INTERNATIONAL STANDARD ISO/IEC 8825-3:2008
TECHNICAL CORRIGENDUM 1

Published 2012-12-01

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION • МЕЖДУНАРОДНАЯ ОРГАНИЗАЦИЯ ПО СТАНДАРТИЗАЦИИ • ORGANISATION INTERNATIONALE DE NORMALISATION
INTERNATIONAL ELECTROTECHNICAL COMMISSION • МЕЖДУНАРОДНАЯ ЭЛЕКТРОТЕХНИЧЕСКАЯ КОМИССИЯ • COMMISSION ÉLECTROTECHNIQUE INTERNATIONALE

Information technology — ASN.1 encoding rules: Specification of Encoding Control Notation (ECN)

TECHNICAL CORRIGENDUM 1

Technologies de l'information — Règles de codage ASN.1: Spécification de la notation de contrôle de codage (ECN)

RECTIFICATIF TECHNIQUE 1

Technical Corrigendum 1 to ISO/IEC 8825-3:2008 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 6, *Telecommunications and information exchange between systems*, in collaboration with ITU-T. The identical text is published as Rec. ITU-T X.692 (2008)/Cor.1 (10/2011).

ICS 35.100.60 **Ref. No. ISO/IEC 8825-3:2008/Cor.1:2012(E)**

Published in Switzerland

CONTENTS

INTERNATIONAL STANDARD

RECOMMENDATION ITU-T

Information technology – ASN.1 encoding rules: Specification of Encoding Control Notation (ECN)

Conventions used in this corrigendum: Original, unchanged, text is in normal font. *Deleted text is struck-through, thus:* ~~deleted text~~. *Inserted text is underlined, thus:* inserted text.

1 Clause 2

Add a new NOTE to the first paragraph as follows:

NOTE – This Recommendation | International Standard is based on ISO/IEC 10646:2003. It cannot be applied using later versions of this standard.

2 Clause 18.2.3

Change the existing NOTE after 18.2.3 to NOTE 2 modified as follows:

NOTE 2 – An enco~~n~~ding object for a user-defined or implicitly-generated encoding class can be added to such a set, and will take precedence over any encoding which could be obtained by de-referencing.

Add the following new NOTE 1:

NOTE 1 – The encoding objects of the encoding object sets BER, CER, DER do not carry an implied alignment to the next multiple of 8 bits. The encoding objects of the encoding objects of the encoding object sets PER-BASIC-ALIGNED and PER-CANONICAL-ALIGNED do carry an implied alignment to the next multiple of 8 bits only when required by ITU-T Rec. X.691 | ISO/IEC-8825-2

3 Annex D

A number of changes of the ASN.1 and ECN specifications are required .There are also a large number of indentation changes needed. These are not listed separately, instead a complete replacement for Annex D is provided. Replace the whole of Annex D with:

Annex D

Examples

(This annex does not form an integral part of this Recommendation | International Standard)

This annex contains examples of the use of ECN. The examples are divided into five groups:

- General examples, which show the look-and-feel of ECN definitions (D.1).
- Specialization examples, which show how to modify some parts of a standardized encoding. Each example has a description of the requirements for the encoding and a description of the selected solution and possible alternative solutions (D.2).
- Explicitly generated structure examples, which show the use of explicitly generated structures when the same specialized encoding is used several times (D.3).
- A legacy protocol example which shows three ways of handling the problem of a traditional "more-bit" approach to sequence-of termination (D.4).
- A second legacy protocol example, which shows how to construct ECN definitions for a protocol whose message encodings have been specified using a tabular notation (D.5).

D.1 General examples

The examples described in D.1.1 to D.1.14 are part of a complete ECN specification whose ASN.1, EDM, and ELM modules are given in outline in D.1.15, D.1.16 and D.1.17, and are given completely in a copy of this annex which is available from the website cited in Annex F.

D.1.1 An encoding object for a boolean type

D.1.1.1 The ASN.1 assignment is:

```
Married ::= BOOLEAN
```

D.1.1.2 The encoding object assignment (see 23.3.1) is:

```
booleanEncoding #BOOLEAN ::= {
ENCODING-SPACE
        SIZE 1
            MULTIPLE OF bit
TRUE-PATTERN bits:'1'B
FALSE-PATTERN bits:'0'B}
marriedEncoding-1 #Married ::= booleanEncoding
```

D.1.1.3 There is no pre-alignment, and the encoding space is one bit, so "**Married**" is encoded as a bit-field of length 1. Patterns for **TRUE** and **FALSE** values (in this case a single bit) are '1'B and '0'B respectively.

D.1.1.4 The values specified above are the values that would be set by default (see 23.3.1) if the corresponding encoding properties were omitted, so the same encoding can be achieved with less verbosity by:

```
marriedEncoding-2 #Married ::= {
ENCODING-SPACE
SIZE 1}
```

D.1.1.5 This encoding for a boolean is, of course, just what PER provides, and another alternative is to specify the encoding using the PER encoding object for boolean by way of the syntax provided by 17.3.1.

```
marriedEncoding-3 #Married ::= {
ENCODE WITH PER-BASIC-UNALIGNED}
```

D.1.1.6 As these examples show, there are often cases where ECN provides multiple ways to define an encoding. It is up to the user to decide which alternative to use, balancing verbosity (stating explicitly values that can be defaulted) against readability and clarity.

D.1.2 An encoding object for an integer type

D.1.2.1 The ASN.1 assignments are:

```
EvenPositiveInteger ::= INTEGER (1..MAX) (CONSTRAINED  BY {-- Must be even --})
EvenNegativeInteger ::= INTEGER (MIN..-1) (CONSTRAINED BY {-- Must be even --})
```

D.1.2.2 The encoding object assignments are:

```
evenPositiveIntegerEncoding #EvenPositiveInteger ::= {
USE #NonNegativeInt
MAPPING TRANSFORMS {{INT-TO-INT divide:2}}
WITH PER-BASIC-UNALIGNED}
#NonNegativeInt ::= #INT(0..MAX)
evenNegativeIntegerEncoding #EvenNegativeInteger ::= {
USE #NonPositiveInt
        MAPPING TRANSFORMS {{INT-TO-INT divide:2                    --
        Note: -1 / 2 = 0 - see clause 24.3.7 -- }}
WITH PER-BASIC-UNALIGNED}
#NonPositiveInt ::= #INT(MIN..0)
```

D.1.2.3 An even value is divided by two, and is then encoded using standardized PER encoding rules for positive and negative integer types.

D.1.3 Another encoding object for an integer type

D.1.3.1 Here we assume the requirement to define an encoding object which encodes an integer in a two-octet field starting at an octet boundary.

D.1.3.2 The ASN.1 assignment is:

```
Altitude ::= INTEGER (0..65535)
```

D.1.3.3 The Encoding object assignment (see 23.6.1 and 23.7.1) is:

```
integerRightAlignedEncoding #Altitude ::= {
ENCODING {
        ALIGNED TO NEXT octet
        ENCODING-SPACE
                SIZE 16}}
```

D.1.4 An encoding object for an integer type with holes

D.1.4.1 The ASN.1 assignment is:

```
IntegerWithHole ::= INTEGER (-256..-1 | 32..1056)
```

D.1.4.2 The encoding object assignment (see 19.5.2) is:

```
integerWithHoleEncoding #IntegerWithHole ::= {
USE #IntFrom0To1280
MAPPING ORDERED VALUES
WITH PER-BASIC-UNALIGNED}
#IntFrom0To1280 ::= #INT (0..1280)
```

D.1.4.3 "`IntegerWithHole`" is encoded as a positive integer. Values in the range -256..-1 are mapped to values in the range 0..255 and values in the range 32..1056 are mapped to 256..1280.

D.1.5 A more complex encoding object for an integer type

D.1.5.1 The ASN.1 assignments are:

```
PositiveInteger ::= INTEGER (1..MAX)
NegativeInteger ::= INTEGER (MIN..-1)
```

D.1.5.2 The encoding object assignments are:

```
positiveIntegerEncoding #PositiveInteger ::=
integerEncoding
negativeIntegerEncoding #NegativeInteger ::=
integerEncoding
```

D.1.5.3 Values of "**PositiveInteger**" and "**NegativeInteger**" types are encoded by the encoding object "**integerEncoding**" as a positive integer or as a twos-complement integer respectively. This is defined below, and provides different encodings depending on the bounds of the type to which it is applied.

D.1.5.4 The "**integerEncoding**" encoding object defined here is very powerful, but quite complex. It contains five encoding objects of the class **#CONDITIONAL-INT**; they all define an octet-aligned encoding. When the integer values being encoded are bounded, the number of bits is fixed; when the values are not bounded, the type is required to be the last in a PDU, and the value is right justified in the remaining octets of the PDU.

D.1.5.5 The definition of the encoding object (see 23.6.1 and 23.7.1) is:

```
integerEncoding #INT ::= {ENCODINGS {
{ IF unbounded-or-no-lower-bound
        ENCODING-SPACE
            SIZE variable-with-determinant
            DETERMINED BY container
            USING OUTER
        ENCODING twos-complement} ,
{ IF bounded-with-negatives
        ENCODING-SPACE
            SIZE fixed-to-max
        ENCODING twos-complement} ,
{ IF semi-bounded-with-negatives
        ENCODING-SPACE
            SIZE variable-with-determinant
            DETERMINED BY container
            USING OUTER
        ENCODING twos-complement} ,
{ IF semi-bounded-without-negatives
        ENCODING-SPACE
            SIZE variable-with-determinant
            DETERMINED BY container
            USING OUTER
        ENCODING positive-int} ,
{ IF bounded-without-negatives
        ENCODING-SPACE
            SIZE fixed-to-max
        ENCODING positive-int}}}
```

D.1.6 Positive integers encoded in BCD

D.1.6.1 This example shows how to encode a positive integer in BCD (Binary Coded Decimal) by successive transforms: from integer to character string then from character string to bitstring.

D.1.6.2 The ASN.1 assignment is:

```
PositiveIntegerBCD ::= INTEGER(0..MAX)
```

D.1.6.3 The encoding object assignment (see 19.4, 24.1 and 23.4.1) is:

```
positiveIntegerBCDEncoding #PositiveIntegerBCD ::= {
USE #CHARS
MAPPING TRANSFORMS{{
        INT-TO-CHARS
        -- We convert to characters (e.g., integer 42
            -- becomes character string "42") and encode the characters
            -- with the encoding object "numeric-chars-to-bcdEncoding"
                SIZE variable
                PLUS-SIGN FALSE}}
            WITH numeric-chars-to-bcdEncoding }
numeric-chars-to-bcdEncoding #CHARS ::= {
ALIGNED TO NEXT nibble
        TRANSFORMS {{
        CHAR-TO-BITS
        -- We convert each character to a bitstring
        --(e.g., character "4" becomes '0100'B and "2" becomes
        -- '0010'B)
        AS mapped
            CHAR-LIST { "0","1","2","3",
                        "4","5","6","7",
```

```
                "8","9"}
        BITS-LIST { '0000'B, '0001'B, '0010'B, '0011'B,
                '0100'B, '0101'B, '0110'B, '0111'B,
                '1000'B, '1001'B }}}
REPETITION-ENCODING {
        REPETITION-SPACE
        -- We determine the concatenation of the bitstrings for the
        -- characters and add a terminator (e.g.,
        -- '0100'B + '0010'B becomes '0100 0010 1111'B)
            SIZE variable-with-determinant
            DETERMINED BY pattern
            PATTERN bits:'1111'B}}
```

D.1.6.4 The positive number is first transformed into a character string by the int-to-chars transform using the options variable length and no plus sign, and in addition the default option of no padding, giving a string containing characters "0" to "9". Then the character string is encoded such that each character is transformed into a bit pattern, '0000'B for "0", '0001'B for "1"…, '1001'B for "9". The bitstring is aligned on a nibble boundary and terminates with a specific pattern '1111'B.

D.1.6.5 A more complex alternative, not shown here, but commonly used, would be to embed the BCD encoding in an octet string, with an external boolean identifying whether there is an unused nibble at the end or not.

D.1.7 An encoding object of class #BITS

D.1.7.1 This example defines an encoding object of class **#BITS** (see 23.2.1) for a bitstring that is octet-aligned, padded with 0, and terminated by an 8-bit field containing '00000000'B (it is assumed that an abstract value never contains eight successive zeros):

D.1.7.2 The ASN.1 assignment is:

```
Fax ::= BIT STRING (CONSTRAINED BY
{-- must not contain eight successive zero bits --})
```

D.1.7.3 The encoding object assignment (see 23.2.1, 23.13.1 and 23.14.1) is:

```
faxEncoding #Fax ::= {
ALIGNED TO NEXT octet
REPETITION-ENCODING {
        REPETITION-SPACE
            SIZE variable-with-determinant
            DETERMINED BY pattern
            PATTERN bits:'00000000'B}}
```

D.1.7.4 This encoding object (of class **#BITS**) contains an embedded encoding object of class **#CONDITIONAL-REPETITION** which specifies the mechanism and the termination pattern.

D.1.7.5 As with many of the examples in this annex, there is heavy reliance here on the defaults provided in clause 23 and advantage is taken of the ability to define encoding objects in-line rather than separately assigning them to reference names which are then used in other assignments.

D.1.8 An encoding object for an octetstring type

D.1.8.1 The ASN.1 assignment is:

```
BinaryFile ::= OCTET STRING
```

D.1.8.2 The encoding object assignment (see 23.9.1) is:

```
binaryFileEncoding #BinaryFile ::= {
ALIGNED TO NEXT octet
PADDING one
REPETITION-ENCODING {
        REPETITION-SPACE
            SIZE variable-with-determinant
            DETERMINED BY container
            USING OUTER}}
```

D.1.8.3 The value is octet-aligned using padding with ones and terminates with the end of the PDU.

D.1.9 An encoding object for a character string type

D.1.9.1 The ASN.1 assignment is:

```
Password ::= PrintableString
```

D.1.9.2 The encoding object assignment (see 23.4.1 and 23.14.1) is:

```
passwordEncoding #Password ::= {
ALIGNED TO NEXT octet
TRANSFORMS {{CHAR-TO-BITS AS compact
        SIZE fixed-to-max
            MULTIPLE OF bit }}
REPETITION-ENCODING {
        REPETITION-SPACE
            SIZE variable-with-determinant
            DETERMINED BY container
            USING OUTER}}
```

D.1.9.3 The string is octet-aligned using padding with "0" and terminates with the end of the PDU; the character-encoding is specified as "`compact`", so each character is encoded in 7 bits using `'0000000'B` for the first ASCII character of type `PrintableString`, `'0000001'B` for the next, and so on.

D.1.10 Mapping character values to bit values

D.1.10.1 The ASN.1 assignment is:

```
CharacterStringToBit ::= IA5String ("FIRST" | "SECOND" | "THIRD")
```

D.1.10.2 The encoding object assignment (see 19.2) is:

```
characterStringToBitEncoding #CharacterStringToBit ::= {
USE #IntFrom0To2
MAPPING VALUES {
        "FIRST" TO 0,
        "SECOND"    TO 1,
        "THIRD"TO 2}
WITH integerEncoding}
#IntFrom0To2 ::= #INT (0..2)
```

where "integerEncoding" is defined in D.1.5.5.

D.1.10.3 The three possible abstract values are mapped to three integer numbers and then those numbers are encoded in a two-bit field.

D.1.11 An encoding object for a sequence type

D.1.11.1 Here we encode a sequence type that has a field "`a`" which carries application semantics (i.e., is visible to the application), but we also want to use it as a presence determinant for a second (optional) integer field "`b`". There is then an octet string that is octet-aligned, and delimited by the end of the PDU. We need to give specialized encodings for the optionality of "`b`", and we use the specialized encoding defined in D.1.8 (by reference to the encoding object "binaryFileEncoding") for the octet string "`c`". We want to encode everything else with PER basic unaligned.

D.1.11.2 The ASN.1 assignment is:

```
Sequence1       ::= SEQUENCE {
a       BOOLEAN,
b       INTEGER OPTIONAL,
c       BinaryFile
-- "BinaryFile" is defined in D.1.8.1 --}
```

D.1.11.3 The ECN assignments (see 17.5 and 23.11.1) are:

```
sequence1Encoding #Sequence1 ::= {
ENCODE STRUCTURE {
        b USE-SET OPTIONAL-ENCODING parameterizedPresenceEncoding {< a >},
        c binaryFileEncoding
        -- "binaryFileEncoding" is defined in D.1.8.2 -- }
        WITH PER-BASIC-UNALIGNED}
parameterizedPresenceEncoding {< REFERENCE:reference >}  #OPTIONAL ::= {
PRESENCE
```

```
        DETERMINED BY field-to-be-used
        USING reference}
```

D.1.11.4 Notice that we did not need to provide the "**DECODERS-TRANSFORMS**" encoding property in the "**parameterizedPresenceEncoding**" encoding object, because the component "**a**" was a boolean, and it is assumed that **TRUE** meant that "b" was present. If, however, "**a**" had been an integer field, or if the application value of **TRUE** for "**a**" actually meant that "**b**" was absent, then we would have included a "**DECODER-TRANSFORMS**" encoding property as in D.2.6.

D.1.12 An encoding object for a choice type

D.1.12.1 A choice type with three alternatives is encoded using the tag number of class context, encoded in a three bit field, as a selector. The encoding object of class **#ALTERNATIVES** specify that the identification handle "**Tag**" is used as determinant; the encoding object of class **#TAG** defines the position of the identification handle (three bits). For each alternative, the value is encoded with PER basic unaligned.

D.1.12.2 The ASN.1 assignment is:

```
Choice ::= CHOICE {
boolean   [1]  BOOLEAN,
integer   [3]  INTEGER,
string    [5]  IA5String}
```

D.1.12.3 The ECN assignments (see 23.1.1 and 23.15.1) are:

```
choiceEncoding-1      #Choice ::= {
ENCODE STRUCTURE {
        boolean   [tagEncoding] USE-SET,
        integer   [tagEncoding] USE-SET,
        string    [tagEncoding] USE-SET
        STRUCTURED WITH {
            ALTERNATIVE
                DETERMINED BY handle
                    HANDLE "Tag"}}
            WITH PER-BASIC-UNALIGNED}
tagEncoding #TAG ::= {
ENCODING-SPACE
        SIZE 3
            MULTIPLE OF bit
EXHIBITS HANDLE "Tag" AT {0 | 1 | 2}}
```

D.1.12.4 Perhaps a neater way of providing the first assignment in D.1.12.3 would be to define a new encoding object set and apply it as follows:

```
MyEncodings #ENCODINGS ::= { tagEncoding } COMPLETED BY PER-BASIC-UNALIGNED
choiceEncoding-2      #Choice ::= {
ENCODE STRUCTURE {
        STRUCTURED WITH {
            ALTERNATIVE
                DETERMINED BY handle
                    HANDLE "Tag"}}
        WITH MyEncodings}
```

D.1.13 Encoding a bitstring containing another encoding

D.1.13.1 A bitstring value encoded with PER basic unaligned, contains the PER basic unaligned encoding of a sequence as an integral number of octets (padded with zeros) but not necessarily aligned on an octet boundary.

D.1.13.2 The ASN.1 assignment are:

```
Sequence2 ::= SEQUENCE {
a         BOOLEAN,
b         BIT STRING (CONTAINING Sequence3) }
Sequence3 ::= SEQUENCE {
a         INTEGER(0..10),
b         BOOLEAN }
```

D.1.13.3 The ECN assignments (see 25.1) are:

```
sequence2Encoding #Sequence2 ::= {
ENCODE STRUCTURE {
        b    { REPETITION-ENCODING {
                REPETITION-SPACE
                    SIZE 8
                        MULTIPLE OF bit}
             CONTENTS-ENCODING {sequence3Encoding}
COMPLETED BY PER-BASIC-UNALIGNED}}
WITH PER-BASIC-UNALIGNED}
sequence3Encoding #Sequence3 ::= {
ENCODE STRUCTURE {
        STRUCTURED WITH sequence3StructureEncoding
        }
WITH PER-BASIC-UNALIGNED }
sequence3StructureEncoding #CONCATENATION ::= {
ENCODING-SPACE
        MULTIPLE OF octet
        VALUE-PADDING
            JUSTIFIED left:0
            POST-PADDING zero
            UNUSED BITS
                DETERMINED BY not-needed }
```

D.1.14 An encoding object set

These encoding object sets contain encoding definitions for some types specified in the ASN.1 module of D.1.15.

```
Example1Encodings #ENCODINGS ::= {
marriedEncoding-1             |
integerRightAlignedEncoding   |
evenPositiveIntegerEncoding   |
evenNegativeIntegerEncoding   |
integerRightAlignedEncoding   |
integerWithHoleEncoding       |
positiveIntegerEncoding       |
negativeIntegerEncoding       |
positiveIntegerBCDEncoding    |
faxEncoding                   |
binaryFileEncoding            |
passwordEncoding              |
characterStringToBitEncoding  |
sequence1Encoding             |
choiceEncoding-1              |
sequence2Encoding             |
        sequence3Encoding }
```

D.1.15 ASN.1 definitions

D.1.15.1 This ASN.1 module groups all the ASN.1 definitions from D.1.1 to D.1.13 together. They will be encoded according to the encoding objects defined in the EDM of D.1.16, together with the PER basic unaligned encoding rules.

```
Example1-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module1(2)}
        DEFINITIONS AUTOMATIC TAGS ::=
        BEGIN
MyPDU ::= CHOICE {
marriedMessage          Married,
altitudeMessage         Altitude
-- etc.
}
Married ::= BOOLEAN
Altitude ::= INTEGER (0..65535)
-- etc.
END
```

D.1.16 EDM definitions

D.1.16.1 This EDM module groups all the ECN definitions from D.1.1 to D.1.13 together.

```
Example1-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module1(3)}
        ENCODING-DEFINITIONS ::=
        BEGIN
EXPORTS Example1Encodings;
IMPORTS #Married, #Altitude, #EvenPositiveInteger, #EvenNegativeInteger,
              #IntegerWithHole, #PositiveInteger, #NegativeInteger, #PositiveIntegerBCD,
              #Fax, #BinaryFile, #Password, #CharacterStringToBit, #Sequence1, #Choice
              FROM Example1-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5)
                                          asn1-module1(2) };
Example1Encodings #ENCODINGS ::= {
              marriedEncoding-1      |
              -- etc
              sequence3Encoding}

       -- etc

       END
```

D.1.17 ELM definitions

The following ELM encodes the ASN.1 module defined in D.1.15, using objects specified in the EDM defined in D.1.16.

```
Example1-ELM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) elm-module1(1)}
        LINK-DEFINITIONS ::=
        BEGIN

        IMPORTS
              Example1Encodings FROM Example-EDM
                    {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module1(3)}
              #MyPDU, #Sequence2 FROM Example1-ASN1-Module
                    {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module1(2)};
ENCODE #MyPDU WITH Example1Encodings
              COMPLETED BY PER-BASIC-UNALIGNED
END
```

D.2 Specialization examples

The examples in this clause show how to modify selected parts of an encoding for given types in order to minimize the size of encoded messages. PER basic unaligned encodings normally produce as compact encodings as possible. However, there are some cases when specialized encodings might be desired:

- There are some special semantics associated with message components that make it possible to remove some of the PER-generated auxiliary fields.
- The user wants different encodings for PER auxiliary fields that are generated by default, such as variable-width determinant fields.

D.2.1 Encoding by distributing values to an alternative encoding structure

D.2.1.1 The ASN.1 assignment is:

```
NormallySmallValues ::= INTEGER (0..1000)
                        -- Usually values are in the range 0..63, but sometimes the whole
              value range
                        -- is used.
```

D.2.1.2 PER would encode the type using 10 bits. We wish to minimize the size of the encoding such that the normal case is encoded using as few bits as possible.

NOTE – In this example we take a simple direct approach. A more sophisticated approach using Huffman encodings is given in E.1.

D.2.1.3 The encoding object assignment (see 19.6) is:

```
normallySmallValuesEncoding-1 #NormallySmallValues ::= {
USE       #NormallySmallValuesStruct-1
MAPPING DISTRIBUTION {
```

```
        0..63       TO small,
        REMAINDER TO large }
WITH PER-BASIC-UNALIGNED}
```

D.2.1.4 The encoding structure assignment is:

```
#NormallySmallValuesStruct-1 ::= #CHOICE {
small       #INT (0..63),
large       #INT (64..1000)}
```

D.2.1.5 Values which are normally used are encoded using the "**small**" field and the ones used only occasionally are encoded using the "**large**" field. The selection between the two is done by a one-bit PER-generated selector field. The length of the "**small**" field is 6 bits and the length of the "**large**" field is 10 bits, so the normal case is encoded using 7 bits and the rare case using 11 bits.

D.2.2 Encoding by mapping ordered abstract values to an alternative encoding structure

D.2.2.1 Example D.2.1 used explicit definition of how value ranges are mapped to fields of the encoding structure. The same effect can be achieved more simply by using "mapping by ordered abstract values". However, as illustration, we here also modify the requirement: Arbitrarily large values may occasionally occur, and the ASN.1 assignment is assumed to have its constraint removed.

D.2.2.2 The encoding object assignments (see 19.5) are:

```
normallySmallValuesEncoding-2 #NormallySmallValues ::= {
USE         #NormallySmallValuesStruct-2
MAPPING ORDERED VALUES
WITH        NormallySmallValuesTag-encoding-plus-PER}
normallySmallValuesTag-encoding #TAG ::= {
ENCODING-SPACE
        SIZE 1}
NormallySmallValuesTag-encoding-plus-PER #ENCODINGS ::=
{normallySmallValuesTag-encoding}
COMPLETED BY PER-BASIC-UNALIGNED
```

D.2.2.3 The encoding structure assignment is:

```
#NormallySmallValuesStruct-2 ::= #CHOICE {
small       [#TAG(0)]       #INT (0..63),
large       [#TAG(1)]       #INT (0..MAX) }
```

D.2.2.4 The result is very similar to D.2.1, but now the values above 64 that are mapped to the field "**large**" are encoded from zero upwards. The two alternatives are distinguished by an index of one bit. Another difference is that the field "large" is left unbounded, so the encoding object can encode arbitrarily large integers, but with the cost of a length field in the "**large**" case. This example can also be used if there is no upper-bound on the values that might occasionally occur ("**large**" is not bounded in the replacement structure). This again illustrates the flexibility available to ECN specifiers to design encodings to suite their particular requirements.

D.2.3 Compression of non-continuous value ranges

D.2.3.1 This example also uses a mapping of ordered abstract values. In this case the mapping is used to compress sparse values in a base ASN.1 specification. The compression could also have been achieved by defining the ASN.1 abstract value "x" to have the application semantics of "2x", then using a simpler constraint on the ASN.1 integer type. The assumption in this example, however, is that the ASN.1 designer chose not to do that, and we are required to apply the compression during the mapping from abstract values to encodings.

D.2.3.2 The ASN.1 assignment is:

```
SparseEvenlyDistributedValueSet ::= INTEGER (2 | 4 | 6 | 8 | 10 | 12 | 14 | 16)
```

D.2.3.3 PER basic unaligned takes only lower bounds and upper bounds into account when determining the number of bits needed to encode an integer. This results in unused bit patterns in the encoding. The encoding can be compressed such that unused bit patterns are omitted, and each value is encoded using the minimum number of bits.

D.2.3.4 The encoding object assignment (see 19.5) is:

```
sparseEvenlyDistributedValueSetEncoding-1 #SparseEvenlyDistributedValueSet ::= {
USE         #IntFrom0To7
MAPPING ORDERED VALUES
WITH PER-BASIC-UNALIGNED}
```

```
#IntFrom0To7 ::= #INT (0..7)
```

D.2.3.5 The eight possible abstract values have been mapped to the range 0..7 and will be encoded in a three-bit field.

D.2.4 Compression of non-continuous value ranges using a transform

D.2.4.1 Example D.2.3 used mapping of ordered abstract values. The same effect can be achieved by using the **#TRANSFORM** class.

D.2.4.2 The encoding object assignment (see 19.4) is:

```
sparseEvenlyDistributedValueSetEncoding-2 #SparseEvenlyDistributedValueSet ::= {
USE     #IntFrom0To7
MAPPING TRANSFORMS {{INT-TO-INT divide: 2}, {INT-TO-INT decrement:1}}
WITH PER-BASIC-UNALIGNED}
```

D.2.4.3 Again, the eight possible abstract values are mapped to the range **0..7** and encoded in a three-bit field.

D.2.5 Compression of an unevenly distributed value set by mapping ordered abstract values

D.2.5.1 The ASN.1 assignment is:

```
SparseUnevenlyDistributedValueSet ::= INTEGER (0|3|5|6|11|8)
-- Out of order to illustrate that order does not matter in the constraint
```

D.2.5.2 The encoding should be such that there are no holes in the encoding patterns used.

D.2.5.3 The encoding object assignment is:

```
sparseUnevenlyDistributedValueSetEncoding #SparseUnevenlyDistributedValueSet ::= {
USE     #IntFrom0To5
MAPPING ORDERED VALUES
WITH PER-BASIC-UNALIGNED}

        #IntFrom0To5 ::= #INT (0..5)
```

D.2.5.4 The six possible abstract values are mapped to the range **0..5** and encoded in a three-bit field. The mapping is as follows: 0→0 , 3→1, 5→2, 6→3, 8→4, and 11→5.

D.2.6 Presence of an optional component depending on the value of another component

D.2.6.1 The ASN.1 assignment is:

```
ConditionalPresenceOnValue ::= SEQUENCE {
a       INTEGER (0..4),
b       INTEGER (1..10),
c       BOOLEAN OPTIONAL
-- Condition: "c" is present if "a" is 0, otherwise "c" is absent --,
d       BOOLEAN OPTIONAL
-- Condition: "d" is absent if "a" is 1, otherwise "d" is present -- }
-- Note the implied presence constraints in comments.
-- Note also that the integer field "a" carries application semantics and
-- has values other than zero and one.
-- If "a" has value 0, both "c" and "d" are present.
-- If "a" has value 1, both "c" and "d" are missing.
-- If "a" has values 3 or 4, "c" is absent and "d" is present.
-- These conditions are very hard to express formally using ASN.1 alone.
```

D.2.6.2 The component "**a**" acts as the presence determinant for both components "**c**" and "**d**", but a PER encoding would produce two auxiliary bits for the optional components. We require an encoding in which these auxiliary bits are absent.

D.2.6.3 The encoding object assignment is:

```
conditionalPresenceOnValueEncoding #ConditionalPresenceOnValue ::= {
ENCODE STRUCTURE {
        c   USE-SET OPTIONAL-ENCODING is-c-present{< a >},
        d   USE-SET OPTIONAL-ENCODING is-d-present{< a >}}
WITH PER-BASIC-UNALIGNED}
is-c-present {<  REFERENCE : a  >} #OPTIONAL ::= {
PRESENCE
        DETERMINED BY field-to-be-used
```

```
        USING a
            DECODER-TRANSFORMS {{INT-TO-BOOL  TRUE-IS {0}}}}
is-d-present {< REFERENCE : a >} #OPTIONAL ::= {
PRESENCE
        DETERMINED BY field-to-be-used
        USING a
            DECODER-TRANSFORMS {{INT-TO-BOOL TRUE-IS {0 | 2 | 3 | 4}}}}
```

D.2.6.4 Here we have a simple, formal, and clear specification of the presence conditions on "**c**" and "**d**" which can be understood by encoder-decoder tools. The ASN.1 comments cannot be handled by tools. The provision of optionality encoding for "**c**" and "**d**" means that the PER encoding for **OPTIONAL** is not used in this case, and there are no auxiliary bits.

D.2.6.5 The parameterized encoding objects "**is-c-present**" and "**is-d-present**" specify how presence of the components is determined during decoding. Note that no transformation is needed (nor permitted) for encoding because the determinant has application semantics (i.e., it is visible in the ASN.1 type definition). However, a good encoding tool will police the setting of "**a**" by the application, to ensure that its value is consistent with the presence or absence of "**c**" and "**d**" that the application code has determined.

D.2.7 The presence of an optional component depends on some external condition

D.2.7.1 The ASN.1 assignment is:

```
ConditionalPresenceOnExternalCondition ::= SEQUENCE {
a       BOOLEAN OPTIONAL
        -- Condition: "a" is present if the external condition "C" holds,
        -- otherwise "a" absent -- }
-- Note that the presence constraint can only be supplied in comment.
```

D.2.7.2 The application code for both a sender and a receiver can evaluate the condition "C" from some information outside the message. The ECN specifier wishes tools to invoke such code to determine the presence of "**a**", rather than using a bit in the encoding.

D.2.7.3 The encoding object assignment is:

```
conditionalPresenceOnExternalConditionEncoding
#ConditionalPresenceOnExternalCondition ::= {
ENCODE STRUCTURE {
        a     USE-SET OPTIONAL-ENCODING is-a-present}
WITH PER-BASIC-UNALIGNED}
is-a-present #OPTIONAL ::=
NON-ECN-BEGIN {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) user-notation(7)}
extern C;
extern channel;
/* a is present only if channel is equal to some value "C" */
int is_a_present() {
        if(channel == C) return 1;
        else return 0; }
NON-ECN-END
```

D.2.7.4 Because the condition is external to the message, the encoding object for determining presence of the component "a" can only be specified by a non-ECN definition of an encoding object. However, while this saves bits on the line, many designers would consider it better to include the bit in the message to reduce the possibility of error, and to make testing and monitoring easier. Such choices are for the ECN specifier.

D.2.8 A variable length list

D.2.8.1 The ASN.1 assignment is:

```
EnclosingStructureForList ::= SEQUENCE {
list       VariableLengthList}
VariableLengthList ::= SEQUENCE (SIZE (0..1023) ) OF INTEGER (1..2)
       -- Normally the list contains only a few elements (0..31),
       -- but it might contain many.
```

D.2.8.2 PER basic unaligned encodes the length of the list using 10 bits even if normally the length is in the range **0..31**. We wish to minimize the size of the encoding of the length determinant in the normal case while still allowing values which rarely occur.

D.2.8.3 The encoding object assignment is:

```
enclosingStructureForListEncoding #EnclosingStructureForList ::= {
USE     #EnclosingStructureForListStruct
MAPPING FIELDS WITH {
        ENCODE STRUCTURE {
            aux-length      list-lengthEncoding,
            list  {
                ENCODE STRUCTURE {
                    STRUCTURED WITH {
                    REPETITION-ENCODING {
                        REPETITION-SPACE
                            SIZE variable-with-determinant
                                MULTIPLE OF repetitions
                            DETERMINED BY field-to-be-set
                            USING aux-length}}}
                WITH PER-BASIC-UNALIGNED }}
        WITH PER-BASIC-UNALIGNED}}
        -- First mapping: use of an encoding structure with an explicit length
        -- determinant.
list-lengthEncoding #AuxVariableListLength ::= {
USE     #AuxVariableListLengthStruct        -- See D.2.8.4.
MAPPING ORDERED VALUES
WITH PER-BASIC-UNALIGNED}
        -- Second mapping: list length is encoded as a choice between
        -- a short form "normally" and a long form "sometimes".
```

D.2.8.4 The encoding structure assignments are:

```
#EnclosingStructureForListStruct ::= #CONCATENATION {
aux-length      #AuxVariableListLength,
list        #VariableLengthList}
#AuxVariableListLength ::= #INT (0..1023)
#AuxVariableListLengthStruct ::= #ALTERNATIVES {
normally  #INT (0..31),
sometimes       #INT (32..1023)}
```

D.2.8.5 The length determinant for the component "**list**" is variable. The length determinant for short list values is encoded using 1 bit for the selection determinant and 5 bits for the length determinant. The length determinant for long list values is encoded using 1 bit for the selection determinant and 10 bits for the length determinant.

D.2.9 Equal length lists

D.2.9.1 The ASN.1 assignment is:

```
EqualLengthLists ::= SEQUENCE {
list1       List1,
list2       List2}
(CONSTRAINED BY {
        -- "list1" and "list2" always have the same number of elements. --
})
List1 ::= SEQUENCE (SIZE (0..1023)) OF BOOLEAN
List2 ::= SEQUENCE (SIZE (0..1023)) OF INTEGER (1..2)
```

D.2.9.2 Both "**list1**" and "**list2**" have the same number of elements, and the ECN specifier wishes to use a single length determinant for both lists. (PER would encode length fields for both components.)

D.2.9.3 The encoding object assignments are:

```
equalLengthListsEncoding #EqualLengthLists ::= {
USE     #EqualLengthListsStruct
MAPPING FIELDS
WITH {
        ENCODE STRUCTURE {
            list1  list1Encoding{< aux-length >},
            list2  list2Encoding{< aux-length >}}
WITH PER-BASIC-UNALIGNED}}
```

The first encoding object is defined with two parameterized encoding objects of classes **#List1** and **#List2** respectively using the length field as an actual parameter. Those two encoding objects use a common parameterized encoding object of class **#REPETITION**.

```
list1Encoding {< REFERENCE : length >} #List1 ::= {
ENCODE STRUCTURE { USE-SET
        STRUCTURED WITH list-with-determinantEncoding {< length >}}
WITH PER-BASIC-UNALIGNED}
list2Encoding {< REFERENCE : length >} #List2 ::= {
ENCODE STRUCTURE { USE-SET
        STRUCTURED WITH list-with-determinantEncoding {< length >}}
WITH PER-BASIC-UNALIGNED}
list-with-determinantEncoding {< REFERENCE : length-determinant >} #REPETITION ::= {
REPETITION-ENCODING {
        REPETITION-SPACE
            SIZE variable-with-determinant
                MULTIPLE OF repetitions
            DETERMINED BY field-to-be-set
            USING length-determinant}}
```

D.2.9.4 The encoding structure assignments are:

```
#EqualLengthListsStruct ::= #CONCATENATION {
aux-length      #AuxListLength,
list1           #List1,
list2           #List2}
#AuxListLength ::= #INT (0..1023)
```

D.2.10 Uneven choice alternative probabilities

D.2.10.1 The ASN.1 assignment is:

```
EnclosingStructureForChoice ::= SEQUENCE {
choice          UnevenChoiceProbability }
UnevenChoiceProbability ::= CHOICE {
frequent1 INTEGER (1..2),
frequent2 BOOLEAN,
common1 INTEGER (1..2),
common2 BOOLEAN,
common3 BOOLEAN,
rare1           BOOLEAN,
rare2           INTEGER (1..2),
rare3           INTEGER (1..2)}
```

D.2.10.2 The alternatives of the choice type have different selection probabilities. There are alternatives which appear very frequently ("`frequent1`" and "`frequent2`"), or are fairly common ("`common1`", "`common2`" and "`common3`"), or appear only rarely ("`rare1`", "`rare2`" and "`rare3`"). The encoding for the alternative determinant should be such that those alternatives that appear frequently have shorter determinant fields than those appearing rarely.

D.2.10.3 The encoding structure assignments are:

```
#EnclosingStructureForChoiceStruct ::= #CONCATENATION {
aux-selector    #AuxSelector,
choice          #UnevenChoiceProbability }
-- Explicit auxiliary alternative determinant for "choice".
#AuxSelector ::= #INT (0..7)
```

D.2.10.4 The encoding object assignments are:

```
enclosingStructureForChoiceEncoding #EnclosingStructureForChoice ::= {
USE     #EnclosingStructureForChoiceStruct
MAPPING FIELDS
WITH {
        ENCODE STRUCTURE {
            aux-selector    auxSelectorEncoding,
            choice {
                ENCODE STRUCTURE  {
                    STRUCTURED WITH  {
                        ALTERNATIVE
                            DETERMINED BY field-to-be-set
                            USING aux-selector}}
                WITH PER-BASIC-UNALIGNED }}
```

```
WITH PER-BASIC-UNALIGNED} }
-- First mapping: inserts an explicit auxiliary alternative
-- determinant.
-- This encoding object specifies that an auxiliary determinant is used
-- as an alternative determinant.
auxSelectorEncoding #AuxSelector ::= {
                USE         #BITS
                -- ECN Huffman
                -- RANGE (0..7)
                -- (0..1) IS 60%
                -- (2..4) IS 30%
                -- (5..7) IS 10%
                -- End Definition
                -- Mappings produced by "ECN Public Domain Software for Huffman encodings,
                -- version 1"
                -- (see E.8)
                MAPPING TO BITS {
                        0 ..  1 TO '10'B    .. '11'B,
                        2 ..  4 TO '001'B .. '011'B,
                        5  TO '0001'B,
                        6 ..  7 TO '00000'B .. '00001'B}
                WITH bitStringEncoding }
                -- Second mapping: Map determinant indexes to bitstrings
bitStringEncoding #BITS ::= {
REPETITION-ENCODING {
        REPETITION-SPACE }}
```

D.2.10.5 In the above, we quantified "frequent", "common", and "rare" as 60%, 30%, and 10%, respectively, and used the public domain ECN Huffman generator (see E.8) to determine the optimal bit-patterns to be used for each range of integer.

D.2.10.6 The above is in a mathematical sense optimal, but how much difference it makes as a percentage of total traffic depends on what the other parts of the protocol consist of. Whilst it costs nothing in implementation effort to produce and use optimal encodings (because tools can be used), the ultimate gains may not be significant.

D.2.11 A version 1 message

D.2.11.1 ASN.1 assignment:

```
Version1Message ::= SEQUENCE {
ie-1            BOOLEAN,
ie-2            INTEGER (0..20)}
```

We want to use PER basic unaligned, but intend to add further fields in version 2, and wish to specify that version 1 systems should accept and ignore any additional material in the PDU.

D.2.11.2 We use two encoding structures to encode the message: one is the implicitly generated encoding structure containing only the version 1 fields, and the second is a structure that we define containing the version 1 fields plus a variable-length padding field that extends to the end of the PDU. The version 1 system uses the first structure for encoding, and the second for decoding. Apart from this approach to extensibility, all encodings are PER basic unaligned. The version 1 decoding structure is:

```
#Version1DecodingStructure ::= #CONCATENATION {
ie-1                #BOOL,
ie-2                #INT (0..20),
future-additions    #PAD}
```

D.2.11.3 The encoding object assignments are:

```
version1MessageEncoding  #Version1Message ::= {
ENCODE-DECODE
        {ENCODE WITH PER-BASIC-UNALIGNED }
DECODE AS IF decodingSpecification}
decodingSpecification #Version1Message ::= {
USE #Version1DecodingStructure
MAPPING FIELDS
WITH {
        ENCODE STRUCTURE {
            future-additions   additionsEncoding{< OUTER >} }
        WITH PER-BASIC-UNALIGNED}}
```

```
additionsEncoding {< REFERENCE:determinant >} #PAD ::= {
ENCODING-SPACE
        SIZE      encoder-option-with-determinant
        DETERMINED BY container
        USING determinant}
```

D.2.12 The encoding object set

This encoding object set contains encoding definitions for some of the types specified in the ASN.1 module named "**Example2-ASN1-Module**" (the rest is encoded using PER basic unaligned).

```
Example2Encodings #ENCODINGS ::= {
normallySmallValuesEncoding-1              |
sparseEvenlyDistributedValueSetEncoding        |
sparseUnevenlyDistributedValueSetEncoding      |
conditionalPresenceOnValueEncoding         |
conditionalPresenceOnExternalConditionEncoding|
enclosingStructureForListEncoding          |
equalLengthListsEncoding                   |
enclosingStructureForChoiceEncoding        |
version1MessageEncoding }
```

D.2.13 ASN.1 definitions

This module groups together all the ASN.1 definitions from D.2.1 to D.2.11 that will be encoded according to the encoding objects defined in the EDM, and also lists the other ASN.1 definitions that will be encoded with the PER basic unaligned encoding rules.

```
Example2-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module2(5)}
        DEFINITIONS AUTOMATIC TAGS ::=
        BEGIN
ExampleMessages ::= CHOICE {
normallySmallValues                NormallySmallValues,
sparseEvenlyDistributedValueSet    SparseEvenlyDistributedValueSet
-- etc.
}
NormallySmallValues ::= INTEGER (0..1000)
SparseEvenlyDistributedValueSet ::= INTEGER (2 | 4 | 6 | 8 | 10 | 12 | 14 | 16)

        -- etc.
        END
```

D.2.14 EDM definitions

```
Example2-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module2(6)}
        ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS Example2Encodings;
IMPORTS #NormallySmallValues, #SparseEvenlyDistributedValueSet,
#SparseUnevenlyDistributedValueSet, #ConditionalPresenceOnValue,
#ConditionalPresenceOnExternalCondition,
#EnclosingStructureForList, #EqualLengthLists, #EnclosingStructureForChoice,
#Version1Message, #List1, #List2, #VariableLength,#UnevenChoiceProbability
FROM Example2-ASN1-Module
{joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module2(5)};
Example2Encodings #ENCODINGS ::= {
normallySmallValuesEncoding-1   |
-- etc.
version1MessageEncoding}
        -- etc.

        END
```

D.2.15 ELM definitions

The following ELM is associated with the ASN.1 module defined in D.2.13, and the EDM defined in D.2.14.

```
Example2-ELM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) elm-module2(4)}
        LINK-DEFINITIONS ::=
        BEGIN
IMPORTS
Example2Encodings FROM Example2-EDM
        {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module2(6)}
#ExampleMessages FROM Example2-ASN1-Module
        {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module2(5)};
ENCODE #ExampleMessages WITH Example2Encodings
COMPLETED BY PER-BASIC-UNALIGNED

        END
```

D.3 Explicitly generated structure examples

The examples described in D.3.1 to D.3.4 show the use of explicitly generated structures to replace an encoding class in an implicitly generated encoding structure with a synonymous class. We then produce specialized encodings by including in the encoding object set an object of the synonymous class.

The examples are presented using the following format:

- The "ASN.1 type assignment". This gives the original ASN.1 type definition.
- The requirement. This lists the required changes from the encodings provided by PER basic unaligned.
- Modification of the implicitly generated encoding structure to produce a new encoding structure.
- The encoding class and encoding object assignments.

D.3.1 Sequence with optional components defined by a pointer

D.3.1.1 The ASN.1 assignment is:

```
Sequence1 ::= SEQUENCE      {
component1    INTEGER       OPTIONAL,
component2    INTEGER       OPTIONAL,
component3    VisibleString }
```

D.3.1.2 Instead of using the PER bit-map for the two components of type integer marked **OPTIONAL**, the presence and the position of those components are determined by pointers at the beginning of the encoding of the sequence. Each pointer contains 0 (component absent) or a relative offset to the encoding of the component which begins on an octet boundary.

D.3.1.3 The encoding class **#INTEGER** is replaced with "**#Integer-with-pointer-concat**" in the encoding object of "**sequence1-encoding**". The class "**#Integer-with-pointer-concat**" is defined as a concatenation structure containing one element which is the replaced element combined with a class in the optionality category "**#Integer-optionality**".

D.3.1.4 Then two encoding objects are defined. The first, "**integer-with-pointer-concat-encoding**" of class **#Integer-with-pointer-concat** receives three parameters: the replaced element, the pointer and the current combined encoding object set (see 22.1.3.7). The second, "**integer-optionality-encoding**" of class "**#Integer-optionality**" receives one parameter, the pointer, which is used to determine the presence of the component. Since **PER-BASIC-UNALIGNED** does not contain an encoding object of class **#CONCATENATION** with optional components, a third encoding object of class **#CONCATENATION** needs to be defined. This object "concat" uses default settings.

D.3.1.5 The encoding class and encoding object assignments are:

```
sequence1-encoding #SEQUENCE ::= {
REPLACE OPTIONALS
WITH #Integer-with-pointer-concat
        ENCODED BY integer-with-pointer-concat-encoding
        INSERT AT HEAD #Pointer
ENCODING-SPACE
SIZE variable-with-determinant
DETERMINED BY container
USING OUTER }
#Pointer ::= #INTEGER
```

```
#Integer-with-pointer-concat {< #Element >} ::= #CONCATENATION {
element   #Element OPTIONAL-ENCODING #Integer-optionality }
#Integer-optionality ::= #OPTIONAL
integer-optionality-encoding{< REFERENCE: start-pointer >}
#Integer-optionality ::= {
ALIGNED TO ANY octet
START-POINTER start-pointer
PRESENCE DETERMINED BY pointer}
integer-with-pointer-concat-encoding
{< #Element, REFERENCE:pointer, #ENCODINGS:EncodingObjectSet >}
#Integer-with-pointer-concat{< #Element >} ::= {
ENCODE STRUCTURE {
        element USE-SET OPTIONAL-ENCODING
            integer-optionality-encoding{< pointer >}
        STRUCTURED WITH concat}
WITH EncodingObjectSet}
concat #CONCATENATION ::= {
ENCODING-SPACE }
```

D.3.2 Addition of a boolean type as a presence determinant

D.3.2.1 The ASN.1 assignment is:

```
Sequence2 ::= SEQUENCE      {
component1    BOOLEAN           OPTIONAL,
component2    INTEGER,
component3    VisibleString   OPTIONAL }
```

D.3.2.2 Instead of using the PER bit-map for components marked "**OPTIONAL**", the presence of an optional component is related to the value of a unique presence bit which is equal to 1 (component absent), or 0 (component present). In that case, the presence bit is inverted.

D.3.2.3 The encoding structures and encoding objects are defined as follows:

The encoding class **#OPTIONAL** is renamed as **#Sequence2-optional** in the "**RENAMES**" clause (see D.3.7). Therefore the "**#Sequence2**" class is implicitly replaced with:

```
#Sequence2 ::= #SEQUENCE   {
component1    #BOOL         OPTIONAL-ENCODING #Sequence2-optional,
component2    #INTEGER,
component3    #VisibleString  OPTIONAL-ENCODING #Sequence2-optional}
```

where:

```
#Sequence2-optional ::= #OPTIONAL
```

Then an encoding object of class "**#Sequence2-optional**" is defined; that object, using the replacement group, replaces the component encoding definition (see 23.11.3.2) with the class "**Optional-with-determinant**".

```
sequence2-optional-encoding #Sequence2-optional ::= {
REPLACE STRUCTURE
WITH #Optional-with-determinant
ENCODED BY optional-with-determinant-encoding}
```

That class, which is parameterized by the original component, belongs to the concatenation category and has two components: the determinant (boolean) and the original component.

```
#Optional-with-determinant{< #Element >} ::= #CONCATENATION {
determinant    #BOOLEAN,
component          #Element OPTIONAL-ENCODING #Presence-determinant}
```

where:

```
#Presence-determinant ::= #OPTIONAL
```

Then an encoding object of class "**#Optional-with-determinant**" is defined; that object has two dummy parameters: the class of the component and an encoding object set used to encode everything except determinant and component optionality:

```
optional-with-determinant-encoding
```

```
{< #Element, #ENCODINGS: Sequence2-combined-encoding-object-set >}
        #Optional-with-determinant {< #Element >}   ::= {
        ENCODE STRUCTURE {
            determinant    determinant-encoding,
            component USE-SET
OPTIONAL-ENCODING if-component-present-encoding{< determinant >} }
WITH Sequence2-combined-encoding-object-set }
```

The encoding is completely specified by the definition of encoding objects "**if-component-present-encoding**" and "**determinant-encoding**":

```
if-component-present-encoding {<REFERENCE:presence-bit>} #Presence-determinant ::= {
PRESENCE
        DETERMINED BY field-to-be-set
        USING presence-bit}
determinant-encoding #BOOLEAN ::= {
ENCODING-SPACE
        SIZE 1
        MULTIPLE OF bit
TRUE-PATTERN bits:'0'B
FALSE-PATTERN bits:'1'B}
```

D.3.3 Sequence with optional components identified by a unique tag and delimited by a length field

D.3.3.1 The ASN.1 assignments are:

```
Octet3 ::= OCTET STRING (CONTAINING Sequence3)
Sequence3 ::=SEQUENCE        {
component1    [0] BIT STRING (SIZE(0..2047))    OPTIONAL,
component2    [1] OCTET STRING (SIZE(0..2047))    OPTIONAL,
component3    [2] VisibleString (SIZE(0..2047))    OPTIONAL }
```

D.3.3.2 Each component is identified by a tag on four bits and the total length of the sequence is specified with a field of eleven bits which precedes the encoding of the first component.

D.3.3.3 The encoding classes **#OCTETS**, **#OPTIONAL** and **#TAG** are renamed respectively as **#Octets3**, **#Sequence3-optional** and **#TAG-4-bits** in the "**RENAMES**" clause (see D.3.7). Then encoding objects of the new encoding classes are defined.

D.3.3.4 The encoding class and encoding object assignments for the octet string are:

```
#Octets3 ::= #OCTET-STRING
octets3-encoding #Octets3 ::= {
REPETITION-ENCODING {
        REPLACE STRUCTURE
        WITH #Octets-with-length
            ENCODED BY octets-with-length-encoding}}
#Octets-with-length{< #Element >} ::= #CONCATENATION {
length      #INT(0..2047),
octets      #Element}
octets-with-length-encoding{< #Element >} #Octets-with-length{< #Element >} ::= {
ENCODE STRUCTURE {
        octets octets-encoding{< length >}}
WITH PER-BASIC-UNALIGNED}
octets-encoding{< REFERENCE:length >} #OCTETS ::= {
REPETITION-ENCODING {
        REPETITION-SPACE
            SIZE variable-with-determinant
                MULTIPLE OF octet
            DETERMINED BY field-to-be-set
            USING length} }
```

D.3.3.5 The encoding class and encoding object assignments for the sequence are:

```
sequence3-encoding #Sequence3 ::= {

ENCODE STRUCTURE {
        STRUCTURED WITH sequence3Structure-encoding }
WITH Sequence3-encodings
        COMPLETED BY PER-BASIC-UNALIGNED }
Sequence3-encodings #ENCODINGS ::= {
sequence3-optional-encoding |
```

```
tag-4-bits-encoding }
#Sequence3-optional ::= #OPTIONAL
sequence3-optional-encoding  #Sequence3-optional ::= {
PRESENCE
        DETERMINED BY container
        USING OUTER}
#TAG-4-bits ::= #TAG
tag-4-bits-encoding  #TAG-4-bits     ::= {
ENCODING-SPACE
        SIZE 4}
```

The following encoding object of class #OUTER specifies that the decoder shall ignore the bits following the encoding of the sequence which were added by the encoder to produce a multiple of octets.

```
outer-encoding #OUTER ::= {
ADDED BITS DECODING silently-ignore }
```

D.3.4 Sequence-of type with a count

D.3.4.1 The ASN.1 assignment is:

```
SequenceOfIntegers ::= SEQUENCE(SIZE(0..63)) OF INTEGER(0..1023)
```

D.3.4.2 The number of elements is encoded in a six-bit field preceding the encoding of the first element.

D.3.4.3 The encoding class **#SEQUENCE-OF** is renamed as **#SequenceOf** in the "**RENAMES**" clause (see D.3.7). An encoding object of the new encoding class is defined. The encoding class and encoding object assignments are:

```
#SequenceOf ::= #REPETITION
sequenceOf-encoding #SequenceOf ::= {
REPETITION-ENCODING {
        REPLACE STRUCTURE
        WITH #SequenceOf-with-count
        ENCODED BY sequenceOf-with-count-encoding}}
#SequenceOf-with-count{< #Element >} ::= #CONCATENATION {
count     #INT(0..63),
elements  #Element }
sequenceOf-with-count-encoding{< #Element >}
#SequenceOf-with-count{< #Element >} ::= {
ENCODE STRUCTURE {
        elements {
            ENCODE STRUCTURE {
                STRUCTURED WITH elements-encoding{< count >}}
            WITH PER-BASIC-UNALIGNED}}
WITH PER-BASIC-UNALIGNED}
elements-encoding{< REFERENCE:count >} #REPETITION ::= {
REPETITION-ENCODING {
        REPETITION-SPACE
            SIZE variable-with-determinant
MULTIPLE OF repetitions
        DETERMINED BY field-to-be-set
            USING count}}
```

D.3.4.4 The count field is encoded using the PER encoding rules for an integer type with the value range constraint (0..63), which gives a six-bit field.

D.3.5 Encoding object sets

The encoding object sets contain encoding objects of classes defined in the EDM module. (Only the first one contains the encoding object of class #SEQUENCE.)

```
Example3Encodings-1  #ENCODINGS ::= {
                sequence1-encoding }
Example3Encodings-2  #ENCODINGS ::= {
                concat                          |
                sequence2-optional-encoding     |
                octets3-encoding                |
                sequenceOf-encoding             |
                sequence3-encoding              |
                outer-encoding }
```

D.3.6 ASN.1 definitions

This module groups together the ASN.1 definitions from D.3.1 to D.3.4 that will be encoded according to the encoding objects defined in the EDM of D.3.7.

```
Example3-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module3(9)}
DEFINITIONS
AUTOMATIC TAGS ::=
BEGIN
Sequence1 ::=   SEQUENCE   {
component1    BOOLEAN          OPTIONAL,
component2    INTEGER          OPTIONAL,
component3    VisibleString    OPTIONAL }

        -- etc.

        END
```

D.3.7 EDM definitions

```
Example3-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module3(10)}
ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS Example3Encodings-1, Example3Encodings-2;
RENAMES
              #OPTIONAL AS #Sequence2-optional
                    IN #Sequence2
              #OCTET-STRING AS #Octets3
                    IN ALL
              #OPTIONAL AS #Sequence3-optional
                    IN #Sequence3
              #TAG AS #TAG-4-bits
                    IN #Sequence3
              FROM Example3-ASN1-Module
                    { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module3(9)};

Example3Encodings-1 #ENCODINGS  ::= {
sequence1-encoding }
Example3Encodings-2 #ENCODINGS  ::= {
concat |
-- etc.
sequenceOf-encoding }
        -- etc.
END
```

D.3.8 ELM definitions

The following ELM is associated with the ASN.1 module defined in D.3.6 and the EDM defined in D.3.7.

```
Example3-ELM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) elm-module3(8)}
LINK-DEFINITIONS ::=
BEGIN

IMPORTS Example3Encodings-1, Example3Encodings-2, #Sequence1, #Sequence2,
#Octet3, #Sequence3, #SequenceOfIntegers
FROM Example3-EDM
    { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module3(10) };

        ENCODE #Sequence1
WITH Example3Encodings-1
COMPLETED BY PER-BASIC-UNALIGNED

ENCODE #Sequence2, #Octet3, #Sequence3, #SequenceOfIntegers
WITH Example3Encodings-2
COMPLETED BY PER-BASIC-UNALIGNED
END
```

D.4 A more-bit encoding example

D.4.1 Description of the problem

D.4.1.1 This example is taken from ITU-T Rec. Q.763 (Signalling System No. 7 – ISDN User Part formats and codes).

D.4.1.2 There is a requirement to produce the following encoding as a series of octets:

8	7 6	5 4 3 2 1
extension indicator	spare	protocol profile

D.4.1.3 Bit 8 is an "extension indicator". If it is 0, there is a following octet in the same format. If it is 1, this is the last octet of the series.

NOTE – The PER encoding of boolean is 1 for **TRUE** and 0 for **FALSE**, and ECN requires that the last element returns **FALSE**, earlier elements **TRUE**. Thus if we use a PER-encoded boolean for the more-bit, we need to apply the "**not**" transform.

D.4.1.4 This is the traditional use of a "more bit", although with the perhaps unusual zero for "more" and one for "last".

D.4.1.5 The example would be simplified if the use of the "extension indicator" had zero and one interchanged, and if there were no "spare" bits, but use of the real example was preferred here.

D.4.1.6 There are four approaches to solving this problem.

D.4.1.7 The first approach is to include a component in the ASN.1 specification to provide the more-bit determinant (see D.4.2). This approach is deprecated for two reasons. The first is that the ASN.1 type definition contains a component which does not carry application semantics. The second is that it requires the application to (redundantly) set this field correctly in each element of the more-bit repetition.

D.4.1.8 The second approach is to use value mappings from an implicitly generated structure to a user-defined encoding structure which includes the more-bit determinant (see D.4.3).

D.4.1.9 The third approach is to use the replacement mechanism to include the more-bit determinant (see D.4.4).

D.4.1.10 The fourth approach is to use head-end insertion of the more-bit determinant. (This is not illustrated here.)

D.4.1.11 All of the last three approaches have their own advantages, and choosing between them is largely a matter of style.

D.4.2 Use of ASN.1 to provide the more-bit determinant

D.4.2.1 In this approach, the ASN.1 reflects all fields in the encoding. This is generally considered "dirty", as fields which should be visible only in the encoding are visible to the application, reducing the "information hiding" that is the strength of ASN.1. In this case the ASN.1 is:

```
ProfileIndication ::= SEQUENCE OF
SEQUENCE {
more-bit            BOOLEAN,
reserved            BIT STRING (SIZE (2)),
protocol-Profile-ID    INTEGER (0..31) }
```

D.4.2.2 The implicitly generated encoding structure is:

```
#ProfileIndication ::= #SEQUENCE-OF {
#SEQUENCE {
        more-bit                #BOOLEAN,
        reserved                #BIT-STRING (SIZE (2)),
        protocol-Profile-ID     #INTEGER (0..31) } }
```

D.4.2.3 First, we produce a generic encoding object for **#SEQUENCE-OF** that uses a more-bit in a field identified as a parameter of the encoding object, and with **BOOLEAN TRUE** (encoded as a single "1" bit by PER) for the last element:

```
more-bit-encoding {< REFERENCE:more-bit >} #SEQUENCE-OF ::= {
REPETITION-ENCODING {
        REPETITION-SPACE
```