



# **The Transfer Syntax Notation One Specification**

## Table of Contents:

1	Introduction .....	4
1.1	Scope.....	4
1.2	References.....	4
1.3	Abbreviations.....	5
2	Package, Import and Conditional .....	6
2.1	Package .....	6
2.2	Import.....	6
2.3	Conditional.....	7
2.4	Names .....	7
2.4.1	Simple Name .....	7
2.4.2	Qualified Name.....	7
3	Constant and Enumeration.....	8
3.1	Integer .....	8
3.2	String.....	8
3.3	Enumeration.....	8
4	Message .....	10
4.1	Bit Field .....	10
4.1.1	Interval.....	10
4.1.2	Enumeration.....	11
4.1.3	Default Value.....	11
4.1.4	Reserved Field .....	12
4.1.5	Alignment .....	12
4.2	String Field.....	12
4.3	Nested Message .....	12
4.3.1	Inline Defined.....	13
4.3.2	Scoping .....	14
4.3.3	Argument.....	14
4.3.4	Recursion.....	15
4.4	Array .....	16
4.4.1	Unbounded Array .....	17
4.4.2	Until.....	18
4.5	Case Of.....	18
4.6	Conditional.....	19
4.7	Block with Size Constraint .....	20
4.8	Variable, Assignment and While Statement .....	20
4.9	Alias .....	21
4.10	Error .....	21
5	Macro.....	23
6	Expressions.....	24
6.1	Boolean Expressions.....	24
6.1.1	Unary Boolean Expressions.....	24
6.1.2	Binary Boolean Expression .....	24
6.2	Integer Expressions.....	24
6.2.1	Unary Integer Expressions.....	24

# The Transfer Syntax Notation One Specification

6.2.2	Binary Integer Expressions.....	25
6.3	Operator Precedence and Associativity .....	25
6.4	Semantics .....	26
7	Lexical Tokens .....	27
7.1	Character Set.....	27
7.2	Line Terminators.....	27
7.3	Comments .....	27
7.3.1	Single Line Comments .....	27
7.3.2	Multi-line Comments.....	27
7.4	White Spaces.....	27
7.5	Identifiers .....	27
7.6	Keywords .....	27
7.7	Literals .....	28
7.7.1	Boolean Literals.....	28
7.7.2	Integer Literals.....	28
7.7.3	String Literals .....	29
7.8	Separators.....	29
7.9	Operators.....	29
A.	BNF Grammar .....	30

## List of Tables:

Table 6-1	Operator Precedence and Associativity.....	25
Table 6-2	Argument/Variable/Bit Field Type Assignment .....	26
Table 7-1	Integer Types and Values .....	28
Table 7-2	Integer Literal Type Assignment.....	28

# 1 Introduction

Transfer Syntax Notation One (TSN.1) is a formal notation for the definition of data types. A data type is a class of information, for example, a message in a communication protocol.

TSN.1 is different from Abstract Syntax Notation One (ASN.1) [1] in the following way: TSN.1 defines the data directly in terms of its binary representation (transfer syntax), thus capturing both the information bits and the encoding bits of a message in a single notation.

In ASN.1, a message is first defined using abstract types such as Integer and Boolean, or type constructs such as Sequence or Set. The base notation does not specify how the message is encoded. To map the abstract definition into concrete bits (transfer syntax), user chooses from one of the standard ASN.1 encoding rules, for example, the Basic Encoding Rule (BER) [2]. Each ASN.1 encoding rule translates the ASN.1 types and type constructs in a standard and uniform way, giving user no control over the final encoding of the data.

ASN.1 does not work for legacy protocols for which the users must retain their existing binary representations. ASN.1 also does not work if users wish to have precise control over the binary representation of their messages, perhaps to save bits in a bandwidth constrained systems such as wireless networks or for some internal data that require proprietary encoding. Although the Encoding Control Notation (ECN) [3] has been added to the ASN.1 standard to alleviate some of these problems, it remains highly impractical to use ASN.1 for this type of data. In some instances, the complexity of applying ECN defeats the purpose of using a formal notation. In other instances, ECN is simply not expressive enough to handle the encoding requirements.

TSN.1 is designed specifically to describe messages that require flexible and custom encodings. It allows users to describe both the information bits and the encoding bits of a message in one notation. Because a message is defined directly in binary, no separate encoding rules is needed.

In terms of expressiveness, it is important to note that all ASN.1 data types can be described in TSN.1 after applying one of the ASN.1 encoding rules. However, it is not always possible to find an equivalent representation in ASN.1 for a TSN.1 data type.

## 1.1 Scope

This document specifies the syntax and semantics of TSN.1. The TSN.1 notation can be applied whenever it is necessary to define the transfer syntax of information.

## 1.2 References

- [1] Abstract Syntax Notation One (ASN.1), Specification of Base Notation, ITU-T Rec. X.680, ISO/IEC 8825-1

## The Transfer Syntax Notation One Specification

- [2] ASN.1 encoding rules: Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER), ITU-T Rec. X.691, ISO/IEC 8825-1
- [3] ASN.1 encoding rules: Specification of Encoding Control Notation, ITU-T Rec. X.692, ISO/IEC 8825-3
- [4] CDMA2000 High Rate Packet Data Air Interface Specification, 3GPP2 C.S0024, TIA IS-856
- [5] IEEE Std 802.16-2004, Air Interface for Fixed Broadband Wireless Access Systems
- [6] ISO/IEC 9899:1999 Programming Languages - C

### 1.3 Abbreviations

<b>Acronym</b>	<b>Definition</b>
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rule
BNF	Backus Naur Form
CR	Carriage Return
ECN	Encoding Control Notation
LF	Line Feed
TSN.1	Transfer Syntax Notation One

## 2 Package, Import and Conditional

TSN.1 source files are organized into packages. Each package has its own set of TSN.1 files. The naming structure for package is hierarchical.

### 2.1 Package

Each TSN.1 file may begin with a package declaration. For example, the “mac\_messages.tsn” file may declare its package as follows:

```
package wimax;
```

This means the file “mac\_messages.tsn” and all the definitions in the file belongs to a package called “wimax”. If no package is specified, then all definitions go into the global default package.

Packages may also be organized in a hierarchical fashion. For example, the package “wimax” may belong to the package “ieee802”. In this case, the package shall be declared as:

```
package ieee802.wimax;
```

TSN.1 files stored in a file system have constraints on their organization to allow easy reference by TSN.1 files in another package. For the above package declaration, “mac\_messages.tsn” shall be placed in the file system under the directory: <tsn\_source\_root>/ieee802/wimax/, where <tsn\_source\_root> is a valid directory in the file system. The directory structure shall preserve the hierarchical structure of the package name.

### 2.2 Import

An import declaration allows one TSN.1 file to access definitions from another TSN.1 file. For example, the following import declaration will import all definitions defined in “mac\_messages.tsn” into the current file:

```
import ieee802.wimax.mac_messages;
```

Note that the import declaration specifies the complete package name of the file to be imported as well as the name of the file without the “.tsn” extension. Import declarations also allow the import of all files under a package using a wildcard:

```
import ieee802.wimax.*;
```

In this example, all TSN.1 files in the package ieee802.wimax are imported into the current file. This, however, does not recursively include any sub-package if there is any.

## 2.3 Conditional

Conditionals at the top level of a TSN.1 file provide the mechanism to conditionally define TSN.1 constants, enumerations, and messages. Conditionals are specified using the *if* keyword with an optional *else*, and they may be nested.

```
if (Mode == MOBILE_STATION)
{
    ...
}
else
{
    ...
}
```

In the above example, “Mode” is a constant whose value can be supplied at compile time, for example, through the command-line of a compiler.

## 2.4 Names

All symbols such as constants, enumerations, macros, and messages are defined at the top level. The symbol names are associated with the package in which they are defined. If no package is specified, the names go to the default global package. Names shall be unique within a package but not necessarily across packages. To refer to a symbol, either a simple name or a qualified name shall be used.

### 2.4.1 Simple Name

The simple name consists of a single identifier, the name shall be defined in the current package or if not present in the current package in one of the imported packages.

### 2.4.2 Qualified Name

If the simple name is defined in more than one imported packages, then it is ambiguous and a qualified name shall be used in its place. A qualified name consists of a package name followed by “.” and ends with an identifier, for example, the “DLMAP” message in the “ieee802.wimax” package shall be referred as “ieee802.wimax.DLMAP”.

## 3 Constant and Enumeration

There are two ways to define integer constants in TSN.1: using a constant definition or using an enumeration.

### 3.1 Integer

An integer definition associates an identifier with an integer value.

```
ONE ::= 1;
```

Subsequent reference to “ONE” will be replaced with the integer value 1. The associated integer value does not need to be an integer literal. It may be any constant expression, which may refer to constant previously defined. For example:

```
TWO ::= ONE + ONE;
```

### 3.2 String

A string definition associates an identifier with a string literal. Strings have no use in describing messages. They are included for external considerations.

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

### 3.3 Enumeration

Integer constants may also be defined using an enumeration. The following definition enumerates the IEEE 802.16 [5] MAC Management Message types.

```
wimax_mac_MessageType ::= enumerated
{
    WIMAX_MAC_UCD,
    WIMAX_MAC_DCD,
    WIMAX_MAC_DL_MAP,
    WIMAX_MAC_UL_MAP,
    WIMAX_MAC_RNG_REQ,
    WIMAX_MAC_RNG_RSP,
    WIMAX_MAC_REG_REQ,
    WIMAX_MAC_REG_RSP,

    WIMAX_MAC_PKM_REQ (9),
    WIMAX_MAC_PKM_RSP,

    /* Rest of the message types ... */
}
```

An enumerated literal may be followed by an explicit value specification (an integer constant expression inside a pair of parenthesis), which defines the value for the literal. If the first literal has no explicit value, its value is 0. Each subsequent literal without explicit value has a value one greater than the previous literal. The use of literals with explicit values may produce enumeration literals with the same values or to have gaps in

## The Transfer Syntax Notation One Specification

them. For literals with explicit values, the type of the literal is the type of the constant expression. For literals without explicit values, the type of the literal is the smallest type in which its value can be represented in the following order: signed 32-bit integer, unsigned 32-bit integer, signed 64-bit integer, and unsigned 64-bit integer with signed 32-bit integer as the smallest type. There shall be at least one literal in an enumeration.

## 4 Message

A message definition shall start with an identifier. It is followed by a list of formal arguments(4.3.3), the definition operator “::=”, and the body of the message. The body contains a sequence of field declarations. In general, each field declaration starts with an identifier followed by the size of the field (in unit of bits). For example, the following definition of the IEEE802.16 Uplink Channel Descriptor (UCD) message declares a sequence of 8-bit fields.

```
wimax_mac_UCD() ::=
{
    ConfigurationChangeCount    8;
    RangingBackoffStart          8;
    RangingBackoffEnd            8;
    RequestBackoffStart          8;
    RequestBackoffEnd            8;

    /* Rest of the UCD ... */
}
```

The field ordering and bit ordering are defined by the following rules:

- The order of fields in the bit sequence follows the same order as they appear in the message definition from top to bottom.
- The order of bits within a field is such that the most significant bit appears first in the bit sequence, followed by the second most significant bit, and so on.

### 4.1 Bit Field

The size of a field does not need to be an integer constant. It may be any valid integer expression. The expression may make reference to previously defined bit field in which case the bit field is used as an unsigned integer using two’s complement representation. For example:

```
Length    8;
Info      8 * Length;
```

In this definition, the size of “Info” is the equal to the value of the field “Length” multiplied by 8. For example, if “Length” has a value of 3 in a message, then “Info” will have 24 bits.

A bit field may also represent a signed integer. To mark a bit field as signed, use the *signed* keyword after the size.

```
Temperature 16 signed;
```

#### 4.1.1 Interval

By default, the value range for a bit field is determined by the size of that field. An unsigned bit field with  $n$  bits has values between 0 and  $2^n-1$  inclusive, whereas a signed bit field has values between  $-2^{n-1}$  and  $2^{n-1} - 1$ . For example, the range for “Length” in the

## The Transfer Syntax Notation One Specification

previous declaration is between 0 and 255. To limit this range to something other than the default, an interval may be specified following the field declaration:

```
Length 8 (0 .. 15);
Info   8 * Length;
```

### 4.1.2 Enumeration

If the values of a bit field belong to an enumeration, you may specify the name of the enumeration in the bit field declaration.

```
MessageType 8 enumerated wimax_mac_MessageType;
```

You may also define the enumeration inline.

```
MessageType 8 enumerated wimax_mac_MessageType
{
    WIMAX_MAC_UCD,
    WIMAX_MAC_DCD,
    WIMAX_MAC_DL_MAP,

    /* Rest of the message types ... */
};
```

You may omit the name for the enumeration, in which case the name of the enumeration will be the same as the name of the bit field.

```
MessageType 8 enumerated
{
    WIMAX_MAC_UCD,
    WIMAX_MAC_DCD,
    WIMAX_MAC_DL_MAP,

    /* Rest of the message types ... */
};
```

In the example above, the anonymous enumeration is named “MessageType”, you may reuse the enumeration for subsequent fields, but the name is only visible within the current message. If the name shadows a definition in a parent message, then the shadowed definition shall no longer be accessed in the current message and its nested messages.

### 4.1.3 Default Value

Default values may be specified for bit fields:

```
MessageType 8 = WIMAX_MAC_UCD;
```

In this example, “MessageType” has a default value equal to that of the constant “WIMAX\_MAC\_UCD”.

#### 4.1.4 Reserved Field

Reserved fields are anonymous bit fields. They are declared using the keyword *reserve* followed by the number of bits to be reserved.

```
reserve 6;
```

#### 4.1.5 Alignment

Alignment is designed to align a message to a certain bit boundary.

```
align(8);
```

If the above *align* expression appears at the end of a message, the size of the message will always be an integer multiple of 8 bits, i.e. octet aligned. A remainder may also be specified for an alignment. For example, the expression “align(8, 1)” makes a message octet aligned plus one bit, which means the size of the message is of the form, 1, 9, 17, and etc.

## 4.2 String Field

A string field is a field that consists of a sequence of characters terminated by a null character or a maximum length. Each character of the string is a bit field with a size that shall be less than 32. A string field shall have a default maximum length. If the length of the string, not including the null character, is less than the maximum length, then the null character is used to explicitly terminate the string. If the length is equal to the maximum length, then the string is not terminated by the null character but when the maximum length is reached. The default null character has a value of “0”.

```
DomainName 8 string;
```

This declares a string with the default maximum size and the default null character. Each character in this string is 8-bit. To specify a null character other than the default:

```
DomainName 8 string(0xFF);
```

To specify a maximum size other than the default:

```
DomainName 8 string[128];
```

Or to specify both:

```
DomainName 8 string(0xFF)[128];
```

## 4.3 Nested Message

A nested message is a field that is of the type of another message. The nested message may be either defined inline or refer to a previously defined message.

### 4.3.1 Inline Defined

```
Message() ::=
{
  Header :
  {
    Type      8;
    SrcAddress 32;
    DstAddress 32;
  }
  /* ... */
}
```

In the example above, the nested message “Header” is defined inline after the colon. The message has three fields. The name of the nested message will be the same as the field name “Header”. You may also name the nested message explicitly by putting an identifier after the colon and before the left curly brace. The name is only visible within the current message definition. If the name shadows a definition in a parent message, then the shadowed definition shall no longer be accessed in the current message and its nested messages.

A size constraint may also be used after the field name.

```
Message() ::=
{
  HeaderLength 8;

  Header HeaderLength * 8 :
  {
    Type      8;
    SrcAddress 32;
    DstAddress 32;
  }
  /* ... */
}
```

The size constraint specifies that the nested message “Header” shall only have “HeaderLength \* 8” bits. This is equivalent to:

## The Transfer Syntax Notation One Specification

```
Message() ::=
{
    HeaderLength 8;

    reserve HeaderLength * 8 :
    {
        Header :
        {
            Type          8;
            SrcAddress    32;
            DstAddress    32;
        }
    }

    /* ... */
}
```

### 4.3.2 Scoping

When a nested message is defined inline, a new scope is introduced. This means the nested message definition may reuse the field names already declared in the enclosing scope without conflicts. Also, the enclosed scope may refer to the fields declared in the enclosing scope(s). When a field is being referenced in an expression, it is first looked up in the current scope, and then the immediate enclosing scope, and so on. There is no restriction on the number of nested scopes.

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

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

In the above definition for PSMM, the bit fields “PILOT\_STRENGTH” and “KEEP” are declared both in PSMM and in the inline nested message definition for “Pilot”, which is legal since they are in a new scope.

### 4.3.3 Argument

Messages may have arguments. Message arguments may be either simple bit fields or messages. By default, bit field arguments are passed by value. If you want to pass them by reference, precede the argument declaration with the keyword *var*.

## The Transfer Syntax Notation One Specification

```
MsgHeader(Version 8) ::=
{
  Type          8;
  SrcAddress    32;
  DstAddress    32;
}

MsgPayload(Version 8, Header : MsgHeader) ::=
{
  if(Header.Type == 0)
  {
    /* ... */
  }

  if(Version == 1)
  {
    /* ... */
  }
}

Message(Version 8) ::=
{
  Header  : MsgHeader(Version);
  Payload : MsgPayload(Version, Header);
}
```

### 4.3.4 Recursion

Nested messages may be declared recursively.

## The Transfer Syntax Notation One Specification

```
wimax_ie_ARQFeedback() ::=
{
    CID          16;
    LAST         1;
    ACKType      2;
    BSN          11;
    NumACKMaps   2;

    if(ACKType != 1)
    {
        ACKMaps[NumACKMaps + 1] :
        {
            if(ACKType != 3)
            {
                SelectiveACKMap 16;
            }
            else
            {
                SequenceFormat 1;

                if(SequenceFormat == 0)
                {
                    SequenceACKMap 2;
                    Sequence1Length 6;
                    Sequence2Length 6;
                    reserve 1;
                }
                else
                {
                    SequenceACKMap 3;
                    Sequence1Length 4;
                    Sequence2Length 4;
                    Sequenc32Length 4;
                }
            }
        }
    }

    if(LAST != 1)
    {
        Next : wimax_ie_ARQFeedback;
    }
}
```

If the “LAST” bit is not set to 1, then there is another ARQ feedback IE in the list.

### 4.4 Array

Bit fields and nested messages may be elements of a single dimensional array.

## The Transfer Syntax Notation One Specification

```
Enhanced_DL_MAP_IE() ::=
{
    NumAssignment  4;

    Assignments[NumAssignment] :
    {
        if(INC_CID == 1)
        {
            NumCID  8;

            CIDs[NumCID]  16;
        }

        DIUC                4;
        Boosting             3;
        RepetitionCodingIndication  2;
        RegionID             8;
        reserve              3;
    }
}
```

In the above example, “Assignments” is an array of nested messages. The size of the array is specified by the field “NumAssignment”. “CIDs” is also an array with each element a 16-bit field. Default values may be specified for bit field arrays by using curly braces and comma separated constants. For example,

```
CIDs[NumCID]  16 = {0, 1, 2};
```

### 4.4.1 Unbounded Array

If an array size is not specified, then it is an unbounded array, indicating zero or more occurrences of the elements. Unbounded array shall be the last field of a message.

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

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

For unbounded arrays, an optional range may also be used.

## The Transfer Syntax Notation One Specification

```
is856_pktapp_LocationNotification() ::=
{
    LocationType 8;

    Location [(0 .. 1)] : is856_pktapp_Location;
}
```

In the above example, the “Location” array may have zero or one element.

### 4.4.2 Until

An optional *until* may be used after an unbounded array to specify how the array terminates. When an *until* is present, it is assumed that each element of the array is preceded by a leading bit, whose value indicates if another element is to follow. This termination value is specified as an argument to *until* and shall be either zero or one.

```
ts44060_PacketAccessReject() ::=
{
    PAGE_MODE 2 enumerated ts44060_PageModeIE;

    Reject : ts44060_RejectStruct;

    AdditionalReject[] until(0) : ts44060_RejectStruct;
}
```

In the example above, the “AdditionalReject” array continues as long as the leading bit before an element has a value of one.

## 4.5 Case Of

The case of construct in TSN.1 is used to group multiple related fields together such as messages of a protocol, information elements, TLVs, etc. At most one of the fields shall be present at a time. The expression between the keyword *case* and *of* is used to match with the case labels. Multiple values including value ranges may be used for case labels. They shall be separated by commas. A default case may be specified using the ‘\_’ character:

## The Transfer Syntax Notation One Specification

```
wimax_tlv_Common() ::=
{
  Type 8;

  expand wimax_tlv_Length;

  Value Length * 8 : case Type of
  {
    143 =>
      VendorInfo Length * 8;
    144 =>
      VendorID 24;
    145 =>
      UplinkServiceFlow : wimax_tlv_ServiceFlow;
    146 =>
      DownlinkServiceFlow : wimax_tlv_ServiceFlow;
    147 =>
      CurrentTransmitPower 8;
    148 =>
      MACVersion 8;
    149 =>
      HMACTuple : wimax_tlv_HMACTuple;
    150 =>
      CMACTuple : wimax_tlv_CMACTuple(Length);
    151 =>
      ShortHMACTuple : wimax_tlv_ShortHMACTuple;
    - =>
      UnexpectedTLV : wimax_tlv_UnexpectedTLV(Length);
  }
}
```

The size constraint “Length \* 8” is optional. When it is present, it specifies that each branch shall only have “Length \* 8” number of bits. This is equivalent to:

```
wimax_tlv_Common() ::=
{
  Type 8;

  expand wimax_tlv_Length;

  reserve Length * 8 :
  {
    Value : case Type of
    {
      ...
    }
  }
}
```

## 4.6 Conditional

Conditionals provide the mechanism to conditionally include certain fields in a message. Conditionals are specified using the *if* keyword with an optional *else*, and they may be nested. The guard condition shall be a valid Boolean expression 6.1.

## The Transfer Syntax Notation One Specification

```
Anchor_BS_switch_IE() ::=
{
    NumAnchorBSswitch  4;

    AnchorBSswitches[NumAnchorBSswitch] :
    {
        ReducedCID  12;
        ActionCode  2;

        if(ActionCode == 1)
        {
            ActionTime_A  3;
            TEMP_BS_ID    3;
            reserve        2;
        }

        if(ActionCode == 0 || ActionCode == 1)
        {
            CQICHAllocationIndicator  2;
        }
    }
}
```

### 4.7 Block with Size Constraint

A size constraint may be specified for a block of fields.

```
ts24008_CallControlCapabilitiesIE() ::=
{
    Length  8;

    reserve Length * 8 :
    {
        MaxNumSupportedBearers  4;
        reserve                  1;
        ENICM                    1;
        PCP                      1;
        DTMF                     1;

        reserve                  4;
        MaxNumSpeechBearers      4;
    }
}
```

The size of the inner block shall have at most “Length \* 8” number of bits.

### 4.8 Variable, Assignment and While Statement

Variables are declared using the keyword “var”. They are not part of a message and shall only be used to help describe the message. They shall be declared at the beginning of a message body and are subject to the same scoping rules as message fields. A variable has a default value of zero. Its value may be updated using an assignment statement.

Variables and assignment statements may also be used inside a while statement. Field declarations, on the other hand, shall not appear inside a while loop.

## The Transfer Syntax Notation One Specification

```
wimax_mac_MOB_TRF_IND() ::=
{
  var i 8;
  var n 8;

  FMT 1;

  if(FMT == 0)
  {
    SLPIDGroupIndicationBitmap 32;

    /* Count the number of 1's in the bitmap */
    while(i < 32)
    {
      if((SLPIDGroupIndicationBitmap >> i) & 0x1 == 1)
      {
        n = n + 1;
      }

      i = i + 1;
    }

    TrafficIndicationBitmap[n] 32;
  }
  else
  {
    NumPos 8;
    SLPIDs[NumPos] 10;
  }

  /* Rest of the message ... */
}
```

### 4.9 Alias

If two messages have exactly the same structure, one may be defined in terms of the other.

```
Response() ::=
{
  /* ... */
}

Confirm() ::= Response;
```

### 4.10 Error

To report an error in a message, use the error followed by an optional integer error code. Error code shall not be zero.

## The Transfer Syntax Notation One Specification

```
INVALID_PORT_NUMBER_ERROR = 1;

Request() ::=
{
    Port 16;

    If(Port == 0)
    {
        error(INVALID_PORT_NUMBER_ERROR);
    }
}
```

## 5 Macro

Some message definitions share common fields and structures. These common elements may be first defined in a macro. The macro may be expanded later in the form of a textual substitution inside a message. Macros may also take arguments. During the macro expansion, each argument is replaced by the supplied expression.

```
wimax_tlv_Length ::= macro
{
    LengthMSB 1;

    if(LengthMSB == 0)
    {
        Length 7;
    }
    else
    {
        LengthSize 7 (0 .. 2);

        Length LengthSize * 8;
    }
}

wimax_tlv_Common() ::=
{
    Type 8;

    expand wimax_tlv_Length;

    Value Length * 8 : case Type of
    {
        /* TLV values ... */
    }
}
```

## 6 Expressions

### 6.1 Boolean Expressions

Boolean expressions are used as the guard condition in conditionals 4.6.

#### 6.1.1 Unary Boolean Expressions

The unary Boolean expressions are:

- The Boolean constant expression (true and false)
- The logical-complement expression (!)

#### 6.1.2 Binary Boolean Expression

The binary Boolean expressions are:

- The numerical comparison expression (==, !=, <, >, <=, and >=)
- The conditional-and and conditional-or expressions (&& and ||)

### 6.2 Integer Expressions

Integer expressions are used to specify field size, array dimension, or as part of a Boolean expression.

#### 6.2.1 Unary Integer Expressions

The unary integer expressions are:

- The integer constant expression
- The id expression
- The message subfield expression (.)
- The array subscript expression ([])
- The alignment expression (align)
- The count1s expression
- The unary plus expression (+)
- The unary minus expression (-)
- The bitwise complement expression (~)

##### 6.2.1.1 Subfield Expressions

The subfield expression is used to reference the field of a nested message. For example:

## The Transfer Syntax Notation One Specification

```
Header() ::=
{
    Type 8;
}

Message () ::=
{
    header : Header;

    if(header.Type == 0)
    {
        ...
    }
}
```

### 6.2.1.2 Subscript Expression

The subscript expression is used to reference an array element. For example,

```
Message () ::=
{
    ActiveCellId      8;
    NumCellId         8;
    CellIds[NumCellId] 16;

    if(CellIds[ActiveCellId] == 0x3a8e)
    {
        ...
    }
}
```

### 6.2.1.3 Count1s Expression

The count1s expression is used to count the number of “1”s in a field. For example,

```
Message () ::=
{
    Bitmap 8;

    Ids[count1s(Bitmap)] 16;
}
```

## 6.2.2 Binary Integer Expressions

The unary integer expressions are:

- The multiplicative expressions (\*, /, and %)
- The additive expressions (+ and -)
- The bit shift expressions (<< and >>)
- The bitwise expressions (&, |, and ^)

## 6.3 Operator Precedence and Associativity

Operators listed in the following table are of decreasing precedence.

**Table 6-1 Operator Precedence and Associativity**

Operator	Description	Associativity
()	Parentheses (grouping)	left-to-right
+ - ! ~	Unary plus/minus Logical complement/bitwise complement	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left/bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise and	left-to-right
^	Bitwise exclusive or	left-to-right
	Bitwise inclusive or	left-to-right
&&	Logical and	left-to-right
	Logical or	left-to-right

## 6.4 Semantics

The semantics of Boolean and integer expressions is consistent with the ISO C99 standard [5]. The only exception is that TSN.1 Boolean expressions have Boolean type as opposed to the scalar type in C. Therefore, Boolean expression shall only be used where a Boolean value is expected and integer expression where an integer value is expected.

When message arguments, variables, and bit fields are used in an integer expression, their types are assigned according to the following table. This is consistent with the “implicit conversion” as specified in [5].

**Table 6-2 Argument/Variable/Bit Field Type Assignment**

Size (bits)	Sign	Type
<= 16	signed and unsigned	signed 32-bit integer
> 16 and <= 32	signed	signed 32-bit integer
> 16 and <= 32	unsigned	unsigned 32-bit integer
> 32 and <= 64	signed	signed 64-bit integer
> 32 and <= 64	unsigned	unsigned 64-bit integer

## 7 Lexical Tokens

### 7.1 Character Set

TSN.1 uses the ASCII character set.

### 7.2 Line Terminators

Lines are terminated by the ASCII characters LF, CR, or CR followed by LF.

### 7.3 Comments

TSN.1 allows both single line comments and multi-line comments.

#### 7.3.1 Single Line Comments

Single line comments start with “//” and end at the line terminators

```
// This is an example of a single line comment.
```

#### 7.3.2 Multi-line Comments

Multi-line comments start with “/\*” and end with “\*/”. Nested multi-line comments are illegal.

```
/*  
 * This is an example of a multi-line comment.  
*/
```

### 7.4 White Spaces

White space is defined as the ASCII space, horizontal tab, and form feed characters, as well as line terminators.

### 7.5 Identifiers

An identifier shall begin with a letter or an underscore followed by any number of letters, digits, or underscores. A letter is any of the following ASCII characters [A-Z] or [a-z]; a digit is one of [0-9]. Identifiers are case sensitive. They shall not have the same spelling as any of the keywords.

### 7.6 Keywords

The following character sequences are reserved for use as keywords and shall not be used as identifiers:

# The Transfer Syntax Notation One Specification

package	import	enumerated	macro
expand	var	optional	if
else	case	of	reserve
align	while	signed	until
count1s	true	false	error

## 7.7 Literals

### 7.7.1 Boolean Literals

There are two Boolean literals:

true false

### 7.7.2 Integer Literals

An integer literal may be expressed in decimal, binary, or hexadecimal.

A decimal numeral is either the single character 0, or consists of an ASCII digit from 1 to 9 optionally followed by one or more ASCII digits from 0 to 9.

A binary numeral consists of the leading ASCII characters “0b” or “0B” followed by one or more ASCII binary digits.

A hexadecimal numeral consists of the leading ASCII characters “0x” or “0X” followed by one or more ASCII hexadecimal digits. Hexadecimal digits with values 10 through 15 are represented by the ASCII letters ‘a’ through ‘f’ or ‘A’ through ‘F’, respectively; each letter used as a hexadecimal digit may be uppercase or lowercase.

Integer literals may be either 32-bit or 64-bit and they may be signed or unsigned.

**Table 7-1 Integer Types and Values**

Type	Values (inclusive)
signed 32-bit integer	-2147483648 through 2147483647
unsigned 32-bit integer	0 through 4294967295
signed 64-bit integer	-9223372036854775808 through 9223372036854775807
unsigned 64-bit integer	0 through 18446744073709551615

An optional suffix may also be used to specify the type of the integer, “u” or “U” for unsigned and “l” or “L” for 64-bit. An integer literal is assigned to a type in the corresponding list in Table 7-2 in which its value can be first represented.

**Table 7-2 Integer Literal Type Assignment**

Suffix	Decimal	Hexadecimal
none	signed 32-bit integer signed 64-bit integer	signed 32-bit integer unsigned 32-bit integer signed 64-bit integer unsigned 64-bit integer

## The Transfer Syntax Notation One Specification

u or U	unsigned 32-bit integer unsigned 64-bit integer	unsigned 32-bit integer unsigned 64-bit integer
l or L	signed 64-bit integer	signed 64-bit integer unsigned 64-bit integer
Both u or U and l or L	unsigned 64-bit integer	unsigned 64-bit integer

### 7.7.3 String Literals

A string literal consists of zero or more character enclosed in double quotes. The following characters are not allowed inside a string: CR, LF. Double quotes shall be escaped with the backward slash character, for example, “\””.

## 7.8 Separators

The following ASCII characters are the separators:

( ) [ ] { } ; : , . => .. \_

## 7.9 Operators

The following ASCII characters are the operators:

::= + - \* / % && || == != > < >= <= ! << >> & | ^ ~

## A. BNF Grammar

```
<compilation-unit> ::=
  <package-decl-opt> <import-decls> <specifications>

<package-decl-opt> ::=
  "package" <package-id> ';'
  | empty

<import-decls> ::=
  <import-decls> <import-decl>
  | empty

<import-decl> ::= "import" <package-id-star> ';'

<specifications> ::=
  <specifications> <specification>
  | empty

<specification> ::=
  '{' <specifications> '}'
  | <conditionals>
  | <definition>

<conditional> ::=
  "if" '(' <expression> ')' <specification>
  | "if" '(' <expression> ')' <specification> "else" <specification>

<qualified-id> ::=
  <package-id>

<package-id-star> ::=
  <package-id>
  | <package-id> '.' '*'

<package-id> ::=
  identifier
  | <package-id> '.' identifier

<definition> ::=
  identifier "==" <expression> ';'
  | identifier "==" string ';'
  | identifier "==" "enumerated" '{' <enum-literals> '}'
  | identifier "==" "macro" <declaration-block>
  | identifier '(' <macro-arguments-opt> ')' "=="
  "macro" <declaration-block>
  | identifier '(' <argument-decls-opt> ')' "=="
  '{' <variable-decls> <declarations> '}'
  | identifier '(' ')' "==" <qualified-id> ';'
  | ';'

<enum-literals> ::=
  <enum-literals> ',' <enum-literal>
  | <enum-literal>
```

## The Transfer Syntax Notation One Specification

```
<enum-literal> ::=
  identifier
  | identifier '(' <expression> ')'
```

```
<macro-arguments-opt> ::=
  <macro-arguments>
  | empty
```

```
<macro-arguments> ::=
  <macro-arguments> ',' identifier
  | identifier
```

```
<argument-decls-opt> ::=
  <argument-decls>
  | empty
```

```
<argument-decls> ::=
  <argument-decls> ',' <argument-decl>
  | <argument-decl>
```

```
<argument-decl> ::=
  identifier <expression> <signed-opt>
  | "var" identifier <expression> <signed-opt>
  | identifier ':' <qualified-id>
```

```
<variable-decls> ::=
  <variable-decls> <variable-decl>
  | empty
```

```
<variable-decl> ::=
  "var" identifier <expression> <signed-opt>
  <default-value-opt> ';' 
```

```
<declarations> ::=
  <declarations> <declaration>
  | empty
```

```
<declaration> ::=
  <declaration-block>
  | <macro-expansion-decl>
  | <optional-decl>
  | <conditional-decl>
  | <field-decl>
  | <while-loop-decl>
  | <assignment-decl>
  | ';' 
```

```
<declaration-block> ::=
  '{' <declarations> '}' 
```

```
<macro-expansion-decl> ::=
  "expand" <qualified-id>
  | "expand" <qualified-id> '(' <expression-list-opt> ')'
```

```
<optional-decl> ::=
  "optional" identifier ':' <declaration>
```

## The Transfer Syntax Notation One Specification

```
<conditional-decl> ::=
  "if" '(' <expression> ')' <declaration>
  | "if" '(' <expression> ')' <declaration> "else" <declaration>

<field-decl> ::=
  identifier <array-opt> <expression> <signed-opt>
  <interval-opt> <default-value-opt> ';'
  | identifier <array-opt> <expression> <signed-opt>
  <interval-opt> "enumerated" <qualified-id>
  <default-value-opt> ';'
  | identifier <array-opt> <expression> <signed-opt>
  <interval-opt> "enumerated" identifier-opt
  '{' <enum-literals> '}' <default-value-opt> ';'
  | identifier <array-opt> <expression> "string"
  <null-char-opt> <max-num-char-opt> ';'
  | identifier <array-opt> <expression-opt> ':'
  <qualified-id> ';'
  | identifier <array-opt> <expression-opt> ':'
  <qualified-id> '(' <expression-list-opt> ')' ';'
  | identifier <array-opt> <expression-opt> ':' identifier-opt
  '{' <variable-decls> <declarations> '}'
  | identifier <expression-opt> ':' "case" <expression> "of"
  '{' <case-decls> '}'
  | "reserve" <expression> ':' <declaration-block>
  | "reserve" <expression> ';'
  | <align-expression> ';'

<while-loop-decl> ::=
  "while" '(' <expression> ')' <declaration>

<assignment-decl> ::=
  identifier '=' <expression> ','

<null-char-opt> ::=
  '(' <expression> ')'
  | empty

<max-num-char-opt> ::=
  '[' <expression> ']'
  | empty

<case-decls> ::=
  <case-decls> <case-decl>
  | empty

<case-decl> ::=
  <case-labels> "=>" <field-decl>
  | '_' "=>" <field-decl>

<case-labels> ::=
  <case-labels> ',' <case-label>
  | <case-label>

<case-label> ::=
  <expression> ".." <expression>
  | <expression>
```

## The Transfer Syntax Notation One Specification

```
<array-opt> ::=
  '[' <expression-opt> ']' <until-opt>
  | '[' <interval> ']' <until-opt>
  | empty

<until-opt> ::=
  "until" '(' integer ')'
  | empty

<signed-opt> ::=
  "signed"
  | empty

<default-value-opt> ::=
  '=' <expression>
  | '=' '{' <expression-list-opt> '}'
  | empty

<expression-opt> ::=
  <expression>
  | empty

<interval-opt> ::=
  <interval>
  | empty

<interval> ::=
  '(' <expression> ".." <expression> ')

<expression-list-opt> ::=
  <expression-list>
  | empty

<expression-list> ::=
  <expression-list> ',' <expression>
  | <expression>

<expression> ::=
  | <expression> '+' <expression>
  | <expression> '-' <expression>
  | <expression> '*' <expression>
  | <expression> '/' <expression>
  | <expression> '%' <expression>
  | <expression> "&&" <expression>
  | <expression> "||" <expression>
  | <expression> "==" <expression>
  | <expression> "!=" <expression>
  | <expression> ">=" <expression>
  | <expression> "<=" <expression>
  | <expression> '>' <expression>
  | <expression> '<' <expression>
  | <expression> "<<" <expression>
  | <expression> ">>" <expression>
  | <expression> '&' <expression>
  | <expression> '|'| <expression>
  | <expression> '^' <expression>
  | "true"
```

## The Transfer Syntax Notation One Specification

```
| "false"  
| integer  
| identifier  
| <expression> '.' identifier  
| <expression> '[' <expression> ']'  
| "countls" <expression>  
| <align-expression>  
| '+' <expression>  
| '-' <expression>  
| '!' <expression>  
| '~' <expression>  
| '(' <expression> ')'  
  
<align-expression> ::=  
  "align" '(' <expression> ',' <expression> ')'  
| "align" '(' <expression> ')'  
  
<identifier-opt> ::=  
  identifier  
| empty
```