Temporary document **018**

European Telecommunications Standards Institute
MTS#31
24 to 26 October 2000
Sophia-Antipolis

Source:	**STF169 leader**
Title:	**DTS/MTS-00068: Specification of ASN.1 Encoding Control Notation**
Date:	**13 October 2000**
Document for:	**Information**
Agenda item:	**6.6**

DTS/MTS-00068 V1.1.1 (2000-10)

**Methods for Testing and Specification (MTS);
Abstract Syntax Notation 1 (ASN.1) encoding rules;
Specification of Encoding Control Notation (ECN);
(ISO Draft Standard 8825-3,
ITU-T Draft Recommendation X.692)**

Reference
DTS/MTS-00068

Keywords
ASN.1, protocol, specification

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Important notice

Individual copies of the present document can be downloaded from:
http://www.etsi.org

The present document may be made available in more than one electronic version or in print. In any case of existing or perceived difference in contents between such versions, the reference version is the Portable Document Format (PDF). In case of dispute, the reference shall be the printing on ETSI printers of the PDF version kept on a specific network drive within ETSI Secretariat.

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at http://www.etsi.org/tb/status/

If you find errors in the present document, send your comment to:
editor@etsi.fr

Copyright Notification

Reproduction is only permitted for the purpose of standardization work undertaken within ETSI.
The copyright and the foregoing restrictions extend to reproduction in all media.

© European Telecommunications Standards Institute yyyy.
All rights reserved.

Contents

Intellectual Property Rights 6
Foreword 6
Introduction 6
1 Scope 7
2 Normative references 7
2.1 Identical Recommendations | International Standards 7
2.2 Additional references 7
3 Definitions 8
3.1 ASN.1 Definitions 8
3.2 ECN-specific definitions 8
4 Abbreviations 10
5 Definition of ECN syntax 10
6 Encoding conventions and notation 10
7 The ECN character set 11
8 ECN lexical items 11
8.1 Encoding object references 12
8.2 Encoding object set references 12
8.3 Encoding class references 12
8.4 Reserved word items 12
8.5 Reserved encoding class name items 12
8.6 Transparent token item 13
9 ECN Concepts 13
9.1 Structure of Encoding Control Notation (ECN) specifications 13
9.2 Encoding classes 13
9.3 Encoding structures 14
9.4 Encoding objects 14
9.5 Encoding object sets 14
9.6 Defining new encoding classes 14
9.7 Defining encoding objects 15
9.8 Differential encoding-decoding 15
9.9 Properties of encoding objects 16
9.10 Parameterization 16
9.11 Governors 16
9.12 General aspects of encodings 17
9.13 Encoding structure field-references and determinants 17
9.14 Mapping abstract values onto fields of encoding structures 17
9.15 Contents of Encoding Definition Modules 18
9.16 Contents of the Encoding Link Module 18
9.17 Application of encodings 21
9.18 Combined encoding object set 21
9.19 Application point 21
9.20 Conditional encodings 22
9.21 Changes to ASN.1 Recommendations | International Standards 22
10 Identifying encoding classes, encoding objects, and encoding sets 23
11 Encoding ASN.1 types 25
11.1 General 25
11.2 Built-in encoding classes used for implicit encoding structures 26
11.3 Simplification and expansion of ASN.1 notation for encoding purposes 26
11.4 The implicit encoding structure 28
12 The Encoding Link Module (ELM) 28
12.1 Structure of an ELM module 28

12.2 Encoding a type 29
12.3 Encoding multiple types 29

13 Application of encodings 30
13.1 General 30
13.2 The combined encoding object set and its application 30

14 The Encoding Definition Module (EDM) 32

15 Encoding class assignments 33
15.1 General 33
15.2 Encoding structure definition 35
15.3 Alternative encoding structure 37
15.4 Repetition encoding structure 37
15.5 Concatenation encoding structure 37

16 Encoding object assignments 38
16.1 Categories of encoding object assignments 38
16.2 Encoding with a defined syntax 39
16.3 Encoding with encoding object sets 40
16.4 Encoding using value mappings 40
16.5 Encoding an encoding structure 40
16.6 Differential encoding-decoding 42
16.7 User-defined encoding-functions 42

17 Encoding object set assignments 43

18 Mapping values 44
18.1 General 44
18.2 Mapping by explicit values 45
18.3 Mapping by matching fields 47
18.4 Mapping by #TRANSFORM encoding objects 47
18.5 Mapping by abstract value ordering 48
18.6 Mapping by value distribution 49
18.7 Mapping integer values to bits 50

19 Built-in encoding classes supported by defined syntax 51
19.1 General 51
19.2 Common types 52
19.3 The #TRANSFORM encoding class 54
19.3.1 Source class and target encoding class 54
19.3.2 The int-to-int transforms 54
19.3.3 The bool-to-bool transforms 55
19.3.4 The bool-to-int transforms 55
19.3.5 The int-to-bool transforms 55
19.3.6 The int-to-chars transforms 55
19.3.7 The int-to-bits transforms 56
19.3.8 The bits-to-int transforms 56
19.3.9 The char-to-bits transforms 56
19.3.10 The bits-to-char transforms 57
19.3.11 The bit-to-bits transforms 57
19.3.12 The bits-to-bit transforms 57
19.3.13 The bits-to-fixed-units transform 57
19.4 The pre-alignment parameters 58
19.5 The padding parameter group 58
19.6 The pad-padding parameter group 58
19.7 The bit reversal parameters 58
19.8 Encoding space parameters 58
19.9 Determination mechanisms 59
19.9.1 General 59
19.9.2 Use of a length determinant 59
19.9.3 Unused bits determination 60
19.9.4 End of container length determination 60
19.9.5 Special pattern length determination 60
19.10 Definition of handles 60
19.11 Use of handles 61

19.12 Value-encoding for Nul ... 61
19.13 Value-encoding for Bool ... 61
19.14 Value-encoding for Int ... 62
19.15 The concatenation procedure parameters ... 62
19.16 Repetition encoding ... 62
19.17 Value-encoding for Bits ... 62
19.18 Value-encoding for Octets ... 63
19.19 Value-encoding for Chars ... 63
19.20 The ordering procedure parameters ... 63
19.21 Contained type encoding ... 64
19.22 The #OUTER encoding class ... 64

Annex A (Normative): Specification of Encoding Classes ... 65
A.1 Commonly-used type definitions ... 65
A.2 Groups of parameters ... 66
A.3 The #TRANSFORM encoding class ... 66
A.4 Defining encoding objects for alternative classes ... 68
A.5 Defining encoding objects for #BITS and #BIT-STRING classes ... 68
A.6 Defining encoding objects for #BOOL and #BOOLEAN classes ... 69
A.7 Defining encoding objects for #CHARS and other character string classes ... 70
A.8 Defining encoding objects for concatenation classes ... 71
A.9 Defining encoding objects for #INT, #CONDITIONAL-INT, #INTEGER and #ENUMERATED classes ... 72
A.10 Defining encoding objects for #NUL and #NULL classes ... 73
A.11 Defining encoding objects for #OCTETS and #OCTET-STRING classes ... 74
A.12 Defining encoding objects for optionality classes ... 75
A.13 Defining encoding objects for the #PAD class ... 75
A.14 Defining #REPETITION, #CONDITIONAL-REPETITION, #SEQUENCE-OF, #SET-OF class encodings ... 76
A.15 Defining encoding objects for #OUTER class ... 77

Annex B (Normative): Addendum to ITU-T Rec. X.680 | ISO/IEC 8824-1 ... 78
B.1 Exports and imports statements ... 78
B.2 Absolute reference ... 78
B.3 Addition of "REFERENCE" ... 79
B.4 Notation for character string values ... 79

Annex C (Normative): Addendum to ITU-T Rec. X.681 | ISO/IEC 8824-2 ... 80
C.1 Definitions ... 80
C.2 Additional lexical items ... 80
C.3 Addition of "ENCODING-CLASS" ... 80
C.4 FieldSpec additions ... 80
C.5 Encoding object field spec ... 80
C.6 Encoding object set field spec ... 81
C.7 Encoding object list field spec ... 81
C.8 Encoding object list notation ... 81
C.9 Primitive field names ... 81
C.10 Additional reserved words ... 82
C.11 Definition of encoding objects ... 82
C.12 Additions to "Setting" ... 82
C.13 Encoding class field type ... 83

Annex D (Normative): Addendum to ITU-T Rec. X.683 | ISO/IEC 8824-4 ... 84
D.1 Parameterized assignments ... 84
D.2 Parameterized encoding assignments ... 84
D.3 Referencing parameterized definitions ... 85
D.4 Actual parameter list ... 85

Annex E (Informative): Examples ... 86
E.1 General examples ... 86
E.1.1 An encoding object set ... 86
E.1.2 An encoding object for a boolean type ... 86
E.1.3 An encoding object for an integer type ... 87
E.1.4 Another encoding object for an integer type ... 87
E.1.5 Encodings of values of integer types with holes ... 87
E.1.6 A more complex encoding object for an integer type ... 88

E.1.7 Positive integers encoded in BCD ... 88
E.1.8 An encoding object of class #BITS ... 89
E.1.9 An encoding object of class #OCTETS ... 90
E.1.10 An encoding object of class #CHARS ... 90
E.1.11 Mapping character values to bit values ... 90
E.1.12 Encoding a sequence type ... 90
E.1.13 ELM definitions ... 91
E.1.14 ASN.1 definitions ... 91
E.1.15 EDM definitions ... 92
E.2 Specialization examples ... 92
E.2.1 The encoding object set ... 92
E.2.2 Encoding by distributing values to an alternative encoding structure ... 92
E.2.3 Encoding by mapping ordered abstract values to an alternative encoding structure ... 93
E.2.4 Compression of non-continuous value ranges ... 93
E.2.5 Compression of non-continuous value ranges using a transformation ... 94
E.2.6 Compression of an unevenly distributed value set by mapping ordered abstract values ... 94
E.2.7 An optional component's presence depends on the value of another component ... 94
E.2.8 The presence of an optional component depends on some external condition ... 95
E.2.9 A variable length list ... 96
E.2.10 Equal length lists ... 96
E.2.11 Uneven choice alternative probabilities ... 97
E.2.12 A version 1 message ... 98
E.2.13 ELM definitions ... 98
E.2.14 ASN.1 definitions ... 99
E.2.15 EDM definitions ... 99
E.3 Legacy protocol example ... 99
E.3.1 Introduction ... 99
E.3.2 Encoding definition for the top-level message structure ... 101
E.3.3 Encoding definition for a message structure ... 101
E.3.4 Encoding for the sequence type "B" ... 102
E.3.5 Encoding the octet-aligned sequence type for the legacy protocol ... 102
E.3.6 Encoding for an octet-aligned sequence-of type with a length determinant ... 102
E.3.7 Encoding for an octet-aligned sequence-of type which continues to the end of the PDU ... 103
E.3.8 ELM definitions ... 103
E.3.9 EDM definitions ... 103

Annex F (Informative): Support for Huffman encodings ... 104

Annex G: Additional Information on the Encoding Control (Informative) Notation (ECN) ... 106

Annex H (Informative): Summary of the ECN notation ... 107
H.1 Terminal symbols ... 107
H.2 Productions ... 108

Intellectual Property Rights

Foreword

Introduction

This Technical Specification defines the Encoding Control Notation (ECN) used to specify encodings (of ASN.1 types) that differ from those provided by standardized encoding rules such as the Basic Encoding Rules (BER) and the Packed Encoding Rules (PER).

1 Scope

This TS defines a notation for specifying encodings of ASN.1 types or of parts of types.

It provides several mechanisms for such specification, including:

? direct specification of the encoding using standardized notation;

? specification of the encoding by reference to standardized encoding rules;

? specification of the encoding of an ASN.1 type by reference to an encoding structure;

? specification of the encoding using a user-defined encoding-function.

It also provides the means to link the specification of encodings to the type definitions to which they are to be applied.

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this TS. At the time of publication, the editions indicated were valid. All Recommendations and International Standards are subject to revision, and parties to agreements based on this TS are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards

[1] ITU-T Recommendation X.680 (1997) | ISO/IEC 8824-1:1998, "Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation"

[2] ITU-T Recommendation X.681 (1997) | ISO/IEC 8824-2:1998, "Information technology – Abstract Syntax Notation One (ASN.1): Information object specification."

[3] ITU-T Recommendation X.682 (1997) | ISO/IEC 8824-3:1998, "Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification."

[4] ITU-T Recommendation X.683 (1997) | ISO/IEC 8824-4:1998, "Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications."

[5] ITU-T Recommendation X.690 (1997) | ISO/IEC 8825-1:1998, "Information technology – ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER)."

[6] ITU-T Recommendation X.691 (1997) | ISO/IEC 8825-2:1998, "Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)."

NOTE Notwithstanding the ISO publication date, the above specifications are normally referred to as "ASN.1:1997".

2.2 Additional references

ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane*.

NOTE The above reference shall be interpreted as a reference to ISO/IEC 10646-1 together with all its published amendments and technical corrigenda.

3 Definitions

For the purposes of this TS, the following definitions apply.

3.1 ASN.1 Definitions

This TS uses the terms defined in 3 of ITU-T Rec. X.680 | ISO/IEC 8824-1, ITU-T Rec.X.681 | ISO/IEC 8824-2, ITU-T Rec. X.682 | ISO/IEC 8824-3, ITU-T Rec. X.683 | ISO/IEC 8824-4, ITU-T Rec. X.690 | ISO/IEC 8825-1 and ITU-T Rec. X.691 | ISO/IEC 8825-2.

3.2 ECN-specific definitions

alignment point: The point in an encoding (usually its start) which serves as a reference point when an encoding specification requires alignment to some boundary.

bit-field: Contiguous bits or octets in an encoding which are decoded as a whole, and which either represent an abstract value, or provide information (such as a length determinant for some other field) needed for successful decoding, or both.

NOTE It is in legacy protocols that "or both" sometimes occurs.

bounds condition: A condition on the existence of bounds of an integer field (and whether they allow negative values or not) which, if satisfied, means that specified encoding rules are to be applied.

choice determinant: A bit-field which determines which of several possible encodings (each representing different abstract values) is present in some other bit-field.

combined encoding object set: A temporary set of encoding objects produced by the combination of two sets of encoding objects for the purposes of applying encodings.

conditional encoding: An encoding which is to be applied only if some specified bounds condition or size range condition is satisfied.

containing type: An ASN.1 type (or encoding structure field) where a contents constraint has been applied to the values of that type (or to the values associated with that encoding structure field).

NOTE The ASN.1 types to which a contents constraint can be applied are the BIT STRING and the OCTET STRING types.

current application point: The point in an encoding structure at which a combined encoding object set is being applied.

differential encoding-decoding: The specification of rules for a decoder that require the acceptance of encodings that cannot be produced by an encoder conforming to the current specification.

NOTE Differential encoding-decoding supports the specification of decoding by a decoder (conforming to an initial version of a standard) which is intended to enable it to successfully decode encodings produced by a later version of a standard. This is sometimes referred to as support for extensibility.

encodable item: Those parts of an encoding (or procedures involved in encoding or decoding) that can be independently determined using the Encoding Control Notation.

NOTE Encodable items includes not only the encoding of values of ASN.1 primitive types, but also elements of the procedures used in encoding or decoding, for example, those that are used to determine the end of repetitions, or the presence or absence of optional elements.

encoding class: The set of all possible encoding specifications for an encodable item.

NOTE Encoding classes are defined for the encoding of primitive ASN.1 types (and for user-defined types), but are also defined for the procedures associated with the use of "OPTIONAL" in ASN.1 type definitions and for **encoding constructors**.

encoding constructor: Encodable items (supported by encoding classes) that define procedures for combining, selecting, or repeating other encodable items. (Examples are the #ALTERNATIVES, #CHOICE, #CONCATENATION, #SEQUENCE, etc classes).

Encoding Definition Modules (EDM): Modules that define encodings for application in the Encoding Link Module.

Encoding Link Module (ELM): The (unique, for any given application) module that assigns encodings to ASN.1 types.

encoding object: The specification of an encoding for an encodable item.

encoding object set: A set of encoding objects.

NOTE An encoding object set is normally used in the Encoding Link Module to determine the encoding of all the top-level types used in an application.

encoding-space: The number of bits (or octets or words) used to encode an abstract value into a bit-field (see Figure 1).

encoding structure: The structure of an encoding, defined either from the structure of an ASN.1 type definition, or in an EDM using primitive bit-fields and encoding constructors.

NOTE Use of an encoding structure is only one of several mechanisms (but an important one) that the Encoding Control Notation provides for the definition of encodings for ASN.1 types.

extensibility: Notations in an early version of a standard that are designed to maximize the interworking of implementations of that early version with the expected implementations of a later version of that standard.

governor: A part of an ECN specification which determines the syntactic form of some other part of the ECN specification.

NOTE A governor is typically notation that specifies an encoding class, and it determines the syntax to be used for the definition of an encoding object (of that class). The concept is the same as the concept of a type definition in ASN.1 acting as the governor for ASN.1 value notation.

identification handle: Part of the encoding of an encodable item which is the same for the encoding of all abstract values of that encodable item, and which serves to distinguish encodings of values of that encodable item from the encoding of values of other encodable items.

NOTE The ASN.1 Basic Encoding Rules use tags to provide identification handles in BER encodings.

implicit encoding class: An encoding class which is an implicit encoding structure.

implicit encoding structure: The encoding structure that is implicitly generated and exported whenever a type is defined in an ASN.1 module.

incomplete combined encoding object set: A combined encoding object set that does not contain sufficient encoding objects to determine the encoding of the type(s) to which it is applied.

initial application point: The point in an encoding structure at which any given combined encoding object set is first applied (in the ELM and in EDMs).

length determinant: A bit-field that determines the length of some other bit-field.

negative integer value: A value less than zero.

non-negative integer value: A value greater than or equal to zero.

non-positive integer value: A value less than or equal to zero.

optional element: A bit-field that is sometimes included (to encode an abstract value) and is sometimes omitted.

positive integer value: A value greater than zero.

presence determinant: A bit-field that determines whether an optional element is present or not.

self-delimiting encoding: An encoding for a set of abstract values such that there is no abstract value that has an encoding that is an initial sub-string of the encoding of any other abstract value.

NOTE This includes not only fixed-length encodings of a bounded integer, but also encodings generally described as "Huffman encodings" (see Annex F).

size range condition: A condition on the existence of effective size constraints on a string or repetition field (and whether the constraint includes zero, and/or allows multiple sizes) which, if satisfied, means that specified encoding rules are to be applied

source governor (or source class): The governor that determines the notation for specifying abstract values associated with a source class when mapping them to a target class.

target governor (or target class): The governor that determines the notation for specifying abstract values associated with a target class when mapping to them from a source class.

top-level type: Those ASN.1 types in an application that are used by the application in ways other than to define the components of other ASN.1 types.

NOTE 1 – Top-level types may also be used (but usually are not) as components of other ASN.1 types.

NOTE 2 – Top-level types are sometimes referred to as "the application's messages", or "PDUs". Such types are normally treated specially by tools, as they form the top-level of programming language data-structures that are presented to the application.

transformation function (or transformations): Encoding objects of the class #TRANSFORM which specify the mapping of abstract values associated with some class into other abstract values associated with the same or a different class.

NOTE Transformation functions can be used, for example, to specify simple arithmetic operations on integer values, or to map integer values into character or bit strings.

user-defined encoding-function: A piece of non-ECN notation (for example, natural language) that provides an encoding specification for an encoding class.

value-encoding: The way in which an encoding-space is used to represent an abstract value (see Figure 1).

4 Abbreviations

ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules of ASN.1
DER	Distinguished Encoding Rules of ASN.1
ECN	Encoding Control Notation for ASN.1
EDM	Encoding Definition Module
ELM	Encoding Link Module
PDU	Protocol Data Unit
PER	Packed Encoding Rules of ASN.1

5 Definition of ECN syntax

This TS employs the notational convention defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 5, but uses the term "ECN lexical item" as a synonym for the term "item" used in that clause.

This TS employs the notation for information object classes defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 as modified by Annex C.

This TS references productions defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 as modified by Annex B, ITU-T Rec. X.681 | ISO/IEC 8824-2 as modified by Annex C, and ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by Annex D.

6 Encoding conventions and notation

This TS defines the value of each octet in an encoding by use of the terms "most significant bit" and "least significant bit".

NOTE Lower layer specifications use the same notation to define the order of bit transmission on a serial line, or the assignment of bits to parallel channels.

For the purposes of this TS, the bits of an octet are numbered from 8 to 1, where bit 8 is the "most significant bit" and bit 1 the "least significant bit".

For the purposes of this TS, encodings are defined as a string of bits starting from a "leading bit" through to a "trailing bit". On transmission, the first eight bits of this string of bits starting with the "leading bit" shall be placed in the first transmitted octet with the leading bit as the most significant bit of that octet. The next eight bits shall be placed in the next octet, and so on. If the encoding is not a multiple of eight bits, then the remaining bits shall be transmitted as if they were bits 8 downwards of a subsequent octet.

NOTE A complete ECN encoding is not necessarily always a multiple of eight bits, but an ECN specification can determine the addition of padding to ensure this property.

When figures are shown in this TS, the "leading bit" is always shown on the left of the figure.

7 The ECN character set

Use of the term "character" throughout this document refers to the characters specified in ISO 10646-1, and full support for all possible ECN specifications can require the representation of all these characters.

With the exception of comment (as defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.6), user-defined encoding-functions and character string values, ECN specifications use only the characters listed in Table 1 .

ECN lexical items consist of a sequence of the characters listed in Table 1 .

NOTE Additional restrictions on the permitted characters for each lexical item are specified in clause 8.

Table 1 – ECN characters

A to Z	(LATIN CAPITAL LETTER A to LATIN CAPITAL LETTER Z)
a to z	(LATIN SMALL LETTER A to LATIN SMALL LETTER Z)
0 to 9	(DIGIT ZERO to DIGIT 9)
:	(COLON)
=	(EQUALS SIGN)
,	(COMMA)
{	(LEFT CURLY BRACKET)
}	(RIGHT CURLY BRACKET)
.	(FULL STOP)
#	(NUMBER SIGN)
(	(LEFT PARENTHESIS)
)	(RIGHT PARENTHESIS)
-	(HYPHEN-MINUS)
'	(APOSTROPHE)
"	(QUOTATION MARK)
|	(VERTICAL LINE)
&	(AMPERSAND)
;	(SEMICOLON)

There shall be no significance placed on the typographical style, size, color, intensity, or other display characteristics.

The upper and lower-case letters shall be regarded as distinct.

8 ECN lexical items

In addition to the ASN.1 (lexical) items specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11, this TS uses ECN lexical items specified in the following subclauses. The general rules specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.1 apply in this clause.

NOTE Annex H lists all lexical items and all the productions used in the ECN Specification, identifying those that are defined in ITU-T Rec.X.680 | ISO/IEC 8824-1, ITU-T Rec. X.681 | ISO/IEC 8824-2 and ITU-T Rec. X.683 | ISO/IEC 8824-4.

8.1 Encoding object references

Name of item - encodingobjectreference

An "encodingobjectreference" shall consist of the sequence of characters specified for a "valuereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.4. In analyzing an instance of use of this notation, an "encodingobjectreference" is distinguished from an "identifier" by the context in which it appears.

8.2 Encoding object set references

Name of item - encodingobjectsetreference

An "encodingobjectsetreference" shall consist of the sequence of characters specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2. It shall not be one of the character sequences listed in 8.4

8.3 Encoding class references

Name of item - encodingclassreference

An "encodingclassreference" shall consist of the character "#" followed by the sequence of characters specified for a "typereference" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.2. It shall not be one of the character sequences listed in 8.5 except in an EDM imports list or in an "ExternalEncodingClassReference" production .

8.4 Reserved word items

Names of reserved word items:

```
ALL                     FALSE                     REFERENCE
AS                      FIELDS                    REMAINDER
BEGIN                   IF                        SIZE
BER                     LINK-DEFINITIONS          STRUCTURE
BITS                    MAPPING                   STRUCTURED
BY                      MAX                       TO
CER                     MIN                       TRANSFORMS
COMPLETED               MINUS-INFINITY            TRUE
DECODE                  NULL                      UNION
DER                     OPTIONAL-ENCODING         USE
DISTRIBUTION            ORDERED                   USER-FUNCTION-BEGIN
ENCODE                  OUTER                     USER-FUNCTION-END
ENCODING-CLASS          PER-basic-aligned         VALUES
ENCODE-DECODE           PER-basic-unaligned       WITH
ENCODING-DEFINITIONS    PER-canonical-aligned
END                     PER-canonical-unaligned
EXCEPT                  PLUS-INFINITY
```

Items with the above names shall consist of the sequence of characters in the name.

NOTE The words (see ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.9) used in the definition of encoding classes (within a "WITH SYNTAX" clause) in Annex A are not reserved words.

8.5 Reserved encoding class name items

Names of reserved encoding class name items:

```
#ALTERNATIVES         #CHARS                      #ENUMERATED
#BITS                 #CHOICE                     #EXTERNAL
#BIT-STRING           #CONCATENATION              #GeneralizedTime
#BMPString            #CONDITIONAL-INT            #GeneralString
#BOOL                 #CONDITIONAL-REPETITION     #GraphicString
#BOOLEAN              #EMBEDDED-PDV               #IA5String
#CHARACTER-STRING     #ENCODINGS                  #INT
```

```
#INTEGER                 #OUTER                  #SET-OF
#NUL                     #PAD                    #TeletexString
#NULL                    #PrintableString        #TRANSFORM
#NumericString           #REAL                   #UniversalString
#OBJECT-IDENTIFIER       #RELATIVE-OID           #UTCTime
#OCTETS                  #REPETITION             #UTF8String
#OCTET-STRING            #SEQUENCE               #VideotexString
#OPEN-TYPE               #SEQUENCE-OF            #VisibleString
#OPTIONAL                #SET
```

Items with the above names shall consist of the sequence of characters in the name.

8.6 Transparent token item

Name of item - anystringexceptuserfunctionend

An "anystringexceptuserfunctionend" shall consist of one or more characters from the ISO 10646-1 character set, except that it shall not be the character sequence "USER-FUNCTION-END" nor shall that character sequence appear within it.

9 ECN Concepts

This clause describes the main concepts underlying this ITU-T TS.

9.1 Structure of Encoding Control Notation (ECN) specifications

ECN specifications consist of one or more Encoding Definition Modules (EDMs) which define encoding rules for ASN.1 types, and a single Encoding Link Module (ELM) that applies those encoding rules to ASN.1 types.

9.2 Encoding classes

An encoding class is an implicit property of all ASN.1 types, and represents the set of all possible encoding specifications for that type. It provides a reference that allows Encoding Definition Modules to define encoding rules for the type. Encoding class names begin with the character "#".

EXAMPLE: Encoding rules for the ASN.1 type "INTEGER" are defined by reference to the encoding class #INTEGER, and encoding rules for a user-defined type "My-Type" are defined by reference to the encoding class #My-Type.

There are several kinds of encoding classes:

- **Built-in encoding classes.** There are built-in encoding classes with names such as #INTEGER and #BOOLEAN. These enable the definition of special encodings for primitive ASN.1 types. There are also built-in encoding classes for encoding constructors such as #SEQUENCE, #SEQUENCE-OF and #CHOICE, and for the definition of encoding rules for handling optionality through #OPTIONAL. Finally, there are some built-in classes (#OUTER, #TRANSFORM and others) that allow the definition of encoding procedures which are part of the encoding/decoding process, but which do not directly relate to any actual bit-field or ASN.1 construct.

- **Implicit encoding classes** with names consisting of the character "#" followed by the "typereference" name appearing in a "TypeAssignment" in an ASN.1 module. Such encoding classes are implicitly generated whenever a type is defined in an ASN.1 module, and can be imported into an Encoding Definition Module to enable the definition of special encodings for the corresponding ASN.1 type. These encoding classes represent the structure of an ASN.1 encoding, and are formed from the built-in encoding classes mirroring the structure of the ASN.1 type definition.

- **Structure-based encoding classes** are encoding classes defined by the ECN user by specifying an encoding structure as a structure made up of bit-fields and encoding constructors. These structure-based encoding classes are similar to the implicit encoding classes, but the ECN user has full control of their structure. These classes enable complex encoding rules to be defined, and are essential for the use of ASN.1 with ECN for specifying legacy protocols, where additional bit-fields are needed in the encoding for determinants.

9.3 Encoding structures

Encoding structure definitions have some similarity to ASN.1 type definitions, and are named with a name beginning with the character "#" then an upper-case letter. Each encoding structure definition defines a new encoding class (the set of all possible encodings of that encoding structure). Encoding structures are formed from fields which are either built-in encoding classes or the names of other encoding structures, combined using encoding constructors (which represent the set of all possible encoding rules that support their type of construction mechanism, and are hence called encoding classes). (See E.2.9 for an example of an encoding structure definition.)

The most basic encoding constructors are #CONCATENATION, #REPETITION, and #ALTERNATIVES, corresponding roughly to ASN.1 sequence (and set), sequence-of (and set-of), and choice types. There is also an encoding class #OPTIONAL that represents the optional presence of encodings, corresponding roughly to ASN.1 "DEFAULT" and "OPTIONAL" markers.

An encoding structure definition defines a structure-based encoding class. Such classes cannot have the same names as implicit encoding classes that are imported into the module in which they are defined

Encoding structure names can be exported and imported between Encoding Definition Modules and can be used whenever an encoding class name is required.

Values of ASN.1 types (primitive or user-defined) can be mapped to fields of an encoding structure, and encoding rules for that structure then provide encodings of the ASN.1 type. (Values mapped to encoding structures can be further mapped to fields of more complex encoding structures.) This provides a very powerful mechanism for defining complex encoding rules.

9.4 Encoding objects

Encoding objects represent the specific definition of encoding rules for a given encoding class. Usually the rules relate to the actual bits to be produced, but can also specify procedures related to encoding and decoding, for example the way in which the presence or absence of optional elements is determined.

In order to fully define the encoding of ASN.1 types (typically the top-level type(s) of an application), it is necessary to define (or obtain from standardized encoding rules) encoding objects for all the classes that correspond to components of those ASN.1 types and for the encoding constructors that are used.

For legacy protocols, this may have to be done by defining a separate encoding object for every component of an ASN.1 type, but it is more commonly possible to use encoding objects defined by standardized encoding rules (such as PER).

Although BER and PER encoding specifications pre-date ECN, within the ECN model they simply define encoding objects for all classes corresponding to the ASN.1 primitive types and constructors (that is, for all the built-in encoding classes). BER and PER are also considered to provide encoding objects for encoding classes used in the definition of encoding structures .

9.5 Encoding object sets

Encoding objects can be grouped into sets in the same way as information objects in ASN.1, and it is these sets of encoding objects that are (in an ELM) applied to an ASN.1 type to determine its encoding. (The governor used when forming these encoding object sets is the reserved word #ENCODINGS.) (See E.1.1 for an example.)

A fundamental rule of encoding object set construction is that any set can contain only one encoding object of a given encoding class. Thus there is no ambiguity when an encoding object set is applied to a type to define its encoding.

There are built-in encoding object sets for all the variants of BER and PER, and these can be used to complete sets of user-defined encoding objects.

9.6 Defining new encoding classes

New encoding classes can be defined as synonyms for an existing encoding class. This enables encoding objects of both the old encoding class and the new encoding class to appear in an encoding set.

All built-in encoding classes are synonyms for one of a small number of primitive encoding classes. Thus #SEQUENCE and #SET are both defined from the #CONCATENATION class, #INTEGER and #ENUMERATED are both defined

from the #INT class, and the classes for the different ASN.1 character string types are all defined from the #CHARS class. A encoding structure (for example one implicitly generated from an ASN.1 type) can contain a mix of the different synonyms, enabling different encodings to be applied to #SEQUENCE and #SET (for example).

Primitive classes are either bit-field classes, or are one of three types of encoding constructor, or are more general encoding procedures (see clause 13bis.3). Encoding objects can be defined for all encoding classes, but encoding structures can only be defined using bit-field classes which are combined using encoding constructors, and the #OPTIONAL class (representing encoding/decoding procedures for resolving optionality).

If a new encoding class is defined as a synonym for an existing encoding class, it retains the general properties of that encoding class as being either:

a) a class containing encodings for bit-fields; or

b) a particular type of constructor; or

c) a general encoding procedure,

and has the same restrictions on its use as the primitive class from which it was defined.

9.7 Defining encoding objects

There are seven mechanisms available for defining an encoding object of a given encoding class. (They are not all available for all encoding classes.)

1. The first is to specify it as the same as some other defined encoding object of the required class. This does nothing more than provide a synonym for encoding objects.

2. The second, available for a restricted set of primitive encoding classes, is to use a defined syntax to specify the information needed to define an encoding object of that class. Much of the information needed is common to all encoding classes, but some of the information always depends on the specific encoding class. (See E.1.2 for an example of defining an encoding object of class #BOOLEAN which contains encodings for the ASN.1 type boolean).

3. The third, available for all encoding classes, is to define an encoding object as the encoding of the required class which is contained in some existing encoding object set. This is mainly of use in naming an encoding object for a particular class that will perform BER or PER encodings for that class.

4. The fourth is to map the abstract values associated with an encoding class ("#A", say) to abstract values associated with another (typically more complex) encoding class ("#B", say), and to define an encoding object for "#B" (using any of the available mechanisms). An encoding object for "#A" can now be defined as the application to "#B" of the encoding object for "#B". (See E.2.9 for an example).

NOTE This is the model underlying the definition of an object for encoding an "INTEGER" type in BER. The "INTEGER" is mapped to an encoding structure that contains a tag class field, a primitive/constructor boolean, a tag number field, and a value part that encodes the abstract values of the original "INTEGER". In fact, the tag number field in BER is also a complex encoding structure, and requires a second mapping to enable its complete definition.

5. The fifth mechanism is to define an encoding object for a class (for example, one corresponding to a user-defined ASN.1 type) by separately defining encoding objects for the components and for the encoding constructor used in defining the type or encoding class.

6. The sixth is to define an encoding object for differential encoding using two separate encoding objects, one of which defines the encoder's behavior, and the other of which tells a decoder what encoding should be assumed.

7. Finally, an encoding object can be defined using a **user-defined encoding-function**. This is a facility to allow use of any desired notation (including natural language) to define the encoding object.

9.8 Differential encoding-decoding

Differential encoding-decoding is the term applied to a specification that requires an implementation to accept when decoding (and to recover abstract values from) bit-patterns that are in addition to those that it is permitted to generate when performing encoding.

Differential encoding-decoding underlies all support for "extensibility" - the ability for an implementation of an earlier version of a standard to have good interworking capability with an implementation of a later version of the standard.

The precise nature of differential encoding-decoding can be quite complex. It normally includes the requirement that a decoder accepts (and silently ignores) padding fields (usually variable length) which later versions of a standard will use for the transfer of information additional to that transferred in the early version communication.

Support for differential encoding-decoding in ECN is provided by syntax that enables the definition of an encoding object (for any class) which encapsulates two encoding objects. The first encoding object defines the rules for encoding that class, and the second encoding object defines the rules that a decoder is required to assume that a communicating encoder is using for encoding.

NOTE In ECN, the rules for decoding (in an early version of a standard) are always expressed by giving the rules for encoding that it should assume its communicating partner is using. The decoding rules are not given as explicit decoding rules. The ECN specifier will ensure that such decoding rules provide any necessary "extensibility".

9.9 Properties of encoding objects

Encoding objects have some general properties. In most cases, they completely define an encoding, but in some cases they are **encoding constructors**, that is, they define only structural aspects of the encoding, requiring encoding objects for the encoding structure's components to complete the definition of an encoding.

Another key feature of an encoding object is that it may require information from the environment where its rules are eventually applied. One aspect of the environment that is fully supported is the presence of bounds in the ASN.1 type definition, provided they are "PER-visible" (see ITU-T Rec. X.691 | ISO/IEC 8825-2, 9.3). Another (not supported in the current version) would be to access the value of the ASN.1 tag class and number of the component that is to be encoded.

NOTE A somewhat different (and not standardized) external dependency would be a user-defined encoding-function for an #ALTERNATIVES encoding class which returns the selected alternative based on external data such as the channel the message is being sent on.

A third key feature is that an encoding object may exhibit an **identification handle**. This is a part of all the encodings that it produces which distinguishes its encodings from encodings of other classes displaying the same identification handle. Identification handles have to be visible to decoders without knowledge of either the encoding class or value that was encoded (but with knowledge of the name of the identification handle value that is being sought). This concept models (and generalizes) the use of tags in BER encodings: the tag value can be determined without knowledge of the encoding class for all BER encodings, and serves to identify the encoding for resolution of optionality, ordering of sets, and choice alternatives.

9.10 Parameterization

As with ASN.1 types, encoding objects, encoding object sets and encoding classes can be parameterized. This is just an extension of the normal ASN.1 mechanism.

A primary use of parameterization is in the definition of an encoding object that needs the identification of a determinant to complete the definition of the encoding. (See E.1.12 for an example of a parameterized ECN definition.)

Dummy parameters may be encoding objects, encoding object sets, encoding classes, encoding structure field references, and values of any of the ASN.1 types used in the built-in encoding classes defined in Annex A, as specified in ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by Annex D.

NOTE The governor for a dummy parameter which is a value of a type used in the definition of a built-in encoding structure uses the "EncodingClassFieldType" defined in Annex C .

9.11 Governors

The concept of a governor and of governed notation will be familiar from ASN.1 value notation, where there is always a type definition that "governs" the value notation and determines its syntax and meaning.

The same concept extends to the definition of encoding objects of a given encoding class. The syntax for defining an encoding object of class #BOOLEAN (for example) is very different from the syntax for defining an encoding object of

class #INTEGER (for example). In all cases where an encoding object definition is required, there is some associated notation that defines the class of that encoding object, and "governs" the syntax to be used in its specification.

If the governed notation is a reference name for an encoding object, then that object is required to be of "the same" class as the governor. In the ECN standard, "the same" means that the notation used for defining the class of the governor and for defining the class when the encoding object was defined are required to be the same sequence of lexical tokens.

9.12 General aspects of encodings

ECN provides support for a number of techniques typically used in defining encoding rules (not just those techniques used in BER or PER). For example, it recognizes that optionality can be resolved in any of three ways: by use of a presence determinant, by use of an identification handle or by reaching the end of a length-delimited container (or the end of the PDU) before the optional element appears.

Similarly, it recognizes that delimitation of repetitions can be done with some form of length count, can be done by the end of a container (or PDU) in which it is the last item, can be done by use of an identification handle on each of the repetitions and on following encodings or can be done by some pattern that can never occur in an encoding in the repeated series. (A simple example of the latter is a null-terminated character string.) ECN supports all these mechanisms for delimitation of repetitions, and similar mechanisms for identification of alternatives and for resolution of optionality. Encoding structure field references and determinants

Use of an "identification handle" is another common technique. In this case, all encodings for values of the repeated item will have the same bit-pattern (their identification) at some place in their encoding (the handle). Encodings for anything that can follow the repetition can be interpreted by a decoder as part of the repetition, but will generate a different identification for the handle. The concept is similar to that of using tags.

In the general case, an encoding may have several identification handles, but to terminate a repetition (or to choose between alternatives, or to resolve optionality), all the potential encodings must exhibit the same identification handle, and must have distinct values for the different encodable items. This is similar to the ASN.1 requirement for distinct tags. Identification handles have names that are required to be unique within an ECN specification.

It is important here to note that ECN allows the definition of encodings in a very flexible way, but cannot guarantee that an encoding specification is correct - that is, that a decoder can successful recover the original abstract values from an encoding. For example, an ECN specifier could assign the same bit-pattern for boolean values true and false. This would be an error, and in this case a tool could fairly easily detect the error. Another error would be to claim that an encoding was self-delimiting (and required no length determinant), when in fact it was not. This error also could be detected by a tool. In more subtle and complex cases, however, a tool may find it very hard to diagnose an erroneous (one that cannot always be successfully decoded) specification. The responsibility for correct specifications rests with the ECN designer, just as it did for those who designed the encoding rules for BER and for PER.

9.13 Encoding structure field-references and determinants

A very common (but not the only) way of determining the presence of an optional field, the length of a repetition, or the selection of an alternative is to include (somewhere in the message) a **determinant** field. Determinant fields have to be identified if this mechanism is used for determination, and this frequently requires a dummy parameter of an encoding object definition, with the actual parameter providing the encoding structure fieldname of the determinant) being supplied when the encoding object is applied to an encoding structure.

A new concept - an **encoding structure field reference** - is introduced to satisfy the need for a dummy parameter that references an encoding structure fieldname. The governor is the reserved word "REFERENCE", and the allowed notation for an actual parameter with this governor is any encoding structure fieldname within the encoding structure to which an encoding object or encoding object set with such a parameter is being applied. (See E.1.12 for an example of references to encoding structure fieldnames.)

9.14 Mapping abstract values onto fields of encoding structures

There are six mechanisms provided for this.

The first is to map individual abstract values associated with one primitive class to another primitive class. This can be used in many ways. For example, values of a character string can be mapped to integer values (and hence encoded as integer values. Values of an enumerated can be mapped to integer values, and so on. (See E.1.11 for an example.)

The second is to map a complete field of one encoding structure into a field of a compatible encoding structure, which can contain additional fields - typically for use as length or choice determinants. (See E.2.9 for an example.)

The third is to map by transforming all the abstract values associated with one encoding class into abstract values associated with a different encoding class, using a transformation function. With this mechanism, it is, for example, possible to map an #INTEGER into a #CHAR to obtain characters that can then be encoded in whatever way is desired (for example, Binary-Coded Decimal or ASCII). (See E.1.7 for an example.) Transformation functions are encoding objects of the class #TRANSFORM. They can not only transform between different encoding classes, they can also be used to define simple arithmetic functions such as multiplication by eight, subtraction of a fixed value, and so on. When applied in succession, they enable general arithmetic to be specified. (See E.1.3 for an example.)

The fourth mapping mechanism is to use a defined ordering of the abstract values of certain types and constructions, and to map according to the ordering. This provides a very powerful means of encoding abstract values associated with one encoding class as if they were abstract values associated with a wholly unrelated encoding class. (See E.1.5 for an example.)

The fifth mechanism is to distribute the abstract values (using value range notation) associated with one encoding class (typically #INTEGER) into the fields of another encoding class. (See E.2.2 for examples.)

The final mechanism allows the ECN specifier to provide an exp licit mapping from integer values (which may have been produced by earlier mappings from, for example, an #ENUMERATED class) to the bits that are to be used to encode those values. This is intended to support Huffman encodings, where the frequency of occurrence of each value is (at least approximately) known, and where the optimum encoding is required. Annex F describes Huffman encodings in more detail, and gives examples of this mechanism, together with a reference to software that will generate the ECN syntax for these mappings, given only the relative frequency with which each value of the integer is expected to be used.

9.15 Contents of Encoding Definition Modules

Encoding Definition Modules contain export and import statements exactly like ASN.1 (but can import only encoding objects, encoding object sets, and encoding classes from other EDM modules, or from ASN.1 modules in the case of implicit encoding classes).

The body of an Encoding Definition Module contains:

- "EncodingObjectAssignment" statements that define and name an encoding object for some encoding class. (There are six forms of this statement)
- "EncodingObjectSetAssignment" statements that define sets of encoding objects.
- "EncodingStructureAssignment" statements that define and name new encoding structures (new encoding classes).

The EDM can also contain parameterized versions of these statements.

Encoding objects can be defined for built-in encoding classes within any Encoding Definition Module. Encoding objects can be defined for implicit encoding classes only in Encoding Definition Modules that import the implicit encoding class from the ASN.1 module that defines the corresponding type.

NOTE If an implicit encoding class happens to have a name that is the same as a built-in encoding class name, it can still be imported into an EDM, but must be referenced in the EDM using an "External<X>EncodingClassReference".

9.16 Contents of the Encoding Link Module

All applications of the Encoding Control Notation require the identification of a single Encoding Link Module.

The Encoding Link Module applies encoding object sets to ASN.1 types (formally, to the implicit encoding structure corresponding to the ASN.1 type). These encoding object sets (or their constituent encoding objects) are imported into the Encoding Link Module from one or more Encoding Definition Modules.

There are restrictions on the application of encoding object sets to ensure that there is no ambiguity about the actual encoding rules that are being applied. For example, it is not permitted for an ELM to apply more than one encoding object set to a specific ASN.1 type.

It is possible in simple cases for an ELM to contain just a single statement (following an imports statement) that applies an encoding object set to the single top-level type of an application. (See E.1.13 for an example.)

Defining encodings for primitive encoding classes

Encoding rules for some primitive encoding classes can be defined using a user-friendly syntax which is specified in the "WITH SYNTAX" clauses of encoding class definitions (see Annex A).

The encoding class definitions in Annex A based on the information object class definition, and its syntax (and associated semantics) is defined by reference to ITU-T Rec. X.681 | ISO/IEC 8824-2 as modified by Annex C.

The encoding class definition specifies the information that has to be supplied in order to define encoding rules for particular encoding classes. The set of encoding rules that can be defined in this way is not, of course, all possible rules, but is believed to cover the encoding specifications that ECN users are likely to require.

These encoding class definitions specify a series of fields (with corresponding ASN.1 types and semantics). Encoding rules are specified by providing values for these fields. The values of these fields are effectively providing the values of a series of parameters which collectively define an encoding.

NOTE The use of the word "parameter" above should not be confused with dummy and actual parameters of an ASN.1 or ECN construct.

The meaning of the values of these fields (parameters) is specified using an encoding model (see Figure 1) where the value of each encodable item produces a **value-encoding** which is placed (left or right justified) into an **encoding-space**.

The encoding-space may have its leading edge aligned to some boundary (such as an octet boundary) by encoding-space pre-padding, and its size can be fixed or variable. The value-encoding fits within it, perhaps left or right justified, and with padding around it.

Finally, the complete encoding-space with the value-encoding and any value pre-padding and value post-padding, is mapped to bits-on-the-line with an optional specification of **bit-reversal**. This handles encodings that require "most significant byte first" or "most significant byte last" for integers, or that require the bits within an octet to be in the reverse of the normal order.

Thus there are three broad categories of information needed:

? the first relates to the encoding-space in which the encoding is placed;

? the second relates to the way an abstract value is mapped to bits (value-encoding), and the positioning of those bits within the encoding-space; and

? the third relates to any require bit-reversals..

Figure 1 shows the encoding-space (with pre-padding) and the value-encoding, (with value pre-padding and value post-padding). Figure 1 also illustrates the specification of an encoding-space unit. The encoding-space is always an integral multiple of this specified number of bits.

If the encoding-space is not the same size for all values encoded by an encoding object, then some additional mechanism is needed to determine the actual encoding-space used in an instance of an encoding.

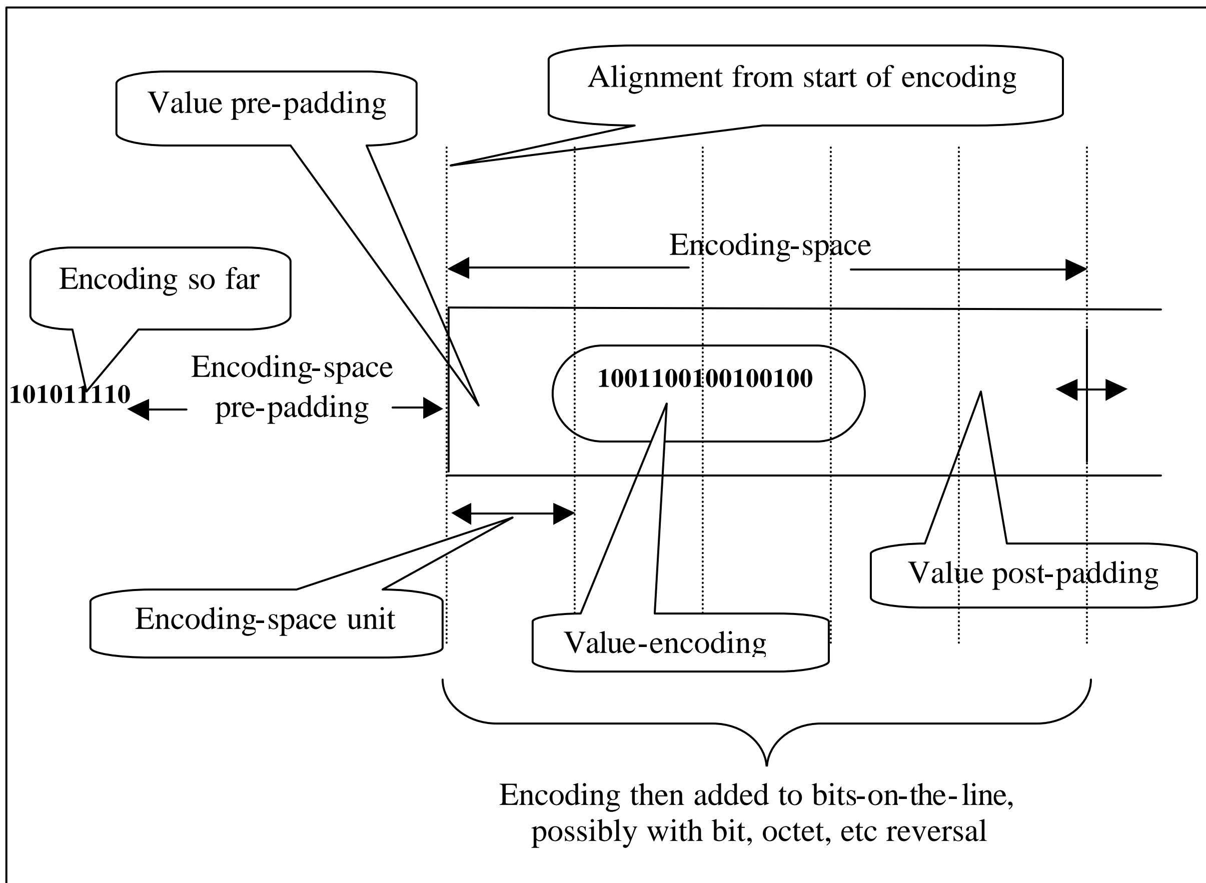


Figure 1 - Encoding-space, value-encoding and padding concepts

The steps in a definition of an encoding for a primitive encoding class are:

Specify the alignment (if any) required for the leading edge of the encoding-space (relative to the **alignment point** - normally the start of the encoding of the top-level type, that is, the type to which an encoding object set is applied in the ELM).

Specify the form of any necessary padding to that point (encoding-space pre-padding).

Specify the encoding of abstract values into bits (value-encoding).

Specify the units of the encoding-space (the encoding-space will always be an integral multiple of these units).

Specify the size of the encoding-space in these units. This may be fixed, fixed using knowledge of integer or size bounds associated with the abstract values to be encoded, or variable (different for each abstract value). The specification may also (in all cases) specify the use of a length determinant that has to be encoded with the length of the field, and either enables decoding or provides redundant information (in the case of a fixed-size encoding-space) that a decoder can check.

Specify the alignment of the value-encoding within the encoding-space.

Specify the form of any necessary padding from the start of the encoding-space to the start of the value-encoding (value pre-padding).

Specify the form of any necessary padding between the end of the value encoding and the end of the encoding-space (value post-padding).

Specify any necessary bit-reversals of the encoding-space contents before adding the bits to the encoding done so far.

Encoding class fields are available to support the specification of the encoding rules for all these steps.

In real cases, only some (or none!) of these fields will have unusual values, and defaults operate if they are not specified. (See E.1.4 for an example of the definition of the encoding for an integer that is right-aligned in a fixed two octet field, starting at an octet boundary.)

9.17 Application of encodings

Application of encodings (encoding rules) to encoding structures is a key part of the ECN work, but is very distinct from the definition of the encoding rules. Final application of encoding rules (to the encoding structure implicitly generated from an ASN.1 type definition) only occurs within an Encoding Link Module, but application of encoding rules to fields of an encoding structure may be used in the definition of an encoding rule for a larger encoding structure.

Encoding rules are applied by reference to an encoding object set (or to a single encoding object). Such application can occur in the definition of the encoding objects for any class (including encoding objects for an implicit encoding class and for a structure-based encoding class). Such application is merely the definition of more encoding objects for that encoding class: The definitive application to an actual type occurs in the Encoding Link Module.

When a set of encoding objects (encoding rules) are being applied, they always result in a complete encoding specification for the encoding class. If, in any given application, encodings are needed for encoding classes (used within an encoding structure being encoded) for which there are no encoding objects in the set being applied, then this is an error .

NOTE Although the specification of the encoding rules will be complete, the actual encoding (for example, the presence or absence of encoding-space pre-padding, or use of the values of bounds referenced in the encoding rules) can only be determined when the encoding is applied to a top-level ASN.1 type.

There are two exceptions to this requirement. The first exception is when the ASN.1-like parameterization mechanism is used to define a parameterized encoding object. In such cases the complete encoding is only defined following instantiation with an actual parameter. The second exception is when an encoding object is defined for an encoding constructor (#CONCATENATION, #ALTERNATIVES, #REPETITION, #SEQUENCE, etc.). In this latter case, the encoding rules associated with the encoding class simply define the rules associated with the structuring aspects. A complete encoding specification for an encoding structure using these encoding classes will require rules for encoding the components of that encoding structure.

NOTE There is a distinction here between encoding objects of class #SEQUENCE (an encoding constructor) and encoding objects for an implicit encoding class "#My-Type" (which happens to be defined using the ASN.1 type "SEQUENCE"). The latter is not an encoding constructor, and encoding objects of this class will provide full encoding rules for the encoding of values of "My-Type".

9.18 Combined encoding object set

In order to provide a complete encoding, the ECN user can supply a primary encoding object set, and a second encoding object set introduced by the reserved words "COMPLETED BY".

The encoding object set that is applied is defined to be the **combined encoding object set** formed by adding to the first set encoding objects for any encoding class for which the first set is lacking an encoding object and the second set contains one . A frequent set to use with "COMPLETED BY" is the built-in set "PER-basic-unaligned". (See E.1.13 for an example of the application of a combined encoding object set.)

9.19 Application point

In any given application of encodings, there is a defined starting point (for the ELM, it is the top-level type(s) to which encodings are being applied). This is called the "initial application point" for the type that is being encoded by the ELM.

The combined encoding object set is conceptually applied to an encoding structure corresponding to the ASN.1 type, and it is the encodings defined for this encoding structure that encode the abstract values of the ASN.1 type.

If there is an encoding object in the combined encoding object set that matches the encoding class (initially an implicit encoding class) at the application point, it is applied and the process terminates. Otherwise the type at the application point is "expanded". There has to be an encoding object for the class of the encoding constructor (#CHOICE, #SEQUENCE, #SEQUENCE-OF, etc), and the application point then passes to each component (as a parallel activity).

9.20 Conditional encodings

Mention has already been made of the #TRANSFORM encoding class as a means of performing simple arithmetic on integer values . This encoding class does, however, play a more fundamental role in the specification of encodings for some primitive fields. In general, the specification of encodings for many of the primitive types is a two or a three stage process, using encoding objects of class #TRANSFORM and (for example) of class #CONDITIONAL-INT or #CONDITIONAL-REPETITION".

The #TRANSFORM, #CONDITIONAL-INT, and #CONDITIONAL-REPETITION encoding classes are restricted in their use. Encoding objects can only be defined for these classes using either the syntax of Annex A, or a user-defined encoding-function, and they can only be used in the definition of other encoding objects. They cannot appear in encoding object sets or be applied directly to encode fields of encoding structures.

Encoding specifications for #INT proceeds as follows: First, one or more #TRANSFORM encoding objects are defined that map integer values into #BITS (possibly using bounds information), producing either a variable length set of bit strings or fixed length bit strings. These transformations can specify either two's complement or positive integer encodings. Secondly, encodings (of the #CONDITIONAL-INT encoding class) are defined for a particular **bounds condition,** specifying the container size (and how it is delimited), the transformation of the integer to bits (by reference to a #TRANSFORM encoding object), and the way these bits fit into the container. (An example of a bounds condition is the existence of an upper bound and a non-negative lower bound.) This is called a **conditional encoding**. Finally, the #INT encoding is defined as a list of these conditional encodings, with the actual encoding to be applied in any given circumstance being the one that is earliest in the list whose bounds condition is satisfied.

Encoding specifications for repetitions use the #CONDITIONAL-REPETITION encoding class, which defines the way in which the encoding-space for the repeated items is delimited and how the repeated encodings are to be placed into it, for a given **range condition**, again producing a conditional encoding. As with the #INT encoding class, the final encoding is defined as a list of conditional encodings.

Encoding specifications for the #OCTETS encoding class proceeds as follows: First, one or more #CONDITIONAL-REPETITION encoding objects are defined to take each of the octets in the octet string and to concatenate them into a delimited container. (The definition of this encoding object is not specific to encoding #OCTETS). The final encoding of #OCTETS is defined as a list of #CONDITIONAL-REPETITION encoding objects.

Encoding specifications for #BITS proceeds as follows: First, #TRANSFORM encoding objects are defined to map a single bit into a bit-string, similar to the encoding of an integer into bits, but in this case the mapping of the bit must be to a self-delimiting string. Secondly, one or more #CONDITIONAL-REPETITION encoding objects are defined for the repetition of the bits (these could be the same encoding objects that were defined for use with #REPETITION or #OCTETS). Finally, the #BITS encoding is defined as a list of the #CONDITIONAL-REPETITION encoding objects, with a transformation associated with each encoding object for the mapping of a bit into a bit-string.

Encoding specifications for #CHARS proceeds as follows: First, #TRANSFORM encoding objects are defined to map a single character to a self-delimiting bit-string, using several possible mechanisms for defining the encoding of the character, and using the effective alphabet constraint where it is available. Secondly, #CONDITIONAL-REPETITION encoding objects are defined, and finally the #CHARS encoding is defined as a list of these with an associated #TRANSFORM encoding object.

9.21 Changes to ASN.1 Recommendations | International Standards

This TS references other ASN.1 Recommendations | International Standards in order to define its notation without repetition. For such references to be correct, the semantics of the notation (for example the "IMPORTS" clause, parameterization, and information object definition) needs to be extended to recognize the reference names of encoding classes, encoding objects, and so on that form part of ECN.

There is also a need to extend the information object class notation to allow fields that are **lists** of values or objects, not just sets, in order to allow the use of that notation in the definition of ECN syntax for the definition of encoding objects of certain classes.

These modifications to other ASN.1 Recommendations | International Standards are specified in Annexes B to D, and are solely for the purposes of this TS.

10 Identifying encoding classes, encoding objects, and encoding sets

Many of the productions within this TS require that an encoding class, encoding object, or encoding object set be identified.

For each of these, there are four ways in which identification can be made:

Using a simple reference name.

Using a built-in reference name (not applicable for encoding objects, as there are no built-in encoding objects).

Using an external reference.

Using a parameterized reference.

In-line definition.

NOTE The parameterized reference form may be based on a simple reference name or on an external reference.

There are productions (or lexical items) for all of these means of identification. There are also productions that allow all four alternatives, where applicable. Finally, there are productions that allow all except the last of these (in-line definitions). These lexical items or production names are used where appropriate in other productions, and are defined in the remainder of this clause.

The lexical items for use of a reference name are:

```
encoding class          "encodingclassreference"
encoding object         "encodingobjectreference"
encoding object set     "encodingobjectsetreference"
```

An "encodingclassreference" is a name which is either:

a) assigned an encoding class in an "EncodingClassAssignment" or is

b) imported into an EDM from some other EDM in which it is either assigned an encoding class or is imported; or is

imported as the name of an implicit encoding class from an ASN.1 module or from an EDM module into which it was imported.

NOTE Encoding classes cannot be imported into an ELM.

An "encodingclassreference" shall not be imported from an EDM module if the referenced module has an "EXPORTS" clause and the " encodingclassreference" does not appear as a symbol in that "EXPORTS" clause.

NOTE If the referenced module has no "EXPORTS" clause, this is equivalent to exporting everything.

An implicit encoding class never appears in the "EXPORTS" clause of any ASN.1 module, but can always be imported from any ASN.1 module which it is either defined or imported.

An "encodingobjectreference" is a name which is either:

a) assigned an encoding object in an "EncodingObjectAssignment" in an EDM; or is

b) imported into an EDM or an ELM from some other EDM in which it is either assigned an encoding object or is imported.

An "encodingobjectreference" shall not be imported from an EDM or ELMif the referenced module has an "EXPORTS" clause and the "encodingobjectreference" does not appear as a symbol in that "EXPORTS" clause.

NOTE If the referenced module has no "EXPORTS" clause, this is equivalent to exporting everything.

An "encodingobjectsetreference" is a name which is either:

a) assigned an encoding object set in an "EncodingObjectSetAssignment" in an EDM; or is

imported into an EDM or an ELM from some other EDM in which it is either assigned an encoding object set or is imported.

An "encodingobjectsetreference" shall not be imported from an EDM or ELM if the referenced module has an "EXPORTS" clause and the "enodingobjectsetreference" does not appear as a symbol in that "EXPORTS" clause.

NOTE If the referenced module has no "EXPORTS" clause, this is equivalent to exporting everything.

The productions for use of a built-in reference name are:

```
encoding class              "BuiltinEncodingClassReference"
encoding object set         "BuiltinEncodingObjectSetReference"
```

The productions for use of an external reference name are:

```
ExternalEncodingClassReference ::=
        modulereference "." encodingclassreference |
        modulereference "." BuiltinEncodingClassReference
ExternalEncodingObjectReference ::=
        modulereference
        "."
        encodingobjectreference
ExternalEncodingObjectSetReference ::=
        modulereference
        "."
        encodingobjectsetreference
```

The "modulereference" is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.5, and identifies a module which is referenced in the imports list of the EDM or ELM.

The "ExternalEncodingClassReference" alternative that includes a "BuiltinEncodingClassReference" shall be used in the body of an EDM if and only if there is an implicit encoding class (whose name is the same as that of a "BuiltinEncodingClassReference") which is either:

a) defined implicitly in the ASN.1 module referenced by the "modulereference"; or

b) imported into another EDM referenced by the "modulereference".

NOTE The "BuiltinEncodingClassReference" name can appear as a "Symbol" in the "IMPORTS" clause

The productions defined above shall be used if and only if the corresponding simple reference name has been imported from the module identified by the "modulereference", and either:

a) identical reference names have been imported from different modules; or

b) both conditions hold.

A parameterized reference is a reference name defined in a "ParameterizedAssignment" and supplied with an actual parameter in accordance with the syntax specified in Annex D. The productions involved are:

```
encoding classes        "ParameterizedEncodingClassAssignment"
                        "ParameterizedEncodingClass"

encoding objects        "ParameterizedEncodingObjectAssignment"
                        "ParameterizedEncodingObject"

encoding object sets    "ParameterizedEncodingObjectSetAssignment"
                        "ParameterizedEncodingObjectSet"
```

The productions that allow all four forms of identification are:

```
encoding classes        "EncodingClass"
encoding objects        "EncodingObject"
encoding object sets    "EncodingObjectSet"
```

The productions which allow all forms except in-line definition are:

```
encoding classes        "DefinedEncodingClass"
encoding objects        "DefinedEncodingObject"
encoding object sets    "DefinedEncodingObjectSet"
```

The "DefinedEncodingClass" is:

```
DefinedEncodingClass ::=
        encodingclassreference                  |
        ExternalEncodingClassReference          |
        BuiltinEncodingClassReference           |
        ParameterizedEncodingClass
```

The "DefinedEncodingObject" is:

```
DefinedEncodingObject ::=
        encodingobjectreference                     |
        ExternalEncodingObject                      |
        ParameterizedEncodingObject
```

The "DefinedEncodingObjectSet" is:

```
DefinedEncodingObjectSet ::=
        encodingobjectsetreference              |
        ExternalEncodingObjectSetReference      |
        BuiltinEncodingObjectSetReference       |
        ParameterizedEncodingObjectSet
```

11 Encoding ASN.1 types

11.1 General

For all ASN.1 types, there is a corresponding encoding structure. This encoding structure is implicitly generated for each ASN.1 type assignment, and is automatically exported from the ASN.1 module. (It does, however, have to be imported into an EDM module if it is to be used.) The name of the corresponding encoding structure is the name of the type preceded by a character "#". This encoding structure defines an encoding class called an **implicit encoding class**, and the corresponding encoding structure is called an **implicit encoding structure**.

The encoding of an ASN.1 type is formally defined as the result of encodings applied to the corresponding (implicit) encoding structure. The encodings are applied by statements in the ELM, using encoding objects in a combined encoding object set.

The implicit encoding structure is defined by first simplifying and expanding the ASN.1 notation, and then by mapping ASN.1 types, constructors and component names into corresponding built-in encoding classes, encoding constructors and encoding structure fieldnames.

Each field of the implicit encoding structure has associated with it the abstract values of the corresponding type, and constraint-related information derived from the ASN.1 type definition .

In the present version of this TS, only the outermost tag of a type is visible to ECN (that is, can be used in specifying ECN encodings).

NOTE The only mechanism currently provided for using this information is in the determination of a canonical order for elements, for example for #CHOICE or #SET encodings.

This clause specifies:

a) The built-in encoding classes that are used in defining the implicit encoding structures corresponding to ASN.1 types.

NOTE – Clause 15 specifies additional classes that are used in the explicit definition of encoding structures by an ECN user.

b) Transformations of the ASN.1 syntax (simplification and expansion).

c) The encoding structure that is implicitly generated for an ASN.1 type.

11.2 Built-in encoding classes used for implicit encoding structures

The encoding classes used for implicit encoding structures, and the ASN.1 types or constructors to which they correspond are listed in Table 2 below.

Table 2: Encoding classes for ASN.1 notation

Encoding Class	ASN.1 notation
#BIT-STRING	**BIT STRING**
#BOOLEAN	**BOOLEAN**
#CHARACTER-STRING	**CHARACTER STRING**
#CHOICE	**CHOICE**
#EMBEDDED-PDV	**EMBEDDED PDV**
#ENUMERATED	**ENUMERATED**
#EXTERNAL	**EXTERNAL**
#INTEGER	**INTEGER**
#NULL	**NULL**
#OBJECT-IDENTIFIER	**OBJECT IDENTIFIER**
#OCTET-STRING	**OCTET STRING**
#OPEN-TYPE	**OPEN-TYPE** (This type is produced as a simplification of the ASN.1 notation)
#OPTIONAL	**OPTIONAL**
#REAL	**REAL**
#RELATIVE-OID	**RELATIVE-OID**
#SEQUENCE	**SEQUENCE**
#SEQUENCE-OF	**SEQUENCE OF**
#SET	**SET**
#SET-OF	**SET OF**
#GeneralizedTime	**GeneralizedTime**
#UTCTime	**UTCTime**
#BMPString	**BMPString**
#GeneralString	**GeneralString**
#GraphicString	**GraphicString**
#IA5String	**IA5String**
#NumericString	**NumericString**
#PrintableString	**PrintableString**
#TeletexString	**TeletexString**
#UniversalString	**UniversalString**
#UTF8String	**UTF8String**
#VideotexString	**VideotexString**
#VisibleString	**VisibleString**

11.3 Simplification and expansion of ASN.1 notation for encoding purposes

ECN assumes that certain ASN.1 syntactic constructs have been expanded (or reduced) into equivalent or simpler constructions.

NOTE The types defined by the simpler constructions are capable of carrying the same set of abstract values as the original ASN.1 syntactic structures, and those abstract values are mapped.

The expansion or simplification of ASN.1 syntactic productions is either:

a) fully-defined below; or

b) fully-defined in Annex F of ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.680 | ISO/IEC 8824-1 with all published amendments; or

c) fully-defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 with all published amendments.

d) fully-defined in ITU-T Rec. X.683 | ISO/IEC 8824-4 with all published amendments.

The ASN.1 syntactic constructs removed by the expansions and simplifications below are not referenced further in this TS.

The following expansions and simplifications shall be made:

1. The following transformations are not recursive and hence are applied only once:

   - Automatic allocation of values to enumerations (if applicable) shall be performed. The "ENUMERATED" syntax shall be replaced by the #ENUMERATED encoding class with an upper bound and lower bound set.

NOTE – The actual names of enumerations are not visible to ECN.

   - All "ValueSetTypeAssignment"s shall be replaced by their equivalent "TypeAssignment"s with subtype constraints.
   - The ASN.1 "INSTANCE OF" construction shall be expanded into its equivalent sequence type. (defined in Annex F of ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.680 | ISO/IEC 8824-1 with all published amendments)
   - "TypeFromObject" shall be replaced with the type that is referenced. (defined in Annex F of ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.680 | ISO/IEC 8824-1 with all published amendments)
   - "ValueSetFromObjects" shall be replaced with the type that is referenced. (defined in Annex F of ITU-T Rec. X.680 | ISO/IEC 8824-1 and ITU-T Rec. X.680 | ISO/IEC 8824-1 with all published amendments)

2. The following transformations shall be applied recursively in the specified order, until a fix-point is reached:

   - All ASN.1 parameterization shall be fully resolved by the substitution of actual parameters for dummy parameters. (fully-defined in ITU-T Rec. X.683 | ISO/IEC 8824-4 with all published amendments)
   - All "ComponentsOf"s shall be expanded to their full form.
   - All uses of "SelectionType" shall be resolved.

3. The following simplications shall then be applied:

   - Named number lists in integer type definitions shall be removed. Named numbers are not visible to ECN. ECN sees a single "INTEGER" type.
   - Named bit lists in bit string definitions shall be removed. Named bits are not visible to ECN.
   - All non-PER-visible constraint notation (except the contents constraint) shall be discarded. PER-visible constraints shall be resolved to provide the following values that can be referenced in the definition of encoding rules:

     i) An upper bound on integers and enumerations;

     ii) A lower bound on integers and enumerations;

     iii) The PER effective alphabet and effective size constraints (see ITU-T Rec. X.691 | ISO/IEC 8825-2, 9.3).

   - The existence of a contents constraint, the contents type, and the presence or absence of an "ENCODED BY" clause becomes a property associated with the abstract values of such a constrained "OCTET STRING" or "BIT STRING" type, and the constraint shall then be discarded.

NOTE – When specifying encodings for values with an associated contents constraint, a separate combined encoding object set can be supplied to encode the contents type. This can be specified to over-ride or not to over-ride any "ENCODED BY" that is present, as a designer's option.

   - All tagging shall be ignored in the mapping to encoding structures, but (in order to model BER encodings and PER procedures) the outermost tag of a type becomes a property of the field of the encoding structure to which the corresponding values are mapped.
   - "DEFAULT Value" shall be replaced by "OPTIONAL" and the default value is associated with the field of the structure to which the ASN.1 component is mapped.
   - "T61String" shall be replaced by "TeletexString".
   - "ISO646String" shall be replaced by "VisibleString".

- All occurrences of "ObjectClassFieldType" (fully-defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 with all published amendments) that refer to a type field, a variable-type value field, or a variable-type value set field shall be replaced by the type "OPEN-TYPE".

With these transformations, all ASN.1 type-related constructs have corresponding implicit encoding classes, listed in Table 2, and the implicitly generated encoding structure shall be constructed by mapping from column 1 to column 2 of Table 2.

11.4 The implicit encoding structure

The implicit encoding structure has the same structure as the ASN.1 type definition, with:

a) ASN.1 component identifiers mapped to encoding structure fieldnames.

b) ASN.1 types and constructors in column 2 of Table 2 mapped to the built-in encoding classes in column 1 of Table 2.

c) ASN.1 "DefinedType"s mapped to an encoding class name derived from the typereference by the addition of a character "#".

d) Abstract values are mapped from the corresponding field of the type definition to the corresponding encoding structure field.

e) Upper and lower bounds on integer and enumerated types and all effective size constraints and effective alphabet constraints (see ITU-T Rec. X.691 | ISO/IEC 8825-2, 9.3) are mapped from the type definition to the corresponding field of the encoding structure.

All implicit encoding structures can be encoded by the built-in encoding object sets and will produce the same encodings as are specified by the corresponding TS for those encodings.

12 The Encoding Link Module (ELM)

12.1 Structure of an ELM module

The "ELMDefinition" is:

```
ELMDefinition ::=
        ModuleIdentifier
        LINK-DEFINITIONS
        "::="
        BEGIN
        ELMModuleBody
        END
```

In any given application of ECN, there shall be precisely one ELM which determines the encoding of all the messages used in that application.

NOTE The ASN.1 type(s) defining "messages" are often referred to as "top-level types".

The production "ModuleIdentifier" (and its semantics) is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.1.

The "ModuleIdentifier" provides unambiguous identification of any module in the set of all ASN.1, ELM, and EDM modules.

The "ELMModuleBody" is:

```
ELMModuleBody ::=
        Imports ?
        EncodingApplicationList

EncodingApplicationList ::=
        EncodingApplication
        EncodingApplicationList ?
```

The production "Imports" (and its semantics) is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.1, 12.15, and 12.16, as modified by Annex B.

The "Imports" makes available within the ELM encoding objects and encoding object sets defined in EDMs for application to ASN.1 types.

The ASN.1 types to be encoded are not imported, but are referenced directly by the "AbsoluteReference" notation defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 14, as modified by Annex B.

The "EncodingApplicationList" is required to contain at least one "EncodingApplication" as the sole function of an ELM is to apply encodings.

The "EncodingApplication" is:

```
EncodingApplication ::=
        TypeApplication                         |
        ModuleApplication
```

12.2 Encoding a type

A "TypeApplication" is:

```
TypeApplication ::=
        ENCODE
        AbsoluteReference
        CombinedEncodings
```

A "TypeApplication" defines the encoding of an ASN.1 type whose reference name is defined in (or imported into) an identified ASN.1 module. It identifies the type using the "AbsoluteReference" notation defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, clause 14, as modified by Annex B. The encoding of the type is specified by the "CombinedEncodings" applied to the implicit encoding structure as specified in 13.2.

NOTE 1 – It will be common for an ELM to encode a single type of a single module, but where multiple types **are** encoded, ECN tool-vendors may (but need not) assume that this implicitly identifies top-level types needing support in generated data-structures.

NOTE 2 – An ELM may encode types in different modules (perhaps for different environments), and may even encode separately and differently two instances of the same type if it is exported/imported between modules. Such use is not expected to be common but is not forbidden.

Encodings applied to a type within an ELM are linked solely to the use of that type as application messages. They have no implications on the encoding of that type when referenced by other types or when exported from that module and imported into a different module.

"TypeApplication"s within an ELM shall all have distinct "AbsoluteReference"s.

NOTE The rules of application of encodingsmean that a "TypeApplication" completely defines the encoding of a type unless it contains an instance of a content constraint.

The encoding of the type in a content constraint is that specified by the encoding object applied to the containing "OCTET STRING" or "BIT STRING", and can be any combined encoding object set, or can be the combined encoding object set that was applied to the #OCTET-STRING or #BIT-STRING.

12.3 Encoding multiple types

A "ModuleApplication" is:

```
ModuleApplication ::=
        ENCODE
        ModuleIdentifier
        CombinedEncodings
```

A "ModuleApplication" defines the encodings of all ASN.1 types whose reference names are defined in (or imported into) the module identified by the "ModuleIdentifier", and which are not referenced by other types within that module. The effect is completely equivalent to listing each of these types in separate "TypeApplication"s.

An ECN specification shall not define the encoding of a type with a "TypeApplication" if its encoding is also defined by a "ModuleApplication".

13 Application of encodings

13.1 General

Encodings are applied by the ELM to a type (or independently to multiple types) using a "CombinedEncodings" definition as specified below. This clause specifies the application of "CombinedEncodings" to an encoding structure.

In the ELM, the application is to the encoding structure implicitly generated from the type named in the "EncodingApplication". Later clauses also specify the application of encodings to all or part of an encoding structure definition. This clause is applicable in both cases.

The "CombinedEncodings" is:

```
CombinedEncodings ::=
        WITH
        PrimaryEncodings
        CompletionClause ?

CompletionClause ::=
        COMPLETED BY
        SecondaryEncodings

PrimaryEncodings ::= EncodingObjectSet

SecondaryEncodings ::= EncodingObjectSet
```

A "ParameterizedEncodingObjectSet" that is instantiated in the ELM shall not contain dummy parameters.

"EncodingObjectSet" is defined in 16.3.

The use of "CombinedEncodings" is specified in 13.2.

13.2 The combined encoding object set and its application

A **combined encoding object set** is formed from the "CombinedEncodings" production as follows:

If there is no "CompletionClause", then the "PrimaryEncodings" form the combined encoding object set.

Otherwise, all encoding objects in the "PrimaryEncodings" are placed in the combined encoding object set, and every encoding object in the "SecondaryEncodings" is added to the combined encoding object set if (and only if) there is no encoding object already in the combined encoding object set with the same encoding class.

Following the conceptual construction of the combined encoding object set, encoding commences with the "encodingclassreference" name of the encoding structure identified (by reference to the associated type for applications in the ELM) in the encoding application.

Where the encoding applications in the ELM involve several types, the rules of 12.2 and 12.3 ensure that applications are non-overlapping. They proceed independently. Similarly, the application of encodings to encoding structures in EDMs are always non-overlapping. The following sub-clauses provide the rules for application to a single encoding structure.

Encoding is performed at an **application point**. The application point is initially the "encodingclassreference" for the implicit encoding class (when application is in the ELM or is a component of an encoding structure (when application is in the EDM).

The term "component" in the following text refers to any of the following:

a) The alternatives of a #CHOICE or of an #ALTERNATIVES.

b) The field following #SEQUENCE-OF, #SET-OF or #REPETITION.

c) The components of a #SEQUENCE, #SET or #CONCATENATION.

d) A contained type.

At later stages in these procedures, the application point may be on any of the following:

a) An encoding structure reference name. This is completely encodable using the specification in an encoding object of the same class if one is present in the combined encoding object set.

b) A built-in encoding class that is not an encoding constructor and is not a #BITS, #BIT-STRING, #OCTETS, or #OCTET-STRING class with a contained type associated with the values. This is completely encodable using the specification in an encoding object of the same class.

c) An encoding constructor. The construction procedures can be determined by the specification contained in an encoding object of the encoding constructor class, but that encoding object does not determine the encoding of the components.

d) A #BITS, #BIT-STRING, #OCTETS, or #OCTET-STRING encoding class with a contained type associated with the values. The encoding of the contained type depends on whether there is an "ENCODED BY" present, and on the specification of the encoding object being applied .

The classes #ALTERNATIVES, #CHOICE, #CONCATENATION, #SEQUENCE, #SET, #REPETITION, #SEQUENCE-OF, #SET-OF are encoding constructors.

Encoding proceeds as follows:

- If there is an object in the combined encoding object set of the same class as the current application point, then that encoding object is applied.

- If the encoding class does not have any components then that application completely determines the encoding of the class and terminates these procedures. Otherwise these procedures are applied recursively to each component unless it is a contained type.

- If the component is a #BITS, #BIT-STRING, #OCTETS, or #OCTET-STRING encoding class with a contained type associated with the values, then there are four cases that can occur:

  - The contents constraint contains an "ENCODED BY", and the encoding object for this class either does not contain a specification of the encoding of the contained type, or specifies that it should not over-ride an "ENCODED BY". In this case the "ENCODED BY" specification shall be used for the contained type, and the application point passes to the contained type using this encoding specification.

  - The contents constraint contains an "ENCODED BY", but the encoding object for this class contains a specification of the encoding of the contained type, and specifies that it should override an "ENCODED BY". In this case, the specification in the encoding object shall be applied to the contained type, and the application point passes to the contained type using this encoding specification.

  - The contents constraint does not contain an "ENCODED BY" and the encoding object for this class contains a specification of the encoding of the contained type. In this case, the specification in the encoding object is applied to the contained type, and the application point passes to the contained type using this encoding specification.

  - The contents constraint does not contain an "ENCODED BY", and the encoding object for this class does not contain a specification of the encoding of the contained type. In this case the combined encoding object set being applied to the class shall also be applied to the contents type, and the application point passes to the contained type using this encoding specification.

- If there is no encoding object in the combined encoding object set of the same class as the current application point, and the current application point is an encoding structure reference name, then it is de-referenced and these procedures are applied recursively to the new encoding structure.

- Otherwise the ECN specification is in error.

The above algorithm can be summarized as follows: The combined encoding object set is applied in a top-down manner. If in this process an encoding structure reference name is encountered and there is an object in the combined encoding object set that can encode it, that object determines its encoding. Otherwise, the reference name is expanded by de-referencing. If at any stage an encoding is required (and does not exist) for an encoding class that cannot be de-referenced, then the ECN specification is incorrect, and the combined encoding class is said to be incomplete. When a primitive type is reached the encoding terminates with the encoding of that type, except that if it has a contained type,

encoding proceeds to the contained type. When a type with components is reached, the process continues by applying the combined encoding object set to each component.

In the encoding process, encoding objects applied to encoding constructors (and to #OPTIONAL), may require that the encoding objects applied to their components exhibit identification handles (of a given name) to resolve alternatives or optionality or order in a set-like concatenation. If in this case the encodings of the components do not exhibit the required identification handles, then the ECN specification is in error.

NOTE This problem is most likely to arise if BER encoding objects are applied to encoding constructors and not to their components, as BER is heavily reliant on identification handles. PER encoding objects make no use of identification handles.

14 The Encoding Definition Module (EDM)

There are two top-level productions in ECN, the "EDMDefinition" specified in this clause and the "ELMDefinition" specified in clause 12. These specify the syntax for defining EDMs and the ELM respectively.

The production "EDMDefinition" is:

```
EDMDefinition ::=
        ModuleIdentifier
        ENCODING-DEFINITIONS
        "::="
        BEGIN
        EDMModuleBody
        END
```

In any given application of ECN, there are zero, one or more EDMs which define encoding objects for application in the ELM.

NOTE If there are zero EDMs, then only built-in object sets can be used in the ELM.

The production "ModuleIdentifier" (and its semantics) is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.1.

The "ModuleIdentifier" provides unambiguous identification of any module in the set of all ASN.1, ELM, and EDM modules.

The "EDMModuleBody" is:

```
EDMModuleBody ::=
        Exports ?
        Imports ?
        EDMAssignmentList ?
        EDMAssignmentList ::=
        EDMAssignment
        EDMAssignmentList ?

EDMAssignment ::=
        EncodingClassAssignment         |
        EncodingStructureAssignment     |
        EncodingObjectAssignment        |
        EncodingObjectSetAssignment     |
        ParameterizedAssignment
```

The productions "Exports" and "Imports" (and their semantics) are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.1, as modified by Annex B.

The "Exports" makes available for import into other EDMs (and the ELM) any reference name defined in or imported into the current EDM. The "Symbol" in the "Exports" can reference any encoding class (except a built-in encoding class), an encoding object, or an encoding object set. The "Symbol" shall have been defined in this EDM, or imported into it.

The "Imports" makes available (within the EDM) encoding classes, encoding objects and encoding object sets exported from other EDMs or automatically exported from ASN.1 modules.

All ASN.1 modules that define or import type reference names into that module automatically produce and export an implicit encoding class of the same name preceded by the character "#". Such encoding classes can be imported into an EDM from that ASN.1 module.

NOTE Where such names are the same as built-in encoding class names, then the external form of reference has to be used, as specified in Annex B.

Each "EDMAssignment" defines a reference name, and may make use of other reference names. Each reference name used in a module shall either be imported into that module or shall be defined precisely once within that module.

There is no requirement that any reference name used in one assignment be defined (in another assignment statement) textually before its use.

The productions in "EDMAssignment" are defined in subsequent clauses as follows:

```
EncodingClassAssignment              Clause 15
EncodingObjectAssignment             Clause 16
EncodingObjectSetAssignment          Clause 17
ParameterizedAssignment              Annex D
```

NOTE The "ParameterizedAssignment" allows the parameterization of an "EncodingClassAssignment", an "EncodingObjectAssignment", and an "EncodingObjectSetAssignment".

15 Encoding class assignments

15.1 General

The "EncodingClassAssignment" is:

```
EncodingClassAssignment ::=
        encodingclassreference
        "::="
        EncodingClass
```

The "EncodingClassAssignment" assigns the "EncodingClass" to the "encodingclassreference".

Any "EncodingObject" notation that was valid with "EncodingClass" as a governor is valid with "encodingclassreference" as a governor.

An encoding class is in one of the following categories:

A bit-field class.

An alternatives class.

A concatenation class.

A repetition class.

An optionality class.

A general procedure class.

The "EncodingClass" is:

```
EncodingClass ::=
    DefinedEncodingClass     |
    EncodingStructure
```

Any "ExternalEncodingClassReference" which refers to an implicit encoding class is a bit-field class.

NOTE An "EncodingStructure" is always a bit-field class.

The "BuiltinEncodingClassReference" is:

```
BuiltinEncodingClassReference ::=
        BitfieldClassReference           |
        AlternativesClassReference       |
        ConcatenationClassReference      |
        RepetitionClassReference         |
        OptionalityClassReference        |
        GeneralProcedureClassReference
```

The category of these encoding classes is the category implied by the name of the production.

The "BitfieldClassReference" is:

```
BitfieldClassReference ::=
        #NUL                 |
        #BOOL                |
        #INT                 |
        #BITS                |
        #OCTETS              |
        #CHARS               |
        #PAD                 |
        #BIT-STRING          |
        #BOOLEAN             |
        #CHARACTER-STRING    |
        #EMBEDDED-PDV        |
        #ENUMERATED          |
        #EXTERNAL            |
        #INTEGER             |
        #NULL                |
        #OBJECT-IDENTIFIER   |
        #OCTET-STRING        |
        #OPEN-TYPE           |
        #REAL                |
        #RELATIVE-OID        |
        #GeneralizedTime     |
        #UTCTime             |
        #BMPString           |
        #GeneralString       |
        #IA5String           |
        #NumericString       |
        #PrintableString     |
        #TeletexString       |
        #UniversalString     |
        #UTF8String          |
        #VideotexString      |
        #VisibleString
```

The "AlternativesClassReference" is:

```
AlternativesClassReference ::=
        #ALTERNATIVES        |
        #CHOICE
```

The "ConcatenationClassReference" is:

```
ConcatenationClassReference ::=
        #CONCATENATION       |
        #SEQUENCE            |
        #SET
```

The "RepetitionClassReference" is:

```
RepetitionClassReference ::=
        #REPETITION          |
        #SEQUENCE-OF         |
        #SET-OF
```

The "OptionalityClassReference" is:

```
OptionalityClassReference ::=
        #OPTIONAL
```

The "GeneralProcedureClassReference" is::

```
GeneralProcedureClassReference ::=
        #TRANSFORM                    |
        #CONDITIONAL-INT              |
        #CONDITIONAL-REPETITION       |
        #OUTER
```

15.2 Encoding structure definition

The "EncodingStructureAssignment" is:

```
EncodingStructureAssignment ::=
        encodingclassreference
        "::="
        EncodingStructure
```

The "EncodingStructureAssignment" assigns the "EncodingStructure" to the "encodingstructurereference".

An "EncodingStructure" defines a structure-based encoding class using the notation specified below. This notation permits the definition of arbitrary encoding classes using built-in encoding classes and defined encoding classes (which may be implicit encoding classes) for bit-fields and for encoding constructors. All classes defined by "EncodingStructure" are bit-field classes. Examples of an encoding structure assignment illustrating many of the syntactic structures is given in E.2.9.

The "EncodingStructure" is:

```
EncodingStructure ::=
        DefinedEncodingClass              |
        EncodingStructureDefn
```

The "DefinedEncodingClass" shall be a bit-field class.

The "EncodingStructureField" is:

```
EncodingStructureField ::=
        PrimitiveField                |
        ComplexField

PrimitiveField ::=
        #NUL                          |
        #BOOL                         |
        #INT    Bounds?               |
        #BITS       Size?             |
        #OCTETS     Size?             |
        #CHARS          Size?         |
        #PAD

ComplexField ::=
        #BIT-STRING     Size?         |
        #BOOLEAN                      |
        #CHARACTER-STRING             |
        #EMBEDDED-PDV                 |
        #ENUMERATED Bounds?           |
        #EXTERNAL                     |
        #INTEGER        Bounds?       |
        #NULL                         |
        #OBJECT-IDENTIFIER            |
        #OCTET-STRING   Size?         |
        #OPEN-TYPE                    |
        #REAL                         |
        #RELATIVE-OID                 |
        #GeneralizedTime              |
        #UTCTime                      |
        #BMPString      Size?         |
        #GeneralString  Size?         |
        #GraphicString  Size?         |
        #IA5String      Size?         |
        #NumericString  Size?         |
        #PrintableString    Size?     |
        #TeletexString  Size?         |
        #UniversalString    Size?     |
        #UTF8String     Size?         |
```

```
        #VideotexString Size?       |
        #VisibleString  Size?
```

The "PrimitiveField"s represent all possible bit string encodings for corresponding ASN.1 types, and can be assigned values of those types in a value mapping.

The "Bounds" and "Size" specify the bounds or effective size constraint respectively on the abstract values that can be mapped to the field.

NOTE Effective alphabet constraints cannot be assigned in an encoding structure definition. They can only be assigned through the value mappings of clause 18.

"Bounds" and "Size" are:

```
Bounds ::= "(" EffectiveRange ")"

EffectiveRange ::=
            MinMax  |
            Fixed

Size       ::= "(" SIZE SizeEffectiveRange ")"

SizeEffectiveRange ::=
        "(" EffectiveRange ")" |
        EffectiveRange

MinMax ::=
        ValueOrMin
        ".."
        ValueOrMax

ValueOrMin  ::=
        SignedNumber |
        MIN

ValueOrMax  ::=
        SignedNumber |
        MAX

Fixed ::=  SignedNumber
```

"MIN" and "MAX" specify that there is no lower or upper bound respectively. "MIN" shall not be used in "Size". "Fixed" means a single value or a single size. "SignedNumber" is specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 18.1. It shall be non-negative when used in "Size". "ValueOrMin" and "ValueOrMax" specify lower and upper bounds respectively.

The "ComplexFields" represent the group of all bit string encodings for the corresponding ASN.1 type, and can be assigned such values in a value mapping. The encoding structures represented by these encoding classes are normally structures with more than one bit-field, and differ between standardized encoding rules. The precise encoding structures involved are not specified.

The ASN.1 values corresponding to each field are as follows:

```
#NUL                    The null value
#BOOL               The boolean values
#INT                    The integer values
#BITS                   Bit string values
#OCTETS             Octet string values
#CHARS              Character string values
#PAD                    None
```

The #PAD field shall not be assigned ASN.1 values, and is never visible outside the encoding and decoding procedures.

The "EncodingStructureDefn" is:

```
EncodingStructureDefn ::=
        AlternativesStructure          |
        RepetitionStructure            |
        ConcatenationStructure
```

These encoding structures are defined in the following clauses:

```
AlternativeStructure            15.3
RepetitionStructure             15.4
ConcatenationStructure          15.5
```

15.3 Alternative encoding structure

The "AlternativesStructure" is:

```
AlternativesStructure ::=
        AlternativesClass
        "{"
        NamedFields
        "}"

AlternativesClass ::=
        DefinedEncodingClass        |
        AlternativesClassReference

NamedFields ::= NamedField "," +

NamedField ::=
        identifier
        EncodingStructure
```

The "AlternativesStructure" identifies the presence in an encoding of precisely one of the "EncodingStructure"s in its "NamedFields". The "DefinedEncodingClass" shall be an alternatives class . The mechanisms used to identify which of the "EncodingStructure"s is present in an encoding are specified by an encoding object of the "AlternativesClass".

The "AlternativesStructure" is an encoding constructor: when an encoding object set is applied to this structure as specified in clause 13.2, the encoding of the "AlternativesClass" determines the selection of alternatives, and the application point then proceeds to each of the "EncodingStructure"s in its "NamedFields".

15.4 Repetition encoding structure

The "RepetitionStructure" is:

```
RepetitionStructure ::=
        RepetitionClass
        "{"
        EncodingStructure
        "}"
        Size?

RepetitionClass ::=
        DefinedEncodingClass        |
        RepetitionClassReference
```

The "RepetitionStructure" identifies the presence in an encoding of repeated occurrences of the "EncodingStructure" in the production. The optional "Size" construction specifies bounds on the number of repetitions. The mechanisms used to identify how many repetitions of the "EncodingStructure" are present in an encoding are specified by an encoding object of the "RepetitionClass" class. The "DefinedEncodingClass" shall be a repetition class .

The "RepetitionStructure" is an encoding constructor: when an encoding object is applied to this structure as specified in clause 13.2, the encoding of the "RepetitionClass" determines the mechanisms for determining the number of repetit ions, and the application point then proceeds to the "EncodingStructure" in the production.

15.5 Concatenation encoding structure

The "ConcatenationStructure" is:

```
ConcatenationStructure ::=
        ConcatenationClass
        "{"
        ConcatComponents
        "}"
```

```
ConcatenationClass ::=
        DefinedEncodingClass
        ConcatenationClassReference

ConcatComponents ::=
        ConcatComponent "," *

ConcatComponent ::=
        NamedField
        OptionalClass ?

OptionalClass ::=
        DefinedEncodingClass
        OptionalityClassReference
```

The "ConcatenationStructure" identifies the presence in an encoding of zero or one encodings for each of the "EncodingStructure"s in its "NamedField"s. The "DefinedEncodingClass" in the "ConcatenationClass" shall be a concatenation class, and the "DefinedEncodingClass" in the "OptionalClass" shall be an optionality class .

If "OptionalClass" is absent from a "Component", then the "EncodingStructure" in that named field shall appear precisely once in the encoding.

If "OptionalClass" is present, the mechanism used to determine whether there is an encoding of the corresponding "EncodingStructure" is specified by the encoding object which encodes the "OptionalClass"..

The order in which the encodings of each "NamedField" appear in an encoding of the concatenation (and the means of identifying which "NamedField" an encoding represents) is determined by an encoding object of the "ConcatenationClass" class.

The "ConcatenationStructure" is an encoding constructor: when an encoding object is applied to this structure as specified in clause 13.2 the encoding of the "ConcatenationClass" determines the concatenation procedures and the application point then proceeds to each of the "EncodingStructure"s in its named fields.

16 Encoding object assignments

16.1 Categories of encoding object assignments

The "EncodingObjectAssignment" is:

```
EncodingObjectAssignment ::=
        encodingobjectreference
        EncodingClass
        "::="
        EncodingObject
```

The "EncodingObjectAssignment" defines the "encodingobjectreference" as an encoding object reference to the "EncodingObject", which is required to be a production which generates an object of the encoding class "EncodingClass". E.1.3, E.1.8 and E.1.9 provide examples of encoding object assignment for the different syntactic constructions for "EncodingObject" specified below.

The "EncodingClass" is called the governor of the "EncodingObject" notation in this production.

NOTE 1 – Whenever the "EncodingObject" production appears in ECN, there is a governor, and the syntax of the governed notation depends on the encoding class of the governor.

NOTE 2 – The syntax of the governed notation has been designed so that a parser can find the end of it without knowledge of the governor.

The "EncodingObject" is:

```
EncodingObject ::=
        DefinedEncodingObject                |
        DefinedSyntax                        |
        EncodeWith                           |
        EncodeByValueMapping                 |
        EncodeStructure                      |
        DifferentialEncodeDecodeObject       |
        UserDefinedEncodingFunction
```

"DefinedEncodingObject" identifies an encoding object. The "DefinedEncodingObject" shall be of the same encoding class as the governor. In this ECN TS, "the same encoding class" shall be interpreted as meaning that the notation used for defining the two classes shall be the same encoding class reference name.

The remaining productions of "EncodingObject" are defined in the following clauses and provide alternative means of defining encoding objects of the governor class:

```
DefinedSyntax                          16.2 modified by Annex A
EncodeWith                             16.3
EncodeByValueMapping                   16.4
EncodeStructure                        16.5
DifferentialEncodeDecodeObject         16.6
UserDefinedEncodingFunction            16.7
```

16.2 Encoding with a defined syntax

The "DefinedSyntax" production is specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 11.5, as modified by Annex C, and is used for the definition of encoding objects with a governing encoding classes, as specified in Annex A.

This notation for defining encoding objects is only available for the governing encoding classes listed in Table 3 below. The syntax to be used for each class is the "DefinedSyntax" for the encoding class (specified in Annex A) with the name listed in column 1.

NOTE The use of this syntax frequently requires the inclusion of a parameter for a determinant. Parameterized encoding objects with such parameters are only useful for application to an encoding structure in the EDM, they cannot be applied in the ELM.

Table 3: Classes supported by a defined syntax

Encoding Class	Used also for:
#NUL	#NULL
#BOOL	#BOOLEAN
#INT	#INTEGER and #ENUMERATED
#BITS	#BIT-STRING
#OCTETS	#OCTET-STRING
#CHARS	#BMPString, #GeneralString, #GraphicString, #IA5String, #NumericString, #PrintableString, #TeletexString, #UniversalString, #UTF8String, #VideotexString, and #VisibleString
#PAD	
#ALTERNATIVES	#CHOICE
#REPETITION	#SEQUENCE-OF and #SET-OF
#CONCATENATION	#SEQUENCE and #SET
#OPTIONAL	
#CONDITIONAL-INT	
#CONDITIONAL-REPETITION	
#TRANSFORM	
#OUTER	

NOTE This notation enables users to specify encoding objects which encode #SET in the way PER normally encodes #SEQUENCE, and vice versa. Users are expected to be responsible in their use of this notation.

The "DefinedSyntax" for #CHARS can be used for any of: #BMPString, #GeneralString, #GraphicString, #IA5String, #NumericString, #PrintableString, #TeletexString, #UniversalString, #UTF8String, #VideotexString, and #VisibleString.

The information required to specify an encoding object of one of these classes is specified by the definition of the encoding class in Annex A.

If a governor for a value of one of the fields of the encoding class is needed for use in a dummy parameter list, then the notation "EncodingClassFieldType" shall be used.

The semantics associated with the encoding class definitions in Annex A is specified for each encoding class in clause 19.

Where the syntax defined in Annex A requires the provision of a REFERENCE, this can only be supplied in this construction by using a dummy parameter of the encoding object that is being defined.

16.3 Encoding with encoding object sets

The "EncodeWith" is:

```
EncodeWith ::=
        "{" ENCODE CombinedEncodings "}"
```

"CombinedEncodings" and its application to an encoding class is specified in clause 13.

The encoding object defined by the "EncodeWith" is the application of the "CombinedEncodings" to the encoding class that is the governor of the "EncodeWith" notation.

It is a specification error if this does not produce a complete encoding specification for the governor class.

If an encoding object set in the "CombinedEncodings" is parameterized with a parameter that is a REFERENCE, the actual parameter supplied in this construction can only be a dummy parameter of the encoding object that is being defined.

16.4 Encoding using value mappings

The "EncodeByValueMapping" is:

```
EncodeByValueMapping ::=
        "{"
        USE
        EncodingClass
        MAPPING
        ValueMapping
        WITH EncodingObject
        "}"
```

The production "EncodingClass" and its semantics is defined in 15.1. It shall be a bit-field class .

The production "ValueMapping" and shall be a mapping of values from the governor encoding class to the class identified by the "EncodingClass".

The "EncodingObject" shall define an encoding object using notation governed by the class specified by the "EncodingClass" .

The syntax for "EncodingObject" allows both in-line definition of encoding objects (recursive application of this clause) and the use of reference names. E.2.10 gives an example of in-line definition to perform two value mappings in a single assignment.

Where the "EncodingObject" requires the provision of a REFERENCE, this can only be supplied in this construction by using a dummy parameter of the encoding object that is being defined.

Where there are bounds on fields of the "EncodingClass", then values shall not be mapped to those fields that violate the specified bounds.

16.5 Encoding an encoding structure

The "EncodeStructure" is:

```
EncodeStructure ::=
        "{"
        ENCODE STRUCTURE
        "{"
        ComponentEncodingList
        StructureEncoding ?
        "}"
        CombinedEncodings ?
         "}"
```

```
StructureEncoding ::=
        STRUCTURED WITH
        EncodingObject
```

The "EncodeStructure" can be used to define an encoding only if the governor encoding class (after resolution of references) starts with one of the following encoding structures (called the governing encoding constructor):

```
        #ALTERNATIVES
        #CHOICE
        #CONCATENATION
        #REPETITION
        #SEQUENCE
        #SET
        #SEQUENCE-OF
        #SET-OF
```

The "EncodingObject" in the "StructureEncoding", if this production is present, shall be an encoding for the governing constructor encoding, and shall encode that encoding class. If the production is absent, the "CombinedEncodings" shall be present, and shall contain an encoding object of the class of the governing encoding constructor, otherwise the ECN specification is in error. The encodings assigned to the governing constructor are those of this encoding object.

The "ComponentEncodingList" is:

```
ComponentEncodingList ::=
        ComponentEncoding "," *

ComponentEncoding ::=
        NonOptionalComponentEncodingSpec     |
        OptionalComponentEncodingSpec
```

There shall be precisely one "ComponentEncoding" for each component of the governing encoding constructor. The "ComponentEncoding"s shall be in textual order.

NOTE The absence (and assignment to components) of "ComponentEncoding"s can be detected by following named fields, or by the end of the "ComponentEncodingList".

The "NonOptionalComponentEncodingSpec" shall not be used if the component is marked OPTIONAL.

The "OptionalComponentEncodingSpec" shall only be used if the component is marked OPTIONAL.

If the "ComponentEncodingList" is empty, then the "CombinedEncodings" must be present, and is required, on application to the component to provide a complete encoding of that component, otherwise it is an error in the ECN specification.

```
NonOptionalComponentEncodingSpec ::=
        identifier ?
        EncodingObject1

OptionalComponentEncodingSpec ::=
        ComponentEncodingObject                         |
        OptionalEncodingObject                          |
        ComponentAndOptionalEncodingObject

ComponentEncodingObject ::=
        identifier
        EncodingObject1

OptionalEncodingObject ::=
        identifier
        OPTIONAL-ENCODING
        EncodingObject2

ComponentAndOptionalEncodingObject ::=
        identifier
        EncodingObject1
        OPTIONAL-ENCODING
        EncodingObject2

EncodingObject1 ::= EncodingObject
EncodingObject2 ::= EncodingObject
```

The "FieldReference" shall be the "identifier" of the component of the governor. The "identifier" in "NonOptionalComponentEncodingSpec" shall be omitted if and only if the governing encoding constructor is "REPETITION", "SEQUENCE-OF", or "SET-OF".

"EncodingObject1" (if present) in the "ComponentEncoding" shall be governed by the encoding class of the component and shall provide a complete encoding of that component. If it is absent, then the "CombinedEncodings" shall be present in the "EncodeStructure", and shall provide a complete encoding of the component, otherwise it is an error in the ECN specification.

NOTE Use of the "ENCODE WITH" form of "EncodingObject" effectively enables an encoding object set to be applied to the component. (See E.1.3 for an example.)

"EncodingObject2" (if present) in the "OptionalComponentEncodingSpec" shall completely encode the #OPTIONAL encoding class of that component. If it is absent, then the "CombinedEncodings" must be present, and is required to provide an encoding object of the class #OPTIONAL which encodes the optionality of the component, otherwise it is an error in the ECN specification.

If a REFERENCE is needed as an actual parameter of any of the encoding objects or encoding object sets used in this production, then it can either be supplied as a dummy parameter of the encoding object that is being defined, or it can be supplied as any of the "encodingstructurefieldreference"s that are textually present in the construction. If the REFERENCE is required to identify a container, it can also be supplied as "*" (provided the structure being encoded is not an #ALTERNATIVES or a #CHOICE) when it refers to that structure.

NOTE This is the only production in which REFERENCEs can be supplied, except through the use of dummy parameters.

16.6 Differential encoding-decoding

The "DifferentialEncodeDecodeObject" is:

```
DifferentialEncodeDecodeObject ::=
        "{"
        ENCODE-DECODE
        SpecForEncoding
        DECODE AS IF
        SpecForDecoders
        "}"

SpecForEncoding ::= EncodingObject

SpecForDecoders ::= EncodingObject
```

The "DifferentialEncodingObject" specifies separately rules for encoding values associated with the class of the governor of this notation, and rules to be used by decoders for recovering abstract values from encodings that are assumed to have been produced by encoding objects of the class of the governor.

The "SpecForEncoding" shall be applied by encoders. Decoders shall decode as if the encoder had applied the "SpecForDecoders".

NOTE 1 - The "SpecForDecoders" is still an encoding specification. It tells decoders to assume that encoders have used this specification.

NOTE 2 - The behavior of decoders that decode on the assumption that an encoder has used the "SpecForDecoders", but detect encoding errors, is not standardized.

The SpecForEncoding and the SpecForDecoding encoding objects shall not have been defined using "ENCODE-DECODE", nor shall any encoding objects used in their definition have been defined using "ENCODE-DECODE".

16.7 User-defined encoding-functions

The "UserDefinedEncodingFunction" is:

```
UserDefinedEncodingFunction ::=
        USER-FUNCTION-BEGIN
        AssignedIdentifier
        anystringexceptuserfunctionend
        USER-FUNCTION-END
```

The "UserDefinedEncodingFunction" shall specify an encoding object of the governor class . The notation used to do this is contained in "anystringexceptuserfunctionend" and is not standardized.

The production "AssignedIdentifier" and its semantics is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.1, as modified by Annex B. It identifies the notation used in the "anystringexceptuserfunctionend" to specify the encoding.

If the "empty" alternative of "AssignedIdentifier" is used, then the notation is determined by means outside of the ECN specification.

The assignment of object identifiers to any notation for use in "anystringexceptuserfunctionend" follows the normal rules for the assignment of object identifiers as specified in the ITU-T Rec. X.660 | ISO/IEC 9834 series.

17 Encoding object set assignments

The "EncodingObjectSetAssignment" is:

```
EncodingObjectSetAssignment ::=
        encodingobjectsetreference
        #ENCODINGS
        "::="
        EncodingObjectSet

EncodingObjectSet ::=
        DefinedEncodingObjectSet |
        EncodingObjectSetSpec
```

The "EncodingObjectSet" notation is governed by the reserved word #ENCODINGS, and shall satisfy the conditions given below.

"DefinedEncodingObjectSet" is defined in clause 10.

The "EncodingObjectSetSpec" is:

```
EncodingObjectSetSpec ::=
        "{"
            EncodingObjects UnionMark *
        "}"

EncodingObjects ::=
        DefinedEncodingObject    |
        DefinedEncodingObjectSet

UnionMark ::=
        "|" |
        UNION
```

"EncodingObjectSetSpec" defines an encoding object set using one or more encoding objects or encoding object sets.

Encoding objects forming an encoding object set shall all be of distinct encoding classes, and shall not be of the #TRANSFORM, #CONDITIONAL-INT, or #CONDITIONAL-REPETITION encoding classes.

NOTE An encoding object set is used for defining other encoding object sets, for defining encoding objects in the EDM, and for import into the ELM for the application of encodings.

Built-in encoding object sets

The "BuiltinEncodingObjectSetReference" is:

```
BuiltinEncodingObjectSetReference ::=
        PER-basic-aligned                       |
        PER-basic-unaligned                     |
        PER-canonical-aligned                   |
        PER-canonical-unaligned                 |
        BER                                     |
        CER                                     |
        DER
```

These encoding object set names are the sets of encoding objects defined by ITU-T Rec. X.690 | ISO/IEC 8825-1 and ITU-T Rec. X.691 | ISO/IEC 8825-2. The object identifiers for the encoding rules providing these encoding object sets are given in Table 4.

Table 4: Encoding object set names and associated object identifiers

PER-basic-aligned	{joint-iso-itu-t(1) packed-encoding(3) basic(0) aligned(0)}
PER-basic-unaligned	{joint-iso-itu-t(1) packed-encoding(3) basic(0) unaligned(1)}
PER-canonical-aligned	{joint-iso-itu-t(1) packed-encoding(3) canonical(1) aligned(0)}
PER-canonical-unaligned	{joint-iso-itu-t(1) packed-encoding(3) canonical(1) unaligned(1)}
BER	{joint-iso-itu-t(1) asn1(1) basic-encoding(1)}
CER	{joint-iso-itu-t(1) asn1(1) ber-derived(2) canonical-encoding(0)}
DER	{joint-iso-itu-t(1) asn1(1) ber-derived(2) distinguished-encoding(1)}

These encoding object sets are each a complete set of encoding objects which can be applied to any encoding structure (either implicitly generated from an ASN.1 type or user-defined) to specify the corresponding BER or PER encodings.

All the above sets contain the same encoding objects for the classes #INT, #BOOL, #NUL, #CHARS, #OCTETS, #BITS, #CONCATENATION, #REPETITION. They do **not** contain an encoding object for #ALTERNATIVES.

NOTE Use of encoding objects from the built-in encoding object sets for these classes is not expected to be heavily used, but is defined here for completeness.

These encoding classes represent basic building blocks of encodings, and are encoded simply by all the above encoding object sets. (Any encoding that would involve mapping to a more complex encoding structure, or would require a determinant, was not felt appropriate for inclusion in these built-in encoding object sets.)

NOTE This is why there is no support for #ALTERNATIVES and optional elements, or for unbounded integers, strings, repetitions, and so on. Support for these has to be defined explicitly by the ECN user.

The encoding objects for these classes specify encodings as follows:

1. #INT is encoded as a PER-basic-unaligned #INTEGER encoding, provided it is bounded. It is an ECN design error if the #INT does not have both a lower and an upper bound, and this encoding object is applied to it.
2. #BOOL and #NUL are encoded as PER-basic-unaligned #BOOLEAN and #NULL respectively.
3. #CHARS, #OCTETS, and #BITS are encoded as PER-basic-unaligned "UTF8String", #OCTET-STRING, and #BIT-STRING, respectively, provided they are a single size. It is an ECN design error if #CHARS, #OCTETS, or #BITS do not have an effective size constraint restricting them to a single size.
4. #CONCATENATION is encoded as a PER-basic-unaligned encoding of a #SEQUENCE **with no optional elements**. Any optional elements have to have their optionality resolved by use of encoding objects for #OPTIONAL, **which are not present in any of these encoding object sets**.
5. #REPETITION is encoded as a PER-basic-unaligned encoding of #SEQUENCE-OF provided it has a PER-visible size constraint restricting it to a fixed number of iterations. It is an ECN design error if it does not have such a size constraint.

NOTE The BER encoding objects for #OPTIONAL, #SET and #CHOICE require that the components display an identification handle that is a BER tag class and number. Such identification handles can only be obtained in the current version of ECN by using BER encoding objects to encode the components, so BER encoding objects should be used with caution unless they are applied to encoding structures implicitly generated from ASN.1 types.

18 Mapping values

18.1 General

This clause specifies the syntax for mapping values to be encoded by the fields of one encoding class (which may be an implicit encoding class or may be any other encoding class) to the fields of another encoding class.

NOTE The power provided in a single use of this notation has been limited (to avoid complexity). More complex mappings can be achieved by using multiple instances of "EncodeByValueMapping" (see the example in E.1.11). These mapping mechanisms can be extended and generalized, but this will not be done unless further user requirements are identified.

In specifying the "EncodeByValueMapping" notation the "EncodingClass" in the "EncodingObjectAssignment" of which it is a part, is called the source governor or the source encoding class (depending on context). The "EncodingClass" in the "EncodeByValueMapping" itself is called the target governor or the target encoding class (depending on context).

The encodings specified for values mapped to the target encoding class become encodings of those values in the source encoding class.

NOTE 1 – If the total ECN specification maps only some of the values from an ASN.1 type into encodings, that is not an error. It is a constraint imposed by ECN on the values that can be used by the application. Such constraints should normally be identified by comment in either the ASN.1 specification or in the ELM specification.

NOTE 2 – If the total ECN specification maps two values into the same encoding produced by a single encoding object, then that is an ECN specification error. Such errors can be detected by ECN tools, but rules for their avoidance are not complete in this ITU-T TS, and responsibility rests with the ECN user.

The "ValueMapping" is:

```
ValueMapping ::=
        MappingByExplicitValues                 |
        MappingByMatchingFields                 |
        MappingByTransformEncodingObjects       |
        MappingByAbstractValueOrdering          |
        MappingByValueDistribution              |
        MappingIntToBits
```

Tutorial note: All occurrences of this syntax are preceded by the reserved word "MAPPING". E.1.3, E.1.5, E.1.11, and E.2.2 and Annex G give examples of the definition of encodings using each of these value mappings.

The "ValueMapping" productions are specified as follows:

```
        MappingByExplicitValues              18.2
        MappingByMatchingFields              18.3
        MappingByTransformEncodingObjects    18.4
        MappingByAbstractValueOrdering       18.5
        MappingByValueDistribution           18.6
        MappingIntToBits                     18.7
```

NOTE It is frequently the case that several of the value mappings can be used to define the same encoding, but some will produce a more obvious or less verbose specification than others. ECN designers should select carefully the form of value mapping to be used.

18.2 Mapping by explicit values

This clause provides notation for specifying the mapping of values between different primitive encoding classes. E.1.11 gives an example.

This clause uses the notation for ASN.1 values (ASN.1 value notation) specified in ITU-T Rec. X.680 | ISO/IEC 8824-1 for the type which corresponds to an encoding class.

The following table specifies the encoding classes that can act as governors for ASN.1 value notation, and the value notation to be used for each governor. In each case the class may or may not have an associated size or value range constraint.

ECN supports mapping by explicit values (either to or from the encoding class) for all classes listed in column 1 of Table 5. Column 2 of the table specifies the value notation (as either an ASN.1 production or by reference to a clause of ITU-T Rec. X.680 | ISO/IEC 8824-1 or both) that shall be used when the encoding class in column 1 is specified as the governor of the notation. It also specifies the clause in ITU-T Rec.X.680 | ISO/IEC 8824-1 that defines the value notation.

NOTE None of the following ASN.1 value notations can use "DefinedValue"s (as defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 13.1) because "valuereference"s cannot be imported nor defined in an EDM or ELM module.

Table 5: Encoding classes (and value notation) used in mapping by explicit values

Governing encoding class	ASN.1 value notation
#BITS or #BIT-STRING	**"bstring" or "hstring"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.9 and 11.10)
#BMPString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#BOOL or #BOOLEAN	**"BooleanValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 17.3)
#CHARS	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#ENUMERATED	**"SignedNumber"** (present in the EnumeratedType or determined by ITU-T Rec. X.680 | ISO/IEC 8824-1, 19.3)
#GeneralizedTime	**"cstring"** conforming to ITU-T Rec. X.680 | ISO/IEC 8824-1, 41.3)
#GeneralString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#GraphicString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#IA5String	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#INT or #INTEGER	**"SignedNumber"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 18.1)
#NUL or #NULL	**"NullValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 23.3)
#NumericString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#OBJECT-IDENTIFIER	**"DefinitiveIdentifier"** (see Annex B)
#OCTETS or #OCTET-STRING	**"bstring" or "hstring"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.9 and 11.10)
#PrintableString	"RestrictedCharacterStringValue" **(see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)**
#REAL	**"RealValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 20.6)
#RELATIVE-OID	**"RelativeOIDValue"** (see ITU-T Rec. X.680 Amd. 1 | ISO/IEC 8824-1 Amd. 1, 31bis.3)
#TeletexString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#UniversalString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#UTCTime	**"cstring"** (conforming to ITU-T Rec. X.680 | ISO/IEC 8824-1, 42.3)
#UTF8String	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#VideotexString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)
#VisibleString	**"RestrictedCharacterStringValue"** (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7)

The "MappingByExplicitValues" is:

```
MappingByExplicitValues ::=
        VALUES
        "{"
        MappedValues "," +
        "}"

MappedValues ::=
        MappedValue1
        TO
        MappedValue2

MappedValue1 ::= Value

MappedValue2 ::= Value
```

The "MappedValue1" shall be value notation governed by the source governor and "MappedValue2" shall be value notation governed by the target governor . The value in the source specified by "MappedValue1" is mapped to the value in the target specified by "MappedValue2".

There are no bounds or effective size constraints associated with the target encoding class as a result of the "MappingByExplicitValues", but any already present shall not be violated.

18.3 Mapping by matching fields

This mapping is provided primarily to enable the encoding of an ASN.1 type to be defined as the encoding of an encoding structure that has fields corresponding to the components of the type, but also has added fields for determinants.

The "MappingByMatchingFields" is:

```
MappingByMatchingFields ::=
        FIELDS
```

If either the source or the target encoding classes are structure-based encoding classes or implicit encoding classes, then the references are resolved. After resolution, the source and the target encoding classes shall start with the same encoding constructor (#CONCATENATION, #SEQUENCE, #SET, #REPETITION, #SEQUENCE-OF, #SET-OF, #ALTERNATIVES or #CHOICE), and the resulting encoding structures are called the source and target encoding structures respectively.

No further resolution of references takes place during these procedures.

All fieldnames that are now visible in the source encoding structure shall be distinct. All fieldnames that are now visible in the target encoding structure shall be distinct.

NOTE Fieldnames in unresolved references are not visible to these procedures.

For every fieldname that is visible in the source encoding structure, there shall be a component in the target encoding structure with the same fieldname and with the identical encoding class.

All abstract values are mapped from each of the fields in the source encoding structure to the fields with the same name (and encoding class) in the target encoding structure. Additional fields in the target encoding structure do not acquire abstract values. In a correct ECN specification, the value of such fields has to be specified by reference as a determinant.

Bounds and effective size and alphabet constraints on source fields are mapped to the target fields, and replace any bounds and effective size and permitted alphabet constraints already present on the target field.

NOTE Any bounds, effective size and permitted alphabet constraints on the target field are always lost in this mapping.

18.4 Mapping by #TRANSFORM encoding objects

This mapping permits one or more transform encoding objects to be applied to produce the mapping.

The transform encoding class is defined in Annex A. It enables encoding objects to be specified for the following transformations (for example):

a) Mappings of #INT to #INT using simple numerical transformations with constant values (add 1, divide-by 2, etc)

b) Mappings of #INT to #CHARS

c) Mappings of #INT to #BITS

NOTE 1 – Examples of mappings defined with these transformations are given in E.1.3 and E.1.4. The example in E.1.7 shows the use of this production to define BCD encodings of an #INTEGER.

NOTE 2 – One of the possible uses of the #TRANSFORM class is to produce a count in bits (or octets) from a determinant field that has a count in octets (or bits), or to offset such a field.

The "MappingByTransformEncodingObjects" is:

```
MappingByTransformEncodingObjects ::=
        TRANSFORMS
        "{"
        TransformList
        "}"

TransformList ::= Transform "," +

Transform ::= EncodingObject
```

All the "EncodingObject"s in the "TransformList" shall be governed by the encoding class #TRANSFORM. Transform encoding objects are defined with a source encoding class and a target encoding class specified. The source encoding class of the first transform must be the same as the source encoding class for this mapping, and the target encoding class of the last transform must be the same as the target encoding class of this mapping. All other transforms in the list must have a source encoding class which is the same as the target encoding class of the previous transform and a target encoding class which is the same class as the source encoding class of the next transform.

Abstract values are mapped through the successive application of the transforms from the first to the last (in their textual order).

If any bounds listed for the target encoding class are violated in the mapping (by any of the abstract values in the source encoding class), this is not an error, but such values are not mapped, and do not appear in the target encoding class. Thus, there may be no encoding in the resulting specification for such values, and such a restriction should be identified by comment in the ASN.1 or in the ELM.

18.5 Mapping by abstract value ordering

This is a very powerful form of mapping which enables abstract values associated with simple encoding classes to be distributed into the fields of complex encoding structures, and for abstract values associated with complex encoding structures to be mapped to simple encoding classes such as #INT. It is also a means of "compacting" integer values or enumerations into a contiguous set of integer values.

The "MappingByAbstractValueOrdering" is:

```
MappingByAbstractValueOrdering ::=
        ORDERED VALUES
```

For this mapping, all encoding structure reference names are resolved (recursively), and the resulting encoding structure shall contain only #ALTERNATIVES, #CHOICE, #NUL, #NULL, #BOOL, #BOOLEAN, #INT, #INTEGER, #REAL, #ENUMERATED.

An ordering of abstract values is defined for the following encoding classes:

```
#NUL and #NULL
#BOOL and #BOOLEAN
#INT and #INTEGER (but only if they have a lower bound)
#REAL (but only if it is constrained
               to a finite number of values)
#ENUMERATED
```

and any #ALTERNATIVES or #CHOICE containing alternatives all of which have a defined ordering.

#NUL and #NULL have a single abstract value. #BOOL and #BOOLEAN are defined to have "TRUE" before "FALSE". #INT and #INTEGER are defined to have higher integer values following lower integer values. #REAL is defined to have higher values following lower values. #ENUMERATED is defined to have the enumerations in the order of the numerical value assigned to them, higher values following lower values.

NOTE The number of abstract values in #INT and #INTEGER is not necessarily finite.

Any bounds present in the source or destination shall be taken fully into account in determining the ordered set of abstract values.

The ordering of the abstract values of an #ALTERNATIVES or #CHOICE (all of whose alternatives have a defined ordering of abstract values) is defined to be the (ordered) abstract values from the textually first alternative, followed by those from the textually second alternative, and so on to the textually last alternative.

The mapping is defined from the abstract values in the first encoding class to the abstract values in the second encoding class by their position in the above ordering.

Note that the above rules ensure that there is a defined first value in each ordering, and a defined next value. There need not be a defined last value (either or both sets may be infinite).

If the number of abstract values in the destination ordering is less than the number of abstract values in the source ordering, this is not an error. However, the ECN specification will be unable to encode some of the abstract values of the ASN.1 specification and this should be identified by comment in either the ASN.1 module or the ELM.

If the number of abstract values in the destination ordering exceeds those in the source ordering, then there may be some ECN-defined encodings that have no ASN.1 abstract value, and will never be generated.

18.6 Mapping by value distribution

This mapping takes ranges of values from a #INT or #INTEGER encoding class, mapping each range to a different #INT or #INTEGER field in a more complex encoding structure. Fields which receive no abstract values shall have their values determined by the application of determinants.

All encoding structure reference names are resolved (recursively) before the application of this mapping.

The source encoding class shall then be either #INT or #INTEGER.

The target encoding class may be any encoding structure, but all fieldnames in the entire encoding structure shall be distinct.

The "MappingByValueDistribution" is:

```
MappingByValueDistribution ::=
        DISTRIBUTION
        "{"
        Distribution "," +
        "}"

Distribution ::=
        SelectedValues
        TO
        identifier

SelectedValues ::=
        SelectedValue       |
        DistributionRange   |
        REMAINDER

DistributionRange ::=
         DistributionRangeValue1
        ".."
        DistributionRangeValue2

SelectedValue ::= SignedNumber

DistributionRangeValue1 ::= SignedNumber

DistributionRangeValue2 ::= SignedNumber
```

"SignedNumber" is specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 18.1.

"DistributionRangeValue1" shall be less than "DistributionRangeValue2".

The value specified by "SelectedValue" in "SelectedValues", or the set of values greater than or equal to "DistributionRangeValue1" and less than or equal to "DistributionRangeValue2", are mapped to the field specified by "identifier".

The reserved word "REMAINDER" shall only be used once for the last "SelectedValues", and specifies all abstract values in the source encoding class that have not been distributed by earlier "SelectedValues".

A value shall not be mapped to more than one target field, but several "SelectedValues" may have the same "Destination".

Values shall not be mapped to the target that violate any bounds present on the target. This mapping does not affect the bounds on the target.

18.7 Mapping integer values to bits

This mapping takes single values or ranges of values from a #INT or a #INTEGER class, mapping each integer value to a bitstring value.

NOTE This mapping is intended to support self-delimiting encodings of integers, such as Huffman encodings. See Annex F for further discussion and examples of Huffman encodings.

The source encoding class shall be either #INT or #INTEGER, or any implicit encoding class or encoding structure that resolves to #INT or #INTEGER after de-referencing.

The destination encoding class shall be #BITS.

The "MappingIntToBits" is:

```
MappingIntToBits ::=
        TO BITS
        "{"
        MappedIntToBits "," +
        "}"

MappedIntToBits ::=
        SingleIntValMap
        IntValRangeMap
```

Each "SingleIntValMap" maps a single integer value to a single bit string value.

Each "IntValRangeMap" maps a range of contiguous and increasing integer values to a range of contiguous and increasing bit string values.

Bit string values are defined to be contiguous if:

a) They are all the same length in bits.

b) When interpreted as a positive integer value, the corresponding integer values are contiguous and increasing integer values.

Only values specified in the mapping are encodable. Other abstract values of the source are not mapped and cannot be encoded by the encoding object defined by the encoding object assignment using this construct.

NOTE This limitation of the encoding should be reflected by constraints on the ASN.1 type to which it is applied, or by comment in the ASN.1 specification or in the ELM.

The "SingleIntValMap" is:

```
SingleIntValMap ::=
        IntValue
        TO
        BitValue

IntValue ::= SignedNumber
BitValue ::=
         bstring    |
         hstring
```

The "SignedNumber" is specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 18.1.

The "BitStringValue" shall be the value notation for an ASN.1 "BIT STRING" type with no named bits, as specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 21.9.

The "SingleIntValMap" maps the specified integer value to the specified bit string value.

The "IntValRangeMap" is:

```
IntValRangeMap ::=
```

```
            IntRange
            TO
            BitRange

    IntRange ::=
            IntRangeValue1
            ".."
            IntRangeValue2

    BitRange ::=
            BitRangeValue1
            ".."
            BitRangeValue2

    IntRangeValue1 ::= SignedNumber
    IntRangeValue2 ::= SignedNumber
    BitRangeValue1 ::=
            bstring |
            hstring
    BitRangeValue2 ::=
            bstring |
            hstring
```

The bit strings "BitRangeValue1" and "BitRangeValue2" shall be the same number of bits.

The value "IntRangeValue2" shall be greater than the value "IntRangeValue1".

When interpreted as a positive integer encoding (see ITU-T Rec. X.690 | ISO/IEC 8825-1, 8.3.3), "BitRangeValue2" shall represent an integer value ("B", say) greater than that represented by "BitRangeValue1" ("A", say), and the difference between the integer values corresponding to "BitRangeValue2" and "BitRangeValue1" ("B" - "A") shall equal the difference between the values of "IntRangeValue2" and "IntRangeValue1".

The "BitRange" represents the ordered set of bit strings corresponding to the integer values between "A" and "B".

The "IntValRangeMap" maps each of the integers in the specified range to the corresponding bit string value in the "BitRange". Annex F gives examples of an "IntValRangeMap".

19 Built-in encoding classes supported by defined syntax

This clause describes the use of defined syntax to define encoding objects of the classes listed in Annex A.

19.1 General

Annex A specifies a number of encoding classes using the information object class notation of ITU-T X.681 | ISO/IEC 8824-2 as modified by Annex C.

NOTE This TS uses the term "fields of an information (encoding) class". In this clause we use the term "parameters of the encoding class" to avoid confusion with the term "field of an encoding structure". The term "parameter" should, however, not be confused with the use of parameterized assignments.

The use of this notation is restricted to the definition within this TS of a defined syntax for the definition of encoding objects of standardized built-in encoding classes.

NOTE It is recognized that tool vendors may choose to provide extensions to their ECN support using this notation. Such activity is deprecated, as it will reduce inter-working between tools. Where deficiencies are found in Annex A they should be addressed in a defect report and resolved through the normal standardization process.

The use of the defined syntax notation to define encoding objects is specified in 16.2. The defined syntax for defining encoding objects shall be the syntax specified by the "WITH SYNTAX" clauses in Annex A.

The "WITH SYNTAX" clauses impose constraints on the setting of some parameters in conjunction with other parameters to enforce some (but not all) of the semantic constraints expressed in this clause. The semantic constraints in this clause shall be satisfied, even if not enforced by the "WITH SYNTAX".

The defined syntax for each encoding class specifies a number of parameters which require to be supplied with values of the ASN.1 types that are defined in Annex A (or in some cases with encoding objects) in order to provide the information needed to specify an encoding object of that class.

The parameters specified in Annex A operate together in groups. This sub-clause specifies the meaning of values of the types used in the specification of these parameters. Sub-clause 19.3 gives the semantics of the #TRANSFORM encoding class Sub-clauses 19.4 to 19.21 specify the semantics of the groups of parameters (and the encoding classes that use them). Sub-clause 19.22 defines the semantics of the #OUTER encoding class. **It is intended that all these clauses be read in conjunction with the syntactic specifications in Annex A to which they refer.**

In specifying the semantics of the parameter groups, each group name is listed in angle brackets, followed by the encoding classes that use that group (see also Annex A), and an explanation of the semantics of any of the values of those parameters that have not been specified in earlier text.

19.2 Common types

NOTE These common types are formally defined in Annex A. Their use in specific encoding definitions is specified in 19.4 to 19.21.

The type "Unit" specifies a unit for alignment ("1" meaning no alignment), or for the specification of sizes, when it is used in conjunction with "Size". Values are a count in bits, except for "repetitions(0)", which specifies a count of the number of repetitions in a repetition class.

"Size" is used to specify the size of an encoding-space.

The value "uses-determination-mechanism(-3)" is used to specify that the size will vary according to the abstract value being encoded, and that the precise means of determining the size of the encoding-space will be specified using a setting of "Determination-Mechanism" . In this case, there shall be a setting of a parameter giving a value of "Determination-Mechanism".

NOTE The specification of a "Determination-Mechanism" (to determine the encoding-space size) is required in this case, but the provision of a determinant is allowed in all the other cases, to support encodings (similar to BER) that use determinants even when they are redundant.

The value "fixed-to-max(-2)" is used to specify that the encoding-space is to be the same for the encoding of all abstract values of the field to which the encoding object is applied, and is to be the largest size needed. If such a specification is given and applied when a maximum size cannot be determined, this is an ECN specification error.

The value "variable(-1)" is not used for encoding-space sizes. It is allowed only when specifying the size of a target encoding within #TRANSFORM specification. It specifies that the target abstract value (a bitstring or character string) will be of variable length.

The value "empty(0)" specifies that the encoding-space is of zero size, and hence contains no bits. It is permitted only for the encoding of #NUL.

All positive values "x", say, specify an encoding space which is "x" times the value of "Units".

The "DeterminationMechanism" type is used to specify the way in which the size of a variable length encoding-space is to be determined, or the way in which alternatives are selected, or the way in which optionality is resolved.

The value "aux-determinant" specifies that there will be, within the encoding structure, a field that is used to provide the necessary determinant, and that this is an auxiliary field that does not carry application semantics (does not appear within the ASN.1 specification). In this case, the specification will also include parameters that say how an encoder determines the value of this field (possibly using one or more transformations) from:

- the size of the encoding-space (in "Units"); or
- the index of the selected alternative; or
- the presence (identified as boolean TRUE) or absence (boolean FALSE) of an optional element.

The value "app-determinant" specifies that there will be, within the encoding structure, a field that is used to provide the necessary determinant, but that this is a field which appears within the ASN.1 specification, and which carries application semantics. In this case, the encoder does not determine the value of the field, but the ECN specification is now required to include parameters that say how an encoder uses the value of this field (possibly using one or more transformations) to determine:

a) the size of an encoding-space (in "Units"); or

b) the index of a selected alternative; or

c) the presence (identified as boolean TRUE) or absence (boolean FALSE) of an optional element.

The value "container" specifies that the end of the encoding-space is determined by the end of some containing encoding space. In this case a separate parameter will either provide a reference to the containing container, or will specify it as "end-of-encoding:NULL", meaning the end of the PDU.

The value "pattern" specifies that some specified pattern of bits will terminate the encoding space. In this case additional parameters will require the insertion of a specified pattern by an encoder, and its detection by a decoder.

NOTE An example is null-terminated character string.

The value "handle" specifies that determination is obtained through the use of identification handles. More details of the use of identification handles appears in 19.10.

The value "not-needed" specifies that the contents of the encoding-space are self-delimiting, and that, although the size varies between different abstract values, no external determination of the length is needed.

NOTE This is used in the case of a concatenation in which optionality is not determined by the length of this container, nor is the length of the container used to determine the length of a final element. It is also used if the encoding of an integer is known to be variable but self-delimiting, such as a Huffman encoding.

The "Justification" type is used to specify right or left justification of the encoding of a value within a container, with an offset in bits from the ends of the container. The value "left:0" means that the value is placed with the leading bit of the value at the leading edge of the container. The value "left:1" means a gap of 1 bit, and so on. "right:0" means that the trailing bit of the value is placed at the trailing end of the container, and so on. In all cases, the setting of the bits before or after the value encoding is determined by a value of "Padding" and "Pattern".

"Padding" is used to specify the value of the bits in the encoding-space that are not occupied by the value-encoding. If the value is "pattern" then the bits are set according to "Pattern" . If the value is "encoder-option", then the bit values are freely chosen by the encoder. A value of "zero" or of "one" specifies the use of zero or one bits for padding. Separate values of "Padding" and "Pattern" can be specified for three separate forms of padding: pre-alignment of the encoding-space to a defined boundary, pre-padding of a value-encoding if it is right-justified in a fixed-size encoding-space, and post-padding of a value-encoding if it is left-justified in a fixed-size encoding-space. A fourth use of "Padding" is to specify the value to be used in the encoding of a #PAD class.

"Pattern" is used to specify a pattern to fill a padding field, and also to specify the pattern to be used to encode boolean and null values.

If "Pattern" is set to the "bits" or "octets" alternative, it specifies a pattern of length equal to the bit string or octet string. If it is set to "char8", it specifies a (multiple of 8-bits) pattern where each character in the string is converted to its ISO/IEC 10646-1 value as an 8-bit value. If it is set to "char16", it specifies a (multiple of 16-bits) pattern where each character in the string is converted to its ISO/IEC 10646-1 value as a 16-bit value. If it is set to "char32", it specifies a (multiple of 32-bits) pattern where each character in the string is converted to its ISO/IEC 10646-1 value as a 32-bit value. When used in padding, the pattern specified by "Pattern" is truncated and/or replicated as necessary to provide sufficient bits for the field being filled.

When "Pattern" is used to fill padding bits, the leading bits of the pattern are used first and any bits not needed are discarded. If the padding bits are not filled when the pattern is exhausted, it repeats indefinitely.

The "other" value of type "Pattern" is excluded from most uses of this type. When "Pattern" is used to specify the pattern for a #BOOL value ("TRUE" say), then the value "other" can be used to specify the pattern for the other #BOOL value ("FALSE" in this case). When used in this way, "other" means an encoder's option for the pattern. The encoder may use any pattern it chooses, but **it shall be of the same length as the other pattern and shall differ from it in at least one bit position.**

"RangeCondition" is used in the specification of a predicate which tests the existence and nature of bounds on an integer field. The predicate is satisfied if the bounds set for the field are in accordance with the name used in the enumeration. The value "semi-bounded" means that the predicate is satisfied if and only if there is a lower bound, but no upper bound.

NOTE It is not possible for more than one predicate to be satisfied by any given set of bounds.

"SizeRangeCondition" is used to test properties of the bounds in an effective size constraint associated with a field. The predicate is satisfied if the effective size constraint is in accordance with the name in the enumeration, where "fixed-size" means that there is only one size permitted by the effective size constraint.

NOTE Only the "fixed-size" case overlaps with other predicates.

"ReversalSpecification" is used in the final transformation of bits from an encoding-space into an output buffer for transmission (with the reverse transformation being applied for decoding). It is always used in conjunction with a value of "Unit" that specifies a unit size in bits. It is an ECN specification error if the values "reverse-half-units" and "reverse-bits-in-half-units" are used when "Unit" is not an even number of bits. The enumerations specify either no reversal of bits, or reversal of the order of half-units (without changing the order of bits in each half unit), or reversal of the order of bits in each half-unit but without reversing the order of the half-units, or reversal of the order of the bits in each unit. Bit-reversal can be specified for the encoding of all classes that can appear as fields of encoding structures, except the #ALTERNATIVES encoding class, which does not use the encoding-space concept.

19.3 The #TRANSFORM encoding class

NOTE This encoding class is formally defined in Annex A.

19.3.1 Source class and target encoding class

The #TRANSFORM encoding class allows the specification of encoding/decoding procedures (to be associated with an encoding object) which transform certain abstract values into different abstract values. These transforms are used in the definition of value mappings, in the definition of encoding objects for primitive encoding classes, and in converting values from a determinant field to the boolean or integer values needed to express presence or absence, an index for an alternative, or presence determination.

The transformation is determined by the first word ("INT-TO-INT", "BOOL-TO-BOOL", etc) in the #TRANSFORM encoding object definition. This word also determines the group of parameters of #TRANSFORM that are used in the definition of the transformation. All parameters except those immediately following comment lines such as "--<bool-to-bool>" in Annex A are ignored.

The first word in the encoding object definition defines the abstract values that the transformation accepts as input and produces as output, using the notation:

```
"input type"-to-"output type"
```

The "input type" determines the allowed source encoding classes for the transformation and the "output type" determines the allowed target encoding classes for the transformation.

When transformations are used in succession, the output type of one #TRANSFORM encoding object shall be the input type for the next #TRANSFORM encoding object.

For the first and last of a chain of transformations, text in the body of this TS (or in Annex A) specifies the source encoding class for the first transformation and the target encoding class for the last.

For the "int", "bool", "bits", and "chars" input or output types, the encoding classes they can be transformed from or into are specified in Table 6

Table 6 – Permitted encoding classes for input and output types

Input or output type		Permitted encoding classes
int		#INT, #INTEGER, #ENUMERATED
bool		#BOOL, #BOOLEAN
bits	from to	#BITS, #BIT-STRING, #OCTET-STRING #BITS, #BIT-STRING
chars		#CHARS, #GeneralizedTime, #UTCTime, #BMPString, #GeneralString, #GraphicString, #IA5String, #NumericString, #PrintableString, #TeletexString, #UniversalString, #UTF8String, #VideotexString, #VisibleString
fixed-units	to	#BITS

The "char" and "bit" input or output types have no corresponding encoding class, and transformation encoding objects that require these types (or produce these types) can only be used when specifically required in Annex A.

19.3.2 The int-to-int transforms

NOTE Examples of these transforms are given in E.1.13.

A transform is defined by giving a value of a "CHOICE", permitting any given object to specify precisely one arithmetic operation. General arithmetic can, however, be defined by the use of a list of transforms. (Permitted wherever transforms are allowed.)

The transforms "increment", "decrement", "multiply", "negate" have their normal mathematical meaning.

The transform "divide" is defined to have an integer result which is the integer value that is closest to the mathematical (real arithmetic) result, but is no further from zero than that result.

NOTE In programming terms, "divide" truncates towards zero, so a value of -1 with "divide : 2" will give zero.

The transform "modulo" is defined as follows: Let "i" be the original integer value, let the transform be "modulo : j". Let "k" be the result of applying "divide : j" followed by "multiply : j" to "i". Then "modulo : j" applied to "i" is defined to be the same as applying "decrement : k" to "i".

19.3.3 The bool-to-bool transforms

There is only one transform currently defined, which converts boolean "TRUE" to "FALSE", and vice-versa.

19.3.4 The bool-to-int transforms

There is only one transform currently defined, and it produces an integer value of "1" or "0". If "true-zero" is selected, then "TRUE" produces "0" and "FALSE" produces "1". If "true-one" is selected, the reverse applies.

19.3.5 The int-to-bool transforms

There are three means of specifying these transform. The parameter group for these transformations contains three parameters, and at most one of these can be explicitly set. If none of the parameters in the group is set, then the transformation is determined by the default value of the "&int-to-bool" parameter, namely a mapping which transforms zero values to boolean FALSE, and non-zero values to boolean TRUE. Alternatively, the "&int-to-bool" parameter can be explicitly set to either the default value, or to the value "zero-true" which maps zero values to TRUE and non-zero values to false. If the parameter "&Int-to-bool-true-is" is explicitly set (with a value set such as " {1 | 3 | 5}"), then all the values specified for the parameter map into TRUE and all other values map into FALSE. If the parameter "&Int-to-bool-false-is" is explicitly set, then all the values specified for the parameter map into FALSE and all other values map into TRUE.

19.3.6 The int-to-chars transforms

There is only one transform currently defined. Three parameters are used in its definition, but all have default values and can be omitted. The "&int-to-chars-size" parameter specifies either

- a fixed size in characters for the resulting size (a positive value of "Size"); or
- specifies that a variable length string of characters is to be produced (a value of "variable" for "Size"), or
- specifies a fixed-size just largest enough to contain the transformation of all abstract values in the source class (a value of "fixed-to-max" for "Size"). It is an ECN specification error if this is not a finite size.

The integer value is first converted to a decimal representation with no leading zeros and with a pre-fixed "-" ("HYPHEN-MINUS") if it is negative. If "&int-to-chars-plus" is set to true, positive values have a "+" ("PLUS SIGN") pre-fixed to the digits.

The most significant digit shall be at the leading end of the chars string.

If the size specified is "variable", then this is the resulting string of chars.

NOTE In this case it is not an error to specify a value for "&int-to-chars-pad", but the value is ignored.

If size is fixed (either explicitly or by specifying "fixed-to-max"), and the resulting string (in an instance of application of this transform during encoding) is too large for the fixed size, then this is an ECN specification error.

NOTE In general it will only be possible for a tool to check for this error at encode time, as restrictions on possible abstract values may not be formally present in the ASN.1 specification.

If the size is fixed, and the string is smaller than the specified size, then it has either spaces (the ISO 10646 character "SPACE") or zeros (the ISO 10646 character "DIGIT ZERO"), as specified by the value of "&int-to-chars-pad" pre-fixed to produce the specified size.

19.3.7 The int-to-bits transforms

The "&int-to-bits-encoded" parameter selects the encoding of the integer as either a 2's complement encoding or as a positive integer encoding. The definition of these encodings is given in ITU-T Rec. X.690 | ISO/IEC 8825-1, 8.3.2 and 8.3.3.

The most significant bit shall be at the leading end of the bit string.

The integer shall be encoded into the minimum number of bits necessary.

NOTE This means that a positive integer encoding shall not have zero as the leading bit (unless there is a single zero bit in the encoding), and a 2's complement encoding shall not have two successive leading zero or two successive leading one bits E.1.6.

If "positive-int" encoding is selected, and the value to be transformed is negative, this is an ECN specification error.

NOTE In general it will only be possible for a tool to check for this error at encode time, as restrictions on possible abstract values may not be formally present in the ASN.1 specification.

If the size specified is "variable", then this is the resulting string of bits.

NOTE In this case it is not an error to specify a value for "&int-to-bits-unit", but the value is ignored.

If the size is not "variable" or "fixed-to-max", the size of the resulting bits shall be the value of "&int-to-bits-unit" multiplied by "&int-to-bits-size". If the size is "fixed-to-max", then the size of the resulting bits shall be largest size needed for any value of the class to which the transformation is applied. It is an ECN specification error if this is not finite.

If the resulting string (in an instance of application of this transform during encoding) is too large for the fixed size, then this is an ECN specification error.

NOTE In general it will only be possible for a tool to check for this error at encode time, as restrictions on possible abstract values may not be formally present in the ASN.1 specification.

If the string is smaller than the specified size, then for a positive integer encoding it shall have zero bits prefixed. If the encoding is 2's complement, then it shall have bits prefixed equal in value to the original leading bit.

19.3.8 The bits-to-int transforms

The integer value shall be produced by interpreting the bits as 2's complement or a positive integer encoding according to the value of "&bits-to-int-decode", as specified in ITU-T Rec. X.690 | ISO/IEC 8825-1, 8.3.2 and 8.3.3.

19.3.9 The char-to-bits transforms

NOTE ECN supports only characters within the ISO/IEC 10646 character set. Where ASN.1 types such as "GeneralString" are in use, characters outside of this character set can in theory appear. Such characters are not supported by this transformation (or by encoding with defined syntax), although BER and PER encoding objects are able to encode them.

This transform shall only be used in the "&Char-encodings" parameter of the #CHARS defined syntax, and reference to an effective alphabet constraint below refers to the constraint on the encoding structure field to which an encoding object of this class has been applied.

If "&char-to-bits-encode" is set to "mapped", then the transform is specified by the values of "Char-to-bits-chars" and "Char-to-bits-values", both of which shall be specified. They are respectively a list of single characters and of bit string values. These parameters are ignored if "&char-to-bits-encode" is not set to "mapped". The value of "&char-to-bits-unit" and "&char-to-bits-size" shall be ignored.

There shall be an equal number of values in each list, and all character values and all bit string values in the list shall be distinct. The encoding of a character in the "Chars-to-bits-chars" list is the bit string specified in the corresponding

position in the "&Chars-to-bits-values" list. If in an instance of application of this transform a character is to be transformed that is not in the list, this is an ECN specification error.

NOTE In general it will only be possible for a tool to check for this error at encode time, as restrictions on possible abstract values may not be formally present in the ASN.1 specification.

If the value of "&char-to-bits-encode" is "iso10646", then the character is first converted to the numerical value specified in ISO/IEC 10646.

NOTE ISO/IEC 10646 includes the so-called ASCII control characters, which have positions in row 1.

This numerical value is then converted to bits as specified for "int-to-bits" in 19.3.7, using a positive integer encoding, and with the value of "Unit" and "Size" taken from the "&char-to-bits-unit" and "&char-to-bits-size" parameters of "char-to-bits" (which both have default values if not explicitly set).

If "Size" is set to "fixed-to-max", then the effective alphabet constraint associated is inspected. The "Size" for conversion to bits is then set to the minimum number of bits that will accommodate the encoding as a positive integer of the value of any character in the effective alphabet constraint. (The value is either the ISO/IEC 10646 value or the compacted value specified below) If there is no effective alphabet constraint the "fixed-to-max" will produce a 32-bit encoding for all characters.

If "&char-to-bits-encode" is set to "compact", then the effective alphabet constraint is inspected. All characters in the alphabet are placed in canonical order using their value, lowest value first. The first in the list is then assigned the numerical value zero, the next one, and so on. The resulting value is then encoded into bits as specified in 19.3.7.

NOTE The PER encoding of character string types uses "compact" only if the application of this algorithm reduces the number of bits required to encode characters (using "fixed-to-max"). This degree of control is not possible in this version of the ECN specification.

19.3.10 The bits-to-char transforms

This transformation is applied by taking the bitstring in the source class and mapping it to a character.

If "iso10646" is selected, then the bitstring shall be interpreted as a positive integer encoding which contains the ISO 10646 numerical value of a character. It is an ECN specification error if in this application the bitstring contains more than 32 bits.

If "mapped" is selected then the mapping defined in "&Bits-to-char-values" and "&Bits-to-char-chars" shall be applied in a similar way to 9.3.9. It is an ECN specification error if, in an application of this transform, the bitstring is not present in the "&Bits-to-char-values".

If "compact" is selected, the bitstring is first converted to a numeric value, then the effective alphabet constraint associated with the target encoding class is used to determine the character to which that value corresponds, as in 19.3.9.

19.3.11 The bit-to-bits transforms

This transform shall only be used as the first of the "&Encoder-transforms" for the #BITS encoding of A.5.

specify the replacement of each zero bit in the bitstring being encoded with the pattern "&bit-to-bits-zero" and of each one bit with the pattern "&bit-to-bits-one".

It is an ECN specification error if the two patterns are the same, or if one is an initial sub-string of the other.

19.3.12 The bits-to-bit transforms

This transform takes the values of a bitstring and interprets them as a single bit in accordance with the given patterns.

It is an ECN specification error if the two patterns are the same or if the transform is applied to a bitstring that does not match either pattern.

19.3.13 The bits-to-fixed-units transform

This transform takes a bit string and converts it to another bit string whose size is specified by "Unit" and "Size" which is required to specify a numeric size for the resulting bitstring, which acts as a "container". The source bit string is placed

in this container according to "Justification". Any bits in the container before the source bit string are filled in accordance with the "pre-padding" and the "pre-pattern", and any bits in the container after the source bit string are filled in accordance with the "post-padding" and the "post-pattern".

NOTE It is the ECN-specifier's responsibility to ensure that any encoding produced in this way is capable of being decoded.

19.4 The pre-alignment parameters

<Pre-alignment>: #BITS, #BOOL, #CHARS, #CONCATENATION, #INT, #NUL, #OCTETS, #PAD, #REPETITION

The pre-alignment parameters use a value of "Unit" to specify that a container is to start at a multiple of "Unit" bits from the alignment point. The alignment point is the start of the encoding of the type to which an ELM applied an encoding, except when reset for the encoding of a contained type by the use of a #OUTER encoding object . The "Padding" and "Pattern" parameters are used to control the bits that provide padding to the required alignment.

NOTE If "Unit" is set to one, there is no pre-alignment, and the other parameters are ignored.

19.5 The padding parameter group

<Padding>: #BOOL, #CONCATENATION, #CONDITIONAL-INT, #NUL, #CONDITIONAL-REPETITION, #OUTER

The "<Padding >" parameter group is used to define justification and pre and post padding if a value-encoding is smaller than the specified encoding-space.

The padding parameter group uses "Justification" to determine the positioning of a value within a container, and values of "Padding" (with an optional "Pattern") to specify the padding bits before the value and those after it.

NOTE If the value encoding is not fixed length or self-delimiting, then the use of value padding in a fixed size container can in some circumstances make it impossible for a decoder to recover the original abstract values. This would be an ECN specification error.

19.6 The pad-padding parameter group

<Pad padding>: #PAD

The "<Pad padding>" parameter group is used to define the value to be placed into a field encoded by the #PAD encoding class.

19.7 The bit reversal parameters

NOTE Bit reversal can be specified on individual primitive encodings and also for the results of concatenation or repetition. Care should be taken to ensure that one reversal does not negate the other.

<Bit reversal>: #BITS, #BOOL, #CHARS, #CONCATENATION, #INT, #NUL, #OCTETS, #PAD, #REPETITION

All encoding classes apart from #ALTERNATIVES provide for bit-reversal. This has been described in 19.2 ("ReversalSpecification").

NOTE Bit reversal applies to the contents of the container, it does not apply to pre-alignment padding.

19.8 Encoding space parameters

<Encoding space>: #NUL, #BOOL, #PAD, #CONCATENATION, #CONDITIONAL-INT, #CONDITIONAL-REPETITION

The use of "Unit" and "Size" is described in 19.1. There are restriction on the use of these parameters for the different classes.

For #NUL, there is only a single abstract value, and only a fixed "Size" can be specified (but it can be zero - no bits in the encoding).

For #BOOL, there are two values, which may encode to different sizes (for example, the ASCII codes for "TRUE" and for "FALSE"). However, the size of the container is not allowed to be zero, as it requires at least one bit to discriminate the two values.

For #PAD, a zero size is prohibited, as the purpose of #PAD is to provide a "filler".

For the other classes, the full range of values of "Unit" and "Size" are available.

19.9 Determination mechanisms

19.9.1 General

<DeterminationMechanism>: #ALTERNATIVES, #CONDITIONAL-INT, #CONCATENATION, # CONDITIONAL-REPETITION, #OPTIONAL

There are several mechanisms available for determining the length of a container which is not of fixed length, for selection between alternatives, and for determining optionality. These are selected by the value of "DeterminationMechanism" . Depending on the value selected, other parameter groups are required. Some of these groups have default values, but others do not (in particular, those requiring a reference to an encoding structure fieldname that is to be a determinant). Where there is no default specified in Annex A, a value for the parameters of the required group shall be specified if that mechanism is selected.

A value of "aux-determinant" or "app-determinant" requires that the "Detereminant" group be set.

A value of "container" requires that the <Run-out of container> group be set.

A value of "pattern" requires that the "<Append pattern>" group be set.

A value of "handle" requires that the "<Use handle>" group be set.

A value of "not-needed" identifies that the construction is self-delimiting, and that other parameter groups are not required.

NOTE Although not needed, where a <Determinant> group is provided, it may still be set to indicate that an encoder shall encode a determinant value for a length field. Encoders will detect an encoders error if the value in this field is not consistent with the length of the encoding space that it determines.

19.9.2 Use of a length determinant

<Length determinant>: #BOOL, #CONCATENATION, #CONDITIONAL-INT, #NUL, #PAD, #CONDITIONAL-REPETITION

This group shall be set if the "determinant" mechanism is selected.

The <Length determinant> contains a parameter which shall be set to a "REFERENCE" value, that is, a reference to a fieldname which will be visible when the encoding is applied. This value will always be supplied as a dummy parameter, as fieldnames are not visible when the encoding object is defined using defined syntax.

For a length determinant, the determinant is required to produce a count in encoding space units. For a presence determinant it is required to produce a boolean value (TRUE for present, FALSE for absent). For alternative selection, it is required to produce an index value, zero for the first alternative, one for the next, and so on, where "first" is determined by the "<Ordering procedure>" parameter group . However, transforms can optionally be supplied to determine these integer or boolean values from values of the determinant field.

If the determinant was selected by "aux-determinant", then the transformations shall be the "&Encoder-transforms", and the "&Decoder-transforms" shall not be set. The "&Encoder-transforms" define the transformation from the value required for determination to the value that an encoder is required to place in the determinant field.

If the determinant was selected by "app-determinant", then the value of the determinant field is determined by the application, and the transformations shall be the "&Decoder-transforms", and the "&Encoder-transforms" shall not be set. The "&Decoder-transforms" define the transformation from the value in the determinant field to the value required

for determination. An encoder shall not generate encodings where the value in the determinant field is inconsistent with the determination it is to be used for by an encoder.

For the #CONDITIONAL-REPETITION class, there is an additional parameter in this group that specifies the "Units" for the determinant. In this case (only) a value of "repetitions" for "Units" is permitted.

19.9.3 Unused bits determination

<Unused bits>:#CONDITIONAL-INT, #CONDITIONAL-REPETITION

This group is always optional. If absent, any unused bits are always less than the granularity of variability of the value encoding, and decoding will never be ambiguous.

NOTE This is the assertion made by the ECN designer. As in all other cases, it is the designer's sole responsibility to ensure that ECN specifications permit unambiguous decoding.

It is frequently the case that the unit of length determination for a container is greater than the unit of value encoding. (For example, the container length may be determined in octets, but the value encoding may be in units of bits or half-octets.) In such cases the number of unused bits at the end of the container has to be determined by a decoder. The specification of the "<Unused bits>" determinant and encoders transforms tells a decoder that it shall ignore the trailing "n" bits of the encoding-space, where "n" is the value produced by the unused bits determinant, after reversal of specified transformations.

19.9.4 End of container length determination

<Run-out of container>:#CONCATENATION, #CONDITIONAL-INT, #OPTIONAL, #CONDITIONAL-REPETITION

This group (of one parameter) shall be set if the corresponding mechanism is selected.

The parameter contains either a "REFERENCE" or an "end-of-encoding:NULL" specification. The "REFERENCE" shall identify (be the fieldname of) a container within which this element is contained. A decoder will continue to decode bits for the current encodable item until the end of the referenced container is reached.

If "end-of-encoding" is specified, then an encoder will terminate the current item only when there are no more bits left to decode (possible modified by an "unused bits" determinant that might be present).

19.9.5 Special pattern length determination

<Append pattern>: #CONDITIONAL-REPETITION

This parameter group shall be present if the corresponding mechanism is selected.

The use of this group provides a "Pattern" that an encoder is required to place at the end of all repetitions. A decoder shall match bits against this pattern whenever it is looking for another repetition.

NOTE 1 – An example would be a null-terminated list of characters, or a list of positive integers terminated by a "-1".

NOTE 2 – It is the ECN designer's responsibility to ensure that the specified pattern can never occur at the start of the encoding of any value in the repetition.

NOTE 3 - There is no requirement that the size of repetitions and the pattern be the same, but it is an ECN specifiers error if the encoding of a repetition can produce bits that can be decoded as the terminating pattern.

19.10 Definition of handles

<Primitive Handle information>: #BITS, #BOOL, #CHARS, #INT, #NUL, #OCTETS, #PAD

<Concatenation handle information>: #CONCATENATION

<Repetition handle information>: #REPETITION

These groups are used to identify the presence of an identification handle within an encoding.

An identification handle is a named field or set of fields in the encoding of an encodable item which has the same bit-pattern for the encoding of all abstract values of that class.

In the simplest case (primitive fields), an identification handle is given a name, and a set of integer values (a subtype of "INTEGER") which are the bit positions in the encoding (after any pre-alignment) which form the identification handle. The leading bit is called bit zero for this purpose.

In any application of ECN, all identification handles with the same name that are displayed by primitive types shall specify the same set of bits for the location of the identification handle.

The <Concatenation handle information> enables a concatenation to display a new identification handle (handle "HXH" say) based on a list of component numbers (zero identifies the first component). Each of these components is required to exhibit an identification handle, and the list of identification handle names from these components (and the positions of the components in the concatenation) shall be the same for all concatenations displaying the handle "HXH".

The <Repetition handle information> enables a #REPETITION to display an identification handle. This handle shall be displayed by the component of the #REPETITION.

NOTE If this parameter is not specified, then any identification handles on the component are not visible outside of the #REPETITION.

19.11 Use of handles

<Use handle>: #ALTERNATIVES, #CONCATENATION, #OPTIONAL, #CONDITIONAL-REPETITION

The <Use handle> group shall be present if this mechanism is selected for determination of alternatives, resolution of order in a concatenation , determination of optionality, or for termination of a #REPETITION.

When this mechanism is used to select an alternative, all alternatives shall exhibit identification handles with the specified name, and all shall have different bit-patterns in the fields associated with those patterns, for all abstract values of each alternative.

When this mechanism is used to determine the order of a concatenation, then all components shall exhibit identification handles with the specified name, and all shall have different bit-patterns in the fields associated with those patterns, for all abstract values of each component.

When this mechanism is used to resolve optionality, then the optional component and any following components in this container (up to and including the first mandatory component) shall exhibit identification handles with the specified name. All shall have different bit-patterns in the fields associated with those patterns, for all abstract values of each component.

When this mechanism is used to terminate a repetition, the repetition component and any possible following encodable item is required to exhibit the named handle. The repetition terminates when an encoding is encountered which, when decoded in accordance with any encoding that exhibits that handle, does not contain the value of the identification handle for the repetition component.

19.12 Value-encoding for Nul

<Nul value-encoding>: #NUL

The value-encoding parameters for #NUL allow the specification of a "Pattern" value that will represent the NULL abstract value.

NOTE It is also possible to encode null with zero bits by setting the container size to zero.

19.13 Value-encoding for Bool

<Bool value-encoding>: #BOOL

The value-encoding parameters for #BOOL allow the specification of a value for either "TRUE" or "FALSE" (or both), and the specification of "other" for the pattern for the other value (if both are not specified).

If "other" is specified, then pattern for that value shall be an encoder's option, but shall be of the same length as the pattern specified for the other value.

The value "other" shall not be specified for both "TRUE" and "FALSE".

19.14 Value-encoding for Int

<Int value-encoding>: #INT

<Range predicate>: #CONDITIONAL-INT

<Cond-int value-encoding>: #CONDITIONAL-INT

NOTE There are many other parameter groups used in the specification of integer encodings. See Annex A for full details.

The specification of integer encodings uses "<Int value-encoding>", which is either a single encoding object or an ordered list of encoding objects of class #CONDITIONAL-INT.

Each encoding object of class #CONDITIONAL-INT can (optionally) contain a "<Range predicate>". The "<Range predicate>" tests the bounds on the encoding class that is being encoded. The encoding that is applied is the first in the list of conditional encodings whose predicate is satisfied. (A predicate is always satisfied if the "<Range predicate>" is missing). If there is no conditional encoding in the list whose predicate is satisfied for an encoding class to which the specified encoding object is applied, this is an ECN designer's error.

Once a conditional encoding has been selected for use, the <Cond-int value-encoding> in that conditional encoding is applied.

19.15 The concatenation procedure parameters

<Concatenation procedure>: #CONCATENATION, #CONDITIONAL-REPETITION

The single parameter in this group specifies how encodings are to be combined in concatenations and in repetitions. There are two choices: "simple" means that the encodings are concatenated end-to-end, and "pre- aligned" means that any pre-alignment specified for the #CONCATENATION or #CONDITIONAL-REPETITION is also applied to the encoding of each component before concatenation.

NOTE Alignment specifications (and any resulting padding) within a component are with respect to its position **after** any alignment and padding by the container. (The only exception to this is if the alignment point is reset for a contained type using #OUTER)

19.16 Repetition encoding

<Repetition encoding>: #REPETITION

<Size predicate>: #CONDITIONAL-REPETITION

"<Repetition encoding>" is similar to integer encoding. It uses a list of encoding objects of class #CONDITIONAL-REPETITION, each of which contains an optional <Size predicate>. The size predicate tests the effective size constraints on the class being encoded, and the first one that is satisfied provides the encoding for the repetition. If there is no size predicate, then the condition is always satisfied, and the first such encoding is applied. If there is no encoding in the list whose predicate is satisfied, this is an ECN designer's error.

Once a conditional encoding is selected, then the actual encoding for the repetition is determined by the other parameter groups, detailed in Annex A.

19.17 Value-encoding for Bits

<Bits value-encoding>: #BITS

The first parameter in the "<Bits value-encoding>" group contains an ordered list of transformations that are required to transform a single bit into a bit string. If the list is empty, the resulting bit string is the single source bit.

These transformations shall be applied to all bits in the #BITS value being encoded, and the resulting repeated set of bits is treated as a #REPETITION of encodings of the original string of bits.

NOTE It is the ECN specifier's responsibility to ensure that the mapping of individual bits into a bitstring is reversible by a decoder. This requires that the two values specified for one and zero are a self-delimiting set of values.

The second and third parameters of this group provide either a list of conditional repetitions, or a single conditional repetition (precisely one of these parameters shall be set). The first of the conditional repetitions whose predicate is satisfied is selected, and is used to combine the encodings of individual bits into the encoding that is specified by the #BITS encoding object that is being defined.

If there is no conditional repetition whose predicate is satisfied, this is an ECN designer's error.

19.18 Value-encoding for Octets

<Octet value-encoding>: #OCTETS

The two parameters of this group provide either a list of conditional repetitions, or a single conditional repetition (precisely one of these parameters shall be set). The first of the conditional repetitions whose predicate is satisfied is selected, and is used to combine the encodings of individual octets into the encoding that is specified by the #OCTETS encoding object that is being defined.

If there is no conditional repetition whose predicate is satisfied, this is an ECN designer's error.

NOTE This is a simplified case of the encoding for #BITS. The bits in each octet are not transformed: there is simply a list of one or more conditional repetitions that determine how they are combined.

19.19 Value-encoding for Chars

<Chars value-encoding>: #CHARS

The first parameter in the "<Chars value-encoding>" group contains an ordered list of transformations that are required to transform a single character into a self-delimiting (among the set of all encodings of characters in the alphabet) bit string. The list of transforms cannot be empty.

These transformations shall be applied to all characters in the #CHARS value being encoded, and the resulting repeated set of bits is treated as a #REPETITION of encodings of the original string of characters.

NOTE It is the ECN specifier's responsibility to ensure that the mapping of individual characters into a bitstring is reversible by a decoder. This requires that the two values specified for all characters in the effective permitted alphabet are a self-delimiting set of values.

The second and third parameters of this group provide either a list of conditional repetitions, or a single conditional repetition (precisely one of these parameters shall be set). The first of the conditional repetitions whose predicate is satisfied is selected, and is used to combine the encodings of individual characters into the encoding that is specified by the #CHARS encoding object that is being defined.

If there is no conditional repetition whose predicate is satisfied, this is an ECN designer's error.

NOTE This is identical to the encoding for #BITS, except that it is now transformations from a character to bits that has to be specified.

19.20 The ordering procedure parameters

<Ordering procedure>: #ALTERNATIVES, #CONCATENATION

This parameter group is used when a canonical order is needed for components, in particular for components of an #ALTERNATIVE if an index is needed for a determinant that will identify an alternative, or for the ordering of the components in a #CONCATENATION.

The parameter allows specification of "textual", "tag", or "option-with-handle".

A value of "textual" means that the order is based on the textual order in the encoding structure (which may be an implicit encoding structure for an ASN.1 type) definition.

A value of "tag" means that the order is the canonical tag order of the outermost tag as specified in ITU-T Rec. X.680 | ISO/IEC 8824.

A value of "option-with-handle" means that (for concatenation) the order is an encoder's option, with identification of an alternative or a component being done by a specified identification handle. In this case parameters of the "<Use handle>" group shall be set to specify an identification handle.

All components of the #CONCATENATION or alternatives of the #ALTERNATIVES shall exhibit the specified identification handle, and shall have distinct values for the fields which make up the identification handle.

19.21 Contained type encoding

<Contained type encoding>: #BITS, #OCTETS

This parameter group is optional, and provides either one or two encoding object sets. If two are provided, they are combined according to clause 13.2 to produce a combined encoding object set.

If this encoding object is used to encode a #BITS or #OCTETS class that has a contained type, then it is the combined encoding object set defined by this parameter group that encodes the contained type if it has no "ENCODED BY" construct.

If there is an "ENCODED BY" construct, then it provides the encoding unless the "&over-ride-encoded-by" parameter is set "TRUE", in which case the combined encoding object set in the encoding object is applied.

If the group is missing, then the contained type is encoded with the combined encoding object set that was applied to the #BITS or #OCTETS if there is no "ENCODED BY", otherwise the encoding is that specified by the "ENCODED BY".

19.22 The #OUTER encoding class

NOTE This encoding class is formally defined in Annex A.

Encoding objects of the #OUTER class are applied to the entire encoding of a type which is encoded by either:

a) application of an encoding in the ELM; or

b) application of an encoding to a contained type.

This encoding object performs two functions:

- The first is to specify that the enPadding" and "Pattern". If "Unit" is set to one, there will be no padding.
- The second function is to deter tire encoding is to be made a multiple of "Units" bits by the addition (if necessary) of padding bits specified by "mine whether the alignment point is to be reset to the start of the contained type encoding ("reset") when encoding of a contained type is commenced, or whether the alignment point is to be the same as that for the containing type.

Annex A (Normative): Specification of Encoding Classes

The encoding classes defined in this Annex are used in the definition of the "DefinedSyntax" using notation specified in ITU-T Rec. X.681 | ISO/IEC 8824-1, 10, as modified (for the sole purpose of this annex) by Annex C. Encoding objects specified in accordance with 16.2 shall defined using the syntax specified by the "WITH SYNTAX" clause for their corresponding encoding class.

A.1 Commonly-used type definitions

The following types and values are used in several places in the specification of parameters (of the encoding classes that are to be used in the definition of encoding objects).

The types and their semantics are described in clause 19.1.

NOTE All ASN.1 type definitions given here assume automatic tags and no extensibility.

```
Unit ::= INTEGER
         {repetitions(0), bit(1), nibble(4), octet(8),
          word16(16), dword32(32)} (0..256)
-- Default is always "bit".

Size ::= INTEGER
         { uses-determination-mechanism(-3), fixed-to-max(-2), variable(-1),
empty(0)} (-3..MAX)
-- Default is usually "variable", else OPTIONAL

Justification ::= CHOICE
         { -- in bits
           left            Size (0..MAX),
           right           Size (0..MAX)}
-- Default is always "right:0".

Padding ::= ENUMERATED
         {zero, one, pattern, encoder-option}
-- Default is always "zero".

Pattern ::= CHOICE
         {bits                     BIT STRING,
          octets                   OCTET STRING,
          char8                    IA5String,
          char16                   BMPString,
          char32                   UniversalString,
          other-than               NULL }
-- Default is always "bits:'0'B"

other Pattern ::= other-than:NULL
-- Provided to make the value notation for Pattern more user-friendly.

RangeCondition ::= ENUMERATED
        {unbounded-or-no-lower-bound,
         semi-bounded-with-negatives,
         bounded-with-negatives,
         semi-bounded-without-negatives,
         bounded-without-negatives }
-- Never defaulted.

SizeRangeCondition ::= ENUMERATED
        {no-ub-with-zero-lb,
         ub-with-zero-lb,
         no-ub-with-non-zero-lb,
         ub-with-non-zero-lb,
         fixed-size}
-- Never defaulted.

ReversalSpecification ::= ENUMERATED
        {no-reversal,
         reverse-bits-in-units,
         reverse-half-units,
```

```
         reverse-bits-in-half-units}
-- Default is always "no-reversal".

DeterminationMechanism ::= ENUMERATED
        {aux-determinant,app-determinant, container,pattern,handle,not-needed}
```

A.2 Groups of parameters

NOTE Many parameters are formally OPTIONAL, but are required if certain values are supplied for other parameters. The WITH SYNTAX clause sometimes enforces this, but the rules in clause 19 always apply.

The parameter groups used in the following definitions of encoding classes are defined in clause 19:

<Append pattern>
<Bit reversal>
<Bits value-encoding>
<Bool value-encoding>
<Chars value-encoding>
<Concatenation handle information>
<Concatenation procedure>
<Cond-int value-encoding>
<Contained type encoding>
<Determination mechanism>
<Determinant>
<Encoding space>
<Int value-encoding>
<Nul value-encoding>
<Octet value-encoding>
<Ordering procedure>
<Pad padding>
<Padding>
<Pre-alignment>
<Primitive handle information>
<Range predicate>
<Repetition encoding>
<Repetition handle information>
<Run-out of container>
<Size predicate>
<Unused bits>
<Use handle>

A.3 The #TRANSFORM encoding class

This encoding class and its semantics is described in clause 19.

```
#TRANSFORM ::= ENCODING-CLASS {
-- <int-to-int>
        &int-to-int                CHOICE
                                 { increment       INTEGER (1..MAX),
                                   decrement       INTEGER (1..MAX),
                                   multiply        INTEGER (1..MAX),
                                   divide          INTEGER (1..MAX),
                                   negate          NULL,
                                   modulo          INTEGER (2..MAX)} OPTIONAL,

-- <bool-to-bool>
        &bool-to-bool            CHOICE
                                 {  not        NULL
                                    -- Others may be added -- } OPTIONAL,

-- <bool-to-int>
        &bool-to-int             ENUMERATED {true-zero, true-one}
                                 DEFAULT true-one,
```

```
-- <int-to-bool>
      &int-to-bool            ENUMERATED {zero-true, zero-false}
                              DEFAULT zero-false,
      &Int-to-bool-true-is    INTEGER OPTIONAL,
      &Int-to-bool-false-is   INTEGER OPTIONAL,

-- <int-to-chars>
      &int-to-chars-size          Size (fixed-to-max | variable | 1..MAX) DEFAULT variable,
      &int-to-chars-plus          BOOLEAN DEFAULT FALSE,
      &int-to-chars-pad           ENUMERATED
                                  {space, zero} DEFAULT zero,

-- <int-to-bits>
      &int-to-bits-encoded     ENUMERATED
                                  {positive-int, twos-complement}
                                  DEFAULT twos-complement,
      &int-to-bits-unit           Unit (1..MAX) DEFAULT bit,
      &int-to-bits-size           Size (fixed-to-max | variable | 1..MAX)
                                  DEFAULT variable,

-- <bits-to-int>
      &bits-to-int-decoded        ENUMERATED
                                  {positive-int, twos-complement}
                                  DEFAULT twos-complement,

-- <char-to-bits>
      &char-to-bits-encoded  ENUMERATED
                                  {iso10646, compact, mapped}
                                  DEFAULT compact,
      &Char-to-bits-chars         UniversalString (SIZE(1)) ORDERED OPTIONAL,
      &Char-to-bits-values        BIT STRING ORDERED OPTIONAL,
      &char-to-bits-unit          Unit (1..MAX) DEFAULT bit,
      &char-to-bits-size          Size (fixed-to-max | variable | 1..MAX)
                                  OPTIONAL,

-- <bits-to-char>
      &bits-to-char-decoded   ENUMERATED
                                  {iso10646, compact, mapped}
                                  DEFAULT compact,
      &Bits-to-char-values        BIT STRING ORDERED OPTIONAL,
      &Bits-to-char-chars         UniversalString (SIZE(1)) ORDERED OPTIONAL,

-- <bit-to-bits>
      &bit-to-bits-one            Pattern DEFAULT bits:'1'B,
      &bit-to-bits-zero           Pattern DEFAULT bits:'0'B,

-- <bits-to-bit>
      &bits-to-bit-one                    Pattern DEFAULT bits:'1'B,
      &bits-to-bit-zero                   Pattern DEFAULT bits:'0'B,

-- <bits-to-fixed-units>
      &bits-to-fixed-units-unit           Unit(1..MAX) DEFAULT bit,
      &bits-to-fixed-units-size           Size (1..MAX) OPTIONAL,
      &bits-to-fixed-units-justification Justification DEFAULT right:0,
      &bits-to-fixed-units-pre-padding    Padding  DEFAULT zero,
      &bits-to-fixed-units-pre-pattern    Pattern (ALL EXCEPT other)
                                          DEFAULT bits:'0'B,
     &bits-to-fixed-units-post-padding    Padding  DEFAULT zero,
     &bits-to-fixed-units-post-pattern    Pattern (ALL EXCEPT other)
                                          DEFAULT bits:'0'B  }


WITH SYNTAX {
        -- one and only one of the following options can be set:
            [INT-TO-INT &int-to-int]
            [BOOL-TO-BOOL AS &bool-to-bool]
            [BOOL-TO-INT AS &bool-to-int]
            [INT-TO-BOOL
                -- zero or one of the following options can be set:
                [AS &int-to-bool]
                [TRUE-IS &Int-to-bool-true-is]
                [FALSE-IS &Int-to-bool-false-is]]
            [INT-TO-CHARS
                -- zero or more of the following options can be set:
                [SIZE &int-to-chars-size]
                [PLUS-SIGN &int-to-chars-plus]
                [PADDING &int-to-chars-pad]]
            [INT-TO-BITS
                -- zero or more of the following options can be set:
                [AS &int-to-bits-encoded]
```

```
                [SIZE &int-to-bits-size]
                [MULTIPLE OF &int-to-bits-unit]]
            [BITS-TO-INT AS &bits-to-int-decoded]
            [CHAR-TO-BITS
                -- zero or more of the following options can be set:
                [AS &char-to-bits-encoded]
                [CHAR-LIST &Char-to-bits-chars]
                [BITS-LIST &Char-to-bits-values]
                [SIZE &char-to-bits-size]
                [MULTIPLE OF &char-to-bits-unit]]
            [BITS-TO-CHAR
                -- zero or more of the following options can be set:
                [AS &bits-to-char-decoded]
                [BITS-LIST &Bits-to-char-values]
                [CHAR-LIST &Bits-to-char-chars]]
            [BIT-TO-BITS
                -- zero or more of the following options can be set:
                [ZERO-PATTERN &bit-to-bits-zero]
                [ONE-PATTERN &bit-to-bits-one]]
            [BITS-TO-BIT
                -- zero or more of the following options can be set:
                [ZERO-PATTERN &bits-to-bit-zero]
                [ONE-PATTERN &bits-to-bit-one]]
            [BITS-TO-FIXED-UNITS
                -- zero or more of the following options can be set:
                [SIZE &bits-to-fixed-units-size]
                [MULTIPLE OF &bits-to-fixed-units-unit]
                [JUSTIFIED &bits-to-fixed-units-justification]
                [PRE-PADDING &bits-to-fixed-units-pre-padding
                    [PATTERN &bits-to-fixed-units-pre-pattern]]
                [POST-PADDING &bits-to-fixed-units-post-padding
                    [PATTERN &bits-to-fixed-units-post-pattern]]]
}
```

A.4 Defining encoding objects for alternative classes

The syntax defined here can be used to define encoding objects of class #ALTERNATIVES and #CHOICE.

```
#ALTERNATIVES ::= ENCODING-CLASS {

-- <Determination mechanism>
      &selection-mechanism                     DeterminationMechanism
                (aux-determinant | app-determinant | handle),

-- <Ordering procedure>
      &numbering-order                         ENUMERATED
                                               {textual, tag}
                                               DEFAULT textual,
-- The alternatives are numbered from zero upwards.

-- <Determinant>
      &alternative-determinant      REFERENCE OPTIONAL,
      &Encoder-transforms           #TRANSFORM ORDERED OPTIONAL,
-- Transforms from an integer index to actual  "REFERENCE".
      &Decoder-transforms           #TRANSFORM ORDERED OPTIONAL,
-- Transforms from the actual "REFERENCE" to an integer index.
-- Allowed only if app-determinant is set.

-- <Use handle>
      &handle-id                               PrintableString OPTIONAL}

WITH SYNTAX {
    SELECTION AS &selection-mechanism
        [DETERMINED BY &alternative-determinant
            [ENCODER-TRANSFORMS &Encoder-transforms]
            [DECODER-TRANSFORMS &Decoder-transforms]
        [ORDER &numbering-order]]
        [HANDLE &handle-id]
}
```

A.5 Defining encoding objects for #BITS and #BIT-STRING classes

The syntax defined here can be used to define encoding objects of classes #BITS and #BIT-STRING.

```
#BITS ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit       Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding              Padding DEFAULT zero,
  &encoding-space-pre-pattern              Pattern (ALL EXCEPT other)
                                           DEFAULT bits:'0'B,

-- <Bits value-encoding>
  &Encoder-transforms                      #TRANSFORM ORDERED OPTIONAL,
-- Transforms bit to bits
  &Bits-repetition-encodings               #CONDITIONAL-REPETITION
                                           OPTIONAL,

  &bits-repetition-encoding                #CONDITIONAL-REPETITION
                                           OPTIONAL,

-- Exactly one of the above must be supplied.

-- <Contained type encoding>
  &Primary-encoding-object-set             #ENCODINGS OPTIONAL,
  &Secondary-encoding-object-set           #ENCODINGS OPTIONAL,
  &over-ride-encoded-by                    BOOLEAN OPTIONAL,

-- <Primitive handle information>
  &handle-id                               PrintableString OPTIONAL,
  &Handle-positions                        INTEGER (1..MAX) OPTIONAL,

-- <Bit reversal>
  &bit-reversal                            ReversalSpecification
                                           DEFAULT no-reversal  }

WITH SYNTAX {
    [ENCODING-SPACE ALIGNED TO &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
            [PATTERN &encoding-space-pre-pattern]]]
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [ENCODING  &bits-repetition-encoding]
    [ENCODINGS &Bits-repetition-encodings]
    [CONTAINING &Primary-encoding-object-set
        [COMPLETED BY &Secondary-encoding-object-set]
        [OVERRIDE &over-ride-encoded-by]]
    [HANDLE &handle-id AT &Handle-positions]
    [BIT-REVERSAL &bit-reversal]
}
```

A.6 Defining encoding objects for #BOOL and #BOOLEAN classes

The syntax defined here can be used to define encoding objects of classes #BOOL and #BOOLEAN.

```
#BOOL ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding           Padding DEFAULT zero,
  &encoding-space-pre-pattern           Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Bool value-encoding>
  &value-true-pattern                   Pattern DEFAULT bits:'1'B,
  &value-false-pattern                  Pattern DEFAULT bits:'0'B,

-- <Encoding space>
  &encoding-space-unit                  Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-size                  Size (uses-determination-mechanism | fixed-to-max |
                                                1..MAX) OPTIONAL,

-- <Padding>
  &value-justification                  Justification DEFAULT right:0,
  &value-pre-padding                    Padding DEFAULT zero,
  &value-pre-pattern                    Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,
  &value-post-padding                   Padding DEFAULT zero,
  &value-post-pattern                   Pattern (ALL EXCEPT other)
```

```
                                          DEFAULT bits:'0'B,

-- <Determinant>
  &encoding-space-length-determinant    REFERENCE OPTIONAL,
  &Encoder-transforms                   #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a count in &encoding-space-unit to actual "REFERENCE".

-- <Primitive handle information>
  &handle-id                            PrintableString OPTIONAL,
  &Handle-positions                     INTEGER (1..MAX) OPTIONAL,

-- <Bit reversal>
  &bit-reversal                         ReversalSpecification
                                        DEFAULT no-reversal  }

WITH SYNTAX {
        [ENCODING-SPACE
            [ALIGNED TO &encoding-space-pre-alignment-unit
                [PADDING &encoding-space-pre-padding
                    [PATTERN &encoding-space-pre-pattern]]]
            [SIZE &encoding-space-size]
            [MULTIPLE OF &encoding-space-unit]
            [DETERMINED BY &encoding-space-length-determinant
                [ENCODER-TRANSFORMS &Encoder-transforms]]]
        [VALUE      [TRUE-PATTERN &value-true-pattern]
                            [FALSE-PATTERN &value-false-pattern]
                [JUSTIFIED &value-justification]
                [PRE-PADDING &value-pre-padding
                    [PATTERN &value-pre-pattern]]
                [POST-PADDING &value-post-padding
                    [PATTERN &value-post-pattern]]]
            [HANDLE &handle-id AT &Handle-positions]
            [BIT-REVERSAL &bit-reversal]
}
```

A.7 Defining encoding objects for #CHARS and other character string classes

The syntax defined here can be used to define encoding objects of classes #CHARS, #GeneralizedTime, #UTCTime, #BMPString, #GeneralString, #GraphicString, #IA5String, #NumericString, #PrintableString, #TeletexString, #UniversalString, #UTF8String, #VideotexString, and #VisibleString.

```
#CHARS ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding           Padding DEFAULT zero,
  &encoding-space-pre-pattern           Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Chars value-encoding>
  &Encoder-transforms                   #TRANSFORM ORDERED OPTIONAL,
-- Transforms a char to a fixed-length bit string
  &Chars-repetition-encodings           #CONDITIONAL-REPETITION OPTIONAL,
  &chars-repetition-encoding            #CONDITIONAL-REPETITION OPTIONAL,
-- Exactly one of the above must be supplied.

-- <Primitive handle information>
  &handle-id                            PrintableString OPTIONAL,
  &Handle-positions                     INTEGER (1..MAX) OPTIONAL,

-- <Bit reversal>
  &bit-reversal                         ReversalSpecification
                                        DEFAULT no-reversal  }

WITH SYNTAX {
        [ENCODING-SPACE ALIGNED TO &encoding-space-pre-alignment-unit
            [PADDING &encoding-space-pre-padding
                [PATTERN &encoding-space-pre-pattern]]]
        [ENCODER-TRANSFORMS &Encoder-transforms]
        [ENCODING &chars-repetition-encoding]
        [ENCODINGS &Chars-repetition-encodings]
        [HANDLE &handle-id AT &Handle-positions]
        [BIT-REVERSAL &bit-reversal]
}
```

A.8 Defining encoding objects for concatenation classes

The syntax defined here can be used to define encoding objects of class #CONCATENATION, #SEQUENCE, and #SET.

```
#CONCATENATION ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding           Padding DEFAULT zero,
  &encoding-space-pre-pattern           Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Ordering procedure>
  &concatenation-order                  ENUMERATED
                                        {textual,
                                        tag,
                                        option-with-handle}
                                        DEFAULT textual,

-- <Use handle>
  &order-handle-id                      PrintableString OPTIONAL,

-- <Concatenation procedure>
  &concatenation-procedure              ENUMERATED
                                        {simple, pre-aligned}
                                        DEFAULT simple,

-- <Encoding space>
  &encoding-space-unit                  Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-size                  Size (uses-determination-mechanism | fixed-to-max |
                                                empty | 1..MAX) OPTIONAL,

-- <Padding>
  &value-justification                  Justification DEFAULT right:0,
  &value-pre-padding                    Padding DEFAULT zero,
  &value-pre-pattern                    Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,
  &value-post-padding                   Padding DEFAULT zero,
  &value-post-pattern                   Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Determination mechanism>
  &container-length-mechanism           DeterminationMechanism
                        (aux-determinant | app-determinant | container | handle | not-needed)
                        DEFAULT not-needed,

-- <Determinant>
  &encoding-space-length-determinant     REFERENCE OPTIONAL,
  &Encoder-transforms                    #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a count in &encoding-space-unit to actual "REFERENCE".
  &Decoder-transforms                    #TRANSFORM ORDERED OPTIONAL,
-- Transforms from the actual "REFERENCE" to a count in &encoding-space-unit.
-- Allowed only if app-determinant is set.

-- <Run-out of container>
  &encoding-space-container            CHOICE
                                       {container          REFERENCE,
                                       end-of-encoding NULL}
                                       OPTIONAL,

-- <Concatenation handle information>
  &handle-id                            PrintableString OPTIONAL,
  &Handle-component-numbers             INTEGER OPTIONAL,

-- <Bit reversal>
  &bit-reversal                         ReversalSpecification
                                        DEFAULT no-reversal  }

WITH SYNTAX {
    [ENCODING-SPACE
        [ALIGNED TO &encoding-space-pre-alignment-unit
            [PADDING &encoding-space-pre-padding
                [PATTERN &encoding-space-pre-pattern]]]
        [SIZE &encoding-space-size]
        [MULTIPLE OF &encoding-space-unit]
```

```
        [LENGTH AS &encoding-space-length-mechanism
            [CONTAINED IN &encoding-space-container]
            [DETERMINED BY &encoding-space-length-determinant
                [ENCODER-TRANSFORMS &Encoder-transforms]
                [DECODER-TRANSFORMS &Decoder-transforms]]]]
    [VALUE
        [JUSTIFIED &value-justification]
        [PRE-PADDING &value-pre-padding
            [PATTERN &value-pre-pattern]]
        [POST-PADDING &value-post-padding
            [PATTERN &value-post-pattern]]]
    [CONCAT &concatenation-procedure]
    [ORDER &concatenation-order
        [HANDLED BY &order-handle-id]]
    [HANDLE &handle-id AT &Handle-component-numbers]
    [BIT-REVERSAL &bit-reversal]
}
```

A.9 Defining encoding objects for #INT, #CONDITIONAL-INT, #INTEGER and #ENUMERATED classes

The syntax defined here can be used to define encoding objects of classes #INT, #INTEGER and #ENUMERATED.

```
#INT ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding           Padding DEFAULT zero,
  &encoding-space-pre-pattern           Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Int value-encoding>
  &Integer-encodings                     #CONDITIONAL-INT    OPTIONAL,
  &integer-encoding                      #CONDITIONAL-INT    OPTIONAL,
-- Exactly one of the above must be supplied.

-- <Primitive handle information>
  &handle-id                            PrintableString OPTIONAL,
  &Handle-positions                     INTEGER (1..MAX) OPTIONAL,

-- <Bit reversal>
  &bit-reversal                         ReversalSpecification
                                        DEFAULT no-reversal   }

WITH SYNTAX {
    [ENCODING-SPACE ALIGNED TO &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
            [PATTERN &encoding-space-pre-pattern]]]
    [ENCODING &integer-encoding]
    [ENCODINGS &Integer-encodings]
    [HANDLE &handle-id AT &Handle-positions]
    [BIT-REVERSAL &bit-reversal]
}


#CONDITIONAL-INT ::= CLASS {

-- <Range predicate>
  &range-condition                      RangeCondition  OPTIONAL,

-- <Cond-int value-encoding>
  &Value-transforms                     #TRANSFORM ORDERED OPTIONAL,
-- Transforms from int to bits

-- <Encoding space>
  &encoding-space-unit                  Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-size                  Size (uses-determination-mechanism | fixed-to-max |
                                                empty | 1..MAX) OPTIONAL,
```

```
-- <Padding>
  &value-justification                Justification DEFAULT right:0,
  &value-pre-padding                  Padding DEFAULT zero,
  &value-pre-pattern                  Pattern (ALL EXCEPT other)
                                      DEFAULT bits:'0'B,
  &value-post-padding                 Padding DEFAULT zero,
  &value-post-pattern                 Pattern (ALL EXCEPT other)
                                      DEFAULT bits:'0'B,

-- <Determination mechanism>
  &encoding-space-length-mechanism    DeterminationMechanism
                                      (aux-determinant | app-determinant |
                                       container | not-needed)
                                      OPTIONAL,

-- <Determinant>
  &encoding-space-length-determinant  REFERENCE OPTIONAL,
  &Encoder-transforms                 #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a count in &encoding-space-unit to actual "REFERENCE".
  &Decoder-transforms                 #TRANSFORM ORDERED OPTIONAL,
-- Transforms from the actual "REFERENCE" to a count in &encoding-space-unit.
-- Allowed only if app-determinant is set.

-- <Run-out of container>
  &encoding-space-container           CHOICE
                                     {container           REFERENCE,
                                      end-of-encoding NULL}
                                      OPTIONAL,

-- <Unused bits>
  &Unused-space-determinant      REFERENCE OPTIONAL,
  &Unused-space-transforms       #TRANSFORM ORDERED OPTIONAL
-- Transforms from a count in bits to actual "REFERENCE".  -- }

WITH SYNTAX {
            [IF &range-condition][ENCODING-SPACE
            [SIZE &encoding-space-size]
            [MULTIPLE OF &encoding-space-unit]
            [LENGTH AS &encoding-space-length-mechanism
                [CONTAINED IN &encoding-space-container]
                [DETERMINED BY &encoding-space-length-determinant
                    [ENCODER-TRANSFORMS &Encoder-transforms]
                    [DECODER-TRANSFORMS &Decoder-transforms]]]
            [UNUSED DETERMINED BY &Unused-space-determinant
                [ENCODER-TRANSFORMS
                    &Unused-space-transforms]]]
    [VALUE [ENCODER-TRANSFORMS &Value-transforms]
        [JUSTIFIED &value-justification]
        [PRE-PADDING &value-pre-padding
            [PATTERN &value-pre-pattern]]
        [POST-PADDING &value-post-padding
            [PATTERN &value-post-pattern]]]
}
```

A.10 Defining encoding objects for #NUL and #NULL classes

The syntax defined here can be used to define encoding objects of classes #NUL and #NULL.

```
#NUL ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit  Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding         Padding DEFAULT zero,
  &encoding-space-pre-pattern         Pattern (ALL EXCEPT other)
                                      DEFAULT bits:'0'B,

-- <Nul value-encoding>
  &value-pattern                      Pattern (ALL EXCEPT other)
                                      DEFAULT bits:'0'B,

-- <Encoding space>
  &encoding-space-unit                Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-size                Size (empty | 1 .. MAX) OPTIONAL,

-- <Padding>
  &value-justification                Justification DEFAULT right:0,
  &value-pre-padding                  Padding DEFAULT zero,
```

```
  &value-pre-pattern                     Pattern (ALL EXCEPT other)
                                         DEFAULT bits:'0'B,
  &value-post-padding                    Padding DEFAULT zero,
  &value-post-pattern                    Pattern (ALL EXCEPT other)
                                         DEFAULT bits:'0'B,

-- <Determinant>
  &encoding-space-length-determinant     REFERENCE OPTIONAL,
  &Encoder-transforms                    #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a count in &encoding-space-unit to actual "REFERENCE".

-- <Primitive handle information>
  &handle-id                             PrintableString OPTIONAL,
  &Handle-positions                      INTEGER (1..MAX) OPTIONAL,

-- <Bit reversal>
  &bit-reversal                          ReversalSpecification
                                         DEFAULT no-reversal}

WITH SYNTAX {
    [ENCODING-SPACE
        [ALIGNED TO &encoding-space-pre-alignment-unit
            [PADDING &encoding-space-pre-padding
                [PATTERN &encoding-space-pre-pattern]]]
        [SIZE &encoding-space-size]
        [MULTIPLE OF &encoding-space-unit]
            [DETERMINED BY &encoding-space-length-determinant
                [ENCODER-TRANSFORMS
                    &Encoder-transforms]]]
    [VALUE      [NULL-PATTERN &value-pattern]
        [JUSTIFIED &value-justification]
        [PRE-PADDING &value-pre-padding
            [PATTERN &value-pre-pattern]]
        [POST-PADDING &value-post-padding
            [PATTERN &value-post-pattern]]]
    [HANDLE &handle-id AT &Handle-positions]
    [BIT-REVERSAL &bit-reversal]
}
```

A.11 Defining encoding objects for #OCTETS and #OCTET-STRING classes

The syntax defined here can be used to define encoding objects of classes #OCTETS and #OCTET-STRING.

```
#OCTETS ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit     Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding            Padding DEFAULT zero,
  &encoding-space-pre-pattern            Pattern (ALL EXCEPT other)
                                         DEFAULT bits:'0'B,

-- <Octet value-encoding>
  &Octet-repetition-encodings            #CONDITIONAL-REPETITION
                                         OPTIONAL,
  &octet-repetition-encoding             #CONDITIONAL-REPETITION
                                         OPTIONAL,
-- Exactly one of the above must be supplied.

-- <Contained type encoding>
  &Primary-encoding-object-set           #ENCODINGS OPTIONAL,
  &Secondary-encoding-object-set         #ENCODINGS OPTIONAL,
  &over-ride-encoded-by                  BOOLEAN OPTIONAL,

-- <Primitive handle information>
  &handle-id                             PrintableString  OPTIONAL,
  &Handle-positions                      INTEGER (1..MAX) OPTIONAL,

-- <Bit reversal>
  &bit-reversal                          ReversalSpecification
                                         DEFAULT no-reversal   }

WITH SYNTAX {
    [ENCODING-SPACE ALIGNED TO &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
            [PATTERN &encoding-space-pre-pattern]]]
```

```
    [ENCODING  &octet-repetition-encoding]
    [ENCODINGS &Octet-repetition-encodings]
    [CONTAINING &Primary-encoding-object-set
        [COMPLETED BY &Secondary-encoding-object-set]
        [OVERRIDE &over-ride-encoded-by]]
    [HANDLE &handle-id AT &Handle-positions]
    [BIT-REVERSAL &bit-reversal]
}
```

A.12 Defining encoding objects for optionality classes

The syntax defined here can be used to define encoding objects of class #OPTIONAL.

```
#OPTIONAL ::= ENCODING-CLASS {

-- <Determination mechanism>
  &optionality-mechanism                DeterminationMechanism
(aux-determinant | app-determinant | container | handle),

-- <Determinant>
  &optionality-presence-determinant     REFERENCE OPTIONAL,
  &Encoder-transforms                   #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a boolean (TRUE if present) to actual "REFERENCE".
  &Decoder-transforms                   #TRANSFORM ORDERED OPTIONAL,
-- Transforms from the actual "REFERENCE" to a boolean.  Present if TRUE.
-- Allowed only if app-determinant is set.

-- <Run-out of container>
  &encoding-space-container             CHOICE
                                       {container           REFERENCE,
                                        end-of-encoding NULL}
                                        OPTIONAL,

-- <Use handle>
  &handle-id                            PrintableString OPTIONAL}

WITH SYNTAX {
    ENCODING-SPACE AS &optionality-mechanism
        [DETERMINED BY &optionality-presence-determinant
            [ENCODER-TRANSFORMS &Encoder-transforms]
[DECODER-TRANSFORMS &Decoder-transforms]]]
    [CONTAINED IN &encoding-space-container]
    [HANDLE &handle-id]
}
```

A.13 Defining encoding objects for the #PAD class

The syntax defined here can be used to define encoding objects of class #PAD.

```
#PAD ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding           Padding DEFAULT zero,
  &encoding-space-pre-pattern           Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Encoding space>
  &encoding-space-unit                  Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-size                  Size (uses-determination-mechanism | 1..MAX ),

-- <Pad Padding>
  &padding                              Padding DEFAULT zero,
  &pattern                              Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Determinant>
  &encoding-space-length-determinant    REFERENCE OPTIONAL,
  &Encoder-transforms                   #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a count in &encoding-space-unit to actual "REFERENCE".

-- <Primitive handle information>
  &handle-id                            PrintableString OPTIONAL,
  &Handle-positions                     INTEGER (1..MAX) OPTIONAL,
```

```
-- <Bit reversal>
  &bit-reversal                         ReversalSpecification
                                        DEFAULT no-reversal}

WITH SYNTAX {
    ENCODING-SPACE
        [ALIGNED TO &encoding-space-pre-alignment-unit
            [PADDING &encoding-space-pre-padding
                [PATTERN &encoding-space-pre-pattern]]]
        SIZE &encoding-space-size
        [MULTIPLE OF &encoding-space-unit]
        [DETERMINED BY &encoding-space-length-determinant
            [ENCODER-TRANSFORMS
                &Encoder-transforms]]
    [VALUE  PADDING &padding
        [PATTERN &pattern]]
    [HANDLE &handle-id AT &Handle-positions]
    [BIT-REVERSAL &bit-reversal]
}
```

A.14 Defining #REPETITION, #CONDITIONAL-REPETITION, #SEQUENCE-OF, #SET-OF class encodings

The syntax defined here can be used to define encoding objects of class #REPETITION, #SEQUENCE-OF, and #SET-OF.

```
#REPETITION ::= ENCODING-CLASS {

-- <Pre-alignment>
  &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-pre-padding           Padding DEFAULT zero,
  &encoding-space-pre-pattern           Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Repetition-encoding>
  &Repetition-encodings                  #CONDITIONAL-REPETITION OPTIONAL,
  &repetition-encoding                   #CONDITIONAL-REPETITION OPTIONAL,
-- Exactly one of the above must be supplied.

-- <Repetition handle information>
  &handle-id                            PrintableString OPTIONAL,

-- <Bit reversal>
  &bit-reversal                         ReversalSpecification
                                        DEFAULT no-reversal  }

WITH SYNTAX {
    [ENCODING-SPACE ALIGNED TO &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
            [PATTERN &encoding-space-pre-pattern]]]
    [ENCODING &repetition-encoding]
    [ENCODINGS &Repetition-encodings]
    [HANDLE &handle-id]
    [BIT-REVERSAL &bit-reversal]
}


#CONDITIONAL-REPETITION ::= ENCODING-CLASS {
-- <Size predicate>
  &size-range-condition                 SizeRangeCondition  OPTIONAL,

-- <Concatenation procedure>
  &concatenation-procedure              ENUMERATED
                                        {simple, pre-aligned}
                                        DEFAULT simple,

-- <Encoding space>
  &encoding-space-unit                  Unit (ALL EXCEPT repetitions) DEFAULT bit,
  &encoding-space-size                  Size (uses-determination-mechanism | fixed-to-max |
                                                empty | 1..MAX) OPTIONAL,

-- <Padding>
  &value-justification                  Justification DEFAULT right:0,
  &value-pre-padding                    Padding DEFAULT zero,
  &value-pre-pattern                    Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,
```

```
  &value-post-padding                   Padding DEFAULT zero,
  &value-post-pattern                   Pattern (ALL EXCEPT other)
                                        DEFAULT bits:'0'B,

-- <Determination mechanism>
  &repetition-length-mechanism          DeterminationMechanism
                       (aux-determinant | app-determinant | container | pattern | handle),

-- <Determinant>
  &repetition-length-determinant        REFERENCE OPTIONAL,
  &repetition-length-unit               Unit DEFAULT bit,
  &Encoder-transforms                   #TRANSFORM ORDERED OPTIONAL,
-- Transforms from count of &repetition-length-unit to actual "REFERENCE".
  &Decoder-transforms                   #TRANSFORM ORDERED OPTIONAL,
-- Transforms from the actual "REFERENCE" to a count in &repitition-length-unit.
-- Allowed only if app-determinant is set.

-- <Run-out of container>
  &encoding-space-container             CHOICE
                                           {container           REFERENCE,
                                            end-of-encoding NULL}
                                            OPTIONAL,

-- <Unused bits>
  &Unused-space-determinant             REFERENCE OPTIONAL,
  &Unused-space-transforms              #TRANSFORM ORDERED OPTIONAL,
-- Transforms from a count in bits to actual "REFERENCE".

-- <Append pattern>
  &termination-pattern                  BIT STRING OPTIONAL,

-- <Use handle>
  &handle-id                            PrintableString OPTIONAL
}

WITH SYNTAX {
    [IF &size-range-condition]
    ENCODING-SPACE AS &repetition-length-mechanism
        [SIZE &encoding-space-size]
        [MULTIPLE OF &encoding-space-unit]
        [CONTAINED IN &encoding-space-container]
        [DETERMINED BY &repetition-length-determinant
            [ENCODER-TRANSFORMS  &Encoder-transforms]
            [DECODER-TRANSFORMS  &Decoder-transforms]]
        [COUNT &repetition-length-unit]
        [TERMINATOR &termination-pattern]
        [UNUSED DETERMINED BY &Unused-space-determinant
            [ENCODER-TRANSFORMS
                &Unused-space-transforms]]
    [VALUE JUSTIFIED &value-justification
        [PRE-PADDING &value-pre-padding
            [PATTERN &value-pre-pattern]]
        [POST-PADDING &value-post-padding
            [PATTERN &value-post-pattern]]]
    [CONCAT &concatenation-procedure]
    [HANDLE &handle-id]
}
```

A.15 Defining encoding objects for #OUTER class

This encoding class and its semantics is described in clause 19.

```
#OUTER ::= ENCODING-CLASS {
        -- <Padding>
          &post-padding-unit        Unit (1..MAX) DEFAULT bit,
          &post-padding             Padding DEFAULT zero,
          &post-padding-pattern     Pattern (ALL EXCEPT other)
                                    DEFAULT bits:'0'B,
          &alignment-point          ENUMERATED
                                    {unchanged, reset } DEFAULT reset }
         WITH SYNTAX {
           [ALIGNMENT &alignment-point]
           [MULTIPLE OF &post-padding-unit]
           [POST-PADDING &post-padding
               [PATTERN &post-padding-pattern]]
        }
```

Annex B (Normative): Addendum to ITU-T Rec. X.680 | ISO/IEC 8824-1

This annex specifies the modifications that are to be applied when productions and/or clauses from ITU-T Rec. X.680 | ISO/IEC 8824-1 are referenced in this TS.

B.1 Exports and imports statements

The productions "AssignedIdentifier", "Symbol" and "Reference" of 12.1, as well as sub-clauses 12.12, 12.15, and 12.19 of ITU-T Rec. X.680 | ISO/IEC 8824-1 are modified as follows:

```
AssignedIdentifier ::= DefinitiveIdentifier

Symbol ::=
        Reference                       |
        BuiltinEncodingClassReference   |
        ParameterizedReference

Reference ::=
        typereference                   |
        valuereference                  |
        encodingclassreference          |
        encodingobjectreference         |
        encodingobjectsetreference
```

NOTE 1 – The production "AssignedIdentifier" is changed because "valuereference"s can neither be defined nor imported into ELM or EDM modules.

NOTE 2 – "valuereference"s and "typereference"s cannot appear as "Symbol"s in an import or export clause. They are valid as "Symbol"s only when "Symbol" is used to represent a dummy parameter.

When the "SymbolsExported" alternative of "Exports" is selected, then each "Symbol" in "SymbolsExported" shall satisfy one and only one of the following conditions:

a) it is defined in the module from which it is being exported; or

b) it appears exactly once in the "SymbolsImported" alternative of "Imports" in the module from which it is being exported;

When the "SymbolsImported" alternative of "Imports" is selected:

Each "Symbol" in "SymbolsFromModule" shall either

be defined in the body of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule", or

be present precisely once in the "IMPORTS" clause of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule".

NOTE – This does not prohibit the same symbol name defined in two different modules from being imported into another module. However, if the same "Symbol" name appears more than once in the "IMPORTS" clause of module "A", that "Symbol" name cannot be exported from "A" for import to another module "B".

b) All the "SymbolsFromModule" in the "SymbolsFromModuleList" shall include occurrences of "GlobalModuleReference" such that:

i) the "modulereference" in them are all different from each other (whether they are ASN.1, ELM or EDM modules) and from the "modulereference" associated with the referencing module; and

ii) the "AssignedIdentifier", when non-empty, denotes object identifier values which are all different from each other and from the object identifier value (if any) associated with the referencing module.

B.2 Absolute reference

B.2.1 ITU-T Rec. X.680 | ISO/IEC 8824-1, 14.3, and ITU-T Rec. X.680 Corr. 2 | ISO/IEC 8824-1 Corr. 2, clause 4, are modified as follows:

```
AbsoluteReference ::=
        ModuleIdentifier
        "."
        ItemSpec
        ItemSpec ::=
        typereference    |
        valuereference
```

B.3 Addition of "REFERENCE"

NOTE This modification is introduced for the sole purpose of Annex A.

"Type" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.1, is amended as follows:

```
Type ::=
        BuiltinType      |
        ReferencedType   |
        ConstrainedType  |
        REFERENCE
```

B.4 Notation for character string values

B.4.1 The production "CharsDefn" of ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7, is modified as follows:

```
CharsDefn ::=
        cstring      |
        Quadruple    |
        Tuple        |
        AbsoluteReference
```

B.4.2 The "AbsoluteReference" references a character string value (of type IA5String or BMPString) defined in the ASN1-CHARACTER-MODULE (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 37.1).

Annex C (Normative): Addendum to ITU-T Rec. X.681 | ISO/IEC 8824-2

This annex specifies the modifications that are to be applied when productions and/or clauses from ITU-T Rec. X.681 | ISO/IEC 8824-2 are referenced in this TS.

C.1 Definitions

The following definitions are added to ITU-T Rec. X.681 | ISO/IEC 8824-2, 3.4:

encoding class field type: A type specified by reference to some field of an encoding object class.

encoding object field: A field which contains an encoding object of some specified encoding class.

encoding object list field: A field which contains an (ordered) list of encoding objects of some specified encoding class.

encoding object set field: A field which contains a set of encoding objects of some specified encoding class.

C.2 Additional lexical items

NOTE This modification is introduced for the sole purpose of Annex A.

The following definition is added to ITU-T Rec. X.681 | ISO/IEC 8824-2, 7:

7.8bis **Encoding object list field references**

Name of item – encodingobjectlistfieldreference

An "encodingobjectlistfieldreference" shall consist of an ampersand ("&") immediately followed by a sequence of characters as specified for an "objectsetreference" in ITU-T Rec. X.681 | ISO/IEC 8824-2, 7.3.

C.3 Addition of "ENCODING-CLASS"

NOTE This modification is introduced for the sole purpose of Annex A.

For the purpose of Annex A only, replace the reserved word "CLASS" with "ENCODING-CLASS" in ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.3.

C.4 FieldSpec additions

NOTE This modification is introduced for the sole purpose of Annex A.

ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.4, is amended as follows:

```
FieldSpec ::=
                FixedTypeValueFieldSpec      |
                FixedTypeValueSetFieldSpec   |
                EncodingObjectFieldSpec      |
                EncodingObjectSetFieldSpec   |
                EncodingObjectListFieldSpec
```

C.5 Encoding object field spec

NOTE This modification is introduced for the sole purpose of Annex A.

An "EncodingObjectFieldSpec" specifies that the field is an encoding object field :

```
EncodingObjectFieldSpec ::=
        objectfieldreference
        DefinedEncodingClass
        EncodingObjectOptionalitySpec?
```

```
EncodingObjectOptionalitySpec ::= OPTIONAL | DEFAULT EncodingObject
```

The name of the field is "objectfieldreference". The "DefinedEncodingClass" references the encoding class of the encoding object contained in the field (which may be the "EncodingObjectClass" currently being defined). The "EncodingObjectOptionalitySpec", if present, specifies that the field may be unspecified in an encoding object definition, or, in the "DEFAULT" case, that omission produces the following "EncodingObject"which shall be of the "DefinedObjectClass".

C.6 Encoding object set field spec

NOTE This modification is introduced for the sole purpose of Annex A.

An "EncodingObjectSetFieldSpec" specifies that the field is an encoding object set field :

```
EncodingObjectSetFieldSpec ::=
        objectsetfieldreference
        DefinedEncodingClass
        EncodingObjectSetOptionalitySpec?

        EncodingObjectSetOptionalitySpec ::= OPTIONAL | DEFAULT EncodingObjectSet
```

The name of the field is "objectsetfieldreference". The "DefinedEncodingClass" references the class of the encoding objects contained in the field. The "EncodingObjectSetOptionalitySpec", if present, specifies that the field may be unspecified in an encoding object definition, or, in the "DEFAULT" case, that omission produces the following "EncodingObjectSet", all of whose objects shall be of "DefinedEncodingClass".

C.7 Encoding object list field spec

NOTE This modification is introduced for the sole purpose of Annex A.

An "EncodingObjectListFieldSpec" specifies that the field is an encoding object list field :

```
EncodingObjectListFieldSpec ::=
        encodingobjectlistfieldreference
        DefinedEncodingClass
        ORDERED
        EncodingObjectListOptionalitySpec?

        EncodingObjectListOptionalitySpec ::= OPTIONAL | DEFAULT EncodingObjectList
```

The name of the field is "objectlistfieldreference". The "DefinedEncodingClass" references the class of the encoding objects contained in the field. The "EncodingObjectListOptionalitySpec", if present, specifies that the field may be unspecified in an encoding object definition, or, in the "DEFAULT" case, that omission produces the following "EncodingObjectList" all of whose objects shall be of "DefinedEncodingClass".

C.8 Encoding object list notation

```
EncodingObjectList ::= "{" EncodingObject "," * "}"
```

The "EncodingObjectList" is an ordered list of encoding objects, and may contain multiple encoding objects. It is used when the application applies semantics to the order of values in the list.

EXAMPLE: A list of #TRANSFORM encoding objects is applied in the stated order.

NOTE As ASN.1 has no concept of object list reference names or assignments, an object list can only be specified by in-line notation when governed by an object list field type of an encoding class.

C.9 Primitive field names

ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.13, is amended as follows:

The construct "PrimitiveFieldName" is used to identify a field relative to the encoding class containing its specification:

```
PrimitiveFieldName ::=
        valuefieldreference        |
```

```
        valuesetfieldreference      |
        valuelistfieldreference
```

C.10 Additional reserved words

ITU-T Rec. X.681 | ISO/IEC 8824-2, 10.6, is amended as follows:

10.6 A "word" lexical item used as a "Literal" not be one of the following:

```
ALL                     END                       PER-canonical-unaligned
BEGIN                   FALSE                     PLUS-INFINITY
BER                     MINUS-INFINITY            TRUE
CER                     NULL                      UNION
DER                     PER-basic-aligned         USE
ENCODE                  PER-basic-unaligned       USER-FUNCTION-BEGIN
ENCODE-DECODE           PER-canonical-aligned
```

NOTE 2 This list comprises only those ASN.1 reserved words which can appear as the first item of a "Value", "EncodingObject", "EncodingObjectSet" or "EncodingClass", and also the reserved word "END". Use of other ECN reserved words does not cause ambiguity and is permitted. Where the defined syntax is used in an environment in which a "word" is also an "encodingobjectsetreference", the use as a "word" takes precedence.

C.11 Definition of encoding objects

The restriction imposed by ITU-T Rec. X.681 | ISO/IEC 8824-2, 10.12.d, is removed.

NOTE This only affects the defined syntax for defining an encoding object of class #TRANSFORM (see A.3). It means, for example, that, for a defined syntax such as:

[BOOL-TO-INT [AS &bool-to-int]]

you are allowed to write:

BOOL-TO-INT

when defining an encoding object of this class. In such a case, the "DEFAULT" value associated with the parameter "&bool-to-int" (i.e., "false-zero") is used in the definition of the transformation "BOOL-TO-BOOL".

C.12 Additions to "Setting"

ITU-T Rec. X.681 | ISO/IEC 8824-2, 11.6, is amended as follows:

A "Setting" specifies the setting of some field within an encoding object being defined:

```
Setting ::=
        Value                 |
        ValueSet              |
        EncodingObject        |
        EncodingObjectSet     |
        EncodingObjectList
```

If the field is:

- a value field, the "Value" alternative;
- a value set field, the "ValueSet" alternative;
- an encoding object field, the "EncodingObject" alternative;
- an encoding object set field, the "EncodingObjectSet" alternative;
- an encoding object list field, the "EncodingObjectList" alternative;

shall be selected.

NOTE The setting is further restricted as described in ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.5 to 9.12, and 11.7 to 11.8.

C.13 Encoding class field type

The type that is referenced by this notation depends on the category of the field name. For the different categories of field names.

The notation for an encoding class field type shall be "EncodingClassFieldType":

```
EncodingClassFieldType ::=
        DefinedEncodingClass
        "."
        FieldName
```

where the "FieldName" is as specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.14, relative to the class identified by the "DefinedEncodingClass".

For a fixed-type value or a fixed type value set field, the notation denotes the "Type" that appears in the specification of that field in the definition of the encoding object class.

This notation is not permitted if the field is an encoding object, an encoding object set or an encoding object list field.

The notation for defining a value of this type shall be "FixedTypeFieldVal" as defined in ITU-T Rec. X.681 | ISO/IEC 8825-2, 14.6.

Annex D (Normative): Addendum to ITU-T Rec. X.683 | ISO/IEC 8824-4

This annex specifies the modifications that need to be applied when productions and/or clauses from ITU-T Rec. X.683 | ISO/IEC 8824-4 are referenced in this TS.

D.1 Parameterized assignments

Clauses 8.1 and 8.3 of ITU-T Rec. X.683 | ISO/IEC 8824-4 are modified as follows:

8.1 There are parameterized assignment statements corresponding to each of the assignment statements specified in this TS. The "ParameterizedAssignment" construct is:

```
ParameterizedAssignment ::=
        ParameterizedEncodingObjectAssignment       |
        ParameterizedEncodingClassAssignment        |
        ParameterizedEncodingObjectSetAssignment

ParameterList ::= "{<"  Parameter  ","  +  ">}"

Governor ::=
        DefinedEncodingClass    |
        EncodingClassFieldType  |
        REFERENCE
```

A "DummyReference" in "Parameter" may stand for:

a) an encoding class, in which case there shall be no "ParamGovernor";

an ASN.1 value, in which case the "ParamGovernor" shall be present as a "Governor" that is a type extracted from an encoding class ("EncodingClassFieldType");

an "identifier", in which case the "ParamGovernor" shall be present as a "Governor" that is "REFERENCE";

an "EncodingObject" in which case the "ParamGovernor" shall be present as a "Governor" that is an encoding class ("EncodingClass");

an "EncodingObjectSet", in which case the "ParamGovernor" shall be present as a "Governor" that is "#ENCODINGS".

NOTE "DummyGovernor"s are not allowed in ECN.

D.2 Parameterized encoding assignments

The following productions are added to ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.2:

```
ParameterizedEncodingClassAssignment ::=
        encodingclassreference
        ParameterList
        "::="
        EncodingClass

ParameterizedEncodingObjectAssignment ::=
        encodingobjectreference
        ParameterList
        EncodingClass
        "::="
        EncodingObject

ParameterizedEncodingObjectSetAssignment ::=
        ParameterList
        #ENCODINGS
        "::="
        EncodingObjectSet
```

D.3 Referencing parameterized definitions

The production "ParameterizedReference" of ITU-T Rec. X.683 | ISO/IEC 8824-4, 9.1, is modified as follows:

```
ParameterizedReference ::=
        Reference       |
        Reference "{<"   ">}"
```

The following productions are added to ITU-T Rec. X.683 | ISO/IEC 8824-4, 9.2:

```
ParameterizedEncodingObject ::=
        SimpleDefinedEncodingObject
        ActualParameterList

SimpleDefinedEncodingObject ::=
        ExternalEncodingObjectReference |
        Encodingobjectreference

ParameterizedEncodingObjectSet ::=
        SimpleDefinedEncodingObjectSet
        ActualParameterList

SimpleDefinedEncodingObjectSet ::=
        ExternalEncodingObjectSetReference |
        Encodingobjectsetreference

ParameterizedEncodingClass ::=
        SimpleDefinedEncodingClass
        ActualParameterList

SimpleDefinedEncodingClass ::=
        ExternalEncodingClassReference |
        encodingclassreference
```

D.4 Actual parameter list

ITU-T Rec. X.683 | ISO/IEC 8824-4, 9.5, is modified as follows:

The "ActualParameterList" is:

```
ActualParameterList ::=
        "{<" ActualParameter "," + ">}"

ActualParameter ::=
        Value                   |
        ValueSet                |
        EncodingObject          |
        EncodingObjectSet       |
        EncodingObjectList      |
        identifier              |
        OUTER
```

Annex E (Informative): Examples

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

General examples, which show the look-and-feel of ECN definitions (E.1).

Specialization examples. These examples show how to modify some parts of a standard encoding. Each example has a description of the requirements for the encoding and a description of the selected solution and possible alternative solutions. (E.2)

Legacy protocol examples. These examples show how to construct ECN definitions for a protocol whose message encodings have been specified using a tabular notation (E.3)

E.1 General examples

The examples described in E.1.2 to E.1.12 are part of a complete ECN specification whose ELM, ASN.1, and EDM modules are given in outline in E.1.13, E.1.14 and E.1.15 are given completely in a copy of this Annex which is available from the web-site cited in Annex G.

E.1.1 An encoding object set

```
Example1Encodings #ENCODINGS ::= {
        marriedEncoding          |
        evenPositiveIntegerEncoding |
        evenNegativeIntegerEncoding |
        positiveIntegerEncoding     |
        negativeIntegerEncoding     |
        integerRightAlignedEncoding |
        integerWithHoleEncoding     |
        positiveIntegerBCDEncoding  |
        bitStringEncoding           |
        octetStringEncoding         |
        characterStringEncoding         |
        characterStringToBitEncoding    |
        sequence1Encoding }
```

This encoding object set contains encoding definitions for some types specified in the ASN.1 module named "Example1-ASN1-Module".

E.1.2 An encoding object for a boolean type

The ASN.1 assignment is:

```
Married ::= BOOLEAN
```

The encoding object assignment is:

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

There is no pre-alignment, and the encoding space is one "bit", so "Married" is 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.

The values used above are the values that would be set by default if the settings were omitted, so the same encoding can be achieved with less verbosity by :

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

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 using the syntax of 16.3.

```
marriedEncoding-3 #Married ::= {
        ENCODE WITH PER-basic-unaligned }
```

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.

E.1.3 An encoding object for an integer type

The ASN.1 assignments are:

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

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

The encoding object assignments are:

```
evenPositiveIntegerEncoding #EvenPositiveInteger ::= {
        USE     #INT (0..MAX)
                MAPPING TRANSFORMS { {INT-TO-INT divide:2} }
        WITH { ENCODE WITH PER-basic-unaligned }}

evenNegativeIntegerEncoding #EvenNegativeInteger ::= {
        USE     #INT (MIN..0)
                MAPPING TRANSFORMS { {INT-TO-INT divide:2
                             -- Note: -1 / 2 = 0 - see clause 19 -- } }
        WITH { ENCODE WITH PER-basic-unaligned }}
```

An even value is divided by two, then encoded using standard PER basic unaligned encoding rules for positive and negative integer types.

E.1.4 Another encoding object for an integer type

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

The ASN.1 assignment is:

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

The Encoding object assignment is:

```
integerRightAlignedEncoding #IntegerRightAligned ::= {
        ENCODING-SPACE
            ALIGNED TO octet
        ENCODING {
            ENCODING-SPACE
                SIZE 16
            VALUE
                JUSTIFIED right: 0 } }
```

E.1.5 Encodings of values of integer types with holes

The ASN.1 assignment is:

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

The encoding object assignment is:

```
integerWithHoleEncoding #IntegerWithHole ::= {
        USE     #INT (0..1280)
                MAPPING ORDERED VALUES
        WITH {ENCODE WITH PER-basic-unaligned }}
```

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

E.1.6 A more complex encoding object for an integer type

The ASN.1 assignments are

```
PositiveInteger ::= INTEGER (1..MAX)

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

The encoding object assignments are:

```
positiveIntegerEncoding #PositiveInteger ::=
        integerEncoding

negativeIntegerEncoding #NegativeInteger ::=
        integerEncoding
```

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

The "integerEncoding" encoding object defined here is very powerful, but quite complex. It contains five encoding objects of the class #CONDITIONAL-INT; the encoding of all of these is **octet-aligned**. 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. The definition of the encoding object is:

```
integerEncoding #INT ::= {ENCODINGS {
{ IF unbounded-or-no-lower-bound
    ENCODING-SPACE
        SIZE container-mechanism
        MULTIPLE OF octet
        LENGTH AS container
            CONTAINED IN end-of-encoding:NULL
    VALUE
            ENCODER-TRANSFORMS { {INT-TO-BITS AS twos-complement} } } |
{ IF bounded-with-negatives
    ENCODING-SPACE
        SIZE fixed-to-max
    VALUE
            ENCODER-TRANSFORMS { {INT-TO-BITS AS twos-complement} } } |
{ IF semi-bounded-with-negatives
    ENCODING-SPACE
        SIZE container-mechanism
        MULTIPLE OF octet
        LENGTH AS container
            CONTAINED IN end-of-encoding:NULL
    VALUE
                ENCODER-TRANSFORMS { {INT-TO-BITS AS twos-complement} } } |
{ IF semi-bounded-without-negatives
    ENCODING-SPACE
        SIZE container-mechanism
        MULTIPLE OF octet
        LENGTH AS container
            CONTAINED IN end-of-encoding:NULL
    VALUE
                ENCODER-TRANSFORMS { {INT-TO-BITS AS positive-int} } } |
{ IF bounded-without-negatives
    ENCODING-SPACE
        SIZE fixed-to-max
    VALUE
                ENCODER-TRANSFORMS { {INT-TO-BITS AS positive-int} } } } }
```

E.1.7 Positive integers encoded in BCD

This example shows how to encode a positive integer in BCD by successive transformations: from integer to character string then from character string to bit string.

The ASN.1 assignment is:

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

The encoding object assignments:

```
positiveIntegerBCDEncoding #PositiveIntegerBCD ::= {
    USE #CHARS
    MAPPING TRANSFORMS{{
```

```
        INT-TO-CHARS
        -- Here we convert to characters, so for example, the integer value 42
        -- becomes the 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 ::= {
    ENCODING-SPACE
        ALIGNED TO          nibble
    VALUE
        ENCODER-TRANSFORMS {{
            CHAR-TO-BITS
            -- Here we convert each character to a bit string, so, for example,
            -- the 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 }

        }}
ENCODING {
    ENCODING-SPACE
    -- Here we determine the concatenation of the bit strings for the
    -- characters and add a terminator,  so here
    -- '0100'B + '0010'B becomes '0100 0010 1111'B
            AS              pattern
            TERMINATOR  '1111'B } }
```

The positive number is first transformed into a character string by the int-to-chars transformation 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 bit string is aligned on a nibble boundary and terminates with a specific pattern '1111'B.

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.

E.1.8 An encoding object of class #BITS

This example defines an encoding object (of class #BITS) for a bit string that is to be octet-aligned, using padding with 0, and is terminated by an 8-bit field containing '00000000'B (it is assumed that the abstract value never contains eight successive zeros):

The ASN.1 assignment is:

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

The encoding object assignment is:

```
bitStringEncoding #BitString ::= {
        ENCODING-SPACE
            ALIGNED TO octet
        ENCODING {
            ENCODING-SPACE
                AS pattern
                TERMINATOR '00000000'B } }
```

The encoding object of the class #BITS contains an embedded encoding object of the class #CONDITIONAL-REPETITION which specifies the mechanism and the termination pattern.

As with many of the examples in this Annex, there is heavy reliance here on the defaults provided in Annex A, 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.

E.1.9 An encoding object of class #OCTETS

The ASN.1 assignment is:

```
OctetString ::= OCTET STRING
```

The encoding object assignment is:

```
octetStringEncoding #OctetString ::= {
        ENCODING-SPACE ALIGNED TO octet
                        PADDING one
        ENCODING {
            ENCODING-SPACE
                AS container
                    CONTAINED IN end-of-encoding: NULL } }
```

The value is octet-aligned using padding with "1"s and terminates with the end of the PDU.

E.1.10 An encoding object of class #CHARS

The ASN.1 assignment is:

```
CharacterString ::= PrintableString
```

The encoding object assignment is:

```
characterStringEncoding #CharacterString ::= {
        ENCODING-SPACE
            ALIGNED TO octet
            VALUE
                ENCODER-TRANSFORMS { {CHAR-TO-BITS AS compact} }
            ENCODING {
                ENCODING-SPACE
                    AS container
                    CONTAINED IN end-of-encoding: NULL } }
```

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 zero for the first ASCII character in PrintableString, one for the next, and so on.

E.1.11 Mapping character values to bit values

The ASN.1 assignment is:

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

The encoding object assignment is:

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

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

E.1.12 Encoding a sequence type

Here we encode a sequence type that has a field "a" which carries application semantics (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 the PDU. We need to give specialized encodings for the optionality of b, and we use the specialized encoding defined in E.1.9 (by reference to "octetStringEncoding") for the octet string "c". We want to encode everything else with PER-basic-unaligned.

The ASN.1 assignment is:

```
Sequence1   ::= SEQUENCE{
        a   BOOLEAN,
```

```
        b   INTEGER     OPTIONAL,
        c   OCTET STRING }
```

The ECN assignments are:

```
sequence1Encoding #Sequence1 ::= {
        ENCODE STRUCTURE {
            b OPTIONAL-ENCODING parameterizedPresenceEncoding {< a >},
            c octetStringEncoding}
        WITH {ENCODE WITH PER-basic-unaligned } }

parameterizedPresenceEncoding {< REFERENCE:reference >}  #OPTIONAL ::= {
        ENCODING-SPACE
            AS app-determinant

DETERMINED BY reference}
```

Note that we did not need to provide any "DECODERS-TRANSFORMS" in the "parameterizedPresenceEncoding", because "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" specification as in E.2.7**.**

E.1.13 ELM definitions

The following ELM encodes the ASN.1 module defined in E.1.14, using objects specified in the EDM defined in E.1.15.

```
Example1-ELM LINK-DEFINITIONS

{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(1) }

::=
BEGIN
        IMPORTS Example1Encodings FROM Example-EDM
        { joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(3) };

ENCODE Example1-ASN1-Module { joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(2) }
        WITH Example1Encodings
        COMPLETED BY PER-basic-unaligned

END
```

E.1.14 ASN.1 definitions

This ASN.1 module contains a collection of type definitions for which there are encoding definitions in a separate EDM. The ASN.1 module groups all the ASN.1 definitions from E.1.2 to E.1.12 together They will be encoded according to the encoding objects defined in the EDM of E.1.15, together with the PER basic unaligned encoding rules.

```
Example1-ASN1-Module

{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(2) }

DEFINITIONS AUTOMATIC TAGS : :=

BEGIN

Married ::= BOOLEAN

-- etc.

END
```

E.1.15 EDM definitions

```
Example1-EDM

{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(3) }

ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS Example1Encodings;
IMPORTS #Married,#EvenPositiveInteger,#EvenNegativeInteger,#PositiveInteger,#NegativeInteger,
    #IntegerRightAligned,#IntegerWithHole,#PositiveIntegerBCD,#BitString,
    #OctetString,#CharacterString,#CharacterStringToBit,#Sequence1
    FROM Example1-ASN1-Module { joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(2) };

Example1Encodings #ENCODINGS ::= {
            booleanEncoding |
            -- etc
            sequence1Encoding }

-- etc

END
```

E.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 are 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-generated auxiliary fields that are generated by default, such as variable-width determinant fields.

The examples are presented using the following format:

- The ASN.1 assignment. This shows the original ASN.1 type definition.
- The requirement. This lists the requirement and the problem with the encoding provided by PER basic unaligned.
- The encoding object assignment. This shows the resulting encoding specification.
- Discussion. This discusses how the specialization has been achieved, and other options that might be used.

E.2.1 The encoding object set

```
Example2Encodings #ENCODINGS ::= {
        normallySmallValuesEncoding                         |
        sparseEvenlyDistributedValueSetEncoding             |
        sparseUnevenlyDistributedValueSetEncoding           |
        conditionalPresenceOnValueEncoding                  |
        conditionalPresenceOnExternalConditionEncoding      |
        conditionalComponentValueSetEncoding                |
        enclosingStructureForListEncoding                   |
        equalLenghListsEncoding                             |
        enclosingStructureForChoiceEncoding                 |
        extensibleIntegerEncoding                           |
        extensibleEnumeratedEncoding                        |
        extensibleTopLevelMessageEncoding }
```

This encoding object set contains encoding definitions for some of the types specified in "Example2-ASN1-Module" (the rest are encoded using PER basic unaligned).

E.2.2 Encoding by distributing values to an alternative encoding structure

The ASN.1 assignment is:

```
NormallySmallValues ::= INTEGER (0..1024)
-- Usually values are in the range 0..63, but sometimes the whole value range is used.
```

The requirement is: 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 later.

The encoding object assignment is:

```
normallySmallValuesEncoding #NormallySmallValues ::= {
        USE     #NormallySmallValuesStruct
                MAPPING DISTRIBUTION {
                    0..63       TO small,
                    REMAINDER   TO large }
        WITH    { ENCODE WITH PER-basic-unaligned }}
```

The encoding structure assignment is:

```
#NormallySmallValuesStruct ::= #CHOICE {
        small   #INT (0..63),
        large   #INT (64..1024) }
```

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 fieldThe 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.

This mapping and the encoding is quite straight-forward, but some further gains can be obtained by mapping values 64 upwards into values zero upwards of "large" (whose lower bound would then be zero), or by transforming values of large by a subtraction of 64 before encoding it. Both these options, however, would be more difficult for a reader of the specification to understand and would give only marginal further gains.

E.2.3 Encoding by mapping ordered abstract values to an alternative encoding structure

The example in E.2.2 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.

The encoding object assignment is:

```
normallySmallValuesEncoding-2 #NormallySmallValues ::= {
        USE     #NormallySmallValuesStruct2
                MAPPING ORDERED VALUES
        WITH    { ENCODE WITH PER-basic-unaligned }}
```

The encoding structure assignment is:

```
#NormallySmallValuesStruct2 ::= #CHOICE {
        small   #INT (0..63),
        large   #INT  }
```

The result is very similar to E.2.2, but now the values above 64 that are mapped to "large" are encoded from zero upwards. Another difference in this example from E.2.2 is that "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.

E.2.4 Compression of non-continuous value ranges

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 exa mple, 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.

The ASN.1 assignment is:

```
SparseEvenlyDistributedValueSet ::= INTEGER (0 | 2 | 4 | 6 | 8 | 10 | 12 | 14)
```

The requirement: 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.

The encoding object assignment is:

```
sparseEvenlyDistributedValueSetEncoding #SparseEvenlyDistributedValueSet ::= {
        USE     #INT (0..7)
                MAPPING ORDERED VALUES
        WITH    { ENCODE WITH PER-basic-unaligned }}
```

The eight possible abstract values have been mapped to the range 0..7 and will be encoded in a field of three bits.

E.2.5 Compression of non-continuous value ranges using a transformation

The example E.2.4 used mapping of ordered abstract values. The same effect can be achieved by using the #TRANSFORM class.

The encoding object assignment is:

```
sparseEvenlyDistributedValueSetEncoding-2 #SparseEvenlyDistributedValueSet ::= {
        USE     #INT (0..7)
                MAPPING TRANSFORMS { {INT-TO-INT divide: 2} }
        WITH    { ENCODE WITH PER-basic-unaligned }}
```

Again, the eight possible abstract values are mapped to the range 0..7 and encoded in a field of three bits.

E.2.6 Compression of an unevenly distributed value set by mapping ordered abstract values

The ASN.1 assignment is:

```
SparseUnevenlyDistributedValueSet ::= INTEGER (0|3|5|6|8|11)
```

The requirement is that the encoding should be such that there are no holes in the encoding patterns used.

The encoding object assignment is:

```
sparseUnevenlyDistributedValueSetEncoding #SparseUnevenlyDistributedValueSet ::= {
        USE     #INT (0..5)
                MAPPING ORDERED VALUES
        WITH    { ENCODE WITH PER-basic-unaligned }}
```

The six possible abstract values are mapped to the range 0..5 and encoded in a field of three bits. The mapping is as follows: 0? 0 , 3? 1, 5 ? 2, 6? 3, 8? 4, and 11? 5.

E.2.7 An optional component's presence depends on the value of another component

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
-- 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.
```

Requirement: The component "a" acts as the presence determinant for both of 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.

The encoding object assignment is:

```
conditionalPresenceOnValueEncoding #ConditionalPresenceOnValue ::= {
        ENCODE STRUCTURE {
            c   OPTIONAL-ENCODING is-c-present{<  a  >},
            d   OPTIONAL-ENCODING is-d-present{<  a  >}
        }
        WITH {ENCODE WITH PER-basic-unaligned }}

is-c-present {<  REFERENCE : a  >} #OPTIONAL ::= {
    ENCODING-SPACE
        AS      app-determinant
        DETERMINED BY       a
            DECODER-TRANSFORMS { {INT-TO-BOOL  TRUE-IS { 0 } } } }

is-d-present {< REFERENCE : a >} #OPTIONAL ::= {
    ENCODING-SPACE
        AS      app-determinant
            DETERMINED BY       a
                DECODER-TRANSFORMS { {INT-TO-BOOL TRUE-IS {0|2|3|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.

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 - 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.

E.2.8 The presence of an optional component depends on some external condition

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.
```

Requirement: The application code for both a sender and a receiver can evaluate the condition "C" from some information outside the message. We wish tools to invoke such code to determine the presence of "a", rather than using a bit in the encoding.

The encoding object assignment is:

```
conditionalPresenceOnExternalConditionEncoding #ConditionalPresenceOnExternalCondition ::= {
        ENCODE STRUCTURE {
            a   OPTIONAL-ENCODING is-a-present,
        }
        WITH { ENCODE WITH PER-basic-unaligned }}

is-a-present #OPTIONAL ::=
        USER-FUNCTION-BEGIN
        /* "a" is present if the external condition "C" is true, otherwise absent. */
        is_a_present = (channel == C);
        USER-FUNCTION-END
```

Because the condition is external to the message, the encoding object for determining presence of the component "a" can only be specified as a user-defined function. However, whilst this saves on bit 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.

E.2.9 A variable length list

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.
```

The requirement: PER basic unaligned encodes the length of the list using 10 bits even if normally the length is in the range 0..31. The requirement is to minimize the size of the encoding of the length determinant in the normal case whilst still allowing values which rarely occur.

The encoding object assignment is:

```
enclosingStructureEncoding #EnclosingStructureForList ::= {
        USE     #EnclosingStructureForListStruct
                MAPPING FIELDS
        WITH {
            ENCODE STRUCTURE {
                aux-length  list-lengthEncoding,
                list  {
                    ENCODING {
                        ENCODING-SPACE AS aux-determinant
                        DETERMINED BY aux-length} } } }
        -- First mapping: use an encoding structure with an explicit length determinant.

list-lengthEncoding #AuxVariableListLength ::= {
        USE     #AuxVariableListLengthStruct
                MAPPING ORDERED VALUES
        WITH    { ENCODE WITH PER-basic-unaligned }}
        -- Second mapping: list length is encoded as a choice between a short form "normally"
        -- and a long form "sometimes".
```

The encoding structure assignments are:

```
#EnclosingStructureForListStruct ::= #CONCATENATION {
        aux-length  #AuxVariableListLength,
        list        #VariableLengtList }

#AuxVariableListLength ::= #INT (0..1023)

#AuxVariableListLengthStruct ::= #ALTERNATIVES {
        normally    #INT (0..31),
        sometimes   #INT (32..1023) }
```

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 ength determinant for long list values is encoded using 1 bit for the selection determinant and 10 bits for the length determinant.

E.2.10 Equal length lists

The ASN.1 assignment is:

```
EqualLengthLists ::= SEQUENCE {
        list1   List1,
        list2   List2 }
        -- "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)
```

The requirement is: "list1" and "list2" have the same number of elements, and we wish to use a single length determinant for both lists. (PER would encode length fields for both components.

The encoding object assignment is:

```
equalLengthListsEncoding #EqualLengthLists ::= {
        USE #EqualLengthListsStruct
        MAPPING FIELDS
        WITH {
            ENCODE STRUCTURE {
                list1   list-with-determinantEncoding{< aux-length >},
```

```
            list2   list-with-determinantEncoding{< aux-length >} }
        WITH PER-basic-unaligned } }

list-with-determinantEncoding {< REFERENCE : length-determinant >} ::= #REPETITION {
        ENCODING {
            ENCODING-SPACE AS   aux-determinant
            DETERMINED BY   length-determinant } }
```

The encoding structure assignments are:

```
#EqualLengthListsStruct ::= #CONCATENATION {
        aux-length  #AuxListLength,
        list1       #List1,
        list2       #List2 }

#AuxListLength ::= #INT (0..1023)
```

E.2.11 Uneven choice alternative probabilities

The ASN.1 assignment is:

```
EnclosingStructureForChoice ::= SEQUENCE {
        cho     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) }
```

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.

The encoding structure assignments are:

```
#EnclosingStructureForChoiceStruct ::= #CONCATENATION {
        aux-selector    #AuxSelector,
        cho     UnevenChoiceProbability }
-- Explicit auxiliary alternative determinant for "cho".

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

The encoding object assignment is:

```
enclosingStructureForChoiceEncoding #EnclosingStructureForChoice ::= {
        USE     #EnclosingStructureForChoiceStruct
                MAPPING FIELDS
        WITH {
                ENCODE STRUCTURE {
                    aux-selector    auxSelectorEncoding,
                    cho {
                        SELECTION AS        aux-determinant
                        DETERMINED BY   aux-selector } }
                        WITH { ENCODE WITH PER-basic-unaligned} } }
        -- First mapping: insert an explicit auxiliary alternative determinant.
        -- This encoding object specifies that an auxiliary determinant is used as
        -- 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"
    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 )
    }-- Second mapping: Map determinant indexes to bit strings
```

In the above, we quantified "frequent" and "common" and "rare" as 60%, 30%, and 10%, and used the public domain ECN Huffman generator (see Annex F) at the web-site cited in Annex G to determine the optimal bit-patterns to be used for each category.

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 (provided tools are used) it costs nothing in implementation effort to produce and use optimal encodings, the ultimate gains may not be significant.

E.2.12 A version 1 message

ASN.1 assignment:

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

Requirement: 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.

We use two encoding structures to encode the message: one is the implicit 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 }
```

The encoding object assignments are then:

```
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
            }
        {ENCODE WITH PER-basic-unaligned } }
additionsEncoding #PAD ::= {
    ENCODING-SPACE
        SIZE        uses-determination-mechanism
            DETERMINED BY container
            CONTAINED IN end-of-encoding:NULL
    VALUE PADDING encoder-option }
```

E.2.13 ELM definitions

The following ELM is associated with the ASN.1 module defined in E.2.14, and the EDM defined in E.2.15.

```
Example2-ELM

{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(5) }

LINK-DEFINITIONS ::=
BEGIN
```

```
        IMPORTS Example2Encodings FROM Example-EDM
        { joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(7) };

        ENCODE Example2-ASN1-Module
        { joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(6) }
                WITH Example2Encodings
                COMPLETED BY PER-basic-unaligned

END
```

E.2.14 ASN.1 definitions

This module groups together all the ASN.1 definitions from E.2.1 to E.2.12 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(1) asn1(1) ecn(4) examples(5) module(6) }

DEFINITIONS AUTOMATIC TAGS : :=
BEGIN
NormallySmallValues ::= INTEGER (0..1024)

-- etc

END
```

E.2.15 EDM definitions

```
Example2-EDM ENCODING-DEFINITIONS ::=
{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(7) }
BEGIN
EXPORTS Example2Encodings;
IMPORTS #NormallySmallValues,#SparseEvenlyDistributedValueSet,
        #SparseUnevenlyDistributedValueSet,#ConditionalPresenceOnValueSet,
        #ConditionalPresenceOnExternalCondition,#ConditionalComponentValueSet,
        #EnclosureStructureForList,#EqualLengthLists,#EnclosingStructureForChoice,
        #ExtensibleInteger,#ExtensibleEnumerated,#ExtensibleTopLevelMessage,
        #ExtensibleMessage
        FROM Example1-ASN1-Module

{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(6) };

Example2Encodings #ENCODINGS ::= {
            normallySmallValuesEncoding     |
            -- etc
            extensibleMessageEncoding}
-- etc

END
```

E.3 Legacy protocol example

E.3.1 Introduction

The purpose of the example in this clause is to show how to construct ECN definitions for a protocol whose message encodings have been specified using tabular notation. The following tables contain the contents of the messages (only Message1 has been shown completely):

Message 1:

	8	**7**	**6**	**5**	**4**	**3**	**2**	**1**
Octet 1	Message id							
Octet 2	a			b-flag	c-len			reserved
Octet X	b1		b2	reserved	b3		reserved	
Octet Y	c1				c2			
Octet Y+1	c3						reserved	
Octet Z	d1	d2			d3			reserved

Message 2:

	8	7	6	5	4	3	2	1
Octet 1	Message id							
Octet M	Something							

Message 3:

	8	7	6	5	4	3	2	1
Octet 1	Message id							
Octet M	something							

All the messages have a common heading part (shown in gray in the tables). In this example it is used only for message identification.

Message 1 has three kinds of fields:

- mandatory fields ("a")
- mandatory fields that are determinants for other fields ("b-flag", "c-len")
- optional fields ("b", "c", and "d")

The fields "b", "c" and "d" are composed of sub-fields. In addition fields "c" and "d" may appear multiple times. Optional fields are octet-aligned.

Presence of an optional component is indicated using different methods

- the field "b" is present if the value of the "b-flag" field is 1.
- the field "c" is present if the value of the "c-len" field is greater than 0
- the field "d" is present if there are octets left in the message

The length of a field that can appear multiple times is determined using different methods:

- multiplicity of the field "c" is governed by the determinant field "c-len"
- multiplicity of the field "d" is determined by the end of message

The following ASN.1 module contains definitions for the message structures presented above. The following design decisions have been made:

- there is one encapsulating type which contains the common definitions for all the messages
- auxiliary determinant fields in messages are visible at the ASN.1 level. Note, this is done for simplicity of exposition in this example, but it should be normal practice to keep such fields out of the ASN.1 definition unless they carry real application semantics.
- extensibility is expressed in the form of comments.
- padding is not visible

The ASN.1 module is:

```
LegacyProtocol-ASN1-Module

{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(8) }

DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

LegacyProtocolMessages ::= SEQUENCE {
        message-id  ENUMERATED { message1, message2, message3 },
        messages    CHOICE {
            message1    Message1,
            message2    Message2,
            message3    Message3 } }
        -- The CHOICE is constrained by the value of message-id.
```

```
Message1 ::= SEQUENCE {
        a       A,
        b-flag  BOOLEAN,
        c-len   INTEGER (0..max-c-len),
        b   B   OPTIONAL,   -- determined by "b-flag"
        c   C   OPTIONAL,   -- determined by "c-len"
        d   D   OPTIONAL }  -- determined by end of PDU

A ::= INTEGER (0..7)    -- Values 5..7 are reserved for future use.  Version 1 systems should

B ::= SEQUENCE {
        b1  ENUMERATED { e0, e1, e2, e3 },
        b2  BOOLEAN,
        b3  INTEGER (0..3) }

C ::= SEQUENCE (SIZE (1..max-c-len)) OF C-elem

C-elem ::= SEQUENCE {
        c1  BIT STRING (SIZE (4)),
        c2  INTEGER (0..1024) }

D ::= SEQUENCE (SIZE (0..max-d-len)) OF D-elem

D-elem ::= SEQUENCE {
        d1  BOOLEAN,
        d2  ENUMERATED { f0, f1, f2, f3, f4, f5, f6, f7 },
        d3  INTEGER (0..7) }

max-c-len INTEGER ::= 7

max-d-len INTEGER ::= 20

Message2 ::= SEQUENCE {
        -- something -- }

Message3 ::= SEQUENCE {
        -- something -- }

END
```

The EDM module in E.3.1 contains encoding definitions for the messages specified in the "LegacyProtocol-ASN1-Module" ASN.1 module. The following design decisions have been made:

- padding within octets is explicitly specified as padding fields
- alignment padding is not specified as explicit padding fields

E.3.2 Encoding definition for the top-level message structure

The encoding object "legacyProtocolMessagesEncoding" specifies how the common parts of the legacy protocol messages are encoded. The message identifier is specified in ASN.1 as an enumerated type. PER basic unaligned encodes "message-id" using the minimum number of bits, i.e. 2, but here we would like to have it encoded using 8 bits. In addition, we have to specify that "message-id" is to be used as a determinant for "messages".

The "legacyProtocolMessagesEncoding" is:

```
legacyProtocolMessagesEncoding #LegacyProtocolMessages ::= {
        ENCODE STRUCTURE {
            message-id  int8Encoding,
            messages {
                        SELECTION AS        app-determinant
                        DETERMINED BY   message-id } }
        WITH { ENCODE WITH PER-basic-unaligned } }

int8Encoding #INT ::= {
        USE #INT (0..255)
        MAPPING VALUES
        WITH    { ENCODE WITH PER-basic-unaligned } }
```

E.3.3 Encoding definition for a message structure

The encoding object "message1Encoding" specifies how values of "Message1" are to be encoded:

- The field "b" is present if the field "b-flag" contains value 1.

- The field "c" is present if the field "c-len" does not contain value 0. "c-len" also governs the number of elements in "c".
- The field "d" is present if there are still octets in an encoding for the message.

The encoding object for "Message1" is:

```
message1Encoding #Message1 ::= {
        ENCODE STRUCTURE {
            b   b-encoding
                OPTIONAL-ENCODING {
                    ENCODING-SPACE AS   app-determinant
                    DETERMINED BY   b-flag },
            c   octet-aligned-seq-of-with-ext-determinant{< c-len >}
                OPTIONAL-ENCODING {
                    ENCODING-SPACE AS   app-determinant
                    DETERMINED BY   c-len
                        DECODER-TRANSFORMS { is-c-present } },
            d   octet-aligned-seq-of-until-end-of-container
                OPTIONAL-ENCODING {
                    ENCODING-SPACE AS   container } }
        WITH { ENCODE WITH PER-basic-unaligned } }

is-c-present #TRANSFORM ::= {
        INT-TO-BOOL         FALSE-IS {0} }
```

E.3.4 Encoding for the sequence type "B"

Padding of one bit is inserted between the fields "b2" and "b3" ("aux-reserved"). The encoding of "B" is octet-aligned.

The encoding for "B" is:

```
b-encoding #B ::= {
        USE     #BStruct
                MAPPING FIELDS
        WITH    {
                ENCODE STRUCTURE {
                        aux-reserved        reserved-1-bitEncoding
                        STRUCTURED WITH     octet-aligned-seqEncoding } } }

#BStruct ::= #CONCATENATION {
        b1      #INT (0..3),
        b2      #BOOL,
        aux-reserved    #PAD,
        b3      #INT (0..3) }
reserved-1-bitEncoding  #PAD ::= {
        ENCODING-SPACE
                SIZE        1
        VALUE
                PADDING zero }
```

E.3.5 Encoding the octet-aligned sequence type for the legacy protocol

The encoding is to be octet-aligned. Any needed padding uses 0 as filler bits.

The encoding object definition for the #SEQUENCE constructor is:

```
octet-aligned-seqEncoding #SEQUENCE ::= {
        ENCODING-SPACE
            ALIGNED TO  octet
            PADDING     zero }
```

E.3.6 Encoding for an octet-aligned sequence-of type with a length determinant

One of the sequence-of types used in the legacy protocol has an explicit length determinant.

The encoding is octet-aligned. The number of elements count is determined by the field "len".

```
octet-aligned-seq-of-with-ext-determinant{< REFERENCE : len >} #SEQUENCE-OF ::= {
        ENCODING-SPACE
            ALIGNED TO  octet
            PADDING zero
```

```
        ENCODING {
            ENCODING-SPACE
                AS          app-determinant
                DETERMINED BY   len } }
```

E.3.7 Encoding for an octet-aligned sequence-of type which continues to the end of the PDU

The encoding is octet-aligned. The number of elements is determined by the end of the PDU.

The encoding object is:

```
octet-aligned-seq-of-until-end-of-container #SEQUENCE-OF ::= {
        ENCODING-SPACE
            ALIGNED TO  octet
            PADDING zero
        ENCODING {
            ENCODING-SPACE
                AS          container
                CONTAINED IN        end-of-container: NULL } }
```

E.3.8 ELM definitions

The ELM for the legacy protocol is:

```
LegacyProtocol-ELM-Module

{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(10) }

 LINK-DEFINITIONS ::=
BEGIN
IMPORTS
        LegacyProtocolEncodings;
FROM    LegacyProtocol-EDM-Module
{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(8) };
ENCODE LegacyProtocol-ASN1-Module.LegacyProtocolMessages
        WITH            LegacyProtocolEncodings
        COMPLETED WITH  PER-basic-unaligned
END
```

E.3.9 EDM definitions

The EDM definitions are:

```
LegacyProtocol-EDM-Module
{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(8) }
ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS
        LegacyProtocolEncodings;
IMPORTS
        #LegacyProtocolMessages
FROM    LegacyProtocol-ASN1-Module
{ joint-iso-itu-t(1) asn1(1) ecn(4) examples(5) module(7) };
LegacyProtocolEncodings #ENCODINGS ::= {
        legacyProtocolMessagesEncoding  |
        message1Encoding }

-- etc

END
```

Annex F (Informative): Support for Huffman encodings

Huffman encodings are the optimum encodings for a finite set of integer values, where the frequency with which each value will be transmitted is known.

The encodings are self-delimiting (no length-determinant is needed) and use a small number of bits for frequent values and a larger number of bits for less frequent values.

There are many possible Huffman encodings. For example, given any such encoding, simply change all "1"s to "0"s and vice versa, and you have a different (but just as efficient) Huffman encoding. More subtle changes can also be made to produce other Huffman encodings that are equally efficient.

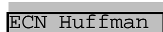
For Huffman encodings to be efficient for decoders, it is desirable that where successive integer values encode into the same number of bits, those bits should define successive integer values when interpreted as a positive integer encoding.

An ECN Huffman encoding has been defined that has this property, and a Microsoft Word 97 macro has been produced that will generate the syntax for a "MappingIntToBits" mapping which is both optimal and easy to decode.

A version of this Annex is available which contains a macro button that will take a specification of the integer values to be encoded and their frequency, and will generate in-line the formal mapping specification conforming to the ECN notation. (The version of this Annex with the associated macro can be obtained from the Web site cited in Annex G).

The following text contains three examples of ECN Huffman specification.

In the version with the macro, double clicking the button below:



will add the ECN Huffman mapping specifications to the text.

The user of the version with the macro may wish to modify the specification of the values to be mapped and their frequencies to see the encodings that are produced in different cases.

NOTE In the version with macros, once encoding specifications have been produced, they can be deleted, the ECN Huffman specification changed, and the macro button again clicked.

The informal syntax for an ECN Huffman specification should be clear from the following examples. All lines start with an ASN.1 comment marker ("--").

The first line (if the macro is to be used) must contain exactly "ECN Huffman" preceded by two hyphens and a space, but following lines are not case sensitive and may contain more or less spaces.

The second line is required, and specifies the lowest and highest values that are to be mapped. The range (upper bound minus lower bound) is limited to 1000, but can include negative values. Not all values in the range need to be mapped.

Percentages are given for either single values or for ranges of values. It is not necessary for percentages to add up to 100%, but a warning is given if they do not.

The "REST" line is optional, and provides frequencies for any values in the range not explicitly listed. If missing, then the mapped values will only be those explicitly specified.

The final line is mandatory, and must contain "End Definition" (in upper or lower case). The formal ECN encoding specification is inserted (by the macro) after this line.

The first example is:

```
my-int-encoding1 #My-Special-1 ::=
{ USE  #BITS
-- ECN Huffman
-- RANGE (-1..10)
-- -1 IS 20%
-- 1 IS 25%
-- 0 IS 15%
-- (3..6) IS 10%
-- Rest IS 2%
-- End Definition
```

```
-- Mappings produced by "ECN Public Domain Software for Huffman encodings, version 1"
        MAPPING TO BITS {
        -1 TO '11'B ,
        (  0 ..  1 ) TO (  '01'B .. '10'B ) ,
         2 TO '0000001'B ,
        (  3 ..  5 ) TO (  '0001'B .. '0011'B ) ,
         6 TO '00001'B ,
        (  7 ..  8 ) TO (  '0000010'B .. '0000011'B ) ,
        (  9 ..  10 ) TO (  '00000000'B .. '00000001'B )
        }
WITH  my-self-delim-bits-encoding }
```

The second example is:

```
my-int-encoding2 #My-Special-2 ::=
{ USE  #BITS
-- ECN Huffman
-- RANGE (-10..10)
-- -10 IS 20%
-- 1 IS 25%
-- 5 IS 15%
-- (7..10) is 10%
-- End Definition
-- Mappings produced by "ECN Public Domain Software for Huffman encodings, version 1"
        MAPPING TO BITS {
        -10 TO '11'B ,
         1 TO '10'B ,
         5 TO '01'B ,
        (  7 ..  10 ) TO (  '0000'B .. '0011'B )
        }
WITH my-self-delim-bits-encoding }
```

The third example is:

```
my-int-encoding3 #My-Special-3 ::=
{ USE  #BITS
-- ECN Huffman
-- RANGE  (0..1000)
-- (0..63) IS 100%
-- REST IS 0%
-- End Definition
-- Mappings produced by "ECN Public Domain Software for Huffman encodings"
        MAPPING TO BITS {
        (  0 ..  62 ) TO (  '000001'B .. '111111'B ) ,
         63 TO '0000001'B ,
        (  64 ..  150 ) TO (  '0000000110101001'B .. '0000000111111111'B ) ,
        (  151 ..  1000 ) TO (  '0000000000000000'B .. '0000000110101000 1'B )
        }
WITH my-self-delim-bits-encoding }
```

Annex G: (Informative) Additional Information on the Encoding Control Notation (ECN)

Additional information and links on the Encoding Control Notation can be found on the following Web site:

http://asn1.elibel.tm.fr/ecn

Annex H (Informative): Summary of the ECN notation

H.1 Terminal symbols

The following terminal symbols are used in this TS

The following items are defined in clause 8:

```
anystringexceptuserfunctionend
encodingobjectreference
encodingobjectsetreference
encodingclassreference
"::="
".."
"{"
"}"
"("
")"
","
"."
"|"
AS
BEGIN
BER
BITS
BY
CER
COMPLETED
DECODE
DER
DISTRIBUTION
ENCODE
ENCODING-CLASS
ENCODE-DECODE
ENCODING-DEFINITIONS
END
FIELDS
IF
LINK-DEFINITIONS
MAPPING
MAX
MIN
OPTIONAL-ENCODING
ORDERED
OUTER
PER-basic-aligned
PER-basic-unaligned
PER-canonical-aligned
PER-canonical-unaligned
REMAINDER
SIZE
STRUCTURE
STRUCTURED
TO
TRANSFORMS
UNION
USE
USER-FUNCTION-BEGIN
USER-FUNCTION-END
VALUES
WITH
```

The following item is defined in Annex B:

```
REFERENCE
```

The following items are defined in ITU-T Rec. X.680 | ISO/IEC 8824-1:

```
bstring
cstring
hstring
identifier
modulereference
number
typereference
"-"
";"
":"
ALL
EXCEPT
EXPORTS
FALSE
FROM
IMPORTS
MINUS-INFINITY
NULL
PLUS-INFINITY
TRUE
```

The following items are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 :

```
word
valuefieldreference
valuesetfieldreference
```

The following items are defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

```
"{<"
">}"
```

H.2 Productions

The following productions are used in this TS, with the items defined in H.1 as terminal symbols:

```
ELMDefinition ::=
    ModuleIdentifier
    LINK-DEFINITIONS
    "::="
    BEGIN
    ELMModuleBody
    END

ELMModuleBody ::=
    Imports ?
    EncodingApplicationList

EncodingApplicationList ::=
    EncodingApplication
    EncodingApplicationList ?

EncodingApplication ::=
    TypeApplication             |
    ModuleApplication

TypeApplication ::=
    ENCODE
    AbsoluteReference
    CombinedEncodings

ModuleApplication ::=
    ENCODE
    ModuleIdentifier
    CombinedEncodings

CombinedEncodings ::=
    WITH
    PrimaryEncodings
    CompletionClause ?

CompletionClause ::=
    COMPLETED BY
    SecondaryEncodings

PrimaryEncodings ::= EncodingObjectSet

SecondaryEncodings ::= EncodingObjectSet

EDMDefinition ::=
    ModuleIdentifier
    ENCODING-DEFINITIONS
    "::="
    BEGIN
    EDMModuleBody
    END

EDMModuleBody ::=
    Exports ?
    Imports ?
    EDMAssignmentList ?

EDMAssignmentList ::=
    EDMAssignment
    EDMAssignmentList ?
```

```
EDMAssignment ::=
EncodingClassAssignment                     |
    EncodingStructureAssignment             |
    EncodingObjectAssignment                |
    EncodingObjectSetAssignment             |
    ParameterizedAssignment

EncodingClassAssignment ::=
    encodingclassreference
    "::="
    EncodingClass

EncodingClass ::=
    DefinedEncodingClass     |
    EncodingStructure

EncodingStructureAssignment ::=
    encodingclassreference
    "::="
    EncodingStructure

EncodingObjectAssignment ::=
    encodingobjectreference
    EncodingClass
    "::="
    EncodingObject

EncodingObjectSetAssignment ::=
    encodingobjectsetreference
    #ENCODINGS
    "::="
    EncodingObjectSet

EncodingStructure ::=
    DefinedEncodingClass         |
    EncodingStructureDefn

EncodingStructureDefn ::=
    AlternativesStructure        |
    RepetitionStructure          |
    ConcatenationStructure

AlternativeStructure ::=
    AlternativesClass
    "{"
    NamedFields
    "}"

AlternativesClass ::=
    DefinedEncodingClass     |
    AlternativesClassReference

NamedFields ::= NamedField "," +

NamedField ::=
    identifier
    EncodingStructure

RepetitionStructure ::=
    RepetitionClass
    "{"
    EncodingStructure
    "}"
    Size?

RepetitionClass ::=
    DefinedEncodingClass         |
    RepetitionClassReference

ConcatenationStructure ::=
    ConcatenationClass
    "{"
    ConcatComponents
    "}"

ConcatenationClass ::=
    DefinedEncodingClass
    ConcatenationClassReference
```

```
ConcatenationClassReference ::=
    #CONCATENATION  |
    #SEQUENCE       |
    #SET

ConcatComponents ::=
    ConcatComponent "," *

ConcatComponent ::=
    NamedField
    OptionalClass ?

OptionalClass ::=
    DefinedEncodingClass
    OptionalityClassReference

DefinedEncodingClass ::=
    encodingclassreference      |
    ExternalEncodingClassReference  |
    BuiltinEncodingClassReference   |
    ParameterizedEncodingClass

DefinedEncodingObject ::=
    encodingobjectreference     |
    ExternalEncodingObject      |
    ParameterizedEncodingObject

DefinedEncodingObjectSet ::=
    encodingobjectsetreference          |
ExternalEncodingObjectSetReference      |
BuiltinEncodingObjectSetReference       |
ParameterizedEncodingObjectSet

BuiltinEncodingObjectSetReference ::=
    PER-basic-aligned               |
    PER-basic-unaligned             |
    PER-canonical-aligned           |
    PER-canonical-unaligned         |
    BER                             |
    CER                             |
    DER

ExternalEncodingClassReference ::=
    modulereference "." encodingclassreference |
    modulereference "." BuiltinEncodingClassReference

ExternalEncodingObjectReference ::=
    modulereference
    "."
    encodingobjectreference

ExternalEncodingObjectSetReference ::=
    modulereference
    "."
    encodingobjectsetreference

EncodingObjectSet ::=
    DefinedEncodingObjectSet |
    EncodingObjectSetSpec

EncodingObjectSetSpec ::=
    "{"
    EncodingObjects UnionMark *
    "}"

EncodingObjects ::=
    EncodingObject      |
    DefinedEncodingObjectSet

UnionMark ::=
    "|" |
    UNION

EncodingObject ::=
    DefinedEncodingObject
    DefinedSyntax               |
    EncodeWith                  |
```

```
    EncodeByValueMapping          |
    EncodeStructure               |
DifferentialEncodeDecodeObject   |
    UserDefinedEncodingFunction

EncodeWith ::= "{" ENCODE CombinedEncodings "}"

EncodeByValueMapping ::=
    "{"
    USE
    EncodingClass
    MAPPING
    ValueMapping
    WITH
    EncodingObject
    "}"

DifferentialEncodeDecodeObject ::=
        {"
    ENCODE-DECODE
    SpecForEncoding
    DECODE AS IF
    SpecForDecoders
    "}"

SpecForEncoding ::= EncodingObject

SpecForDecoders ::= EncodingObject

UserDefinedEncodingFunction ::=
    USER-FUNCTION-BEGIN
    AssignedIdentifier
    anystringexceptuserfunctionend
    USER-FUNCTION-END

EncodeStructure ::=
    "{"
    ENCODE STRUCTURE
    "{"
    ComponentEncodingList
    StructureEncoding ?
    "}"
    CombinedEncodings ?
    "}"

StructureEncoding ::=
    STRUCTURED WITH
    EncodingObject

ComponentEncodingList ::=
    ComponentEncoding "," *

ComponentEncoding ::=
    NonOptionalComponentEncodingSpec
    OptionalComponentEncodingSpec

NonOptionalComponentEncodingSpec ::=
    identifier ?
    EncodingObject1

OptionalComponentEncodingSpec ::=
    ComponentEncodingObject          |
    OptionalEncodingObject           |
    ComponentAndOptionalEncodingObject

ComponentEncodingObject ::=
    identifier
    EncodingObject1

OptionalEncodingObject ::=
    identifier
    OPTIONAL-ENCODING
    EncodingObject2

ComponentAndOptionalEncodingObject ::=
    identifier
    EncodingObject1
    OPTIONAL-ENCODING
```

```
    EncodingObject2

EncodingObject1 ::= EncodingObject
EncodingObject2 ::= EncodingObject

BuiltinEncodingClassReference ::=
    BitfieldClassReference          |
    AlternativesClassReference      |
    ConcatenationClassReference     |
    RepetitionClassReference        |
    OptionalityClassReference       |
    GeneralProcedureClassReference

BitfieldClassReference ::=
    #NUL                    |
    #BOOL                   |
    #INT                    |
    #BITS                   |
    #OCTETS                 |
    #CHARS                  |
    #PAD                    |
    #BIT-STRING             |
    #BOOLEAN                |
    #CHARACTER-STRING       |
    #EMBEDDED-PDV           |
    #ENUMERATED             |
    #EXTERNAL               |
    #INTEGER                |
    #NULL                   |
    #OBJECT-IDENTIFIER      |
    #OCTET-STRING           |
    #OPEN-TYPE              |
    #REAL                   |
    #RELATIVE-OID           |
    #GeneralizedTime        |
    #UTCTime                |
    #BMPString              |
    #GeneralString          |
    #IA5String              |
    #NumericString          |
    #PrintableString        |
    #TeletexString          |
    #UniversalString        |
    #UTF8String             |
    #VideotexString         |
    #VisibleString

OptionalityClassReference ::=
    #OPTIONAL

GeneralProcedureClassReference ::=
    #TRANSFORM              |
    #CONDITIONAL-INT        |
    #CONDITIONAL-REPETITION |
    #OUTER

EncodingStructureField ::=
    PrimitiveField      |
    ComplexField

PrimitiveField ::=
    #NUL                            |
    #BOOL                           |
    #INT        Bounds?             |
    #BITS       Size?               |
    #OCTETS     Size?               |
    #CHARS          Size?           |
    #PAD

ComplexField ::=
    #BIT-STRING     Size?           |
    #BOOLEAN                        |
    #CHARACTER-STRING               |
    #EMBEDDED-PDV                   |
    #ENUMERATED Bounds?             |
    #EXTERNAL                       |
    #INTEGER    Bounds?             |
    #NULL                           |
```

```
    #OBJECT-IDENTIFIER                  |
    #OCTET-STRING   Size?               |
    #OPEN-TYPE                          |
    #REAL                               |
    #RELATIVE-OID                       |
    #GeneralizedTime                    |
    #UTCTime                            |
    #BMPString  Size?                   |
    #GeneralString  Size?               |
    #GraphicString  Size?               |
    #IA5String  Size?                   |
    #NumericString  Size?               |
    #PrintableString    Size?           |
    #TeletexString  Size?               |
    #UniversalString    Size?           |
    #UTF8String Size?                   |
    #VideotexString Size?               |
    #VisibleString  Size?

Bounds ::= "(" EffectiveRange ")"

EffectiveRange ::=
    MinMax  |
    Fixed

Size ::= "(" SIZE SizeEffectiveRange ")"

SizeEffectiveRange ::=
    "(" EffectiveRange ")"  |
    EffectiveRange

MinMax ::=
    ValueOrMin
    ".."
    ValueOrMax

ValueOrMin  ::=
    SignedNumber    |
    MIN
ValueOrMax  ::=
    SignedNumber    |
    MAX
Fixed ::= SignedNumber

ValueMapping ::=
    MappingByExplicitValues             |
    MappingByMatchingFields             |
    MappingByTransformEncodingObjects   |
    MappingByAbstractValueOrdering      |
    MappingByValueDistribution          |
    MappingIntToBits

MappingByExplicitValues ::=
    VALUES
    "{"
    MappedValues "," +
    "}"

MappedValues ::=
    MappedValue1
    TO
    MappedValue2

MappedValue1 ::= Value
MappedValue2 ::= Value

MappingByMatchingFields ::= FIELDS

MappingByTransformEncodingObjects ::=
    TRANSFORMS
    "{"
    TransformList
    "}"

TransformList ::=
    Transform "," +

Transform ::= EncodingObject
```

```
MappingByAbstractValueOrdering ::=
    ORDERED VALUES

MappingByValueDistribution ::=
    DISTRIBUTION
    "{"
    Distribution "," +
    "}"

Distribution ::=
    SelectedValues
    TO
    identifier

SelectedValues ::=
    SelectedValue          |
    DistributionRange      |
    REMAINDER

DistributionRange ::=
    DistributionRangeValue1
    ".."
    DistributionRangeValue2

SelectedValue ::= SignedNumber

DistributionRangeValue1 ::= SignedNumber
DistributionRangeValue2 ::= SignedNumber

MappingIntToBits ::=
    TO BITS
    "{"
    MappedIntToBits "," +
    "}"

MappedIntToBits ::=
    SingleIntValMap
    IntValRangeMap

SingleIntValMap ::=
    IntValue
    TO
    BitValue

IntValue ::= SignedNumber

BitValue ::=
    bstring |
    hstring

IntValRangeMap ::=
    IntRange
    TO
    BitRange

IntRange ::=
    IntRangeValue1
    ".."
    IntRangeValue2

BitRange ::=
    BitRangeValue1
    ".."
    BitRangeValue2

IntRangeValue1 ::= SignedNumber
IntRangeValue2 ::= SignedNumber
BitRangeValue1 ::=
    bstring |
    hstring
BitRangeValue2 ::=
    bstring |
    hstring
```

The following productions are defined ITU-T Rec. X.680 | ISO/IEC 8824-1, as modified by Annex B, with the items defined in H.1 as terminal symbols:

NOTE Struck productions are not allowed in ECN.

```
ModuleIdentifier ::=
    modulereference
    DefinitiveIdentifier ?

DefinitiveIdentifier ::=
    "{" DefinitiveObjIdComponentList "}"

DefinitiveObjIdComponentList ::=
    DefinitiveObjIdComponent                        |
    DefinitiveObjIdComponent DefinitiveObjIdComponentList

DefinitiveObjIdComponent ::=
    NameForm              |
    DefinitiveNumberForm        |
    DefinitiveNameAndNumberForm

NameForm ::= identifier

DefinitiveNumberForm ::= number

DefinitiveNameAndNumberForm ::= identifier "(" DefinitiveNumberForm ")"

Exports ::= EXPORTS SymbolsExported? ";"

SymbolsExported ::= SymbolList

Imports ::= IMPORTS SymbolsImported? ";"

SymbolsImported ::= SymbolsFromModuleList

SymbolsFromModuleList ::=
    SymbolsFromModule               |
    SymbolsFromModuleList  SymbolsFromModule

SymbolsFromModule ::=
    SymbolList
    FROM
    GlobalModuleReference

GlobalModuleReference ::=
    modulereference AssignedIdentifier

AssignedIdentifier ::= DefinitiveIdentifier

SymbolList ::=
    Symbol          |
    SymbolList "," Symbol

Symbol ::=
    Reference       |
    ParameterizedReference

Reference ::=
    valuereference              |
    typereference               |
    identifier                  |
    encodingclassreference      |
    encodingobjectreference     |
    encodingobjectsetreference

AbsoluteReference ::=
    ModuleIdentifier
    "."
    ItemSpec

ItemSpec ::=
    typereference        |
    ItemId "." ComponentId

ItemId ::= ItemSpec

ComponentId ::=
    identifier                |
```

```
    "*"

Value ::=
    BuiltinValue

BuiltinValue ::=
    BitStringValue          |
    BooleanValue            |
    CharacterStringValue    |
    ChoiceValue             |
    EmbeddedPDVValue        |
    EnumeratedValue         |
    ExternalValue           |
    InstanceOfValue         |
    IntegerValue            |
    NullValue               |
    ObjectIdentifierValue   |
    OctetStringValue        |
    RealValue               |
    SequenceValue           |
    SequenceOfValue         |
    SetValue                |
    SetOfValue

BitStringValue ::=
    bstring         |
    hstring

BooleanValue ::=
    TRUE    |
    FALSE

CharacterStringValue ::=
    RestrictedCharacterStringValue  |
    UnrestrictedCharacterStringValue

RestrictedCharacterStringValue ::=
    cstring                 |
    CharacterStringList     |
    Quadruple               |
    Tuple

CharacterStringList  ::=  "{" CharSyms "}"

CharSyms  ::=
    CharsDefn       |
    CharSyms "," CharsDefn

CharsDefn  ::=
    cstring         |
Quadruple           |
Tuple               |
AbsoluteReference

Quadruple   ::= "{"  Group  ","  Plane  ","  Row  ","  Cell "}"
Group       ::= number
Plane       ::= number
Row     ::= number
Cell        ::= number

Tuple ::= "{" TableColumn "," TableRow "}"
TableColumn     ::= number
TableRow    ::= number

UnrestrictedCharacterStringValue ::= SequenceValue

ChoiceValue ::= identifier ":" Value

EmbeddedPDVValue ::= SequenceValue

EnumeratedValue ::= identifier

ExternalValue ::= SequenceValue

IntegerValue ::=
    SignedNumber

SignedNumber ::=
```

```
    number      |
    "-" number

NullValue ::= NULL

ObjectIdentifierValue ::=
    "{" ObjIdComponentList "}"

ObjIdComponentList ::=
    ObjIdComponent              |
    ObjIdComponent ObjIdComponentList

ObjIdComponent ::=
    NameForm    |
    NumberForm  |
    NameAndNumberForm

NameForm ::= identifier
NumberForm ::=
    number

NameAndNumberForm ::= identifier "(" NumberForm ")"

OctetStringValue ::=
    bstring |
    hstring

RealValue ::=
    NumericRealValue    |
    SpecialRealValue

NumericRealValue ::=
    0       |
    SequenceValue

SpecialRealValue ::=
    PLUS-INFINITY   |
    MINUS-INFINITY

SequenceValue ::=
    "{" ComponentValueList "}"  |
    "{"  "}"

ComponentValueList ::=
    NamedValue              |
    ComponentValueList "," NamedValue

NamedValue ::=
    identifier Value    |
    Value

SequenceOfValue ::=
    "{" ValueList "}"   |
    "{"  "}"

ValueList ::=
    Value           |
    ValueList "," Value

SetValue ::=
    "{" ComponentValueList "}"  |
    "{"  "}"

SetOfValue ::=
    "{" ValueList "}"   |
    "{"  "}"

ValueSet ::= "{" ElementSetSpecs "}"

ElementSetSpecs ::=
    RootElementSetSpec
RootElementSetSpec ::= ElementSetSpec

ElementSetSpec ::=
    Unions      |
    ALL Exclusions

Exclusions ::= EXCEPT  Elements
```

```
Unions ::=
    Intersections           |
    UElems UnionMark Intersections

UElems ::= Unions

Intersections ::=
IntersectionElements

IntersectionElements ::= Elements

UnionMark  ::=
    "|" |
    UNION

Elements  ::=
    SubtypeElements
    "(" ElementSetSpec ")"

SubtypeElements ::=
    SingleValue

SingleValue ::= Value
```

The following productions are defined ITU-T Rec. X.681 | ISO/IEC 8824-2, as modified by Annex C, with the items defined in H.1 as terminal symbols:

```
DefinedSyntax ::= "{" DefinedSyntaxList ? "}"

DefinedSyntaxList ::= DefinedSyntaxToken  DefinedSyntaxList  ?

DefinedSyntaxToken ::=
    Literal |
    Setting
Literal ::=
    word    |
    ","
Setting ::=
    Value           |
    ValueSet            |
    EncodingObject      |
    EncodingObjectSet   |
    EncodingObjectList

EncodingObjectList ::= "{" EncodingObject "," * "}"

InstanceOfValue ::= Value

EncodingClassFieldType ::=
    DefinedEncodingClass
    "."
    FieldName

FieldName ::= PrimitiveFieldName "." +

PrimitiveFieldName ::=
    valuefieldreference         |
    valuesetfieldreference
```

The following productions are defined ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by Annex D, with the items defined in H.1 as terminal symbols:

```
ParameterizedAssignment ::=
    ParameterizedEncodingObjectAssignment           |
    ParameterizedEncodingStructureAssignment        |
    ParameterizedEncodingObjectSetAssignment

ParameterizedEncodingObjectAssignment ::=
    encodingobjectreference
    ParameterList
    DefinedEncodingClass
    "::="
    EncodingObject

ParameterizedEncodingStructureAssignment ::=
    encodingclassreference
```

```
    ParameterList
    "::="
    EncodingStructure

ParameterizedEncodingObjectSetAssignment ::=
    encodingobjectsetreference
    ParameterList
    DefinedEncodingClass
    "::="
    EncodingObjectSet

ParameterList ::= "{" Parameter "," + "}"

Parameter ::=
    ParamGovernor ":" DummyReference    |
    DummyReference

ParamGovernor ::=
    Governor        |
    DummyGovernor

Governor ::=
    DefinedEncodingClass    |
    EncodingClassFieldType  |
    REFERENCE

DummyGovernor ::= DummyReference

DummyReference ::= Reference

ParameterizedReference ::=
    Reference   |
    Reference "{"  "}"

ParameterizedEncodingObject ::=
    SimpleDefinedEncodingObject
    ActualParameterList

SimpleDefinedEncodingObject ::=
    ExternalEncodingObjectReference |
    encodingobjectreference

ParameterizedEncodingObjectSet ::=
    SimpleDefinedEncodingObjectSet
    ActualParameterList

SimpleDefinedEncodingObjectSet ::=
    ExternalEncodingObjectSetReference  |
    encodingobjectsetreference

ParameterizedEncodingStructure ::=
    SimpleDefinedEncodingStructure
    ActualParameterList

SimpleDefinedEncodingStructure ::=
    ExternalEncodingClassReference  |
    encodingclassreference

ActualParameterList ::= "{" ActualParameter "," + "}"

ActualParameter ::=
    Value               |
    ValueSet            |
    EncodingObject      |
    EncodingObjectSet   |
    EncodingObjectList  |
    AbsoluteReference
```