



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).

CONTENTS

	<i>Page</i>
1 Clause 2	1
2 Clause 18.2.3	1
3 Annex D	1

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 encoding 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 }
```