ASN.1 for More Effective Network Standards

Alessandro Triglia
sandro@oss.com
OSS Nokalva, Inc.

IEEE 802
Dallas, TX 2010-11-08

ASN.1

- ASN.1 = Abstract Syntax Notation One
- Family of international standards
  - jointly developed and published by ISO/IEC and ITU-T
- Originally developed in the 1980's…
  - …but still alive and well, and still being maintained
- Used in several industries
  - mainly, but not only, telecommunications

ASN.1

- ASN.1 is:
  1) a formal language for specifying the logical structure of data that is to be exchanged between two endpoints
     - independent of hardware platform, operating system, programming language, local representation, etc.
  2) standard sets of rules for encoding instances of logical data structures specified in ASN.1 notation
     - for the purpose of transmission

ASN.1 notation

Examples from P802.16m Draft 9 (1/3)

```
SleepResponseInfo ::= SEQUENCE {
    trafficIndicationFlag         TrafficIndicationFlag,
    listeningWindowExtFlag        ListeningWindowExtFlag,
    nextSleepCycleIndicator       NextSleepCycleIndicator,
    initialSleepCycle             INTEGER (0..15),
    finalSleepCycle               INTEGER (0..1023),
    listeningWindow               INTEGER (0..63),
    listeningSubframeBitmap       BIT STRING (SIZE(8))
}
```

ASN.1 notation

Examples from P802.16m Draft 9 (2/3)

```
MAC-Control-Msg-Type ::= CHOICE {
   -- System information
   aaiSCD                        AAI-SCD,
   aaiSIIAdv                     AAI-SII-ADV,
   aaiULPCNi                     AAI-ULPC-NI,

   -- Network entry / re-entry
   aaiRngReq                     AAI-RNG-REQ,
   aaiRngRsp                     AAI-RNG-RSP,
   aaiRngAck                     AAI-RNG-ACK,
   aaiRngCfm                     AAI-RNG-CFM,
   aaiSbcReq                     AAI-SBC-REQ,
   aaiSbcRsp                     AAI-SBC-RSP,
   aaiRegReq                     AAI-REG-REQ,
   aaiRegRsp                     AAI-REG-RSP,
   ..........................
}
```

ASN.1 notation

Examples from P802.16m Draft 9 (3/3)

```
AAI-GRP-CFG ::= SEQUENCE {
   deletionFlag         ENUMERATED { flowAdded, flowDeleted },
   dlULIndicator        ENUMERATED { dlAllocation, ulAllocation },
   flowID               FID,
   burstSize            INTEGER (0..31) OPTIONAL,
   graInfo              CHOICE {
       graInfoForDeletededFlow    NULL,
       graInfoForAddedFlow        GroupResourceAllocInfo
   },
   ...
}
```

Principles and Benefits of ASN.1

- Separation of concerns
  - The description of the logical structure of a message is kept completely separate from the details of the encoding
- Message descriptions are machine-processable
  - This enables the creation and use of software development tools and testing tools that can read and understand the formal definitions
- Encodings are standardized
  - The problem of specifying detailed encodings and the problem of encoding/decoding messages and their fields do not need to be addressed again and again
- Extensibility
  - It is possible to extend a message description in controlled ways while ensuring backward- and forward-compatibility between different version implementations

Principles and Benefits of ASN.1

- Separation of concerns (1/3)
  - The description of the logical structure of a message is kept completely separate from the details of the encoding
  - A protocol designer can focus on describing the essential (abstract) properties of the data without being distracted by many encoding details
    - Examples: byte order (endianness); how many bits should be assigned to each field; what binary value should be assigned to each option; how to indicate the presence or absence of an optional field; how to align (and whether to align) each field with respect to byte or word boundaries; inclusion of padding bits; and so on

Principles and Benefits of ASN.1

- Separation of concerns (2/3)
  - Message specifications are concise
    - they describe only the logical structure of the data and its most relevant properties
    - essential semantic links to the protocol specification can be provided through a careful choice of names
    - comments can be included to provide explanations, references, and additional requirements as needed
  - A reader of a specification that is moderately familiar with ASN.1 will be able to quickly grasp the structure of the data and the properties that are most relevant to the logic of the protocol
    - The logical structure stands out

Principles and Benefits of ASN.1

- Separation of concerns (3/3)
  - Analogy with 3rd generation programming languages
    - A messaging specification in ASN.1 notation is analogous to “source code”
    - The standard encodings are analogous to “machine code”
    - The same source code can be rendered into machine code in different ways
      - Different processors, kinds of optimizations, sets of runtime requirements
    - The meaning of the source code is usually independent of the processors on which the machine code will be executed
    - The majority of the people who write and read source code are not interested in the details of the machine code
      - But a few of them are

Principles and Benefits of ASN.1

- Message descriptions are machine-processable (1/3)
  - This enables the creation and use of software development tools and testing tools that can read and understand the formal definitions
  - A software development tool can, given a message specification, generate source code, encoder/decoders, and other artifacts that will facilitate and speed up the implementation work
  - A testing tool can process an ASN.1 specification and execute test cases against an implementation
    - There is no need to manually write code that encodes and parses messages in support of testing

Principles and Benefits of ASN.1

- Message descriptions are machine-processable (2/3)
  - The ASN.1 notation is a rigorous formal language, which ensures that any syntactically correct definition will be unambiguous
    - For example, it is impossible to define the same type multiple times, to “forget” to define a type, or to include a definition having insufficient or inconsistent information

Principles and Benefits of ASN.1

- Message descriptions are machine-processable (3/3)
  - A protocol designer can use an ASN.1 tool to verify the syntactic correctness and completeness of a specification at any stage of development
    - Several errors such as missing or syntactically incomplete type definitions can be caught easily and early in the standardization process, because the syntax check will fail
  - A protocol designer or implementer can use an ASN.1 tool to create sample instances of messages conforming to a given specification
    - This facilitates testing and debugging

Principles and Benefits of ASN.1

- Encodings are standardized (1/2)
  - The problem of specifying detailed encodings and the problem of encoding/decoding messages and their fields do not need to be addressed again and again
  - Several standard sets of encoding rules for ASN.1 are available, each with different characteristics:
    - BER – *Basic Encoding Rules*
    - DER – *Distinguished Encoding Rules*
    - PER – *Packed Encoding Rules*
      - PER Aligned
      - PER Unaligned
    - XER – *XML Encoding Rules*

Principles and Benefits of ASN.1

- Encodings are standardized (2/2)
  - Typically, a protocol specification mandates one particular standard set of encoding rules to be used for that protocol
    - Common choices are BER (some earlier standards), DER (security standards), PER Aligned (some 3GPP standards), and PER Unaligned (aviation standards and 3GPP standards)
  - Someone interested in the details of the encodings for a given specification can turn his attention to the standard encoding rules
    - A protocol designer or implementer may need to do this occasionally
    - In most practical cases this is not a difficult task, but it does require some knowledge of the encoding rules
    - Encoding concerns remain separate from logical structure

Principles and Benefits of ASN.1

- Extensibility
  - It is possible to extend a message description in controlled ways while ensuring backward- and forward-compatibility between different version implementations
  - A version-2 receiver will be able to handle any message created by a version-1 sender
  - A version-1 receiver will be able to handle any message created by a version-2 sender (possibly ignoring any parts related to extensions that were defined after version 1)
    - Here “1” and “2” mean any m, n with m < n
  - This mechanism works with any standard encoding rules

ASN.1 standards

- Three sets of standards:
  - ASN.1 notation (X.680, X.681, X.682, X.683)
    - a formal language for the definition of messages
  - Encoding rules
    - BER – *Basic Encoding Rules* (X.690)
    - DER – *Distinguished Encoding Rules* (X.690)
    - PER – *Packed Encoding Rules* (X.691)
    - XER – *XML Encoding Rules* (X.693)
    - *...*
  - Other ASN.1 standards
    - Mapping from XML Schema to ASN.1 (X.694)
    - Fast Infoset (X.891)
    - Fast Web Services (X.892)

Uses of ASN.1

- Some traditional applications of ASN.1:
  - Signaling standards for the public switched telephone network (SS7 family)
  - Network management standards (SNMP, CMIP)
  - Directory standards (X.500 family, LDAP)
  - Public Key Infrastructure standards (X.509, etc.)
  - PBX control (CSTA)
  - IP-based Videoconferencing (H.323 family)
- Some more recent applications:
  - Aeronautical Telecommunication Network
  - Biometrics (BIP, CBEFF, ACBio)
  - Intelligent transportation (SAE J2735)
  - Cellular telephony (GSM, GPRS/EDGE, UMTS, LTE)

Developing an application that uses an ASN.1 specification

1. The application developer submits the ASN.1 specification to an ASN.1 compiler that is part of an ASN.1 toolkit
2. The ASN.1 compiler generates some source code in a programming language (C, C++, Java, etc.)
3. An encoder/decoder for the designated set of ASN.1 encoding rules may either:
   - be an integral part of the source code generated by the ASN.1 compiler from the given ASN.1 schema; or
   - be provided as a separate, pre-built component, typically a library that is part of the ASN.1 toolkit
4. The application developer integrates the generated source code and the encoder/decoder library into his application
5. The resulting application is typically able to create, encode, send, receive, decode, and process messages conforming to the ASN.1 specification

At runtime

1. The <u>sending application</u> creates a message conforming to a certain message type within the ASN.1 specification
   - the message is represented in a data structure that is appropriate to the programming language in use (e.g., a Java class or a C struct)
2. The <u>sending application</u> encodes the message using the designated set of ASN.1 encoding rules
   - BER – DER – PER – XER *…*
3. The encoded message is transferred from the sending endpoint to the receiving endpoint
4. The <u>receiving application</u> decodes the encoded message using the designated set of ASN.1 encoding rules
   - the message is now represented in a data structure appropriate to the programming language in use
5. The <u>receiving application</u> processes the message

Boolean types
Characteristics

- A component whose type is `BOOLEAN` (or a user-defined type derived from `BOOLEAN`) may take as its value one of the values `FALSE` and `TRUE`

BOOLEAN

Usage examples from 802.16m D9

```
AAI-DSA-REQ ::= SEQUENCE {
    .........................
    emergencyIndication             BOOLEAN OPTIONAL,
    .........................
}

EMBSZoneInfoInHandover ::= SEQUENCE {
    serviceFlowUpdateIndicator     BOOLEAN,
    .........................
}

AAI-RNG-RSP ::= SEQUENCE {
    rangingAbortFlag               BOOLEAN,
    .........................
}
```

Integer types
Characteristics (1/2)

- A component whose type is `INTEGER` (or a user-defined type derived from `INTEGER`) may take as its value any integer from a certain set
  - If the `INTEGER` type has no constraints, the permitted value set is the range [–infinity..+infinity]
  - A permitted value set can be specified, for example, by including a value range constraint, as follows:

```
A ::= INTEGER (0..255)
B ::= INTEGER (0..65535)
C ::= INTEGER (-100000..100000)
D ::= INTEGER (1..8)
PhyCarrierIndex ::= INTEGER (0..63)
```

Integer types
Characteristics (2/2)

- It is possible to associate names with some of the values of a user-defined integer type, as in the following example:

```
CsSpecification ::= INTEGER {
                    packetIpv4 (1),
                    packetIpv6 (2),
                    packetEthernet (3),
                    packetIpv4OrIpv6 (14),
                    multiProtocol (15) } (0..255)
```

INTEGER

Usage examples from 802.16m D9

```
PhyCarrierIndex ::= INTEGER (0..63)

FID ::= INTEGER (0..15)

NbrAdvChangeCount ::= INTEGER (0..7)

AmsCapabilities ::= SEQUENCE {
   maxARQBufferSize       INTEGER (0..8388607) OPTIONAL,
   maxNonARQBufferSize    INTEGER (0..8388607) OPTIONAL,
    .........................
}

TargetABSSelection ::= SEQUENCE {
  targetABSID               BSID,
  targetPhyCarrierID        PhyCarrierIndex OPTIONAL,
  servingPhyCarrierID       PhyCarrierIndex OPTIONAL,
    .........................
}
```

Enumerated types
Characteristics (1/2)

- There are no built-in enumerated types
- The ENUMERATED keyword is used to create a (user-defined) enumerated type, as follows:

```
A ::= ENUMERATED { red, yellow, green }
DirIndicator ::= ENUMERATED { uplink, downlink }
```

- A component whose type is an enumerated type may take as its value any one of the names listed in the definition of the enumerated type

Enumerated types

Characteristics (2/2)

- It is possible to associate numbers with some of the names present in the definition of an enumerated type, as follows:

  ```
  A ::= ENUMERATED { red (3), yellow (2), green (1) }
  ```

  - This feature exists for historical reasons and makes sense only for a specification designed to be encoded in BER or DER (the numbers are transmitted in BER and DER). In PER, the numbers are taken into account only for the purpose of determining the order of the enumerations. In the above example, the encodings would not change if the “3” were replaced by a “15”, but would change if the “2” became a “4”.

ENUMERATED

Usage examples from 802.16m D9

```
DirIndicator ::= ENUMERATED { uplink, downlink }

McCapabilities ::= ENUMERATED { noMcModes, basicMcMode,
                                mcAggregation, mcSwitching,
                                mcAggregationAndSwitching }

QosParameter ::= SEQUENCE {
   .........................
   secGrantSize             GrantSize OPTIONAL,
   adaptationMethod         ENUMERATED { absInitiated,
                                         amsInitiated } OPTIONAL,
   accessClass              INTEGER (0..3) OPTIONAL,
   differentiatedBrTimer    INTEGER (1..64) OPTIONAL,
   .........................
   macInOrderDelivery       ENUMERATED { notPreserved,
                                         preserved } OPTIONAL,
   .........................
}
```

Bit string types
Characteristics (1/2)

- A component whose type is `BIT STRING` (or a user-defined type derived from `BIT STRING`) takes as its value a string of bits
  - If the bit string type has no constraints, the string may have any length from zero to infinity
  - A fixed length for the string can be specified by including a single-value size constraint, as follows:

```
A ::= BIT STRING (SIZE(8))
STID ::= BIT STRING (SIZE(12))
```

  - A range of permitted lengths for the string can be specified by including a value-range size constraint, as follows:

```
B ::= BIT STRING (SIZE(0..255))
C ::= BIT STRING (SIZE(1..8))
```

Bit string types
Characteristics (2/2)

- It is possible to assign names to one or more locations within a user-defined bit string type, as in the following example of a simple bitmap:

```
ReportMetric ::= BIT STRING {
        absCINRMean (0),
        absRSSIMean (1),
        relativeDelay (2),
        absRTD (3) } (SIZE(4))
```

- The length of a bit string is an inherent part of its value, and does not need to be provided separately
  - For example, there is no need to include a “length” field before the bit string field

BIT STRING

Usage examples from 802.16m D9

```
BSID ::= BIT STRING (SIZE(48))

STID ::= BIT STRING (SIZE(12))

MACAddress ::= BIT STRING (SIZE(48))

ReentryProOptimization ::= BIT STRING {
             omitSbcMessages (0),
             omitPkmAuthenticationPhase (1),
             omitRegMessages (2),
             omitIPRefresh (3),
             contextAvailability (4) } (SIZE(5))
```

Octet string types
Characteristics (1/2)

- A component whose type is `OCTET STRING` (or a user-defined type derived from `OCTET STRING`) takes as its value a string of octets
  - If the octet string type has no constraints, the string may have any length
  - A fixed length for the string can be specified by including a single-value size constraint, as follows:

```
A ::= OCTET STRING (SIZE(4))
IPv6Address ::= OCTET STRING (SIZE(16))
```

  - A range of permitted lengths for the string can be specified by including a value-range size constraint, as follows:

```
B ::= OCTET STRING (SIZE(0..255))
SMS ::= OCTET STRING (SIZE(1..140))
```

  - The length of an octet string is an inherent part of its value, and does not need to be provided separately

Octet string types
Characteristics (2/2)

- It is possible to specify that an octet string or bit string type is required to contain the encoding of an instance of a certain type, as follows:

```
Layer-1-Message ::= SEQUENCE {
  ..........................
    payload      OCTET STRING
                  (CONTAINING Layer-2-Message),
  ..........................
}
Layer-2-Message ::= CHOICE {
   message1 Message1,
   message2 Message2,
  ..........................
}
```

  - First, an instance of the contained type will be encoded, and then the octets that constitute its encoding will be used as the value of the octet string component
  - The contained type may be any built-in or user-defined type

OCTET STRING

Usage examples from 802.16m D9

```
SMS ::= OCTET STRING (SIZE(1..140))

IPv4Address ::= OCTET STRING (SIZE(4))

IPv6Address ::= OCTET STRING (SIZE(16))

AAI-L2-XFER ::= SEQUENCE {
   transferType        INTEGER { ................ } (0..255),
   transferSubtype     INTEGER (0..15) OPTIONAL,
   payload             OCTET STRING (SIZE(1..999)) OPTIONAL,
   ...
}
```

Sequence types
Characteristics (1/2)

- The `SEQUENCE` keyword is used to create a (user-defined) sequence type, as follows:

```
EMBSZoneInfoItem ::= SEQUENCE {
   embsZoneID                  BIT STRING (SIZE(7)),
   newEMBSZoneID               BIT STRING (SIZE(7)) OPTIONAL,
   physicalCarrierIndex        INTEGER (0..63),
   bitmapAndServiceFlowInfo BitmapAndSfInfo
}
```

- A component whose type is a sequence type takes as its value an ordered list of values, each being a permitted value of one of the components specified in the definition of the sequence type, in the same order
  - Each component type may be any built-in or user-defined type

Sequence types

Characteristics (2/2)

- Each component of a sequence type may be specified as mandatory (by default), optional, or optional with a default value, as follows:

```
EMBSZoneInfoItem ::= SEQUENCE {
   embsZoneID                  BIT STRING (SIZE(7)),
   newEMBSZoneID               BIT STRING (SIZE(7)) OPTIONAL,
   physicalCarrierIndex        INTEGER (0..63) DEFAULT (0)
}
```

  - The indication of whether each optional component is present or absent in a sequence value is an inherent part of the sequence value, and does not need to be provided separately
    - For example, there is no need to include a “flag” field or “bitmap” field before some optional components of a sequence

Choice types
Characteristics

- The `CHOICE` keyword is used to create a (user-defined) choice type, as follows:

```
AddressInfo ::= CHOICE {
   macAddress          MACAddress,
   currentSTID         STID
}
```

- A component whose type is a choice type takes as its value a single component value, which must be a value of one of the alternatives specified in the definition of the choice type
  - Each alternative type may be any built-in or user-defined type
  - The indication of which alternative has been chosen in a choice value is an inherent part of the choice value, and does not need to be provided separately
    - For example, there is no need to include an enumerated type before the choice type

SEQUENCE, CHOICE

Usage examples from 802.16m D9

```
HandoverReentry ::= SEQUENCE {
   stidOrMacAddress   CHOICE {
      stidInfo              SEQUENCE {
         servingBsid            BSID,
         previousSTID           STID
      },
      addressInfo           CHOICE {
         macAddress             MACAddress,
         currentSTID            STID
      }
   },
   akCount                AKCount OPTIONAL,
   fidList                SEQUENCE (SIZE(0..15)) OF FidInfo OPTIONAL,
   ...
}
```

Sequence-of types
Characteristics

- The `SEQUENCE OF` keywords are used to create a (user-defined) sequence-of type, as follows:

```
IntegerList ::= SEQUENCE OF INTEGER
```

- A component whose type is a sequence-of type takes as its value an ordered list of component values, all of the type specified in the definition of the sequence-of type
  - The component type may be any built-in or user-defined type
  - If the sequence-of type has no constraints, the lists may have any length
  - A fixed length for the lists can be specified by including a single-value size constraint, as follows:

```
A ::= SEQUENCE (SIZE(16)) OF SEQUENCE { flag BOOLEAN }
```

  - A range of permitted lengths for the lists can be specified by including a value-range size constraint, as follows:

```
IntegerList ::= SEQUENCE (SIZE(0..1000)) OF INTEGER
```

SEQUENCE OF

Usage examples from 802.16m D9

```
NspInformation ::= SEQUENCE {
   nspIdentifier           SEQUENCE (SIZE(0..255)) OF NSPID,
   verboseNspNameList      SEQUENCE (SIZE(0..255)) OF
                                       VerboseName OPTIONAL
}

AbsInitDsdInfo ::= SEQUENCE {
   fid                     FID,
   embsZoneID              EMBSZoneID,
   embsidFIDMappingArray  SEQUENCE (SIZE(1..15)) OF SEQUENCE {
      embsid                  EMBSID,
      fid                     FID
   }
}
```

Null types
Characteristics

- A component whose type is `NULL` (or a user-defined type derived from `NULL`) may only take as its value the name `NULL`
- Null types are mostly useful as the types of alternatives within choice types, as in the following example:

```
Mode ::= CHOICE {
    hoCmd        HandoverCommand,
    zsCmd        ZoneSwitchCommand,
    hoReject     NULL
}
```

  - Every choice value includes the indication of which alternative has been chosen as well as the value of that alternative. In some cases (as in the above example) and for some of the alternatives, the fact that a certain alternative has been chosen is all one needs to know, and the null type is adequate.

CHOICE , NULL

Usage examples from 802.16m D9

```
AAI-DREG-REQ ::= SEQUENCE {
   deRegReqCode                 CHOICE {
        deregFromABSAndNetwork            NULL,
        deregAndInitIdleMode              DeregAndInitIdleMode,
        unsolicitedDeregRspWithAct05      NULL,
        rejectUnsolicitedDeregRsp         NULL,
        deregToEnterDcrMode               DeregToEnterDcrMode,
        unsolicitedDeregRspWithAct00      NULL,
        ...
   }
}

AAI-GRP-CFG ::= SEQUENCE {
    .............................
    graInfo                      CHOICE {
       graInfoForDeletededFlow           NULL,
       graInfoForAddedFlow               GroupResourceAllocInfo
    },
    ...
}
```

Other major types

- `OBJECT IDENTIFIER`
  - A variable-length string of integers, used as an identifier with global scope
    - Example: `1.1.19785.0.257.8`
  - Each value identifies a node in a tree, which is a hierarchy of registration authorities and numbers allocated by them
- `REAL`
  - A floating-point number
- `IA5String`
  - A US-ASCII character string (7-bit characters)
- `UTF8String`
  - A Unicode character string in UTF-8 format
- `TIME`
  - A variety of types representing time

Extensibility

- Extensibility is a feature of ASN.1 that enables both backward- and forward-compatibility between endpoints implementing different versions of an ASN.1 specification
- Syntax: a "..." symbol ("extension marker") included in a certain position within a type definition makes the type extensible
- There are rules that must be followed when extending a type in a later version
  - First rule: a type that is non-extensible in the very first version cannot be made extensible in any subsequent version (an extension marker may not be added where there was none)

Extensibility

- Extensible integer types:
  - In version 1: `INTEGER (0..255, ...)`
  - In version 2: `INTEGER (0..255, ..., `**`0..587`**`)`
  - In version 3: `INTEGER (0..255, ..., `**`0..587 | 0..15589`**`)`
- Extensible enumerated types:
  - In version 1: `ENUMERATED { red, white, ... }`
  - In version 2: `ENUMERATED { red, white, ...,` **`grey, yellow`** `}`
  - In version 3: `ENUMERATED { red, white, ...,` **`grey, yellow, pink, black`** `}`

Extensibility

- Extensible bit string types:
  - In version 1: `BIT STRING (SIZE(16, ...))`
  - In version 2: `BIT STRING (SIZE(16, ..., 24))`
  - In version 3: `BIT STRING (SIZE(16, ..., 24 | 32))`
- Extensible octet string types:
  - In version 1: `OCTET STRING (SIZE(4..8, ...))`
  - In version 2: `OCTET STRING (SIZE(4..8, ..., 24))`
  - In version 3: `OCTET STRING (SIZE(4..8, ..., 24 | 32))`
- Extensible sequence-of types:
  - In v.1:
    ```
    SEQUENCE (SIZE(0..15, ...))
                            OF SomeType
    ```
  - In v.2:
    ```
    SEQUENCE (SIZE(0..15, ..., 0..255))
                            OF SomeType
    ```
  - In v.3:
    ```
    SEQUENCE (SIZE(0..15, ..., 0..255 | 0..4095))
                            OF SomeType
    ```

Extensibility

- Extensible sequence types:
  - In version 1:

```
EMBSZoneInfoItem ::= SEQUENCE {
   embsZoneID                  BIT STRING (SIZE(7)),
   newEMBSZoneID               BIT STRING (SIZE(7)) OPTIONAL,
   physicalCarrierIndex        INTEGER (0..63),
   ...
}
```

  - In version 2:

```
EMBSZoneInfoItem ::= SEQUENCE {
   embsZoneID                  BIT STRING (SIZE(7)),
   newEMBSZoneID               BIT STRING (SIZE(7)) OPTIONAL,
   physicalCarrierIndex        INTEGER (0..63),
   ...,
   bitmapAndServiceFlowInfo BitmapAndSfInfo
}
```

Extensibility

- Extensible choice types:
  - In version 1:

```
mode          CHOICE {
   hoCmd          HandoverCommand,
   ...
 }
```

  - In version 2:

```
mode          CHOICE {
   hoCmd          HandoverCommand,
   ...,
   zsCmd          ZoneSwitchCommand,
   hoReject       NULL
 }
```

Boolean types
PER encoding summary

- In PER Unaligned, a component whose type is a boolean type is encoded as follows:
  - the value FALSE is encoded as a single ‘0’ bit
  - the value TRUE is encoded as a single ‘1’ bit

Integer types
PER encoding summary

- In PER Unaligned, a component whose type is an integer type (with no extension marker) is encoded as follows:
  - If the integer type has a finite lower bound and a finite upper bound, then the lower bound is subtracted from the value, and the difference is encoded into the minimum number of bits capable of expressing the largest such difference
  - Otherwise, the value is encoded into a variable number of octets, preceded by a length prefix which is usually a single octet

Enumerated types
PER encoding summary

- In PER Unaligned, a component whose type is an enumerated type (with no extension marker) is encoded as follows:
  - If the names in the definition of the enumerated type have numbers associated with them, they are reordered according to those numbers
  - An index, starting from zero and increasing by one, is assigned to each name in order
  - The index of the name that is the value of the enumerated type is encoded into the minimum number of bits capable of expressing the largest such index (possibly zero)

Bit string types
PER encoding summary

- In PER Unaligned, a component whose type is a bit string type (with no extension marker) is encoded as follows:
  - If the bit string type has a fixed length that is less than 65536, then the bits of the string are encoded without any length prefix
  - If the bit string type has a variable length and the length's upper bound is less than 65536, then the length's lower bound is subtracted from the length of the string, and the difference is encoded into the minimum number of bits capable of expressing the largest possible such difference; the bits of the string will follow this prefix
  - In other cases, the bit string is encoded into one or more fragments; each fragment (or the only fragment) will contain at most 65536 bits and will be preceded by a length prefix encoded in a special way

Octet string types
PER encoding summary

- In PER Unaligned, a component whose type is an octet string type (with no extension marker) is encoded as follows:
  - If the octet string type has a fixed length that is less than 65536, then the octets of the string are encoded without any length prefix
  - If the octet string type has a variable length and the length's upper bound is less than 65536, then the length's lower bound is subtracted from the length of the string, and the difference is encoded into the minimum number of bits capable of expressing the largest possible such difference; the octets of the string will follow this prefix
  - In other cases, the octet string is encoded into one or more fragments; each fragment (or the only fragment) will contain at most 65536 octets and will be preceded by a length prefix encoded in a special way

Sequence types
PER encoding summary

- In PER Unaligned, a component whose type is a sequence type (with no extension marker) is encoded as follows:
    1. A bitmap is added that has one bit for each component of the sequence type that is declared as `OPTIONAL` or `DEFAULT`. Each bit of the bitmap indicates whether the corresponding component is present
    2. Each component of the sequence type is encoded (in textual order). Any optional component that is not present in the value of the sequence type is just skipped.

Choice types
PER encoding summary

- In PER Unaligned, a component whose type is a choice type (with no extension marker) is encoded as follows:
  1. An index, starting from zero and increasing by one, is assigned to each alternative of the choice type (in textual order)
  2. The index of the chosen alternative is encoded into the minimum number of bits capable of expressing the largest possible such index (possibly zero)
  3. The chosen alternative of the choice type is encoded

Sequence-of types
PER encoding summary

- In PER Unaligned, a component whose type is a sequence-of type (with no extension marker) is encoded as follows:
  - If the sequence-of type has a fixed length that is less than 65536, then the components of the sequence-of value are encoded without any length prefix
  - If the sequence-of type has a variable length and the length's upper bound is less than 65536, then the length's lower bound is subtracted from the length of the sequence-of value, and the difference is encoded into the minimum number of bits capable of expressing the largest possible such difference; the components of the sequence-of are encoded after this prefix
  - In other cases, the sequence-of is encoded into one or more fragments; each fragment (or the only fragment) will contain at most 65536 components and will be preceded by a length prefix encoded in a special way

Null types
PER encoding summary

- In PER Unaligned, a component whose type is a null type is not encoded

Extensibility
PER encoding summary

- In PER Unaligned, the encodings of types that include an extension marker are modified as follows (1/4):
  - An extension bit is included before the first bit of the encoding of the value, indicating whether the value being encoded is a “root” value or an “extension addition” value
  - Aside from the presence of the extension bit, root values are encoded exactly as if the type were non-extensible
  - The encoding of extension values is often less compact than the encoding of root values, but the rules ensure that any extension values that may be legally added to the type definition in a future version will be encodable

Extensibility
PER encoding summary

- In PER Unaligned, the encodings of types that include an extension marker are modified as follows (2/4):
  - For an extensible integer type, if the value is outside the bounds of the root, the value is encoded in a way that can represent any integer with no bounds
  - For an extensible enumerated type, if the chosen enumeration is beyond the last root enumeration, the enumeration index is encoded in a way that can represent any non-negative integer with no upper bound
  - For an extensible bit string, octet string, or sequence-of type, if the length of the value exceeds the upper bound of the root length, the length is encoded in a way that can represent any non-negative integer with no upper bound

Extensibility
PER encoding summary

- In PER Unaligned, the encodings of sequence types that include an extension marker are modified as follows:
  - Each extension addition is separately “wrapped” in a structure very similar to a variable-length octet string
  - A bitmap is included before the first extension addition, indicating which extension additions (defined in a later version) are present
  - The length prefix of the wrapper allows an earlier-version implementation to skip over the encodings of any extension additions it does not understand

Extensibility
PER encoding summary

- In PER Unaligned, the encodings of choice types that include an extension marker are modified as follows:
  - If the chosen alternative is beyond the last root alternative, the choice index is encoded in a way that can represent any non-negative integer with no upper bound
  - The encoding of an extension alternative is “wrapped” in a structure very similar to a variable-length octet string
  - The length prefix of the wrapper allows an earlier-version implementation to skip over the encoding of an extension alternative it does not understand

Thank you!

Alessandro Triglia
sandro@oss.com
OSS Nokalva, Inc.