsnc_uint8 getTransactionID()
    {
        return TransactionID;
    }
</code>
```

The above code extension inserts the inline function `getTransactionID` into the generated C++ header file inside the class definition of message `is856_admp_UATIRquest`.

## 10.5 Display

The display extension is used to specify the format in which bit fields are to be displayed. The extension can appear either at the top level or right after a bit field declaration. The valid values are: “oct” for octal, “dec” for decimal, “hex” for hexadecimal, “ipv4” for IPv4 address, “ipv6” for IPv6 address, “ascii” for ASCII strings, and “none” to disable the display of the field. The default value is “dec”. There is an optional “def” attribute. When present, all fields in the message will be displayed in the specified format. If the “def” attribute is not present, all messages defined after the display extension will use the specified display. Subsequent display extension in the same file overrides the then current display extension starting from the point where the subsequent display extension is specified.

The display extension has an optional “value\_offset” attribute. The value of this attribute is an integer. In certain type of encoding, for example ASN.1 PER [6], an integer value is represented at an offset to its two’s complement representation. In these cases, the “value\_offset” attribute can be used to specify the offset value to be added to the decoded value before display.

### 10.5.1 Example

```

Foo() ::=
{
    TransactionID  8; <display>hex</display>

    NameLength  8;
    Name        8 * NameLength; <display>ascii</display>

    IPv4Address  32; <display>ipv4</display>
    IPv6Address  128; <display>ipv6</display>

    Address  8  string; <display>ascii</display>
}

Foo2() ::=
{
    TransactionID  8;

    // The range of values is shifted by -5, from
    // [0 - 255] to [-5, 250]
    Value  8; <display value_offset="-5"/>
}

<display def="Foo2">hex</display>

```

## 10.6 Dissect

The dissect extension allows users to specify a custom dissect function for a bit field, a string field, or a message. When the extension appears right after a bit field or a string field, the custom dissection applies only to that instance of the bit field or string field. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by concatenating the message name and the field name with an underscore character in between. The string is then appended with “\_dissect”. The function must conform to the following signature:

```

typedef void DissectFieldType
(
    Foo          *msg,          /* Parent message */
    const char  *field_name,   /* Field name */
    int         field_index,   /* Field index */
    void        *field_value,  /* Pointer to field value */
    tvbuff_t    *tvb,         /* Data buffer */
    guint32     offset,       /* Bit offset of the field */
    gint32      nbit,         /* Number of bit of the field */
    proto_tree  *tree         /* Proto tree */
);

```

When the dissect extension appears at the top level, the “def” attribute must be present to specify the name of the message for which the generated unpack function is to be replaced by a user defined one. This can be handy when the message has complex encodings. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by appending “\_” to the message name. Your function must conform to the following signature:

```
typedef guint32 DissectFuncType
(
    Foo          *msg,          /* The message */
    const char   *field_name,  /* Field name */
    int          field_index,  /* Field index */
    int          proto_index,  /* Protocol index */
    tvbuff_t     *tvb,         /* Data buffer */
    guint32      offset,       /* Bit offset of the field */
    guint32      length,       /* Number of remaining bits */
    packet_info  *pinfo,       /* Packet info */
    proto_tree   *tree         /* Proto tree */
);
```

You can also use dissect extension to add “hooks” using the optional attribute “when.” The valid values are “before” and “after”, which determines whether the supplied function will be called before or after the dissect function is called. The supplied function must conform to the following signature:

```

typedef void DissectHookFuncType
(
    Foo          *msg,          /* The message */
    const char   *field_name,   /* Field name */
    int          field_index,   /* Field index */
    int          proto_index,   /* Protocol index */
    tvbuff_t     *tvb,         /* Data buffer */
    guint32      offset,       /* Bit offset of the field */
    guint32      length,       /* Number of remaining bits */
    packet_info  *pinfo,       /* Packet info */
    proto_tree   *tree         /* Proto tree */
);

```

### 10.6.1 Example

```

Foo() ::=
{
    TransmitPower 16; <dissect/>
}

<dissect def="Foo" when="before"/>

// You can use the code extension to define the custom functions
// inside the TSN.1 file.
<code language="wireshark" file="source">
<![CDATA[
void Foo_predissect
(
    Foo          *msg,
    const char   *field_name,
    int          field_index,
    int          proto_index,
    tvbuff_t     *tvb,
    guint32      offset,
    guint32      length,
    packet_info  *pinfo,
    proto_tree   *tree
)
{
    /* Perform custom predissect action */
}

void Foo_TransmitPower_dissect
(
    Foo          *msg,
    const char   *field_name,
    int          field_index,
    void         *field_value,
    tvbuff_t     *tvb,
    guint32      offset,
    guint32      nbit,
    proto_tree   *tree
)
{
    guint16      TransmitPower;

```

```

proto_item *ti;

TransmitPower = *((guint16 *) field_value);

// Perform the custom dissect
ti = proto_tree_add_uint
(
    tree,
    field_index,
    tvb,
    offset >> 3, /* start */
    ((offset & 0x7) + nbit + 7) >> 3, /* length */
    TransmitPower
);

    proto_item_append_text(ti, " dB");
}
]]>
</code>

```

## 10.7Doc

The doc extension is used to document a TSN.1 file. The Compiler uses the information in the doc extension to generate Doxygen style comments in the output files. The doc extension has an optional attribute “def”. The attribute value specifies the name of the constant, enumeration, or message with which the documentation is associated. If the “def” attribute is omitted, this association can also be established through the normal proximity rule: the documentation for a definition shall appear right before the definition.

As a convenience, the Doxygen style comments “/\*!”, “/\*!<” and “\*/” can be used directly in a TSN.1 file instead of using the “<doc>” and “</doc>” tags. Note that these special comments are merely shorthand of the XML tags. Therefore, they cannot appear anywhere in a TSN.1 file like regular comments. They should only appear where the doc extension is expected.

## 10.7.1 Example

```

/*!
 * @file example.tsn
 * @brief Brief description.
 *
 * Detailed description.
 */

/*!
 * @brief Brief description of constant C
 *
 * Detailed description of constant C.
 */
C ::= 3;

/*!
 * @brief Brief description of enumerated E
 *
 * Detailed description of enumerated E.
 */
E ::= enumerated
{
    Literal1, /*!< First literal. */
    Literal2 /*!< Second literal. */
}

/*!
 * @brief Brief descripton of message N
 *
 * Detailed description of message N.
 */
N() ::=
{
}

/*!
 * @brief Brief descripton of message M
 *
 * Detailed description of message M.
 */
M
(
    Arg1 3, /*!< First argument. */
    Arg2 8 /*!< Second argument. */
) ::=
{
    Field1 3; /*!< First field. */
    Field2 8; /*!< Second field. */

    MsgField : N; /*!< Nested message. */

    InlineMsgField :
    {
        Field1 3; /*!< First field. */
    }
}

```

```

    } /*!< Inline-nested message. */

    CaseField : case Field1 of
    {
        0 =>
            Case0Field  3; /*!< Case 0 field. */

        1 =>
            Case1Field  8; /*!< Case 1 field. */
    } /*!< Case field. */
}

```

## 10.8 Endianness

TSN.1 specifies messages be encoded in the big endian format. You can override this default by using the endianness extension. The endianness extension can appear either at the top level or right after a bit field declaration. The valid values for the extension are “big” and “little”. When the extension appears at the top level of a TSN.1 file, the attribute “def” must be specified. When the attribute value is a valid message name, the specified endianness applies to the entire message.

### 10.8.1 Example

```

Foo() ::=
{
    Value  16; <endianness>little</endianness>
}

Foo2() ::=
{
    Value  32;
}

<endianness def="Foo2">little</endianness>

```

## 10.9 Finalize

The finalize extension allows user to specify custom finalize actions for a message. The required attribute “when” specifies whether the supplied function will be called before or after the message is finalized. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by appending “\_prefinalize” (before) or “\_postfinalize” (after) to the message name. Your function must conform to the following signature:

```

typedef void UserFinalizeFuncType
(
    tsnc_Message *msg
);

```

### 10.9.1 Example

```

Foo() ::=
{
    ...
}

<finalize def="Foo" when="before">my_foo_PreFinalize</finalize>
<finalize def="Foo" when="after"/>

// You can use the code extension to define the functions
// inside the TSN.1 file.
<code language="c/c++" file="source">
<![CDATA[
void my_foo_PreFinalize
(
    tsnc_Message *tsnc_msg,
)
{
    Foo *msg = (Foo *) tsnc_msg;

    /* Your custom finalize code here */
}

void Foo_postfinalize
(
    tsnc_Message *tsnc_msg
)
{
    Foo *msg = (Foo *) tsnc_msg;

    /* Your custom finalize code here */
}
]]>
</code>

```

### 10.10 Initialize

The initialize extension allows user to specify custom initialize actions for a message. The required attribute “when” specifies whether the supplied function will be called before or after the message is initialized. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by appending “\_preinitialize” (before) or “\_postinitialize” (after) to the message name. Your function must conform to the following signature:

```

typedef void UserInitializeFuncType
(
    tsnc_Message *msg
);

```

### 10.10.1 Example

```

Foo() ::=
{
    ...
}

<initialize def="Foo" when="after"/>

// You can use the code extension to define the functions
// inside the TSN.1 file.
<code language="c/c++" file="source">
<![CDATA[
void Foo_postinitialize
(
    tsnc_Message *tsnc_msg
)
{
    Foo *msg = (Foo *) tsnc_msg;

    /* Your custom initialize code here */
}
]]>
</code>

```

## 10.11 Literal

The literal extension is used to associate an arbitrary string to the enum literal for display. It overrides the default string, which is the enum literal itself.

### 10.11.1 Example

```

is856_admp_AirMessageId ::= enumerated
{
    IS856_ADMP_UATI_REQUEST,    <literal>UATI Request</literal>
    IS856_ADMP_UATI_ASSIGNMENT, <literal>UATI Assignment</literal>
    IS856_ADMP_UATI_COMPLETE,  <literal>UATI Complete</literal>
    ...
}

```

## 10.12 Options

The options extension is used to specify compile options inside a TSN.1 file. In case of a conflict, the options in the file overrides the ones specified on the command-line. The options apply to the entire file regardless of where the extension appears and they only apply to the current file. The options do not apply to any imported files, nor does any option in the imported file apply to the importing file. The valid options for this extension are “-int64”, and “-little”.

### 10.12.1 Example

```
<options>-int64</options>
```

## 10.13 Pack

The pack extension allows user to replace the generated pack function with a user defined one. This can be handy when the message has complex encodings. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by appending “\_pack” to the message name. Your function must conform to the following signature:

```
typedef tsnc_Status UserPackFuncType
(
    const tsnc_Message *tsnc_msg,    /* Parent message */
    tsnc_uint8         *buf,         /* Buffer to pack into */
    tsnc_uint32        offset,       /* Start bit offset */
    tsnc_uint32        length,       /* Buffer size in bits */
    void               *nbit_packed /* Number of bit packed */
);
```

You can also use pack extension to add “hooks” using the optional attribute “when.” The valid values are “before” and “after”, which determines whether the supplied function will be called before or after the pack function. The supplied function must conform to the following signature:

```
typedef tsnc_Status UserPackFuncType
(
    tsnc_Message *tsnc_msg,    /* Parent message */
    tsnc_uint8   *buf,        /* Buffer to pack into */
    tsnc_uint32  offset,      /* Start bit offset */
    tsnc_uint32  length,      /* Buffer size in bits */
    void         *nbit_packed /* Number of bit packed */
);
```

### 10.13.1 Example

```
Foo() ::=
{
    ...
}

<pack def="Foo">my_foo_pack</pack>
<pack def="Foo" when="before"/>

// You can use the code extension to define the functions
// inside the TSN.1 file.
<code language="c/c++" file="source">
<![CDATA[
tsnc_Status Foo_prepack
(
    tsnc_Message *tsnc_msg,
    tsnc_uint8   *buf,
    tsnc_uint32  offset,
    tsnc_uint32  length,
    tsnc_uint32  *nbit_packed
)
{
    Foo *msg = (Foo *) tsnc_msg;

    /* Your custom pack code here */

    return TSNC_STATUS_OK;
}

tsnc_Status my_foo_pack
(
    const tsnc_Message *tsnc_msg,
    tsnc_uint8         *buf,
    tsnc_uint32        offset,
    tsnc_uint32        length,
    void               *nbit_packed
)
{
    Foo *msg = (Foo *) tsnc_msg;

    /* Your custom pack code here */

    if(nbit_packed != NULL)
    {
        *nbit_packed = /* number of bit packed */;
    }
}
]!]>
```

```

        return TSNC_STATUS_OK;
    }
}
</code>

```

## 10.14 Print

The print extension allows users to specify a custom print function for a bit field or a string field. The extension has no attribute. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by concatenating the message name and the field name with an underscore character in between. The string is then appended with “\_print”. The function must conform to the following signature:

```

typedef tsnc_int32 PrintFieldFuncType
(
    tsnc_Message *tsnc_msg, /* Parent message */
    void *value, /* Pointer to field value */
    char *text, /* String buffer to print into */
    tsnc_uint32 length, /* Size of text */
    void *user_data /* Reserved */
);

```

The function shall return the number of characters printed on success and “-1” on failure.

### 10.14.1 Example

```

Foo() ::=
{
    TransmitPower 16; <print/>
}

// You can use the code extension to define the custom functions
// inside the TSN.1 file.
<code language="c/c++" file="source">
<![CDATA[
tsnc_int32 Foo_TransmitPower_print
(
    tsnc_Message *msg,
    void         *value,
    char         *text,
    tsnc_uint32  length,
    void         *user_data
)
{
    tsnc_uint16 TransmitPower;

    // Make sure we have enough space to print
    if(length < 12) return -1;

    TransmitPower = *((tsnc_uint16 *) value);

    // Perform the custom print
    return sprintf(text, "%d dB", TransmitPower - 35);
}
]]>
</code>

```

### 10.15 Scan

When you use custom print for a bit field or a string field, you will also need a custom scan function if you want to parse the string back using the scan function. Like the print extension, the scan extension has no attribute. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by concatenating the message name and the field name with an underscore character in between. The string is then appended with “\_scan”. The function must conform to the following signature:

```

typedef tsnc_int32 ScanFieldFuncType
(
    tsnc_Message *tsnc_msg, /* Parent message */
    void         *value,    /* Pointer to field value */
    char         *text,     /* String to scan from */
    void         *user_data /* Reserved */
);

```

The function shall return the number of characters scanned on success and “-1” on failure.

### 10.15.1 Example

```

Foo() ::=
{
    TransmitPower 16; <scan/>
}

// You can use the code extension to define the custom functions
// inside the TSN.1 file.
<code language="c/c++" file="source">
<![CDATA[
tsnc_int32 Foo_TransmitPower_scan
(
    tsnc_Message *tsnc_msg,
    void         *value,
    const char   *text,
    void         *user_data
)
{
    tsnc_int32 TransmitPower;

    // Perform the custom scan.
    if(!sscanf(text, " %d dB", &TransmitPower)) return -1;

    // Convert the value back
    *((tsnc_uint16 *) value) = (tsnc_uint16) (TransmitPower + 35);

    // Skip the scanned characters
    return (tsnc_int32) (strstr(text, "dB") + 2 - text);
}
]]>
</code>

```

## 10.16 Storage

The storage extension is used to specify the storage type of a field. The extension can appear either at the top level or right after a field declaration. The valid values are: “dynamic” for dynamic storage and “static” for automatic storage. By default, bit buffers and nested messages are generated as pointers and use dynamic storage (or automatic storage if the “-static” option is specified). This default can be overridden by using the storage extension. When the extension appears at the top level of a TSN.1 file, the attribute “def” must be specified. When the attribute value is a valid message name, the storage type applies to the entire message.

### 10.16.1 Example

```

Foo() ::=
{
    Length 8;
    Buffer 8 * Length; <storage>static</storage>
    Message :
    {
        ...
    } <storage>static</storage>
}

Foo2() ::=
{
    Length 8;
    Buffer 8 * Length;
    Message :
    {
        ...
    }
}

<storage def="Foo2">static</storage>

```

## 10.17 Type

The type extension directs the Compiler to override the default data type generated for a field. The value can be “float”, “double”, or a user defined custom container. If the value is “float”, the field must be a 32-bit field. If the value is “double”, the field must be a 64-bit field. If the value is a custom container, the field must be an array of nested messages. When custom container is used, there is an optional attribute “language”, which specifies the target language to which the type extension applies. The valid values for this attribute are “c” and “c++”. If the “language” attribute is omitted, it applies to both “c” and “c++”.

If the language attribute is “java”, then the field must be an integer bit field. In this case, a required attribute “object”, whose value must be either “true” or “false”, is used to indicate whether the object version of the primitive integer types shall be used.

If a user defined container type is specified, the container type must implement the following interface functions:

## TSN.1 Compiler User's Manual (For Version 5.4)

```
/* Create a new container */
MyContainer* MyContainer_new(tsnc_uint32 maximum_size);

/* Delete the container and all remaining items */
/* Make sure container == NULL is allowed */
void MyContainer_delete(MyContainer *container);

/* Return an iterator of the container */
MyContainerIterator* MyContainer_iterator(MyContainer *container);

/* Append an item to the container */
void MyContainer_append(MyContainer *container, void *item);

/* Delete the iterator */
/* Make sure iterator == NULL is allowed */
void MyContainerIterator_delete(MyContainerIterator *iterator);

/* Test if the iterator is pointing at the final element */
int MyContainerIterator_has_more(const MyContainerIterator *iterator);

/* Return the next item in the container */
void *MyContainerIterator_get_next(MyContainerIterator *iterator);
```

The example assumes that the custom container type is “MyContainer”. In case the target code is C++, there is also an optional attribute “template”. If set to “true”, the container interface functions uses the C++ template:

```

template <class T>
MyContainer<T>* MyContainer_new(tsnc_uint32 maximum_size)
{
    // Create a new container
}

template <class T>
void MyContainer_delete(MyContainer<T> *container)
{
    // Delete the container and all remaining items
    // Make sure container == NULL is allowed
}

template <class T>
MyContainerIterator<T>* MyContainer_iterator(MyContainer<T> *container)
{
    // Return an iterator of the container
}

template <class T>
void MyContainer_append(MyContainer<T> *container, T item)
{
    // Append an item to the container
}

template <class T>
void MyContainerIterator_delete(MyContainerIterator<T> *iterator)
{
    // Delete the iterator
    // Make sure iterator == NULL is allowed
}

template <class T>
int MyContainerIterator_has_more(const MyContainerIterator<T> *iterator)
{
    // Test if the iterator is pointing at the final element
}

template <class T>
T MyContainerIterator_get_next(MyContainerIterator<T> *iterator)
{
    // Return the next item
}

```

When custom container types are used, make sure you use the “code” extension (section 10.2) to include the header file in which these interface functions are defined. The generated code will invoke these functions to use the custom container.

In the generated message structures, the custom containers are generated as pointers. They are initialized to NULL. You should populate them like any other fields before calling the pack and print functions. For unpack and scan functions, the containers are created automatically when needed. They are deleted by the finalize function.

### 10.17.1 Example

```

Float() ::=
{
    Percentage 32; <type>float</type>
}

Container() ::=
{
    Size 8;
    Elements[Size] : FloatExample; <type>MyContainer</type>

    Elements2[] :
    {
        Value 32;
    } <type template="true">MyContainer2</type>
}

```

## 10.18 Unpack

The unpack extension allows user to replace the generated unpack function with a user defined one. This can be handy when the message has complex encodings. The value of the extension specifies the name of the function to be used. If the value is empty, then a default name is supplied. The default name is formed by appending “\_unpack” to the message name. Your function must conform to the following signature:

```

typedef tsnc_Status UserUnpackFuncType
(
    tsnc_Message      *tsnc_msg,      /* Parent message */
    const tsnc_uint8 *buf,           /* Buffer to unpack from */
    tsnc_uint32       offset,        /* Start bit offset */
    tsnc_uint32       length,        /* Buffer size in bits */
    void              *nbit_packed  /* Number of bit unpacked */
);

```

You can also use unpack extension to add “hooks” using the optional attribute “when.” The valid values are “before” and “after”, which determines whether the supplied function will be called before or after the unpack function is called. The supplied function must conform to the following signature:

```

typedef tsnc_Status UserUnpackFuncType
(
    tsnc_Message      *tsnc_msg,      /* Parent message */
    const tsnc_uint8  *buf,          /* Buffer to unpack from */
    tsnc_uint32       offset,        /* Start bit offset */
    tsnc_uint32       length,        /* Buffer size in bits */
    void              *nbit_packed  /* Number of bit unpacked */
);

```

### 10.18.1 Example

```

Foo() ::=
{
    ...
}

<unpack def="Foo">my_foo_unpack</unpack>
<unpack def="Foo" when="after">Foo_MyPostUnpack</unpack>

// You can use the code extension to define the custom functions
// inside the TSN.1 file.
<code language="c/c++" file="source">
<![CDATA[
tsnc_Status my_foo_unpack
(
    tsnc_Message      *tsnc_msg,
    const tsnc_uint8  *buf,
    tsnc_uint32       offset,
    tsnc_uint32       length,
    void              *nbit_unpacked
)
{
    Foo *msg = (Foo *) tsnc_msg;

    /* Your custom unpack code here */

    if(nbit_unpacked != NULL)
    {
        *nbit_unpacked = /* number of bit unpacked */;
    }

    return TSNC_STATUS_OK;
}

tsnc_Status Foo_MyPostUnpack
(
    tsnc_Message *msg,
    tsnc_uint8   *buf,
    tsnc_uint32  offset,
    tsnc_uint32  length,
    tsnc_uint32  *nbit_packed
)
{
    Foo *msg = (Foo *) tsnc_msg;

```

## TSN.1 Compiler User's Manual (For Version 5.4)

```
        /* Your custom unpack code here */  
        return TSNC_STATUS_OK;  
    }  
    11>  
</code>
```

## References

- [1] The Transfer Syntax Notation One Specification
- [2] ISO/IEC 9899:1999 Programming Languages – C
- [3] Extensible Markup Language (XML) 1.0, W3C Recommendation
- [4] Wireshark: <http://www.wireshark.org/>
- [5] The TSN.1 Server User's Manual
- [6] ASN.1 encoding rules: Specification of Packed Encoding Rules (PER), ITU-T Recommendation X.691, International Standard 8825-2