

ETSI TS 101 969 V1.1.1 (2001-05)

Technical Specification

Methods for Testing and Specification (MTS); Abstract Syntax Notation One (ASN.1) encoding rules; Specification of Encoding Control Notation (ECN)

[Draft ITU-T Recommendation X.692]



Reference

DTS/MTS-00068

Keywords

ASN.1, coding, protocol, MTS

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

No part may be reproduced except as authorized by written permission.
The copyright and the foregoing restriction extend to reproduction in all media.

© European Telecommunications Standards Institute 2001.
All rights reserved.

Contents

Intellectual Property Rights	10
Foreword.....	10
Introduction.....	10
1 Scope.....	12
2 Normative references	12
2.1 Identical International Standards.....	12
2.2 Additional references.....	13
3 Definitions.....	13
3.1 ASN.1 definitions.....	13
3.2 ECN-specific definitions.....	13
4 Abbreviations	16
5 Definition of ECN syntax.....	16
6 Encoding conventions and notation.....	16
7 The ECN character set	17
8 ECN lexical items	17
8.1 Encoding object references	18
8.2 Encoding object set references	18
8.3 Encoding class references	18
8.4 Reserved word items	19
8.5 Reserved encoding class name items.....	20
8.6 Non-ECN item	21
9 ECN Concepts	21
9.1 Encoding Control Notation (ECN) specifications	21
9.2 Encoding classes	21
9.3 Encoding structures	22
9.4 Encoding objects	22
9.5 Encoding object sets.....	23
9.6 Defining new encoding classes	23
9.7 Defining encoding objects	24
9.8 Differential encoding-decoding.....	25
9.9 Encoders options in encodings.....	25
9.10 Properties of encoding objects	25
9.11 Parameterization.....	26
9.12 Governors	26
9.13 General aspects of encodings.....	27
9.14 Identification of information elements.....	27
9.15 Reference parameters and determinants	28
9.16 Replacement classes and structures.....	28
9.17 Mapping abstract values onto fields of encoding structures.....	29
9.18 Contents of Encoding Definition Modules.....	30
9.19 Contents of the Encoding Link Module.....	30
9.20 Defining encodings for primitive encoding classes	30
9.21 Application of encodings	32
9.22 Combined encoding object set	33
9.23 Application point.....	33
9.24 Conditional encodings	34
9.25 Changes to ASN.1 Recommendations International Standards	34
10 Identifying encoding classes, encoding objects, and encoding object sets	35
11 Encoding ASN.1 types.....	38

11.1	General	38
11.2	Built-in encoding classes used for implicitly generated encoding structures	38
11.3	Simplification and expansion of ASN.1 notation for encoding purposes	39
11.4	The implicitly generated encoding structure	41
12	The Encoding Link Module (ELM)	42
12.1	Structure of the ELM	42
12.2	Encoding types	43
13	Application of encodings	43
13.1	General	43
13.2	The combined encoding object set and its application	44
14	The Encoding Definition Module (EDM)	46
15	The renames clause	47
15.1	Explicitly generated and exported structures	47
15.2	Name changes	48
15.3	Specifying the region for name changes	49
16	Encoding class assignments	50
16.1	General	50
16.2	Encoding structure definition	52
16.3	Alternative encoding structure	54
16.4	Repetition encoding structure	55
16.5	Concatenation encoding structure	55
17	Encoding object assignments	56
17.1	General	56
17.2	Encoding with a defined syntax	57
17.3	Encoding with encoding object sets	58
17.4	Encoding using value mappings	58
17.5	Encoding an encoding structure	59
17.6	Differential encoding-decoding	61
17.7	Encoding with encoder's options	61
17.8	Non-ECN definition of encoding objects	62
18	Encoding object set assignments	62
18.1	General	62
18.2	Built-in encoding object sets	63
19	Mapping values	64
19.1	General	64
19.2	Mapping by explicit values	65
19.3	Mapping by matching fields	66
19.4	Mapping by #TRANSFORM encoding objects	67
19.5	Mapping by abstract value ordering	67
19.6	Mapping by value distribution	68
19.7	Mapping integer values to bits	69
20	Defining encoding objects using defined syntax	70
21	Types used in defined syntax specification	71
21.1	The Unit type	71
21.2	The EncodingSpaceSize type	72
21.3	The EncodingSpaceDetermination type	72
21.4	The UnusedBitsDetermination type	73
21.5	The OptionalityDetermination type	73
21.6	The AlternativeDetermination type	74
21.7	The RepetitionSpaceDetermination type	75
21.8	The Justification type	75
21.9	The Padding type	76
21.10	The Pattern and Non-Null-Pattern types	76
21.11	The RangeCondition type	77
21.12	The SizeRangeCondition type	77

21.13	The ReversalSpecification type.....	78
21.14	The ResultSize type.....	78
22	Commonly used encoding parameter groups	79
22.1	Replacement specification	79
22.1.1	Encoding parameters, syntax, and purpose.....	79
22.1.2	Specification restrictions	80
22.1.3	Encoder actions.....	81
22.1.4	Decoder actions	82
22.2	Pre-alignment and padding specification	82
22.2.1	Encoding parameters, syntax, and purpose.....	82
22.2.2	Specification constraints.....	83
22.2.3	Encoder actions.....	83
22.2.4	Decoder actions	83
22.3	Start pointer specification	83
22.3.1	Encoding parameters, syntax, and purpose.....	83
22.3.2	Specification constraints.....	84
22.3.3	Encoder actions.....	84
22.3.4	Decoder actions	84
22.4	Encoding space specification	85
22.4.1	Encoding parameters, syntax, and purpose.....	85
22.4.2	Specification restrictions	85
22.4.3	Encoder actions.....	86
22.4.4	Decoder actions	86
22.5	Optionality determination	87
22.5.1	Encoding parameters, syntax, and purpose.....	87
22.5.2	Specification restrictions	87
22.5.3	Encoder actions.....	88
22.5.4	Decoder actions	89
22.6	Alternative determination	89
22.6.1	Encoding parameters, syntax, and purpose.....	89
22.6.2	Specification restrictions	89
22.6.3	Encoder actions.....	90
22.6.4	Decoder actions	90
22.7	Repetition space specification	91
22.7.1	Encoding parameters, syntax, and purpose.....	91
22.7.2	Specification constraints.....	92
22.7.3	Encoder actions.....	92
22.7.4	Decoder actions	93
22.8	Value padding and justification.....	94
22.8.1	Encoding parameters, syntax, and purpose.....	94
22.8.2	Specification restrictions	95
22.8.3	Encoder actions.....	95
22.8.4	Decoder actions	95
22.9	Identification handle specification.....	96
22.9.1	Encoding parameters, syntax, and purpose.....	96
22.9.2	Specification constraints.....	96
22.9.3	Encoders actions	96
22.9.4	Decoders actions.....	97
22.10	Concatenation specification	97
22.10.1	Encoding parameters, syntax, and purpose.....	97
22.10.2	Specification constraints.....	97
22.10.3	Encoder actions.....	97
22.10.4	Decoder actions	98
22.11	Contained type encoding specification	98
22.11.1	Encoding parameters, syntax, and purpose.....	98
22.11.2	Encoder actions.....	98
22.11.3	Decoder actions	99
22.12	Bit reversal specification	99
22.12.1	Encoding parameters, syntax, and purpose.....	99
22.12.2	Specification constraints.....	99
22.12.3	Encoder actions.....	99

22.12.4	Decoder actions	100
23	Defined syntax specification for bitfield and constructor classes.....	100
23.1	Defining encoding objects for classes in the alternatives category.....	100
23.1.1	The defined syntax.....	100
23.1.2	Purpose and restrictions.....	101
23.1.3	Encoder actions.....	101
23.1.4	Decoder actions	102
23.2	Defining encoding objects for classes in the bitstring category	102
23.2.1	The defined syntax.....	102
23.2.2	Model for the encoding of classes in the bitstring category.....	103
23.2.3	Purpose and restrictions.....	103
23.2.4	Encoder actions.....	104
23.2.5	Decoder actions	104
23.3	Defining encoding objects for classes in the boolean category	104
23.3.1	The defined syntax.....	104
23.3.2	Purpose and restrictions.....	106
23.3.3	Encoder actions.....	106
23.3.4	Decoder actions	107
23.4	Defining encoding objects for classes in the characterstring category.....	107
23.4.1	The defined syntax.....	107
23.4.2	Model for the encoding of classes in the characterstring category.....	108
23.4.3	Purpose and restrictions.....	108
23.4.4	Encoder actions.....	109
23.4.5	Decoder actions	109
23.5	Defining encoding objects for classes in the concatenation category	109
23.5.1	The defined syntax.....	109
23.5.2	Purpose and restrictions.....	111
23.5.3	Encoder actions.....	111
23.5.4	Decoder actions	112
23.6	Defining encoding objects for classes in the integer category.....	112
23.6.1	The defined syntax.....	112
23.6.2	Purpose and restrictions.....	112
23.6.3	Encoder actions.....	112
23.6.4	Decoder actions	112
23.7	Defining encoding objects for the #CONDITIONAL-INT class.....	113
23.7.1	The defined syntax.....	113
23.7.2	Purpose and restrictions.....	114
23.7.3	Encoder actions.....	115
23.7.4	Decoder actions	116
23.8	Defining encoding objects for classes in the null category	116
23.8.1	The defined syntax.....	116
23.8.2	Purpose and restrictions.....	117
23.8.3	Encoder actions.....	117
23.8.4	Decoder actions	118
23.9	Defining encoding objects for classes in the octetstring category	118
23.9.1	The defined syntax.....	118
23.9.2	Model for the encoding of classes in the octetstring category	119
23.9.3	Purpose and restrictions.....	119
23.9.4	Encoder actions.....	120
23.9.5	Decoder actions	120
23.10	Defining encoding objects for classes in the optionality category.....	121
23.10.1	The defined syntax.....	121
23.10.2	Purpose and restrictions.....	121
23.10.3	Encoder actions.....	122
23.10.4	Decoder actions	122
23.11	Defining encoding objects for classes in the pad category.....	122
23.11.1	The defined syntax.....	122
23.11.2	Purpose and restrictions.....	123
23.11.3	Encoder actions.....	123
23.11.4	Decoder actions	124
23.12	Defining encoding objects for classes in the repetition category.....	124

23.12.1	The defined syntax	124
23.12.2	Purpose and restrictions.....	124
23.12.3	Encoder actions.....	124
23.12.4	Decoder actions	125
23.13	Defining encoding objects for the #CONDITIONAL-REPETITION class	125
23.13.1	The defined syntax	125
23.13.2	Purpose and restrictions.....	126
23.13.3	Encoder actions.....	127
23.13.4	Decoder actions	127
23.14	Defining encoding objects for classes in the tag category	128
23.14.1	The defined syntax	128
23.14.2	Purpose and restrictions.....	129
23.14.3	Encoder actions.....	129
23.14.4	Decoder actions	130
23.15	Defining encoding objects for classes in the other categories	130
24	Defined syntax specification for the #TRANSFORM encoding class.....	130
24.1	Summary of encoding parameters and defined syntax.....	130
24.2	Source and target of transforms.....	132
24.3	The int-to-int transform	133
24.4	The bool-to-bool transform.....	133
24.5	The bool-to-int transform.....	134
24.6	The int-to-bool transform.....	134
24.7	The int-to-chars transform	134
24.8	The int-to-bits transform.....	135
24.9	The bits-to-int transform.....	136
24.10	The char-to-bits transform	137
24.11	The bits-to-char transform	138
24.12	The bit-to-bits transform.....	139
24.13	The bits-to-bits transform	139
25	Complete encodings and the #OUTER class.....	140
25.1	General rules for encoding and decoding.....	140
25.2	Encoding parameters, syntax, and purpose for the #OUTER class	140
25.3	Encoder actions for #OUTER	141
25.4	Decoder actions for #OUTER.....	141
Annex A (normative):	Addendum to ITU-T Rec. X.680 ISO/IEC 8824-1.....	143
A.1	Exports and imports clauses	143
A.2	Addition of "REFERENCE"	144
A.3	Notation for character string values.....	144
Annex B (normative):	Addendum to ITU-T Rec. X.681 ISO/IEC 8824-2.....	145
B.1	Definitions.....	145
B.2	Additional lexical items	145
B.3	Addition of "ENCODING-CLASS"	145
B.4	FieldSpec additions.....	146
B.5	Fixed-type value list field spec	146
B.6	Fixed-class encoding object field spec.....	146
B.7	Variable-class encoding object field spec	147
B.8	Fixed-class encoding object set field spec	147
B.9	Fixed-class encoding object list field spec	147
B.10	Encoding class field spec	148
B.11	Encoding object list notation.....	148

B.12	Primitive field names	148
B.13	Additional reserved words	148
B.14	Definition of encoding objects	149
B.15	Additions to "Setting"	149
B.16	Encoding class field type	150
Annex C (normative):	Addendum to ITU-T Rec. X.683 ISO/IEC 8824-4.....	151
C.1	Parameterized assignments	151
C.2	Parameterized encoding assignments	151
C.3	Referencing parameterized definitions	152
C.4	Actual parameter list	153
Annex D (informative):	Examples	154
D.1	General examples	154
D.1.1	An encoding object for a boolean type	154
D.1.2	An encoding object for an integer type	155
D.1.3	Another encoding object for an integer type	155
D.1.4	An encoding object for an integer type with holes	155
D.1.5	A more complex encoding object for an integer type	155
D.1.6	Positive integers encoded in BCD	156
D.1.7	An encoding object of class #BITS	157
D.1.8	An encoding object for an octetstring type	157
D.1.9	An encoding object for a character string type	158
D.1.10	Mapping character values to bit values	158
D.1.11	An encoding object for a sequence type	158
D.1.12	An encoding object for a choice type	159
D.1.13	Encoding a bitstring containing another encoding	160
D.1.14	An encoding object set	160
D.1.15	ELM definitions	160
D.1.16	ASN.1 definitions	161
D.1.17	EDM definitions	161
D.2	Specialization examples	161
D.2.1	Encoding by distributing values to an alternative encoding structure	162
D.2.2	Encoding by mapping ordered abstract values to an alternative encoding structure	162
D.2.3	Compression of non-continuous value ranges	163
D.2.4	Compression of non-continuous value ranges using a transform	163
D.2.5	Compression of an unevenly distributed value set by mapping ordered abstract values	164
D.2.6	Presence of an optional component depending on the value of another component	164
D.2.7	The presence of an optional component depends on some external condition	165
D.2.8	A variable length list	165
D.2.9	Equal length lists	166
D.2.10	Uneven choice alternative probabilities	167
D.2.11	A version 1 message	168
D.2.12	The encoding object set	168
D.2.13	ELM definitions	169
D.2.14	ASN.1 definitions	169
D.2.15	EDM definitions	169
D.3	Explicitly generated structure examples	170
D.3.1	Sequence with optional components defined by a pointer	170
D.3.2	Addition of a boolean type as a presence determinant	170
D.3.3	Sequence with optional components identified by a unique tag and delimited by a length field	171
D.3.4	Sequence-of type with a count	172
D.3.5	Encoding object set	173
D.3.6	ELM definitions	173
D.3.7	ASN.1 definitions	173

D.3.8	EDM definitions.....	174
D.4	Legacy protocol example	174
D.4.1	Introduction.....	174
D.4.2	Encoding definition for the top-level message structure.....	176
D.4.3	Encoding definition for a message structure	176
D.4.4	Encoding for the sequence type "B"	177
D.4.5	Encoding for an octet-aligned sequence-of type with a length determinant.....	177
D.4.6	Encoding for an octet-aligned sequence-of type which continues to the end of the PDU.....	177
D.4.7	ELM definitions	178
D.4.8	EDM definitions.....	178
Annex E (informative):	Support for Huffman encodings.....	179
Annex F (informative):	Additional information on the Encoding Control Notation (ECN)	181
Annex G (informative):	Summary of the ECN notation	182
G.1	Terminal symbols	182
G.2	Productions.....	184
History	198

Intellectual Property Rights

IPRs essential or potentially essential to the present document may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (<http://www.etsi.org/ipr>).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Foreword

This Technical Specification (TS) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

Introduction

The Encoding Control Notation (ECN) is a notation for specifying encodings of ASN.1 types that differ from those provided by standardized encoding rules. ECN can be used to encode all types of an ASN.1 specification, but can also be used with standardized encoding rules such as BER or PER (ITU-T Rec. X.690 | ISO/IEC 8825-1 and ITU-T Rec. X.691 | ISO/IEC 8825-2) to specify only the encoding of types that have special requirements.

An ASN.1 type specifies a set of abstract values. Encoding rules specify the representation of these abstract values as a series of bits. ECN is designed to meet the following encoding needs:

- a) The need to write ASN.1 types (and get the support of ASN.1 tools in implementations) for established ("legacy") protocols where the encoding is already determined and differs from all standardized encoding rules.
- b) The need to produce encodings that are minor variations on standardized rules.

The linkage provided in an ECN specification to an ASN.1 specification is well-defined and machine processable, so encoders and decoders can be automatically generated from the combined specifications. This is a significant factor in reducing both the amount of work and the possibility of errors in making interoperable systems. Another significant advantage is the ability to provide automatic tool support for testing.

These advantages are available with ASN.1 alone when standardized encoding rules suffice, but the ECN work provides these advantages in circumstances where the standardized encoding rules are not sufficient.

NOTE 1: Currently ECN supports only binary-based encodings, but could be extended in the future to cover character-based encodings.

Annex A forms an integral part of the present document, and details modifications to be made to ITU-T Rec. X.680 | ISO/IEC 8824-1 to support the notation used in the present document.

Annex B forms an integral part of the present document, and details modifications to be made to ITU-T Rec. X.681 | ISO/IEC 8824-2 to support the notation used in the present document.

Annex C forms an integral part of the present document, and details modifications to be made to ITU-T Rec. X.683 | ISO/IEC 8824-4 to support the notation used in the present document.

NOTE 2: It is not intended that annexes A, B and C be progressed as amendments to the referenced Recommendations | International Standards. The modifications are solely for the purpose of ECN definition (see clauses 5 and 9.24).

Annex D does not form an integral part of the present document, and contains examples of the use of ECN.

Annex E does not form an integral part of the present document and provides more detail on the support for Huffman encodings in ECN.

Annex F does not form an integral part of the present document, and identifies a Web site providing access to further information and links relevant to ECN.

Annex G does not form an integral part of the present document, and provides a summary of ECN using the notation of clause 5.

1 Scope

The present document 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 non-ECN notation.

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 International Standards contain provisions which, through reference in this text, constitute provisions of the present document. At the time of publication, the editions indicated were valid. All International Standards are subject to revision, and parties to agreements based on the present document are encouraged to investigate the possibility of applying the most recent edition of the Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

2.1 Identical International Standards

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

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

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

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

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)".

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

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

NOTE 2: The above references shall be interpreted as references to the identified Recommendations | International Standards together with all their published amendments and technical corrigenda.

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.

ITU-T Recommendation X.660 | ISO/IEC 9834: "Information technology - Open Systems Interconnection - Procedures for the operation of OSI Registration Authorities: General procedures".

3 Definitions

For the purposes of the present document, the following terms and definitions apply.

3.1 ASN.1 definitions

The present document uses the terms defined in clause 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

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

3.2.2 auxiliary field: field of a replacement structure (that is added in the ECN specification) whose value is set directly by the encoder without the use of any abstract value provided by the application

NOTE: An example of an auxiliary field is a length determinant for an integer encoding or for a repetition.

3.2.3 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 - see 3.2.30) needed for successful decoding, or both

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

3.2.4 bit-field class: encoding class whose objects specify the encoding of abstract values (of some ASN.1 type) into bits

NOTE: Other encoding classes are concerned with more general encoding procedures, such as those required to determine the end of repetitions of bit-field class encodings, or to determine which of a set of alternative bit-field encodings is present.

3.2.5 bounds condition: 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

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

3.2.7 combined encoding object set: temporary set of encoding objects produced by the combination of two sets of encoding objects for the purpose of applying encodings

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

3.2.9 containing type: 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 (using "CONTAINING"/"ENCODED BY") can be applied are the bitstring and the octetstring types.

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

3.2.11 differential encoding-decoding: 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 that standard. This is sometimes referred to as support for extensibility.

3.2.12 encoding class: set of all possible encodings for a specific part of the procedures needed to perform the encoding or decoding of an ASN.1 type

NOTE: Encoding classes are defined for the encoding of primitive ASN.1 types, but are also defined for the procedures associated with ASN.1 tag notation, the use of "OPTIONAL" and for encoding constructors.

3.2.13 encoding class category: grouping of encoding classes with some common characteristics

NOTE: Examples are the integer category, the encoding constructor category, and the bit-field category.

3.2.14 encoding constructor: encoding class whose encoding objects define procedures for combining, selecting, or repeating parts of an encoding.

(Examples are the #ALTERNATIVES, #CHOICE, #CONCATENATION, #SEQUENCE, etc classes).

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

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

3.2.17 encoding object: specification of some part of the procedures needed to perform the encoding or decoding of an ASN.1 type

NOTE: Encoding objects can specify the encoding of primitive ASN.1 types, but can also specify the procedures associated with ASN.1 tag notation, the use of "OPTIONAL" and with encoding constructors.

3.2.18 encoding object set: 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.

3.2.19 encoding parameter: piece of information used to define an encoding using the notation specified in clauses 23, 24 and 25 of the present document

3.2.20 encoding space: number of bits (or octets, words or other units) used to encode an abstract value into a bit-field (see 9.20.5)

3.2.21 encoding structure: structure of an encoding, defined either from the structure of an ASN.1 type definition, or in an EDM using bit-field classes and encoding constructors

NOTE 1: 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.

NOTE 2: Definition of an encoding structure is also the definition of a corresponding encoding class.

3.2.22 explicitly generated encoding structure: encoding structure derived from an implicitly generated encoding structure by use of the renames clause in an EDM

3.2.23 extensibility: provisions 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

3.2.24 fully-qualified name: reference to an encoding class that includes either the name of the EDM module in which that encoding class was defined, or the name of the ASN.1 module in which it was generated

NOTE: A fully-qualified name (see production "ExternalEncodingClassReference" in 10.6) has to be used in the body of a module if the encoding class is an implicitly generated encoding structure whose name is the same as a reserved class name, or if use of the name alone would produce ambiguity due to multiple imports of classes with that name. (See A.1/12.15).

3.2.25 generated encoding structure: implicitly or explicitly generated encoding structure whose purpose is to define the encodings of the corresponding ASN.1 type through application of encodings in the ELM

3.2.26 governor: part of an ECN specification which determines the syntactic form (and semantics) of some other part of the ECN specification

NOTE: A governor is an encoding class reference, 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 reference in ASN.1 acting as the governor for ASN.1 value notation.

3.2.27 identification handle: part of an encoding which serves to distinguish encodings of one encoding class from those of other encoding classes

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

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

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

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

3.2.31 negative integer value: value less than zero

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

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

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

3.2.35 positive integer value: value greater than zero

3.2.36 presence determinant: bit-field that determines whether an optional bit-field is present or not

3.2.37 primitive class: encoding class which is not an encoding structure, and which cannot be de-referenced to some other class (see 16.1.14)

3.2.38 replacement structure: parameterized structure used to replace some or all parts of a construction before encoding the construction

3.2.39 self-delimiting encoding: 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 in the set

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

3.2.40 size range condition: 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

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

3.2.42 start pointer: auxiliary field indicating the presence or absence of an optional bit-field, and in the case of presence, containing the offset from the current position to the bit-field

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

3.2.44 top-level type(s): those ASN.1 type(s) 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.

3.2.45 transforms: encoding objects of the class #TRANSFORM which specify that the encoding of the abstract values associated with some class is to be the encoding of different abstract values associated with the same or a different class

NOTE: Transforms can be used, for example, to specify simple arithmetic operations on integer values, or to map integer values into characterstrings or bitstrings.

3.2.46 value encoding: the way in which an encoding space is used to represent an abstract value (see 9.20.5)

4 Abbreviations

For the purposes of the present document, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
BCD	Binary Coded Decimal
BER	Basic Encoding Rules of ASN.1
CER	Canonical 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

5.1 The present document 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.

5.2 The present document employs the notation for information object classes defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 as modified by annex B.

5.3 The present document references productions defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 as modified by annex A, ITU-T Rec. X.681 | ISO/IEC 8824-2 as modified by annex B, and ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by annex C.

6 Encoding conventions and notation

6.1 The present document 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.

6.2 For the purpose of the present document, the bits of an octet are numbered from 8 to 1, where bit 8 is the "most significant bit" and bit 1 is the "least significant bit".

6.3 For the purposes of the present document, 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.

6.4 When figures are shown in the present document, the "leading bit" is always shown on the left of the figure.

7 The ECN character set

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

7.2 With the exception of comment (as defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.6), non-ECN definition of encoding objects (see 17.8) and character string values, ECN specifications use only the characters listed in table 1.

7.3 ECN lexical items defined in clause 8 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

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

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

7.5 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, clause 11, the present document uses ECN lexical items specified in the following clauses. The general rules specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 11.1 apply in this clause.

NOTE: Annex G lists all lexical items and all the productions used in the present document, 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 analysing 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 (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.19, as modified by A.1) or in an "ExternalEncodingClassReference" (see the note on 14.11).

8.4 Reserved word items

Names of reserved word items:

ALL
AS
BEGIN
BER
BITS
BY
CER
COMPLETED
DECODE
DER
DISTRIBUTION
ENCODE
ENCODING-CLASS
ENCODE-DECODE
ENCODING-DEFINITIONS
END
EXCEPT
EXPORTS
FALSE
FIELDS
FROM
GENERATES
IF
IMPORTS
IN
LINK-DEFINITIONS
MAPPING
MAX
MIN
MINUS-INFINITY
NON-ECN-BEGIN
NON-ECN-END
NULL
OPTIONAL-ENCODING
OPTIONS
ORDERED
OUTER
PER-BASIC-ALIGNED
PER-BASIC-UNALIGNED
PER-CANONICAL-ALIGNED
PER-CANONICAL-UNALIGNED
PLUS-INFINITY
REFERENCE
REMAINDER
RENAMES
SIZE
STRUCTURE
STRUCTURED
TO
TRANSFORMS
TRUE
UNION
USE
USE-SET
VALUES
WITH

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" statement) in clause 23 are not reserved words (see also B.13).

8.5 Reserved encoding class name items

Names of reserved encoding class name items:

```
#ALTERNATIVES
#BITS
#BIT-STRING
#BMPString
#BOOL
#BOOLEAN
#CHARACTER-STRING
#CHARS
#CHOICE
#CONCATENATION
#CONDITIONAL-INT
#CONDITIONAL-REPETITION
#EMBEDDED-PDV
#ENCODINGS
#ENUMERATED
#EXTERNAL
#GeneralizedTime
#GeneralString
#GraphicString
#IA5String
#INT
#INTEGER
#NUL
#NULL
#NumericString
#OBJECT-IDENTIFIER
#OCTETS
#OCTET-STRING
#OPEN-TYPE
#OPTIONAL
#OUTER
#PAD
#PrintableString
#REAL
#RELATIVE-OID
#REPETITION
#SEQUENCE
#SEQUENCE-OF
#SET
#SET-OF
#TAG
#TeletexString
#TRANSFORM
#UniversalString
#UTCTime
#UTF8String
#VideotexString
#VisibleString
```

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

8.6 Non-ECN item

Name of item – anystringexceptnonecnend

An "anystringexceptnonecnend" shall consist of one or more characters from the ISO/IEC 10646-1 character set, except that it shall not be the character sequence "NON-ECN-END" nor shall that character sequence appear within it.

9 ECN Concepts

This clause describes the main concepts underlying the present document.

9.1 Encoding Control Notation (ECN) specifications

9.1.1 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.

9.1.2 The most important part of ECN is the concept of an **encoding structure definition**. ASN.1 is used to define complex abstract values using primitive types and constructors. In the same way, complex encodings can be defined using a similar notation where construction mechanisms are used to combine simple bit-fields into more complex encodings, and eventually into complete messages. This is called encoding structure definition. In using ECN with ASN.1, it is necessary in principle to:

- a) define the abstract syntax (the set of abstract values to be communicated, and their semantics); and
- b) the encoding structure (the structure of fields) used to carry these abstract values; and
- c) to relate the components of the abstract value to the encoding structure fields; and
- d) to define the encoding of each encoding structure field and mechanisms for identifying repetitions of fields and identification of alternatives, etc.

9.1.3 The above process normally takes part in several stages. First an ASN.1 definition is produced detailing the abstract syntax. From this a crude encoding structure is automatically generated (conceptually within the ASN.1 module). This implicitly generated structure contains only fields that carry the application semantics, without fields for things like length determination, alternative selection, and so on.

9.1.4 This structure can be transformed by a series of mechanisms into the structure of fields that is actually required, including all fields needed to support the decoding activity (determinants). These mechanisms all involve some form of replacement of a simple field carrying application semantics by a more complex structure. Such replacements form an important part of ECN specification.

9.1.5 We can further define **encoding objects** for each of the fields in the final structure. These determine not only the encoding of fields, but also the way in which one field determines the length (for example) of another, or has its optionality resolved.

9.1.6 The above definitions occur in Encoding Definition Modules (EDMs). The last step is to apply a set of defined encoding objects to the final encoding structure in order to completely determine an encoding. This is done in the Encoding Link Module (ELM).

9.2 Encoding classes

9.2.1 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 encoding structure fields corresponding to the type. Encoding class names begin with the character "#".

EXAMPLE: Encoding rules for the ASN.1 built-in 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.

9.2.2 There are several kinds of encoding classes:

9.2.2.1 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 (see also 9.3.2), and for the definition of encoding rules for handling optionality through #OPTIONAL. Encoding of tags is supported by the #TAG class. 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.

9.2.2.2 Encoding classes for implicitly generated encoding structures. These have names consisting of the character "#" followed by the "typereference" appearing in a "TypeAssignment" in an ASN.1 module. Such encoding classes are implicitly generated whenever a (non-parameterized) "typereference" is assigned 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.

9.2.2.3 Encoding classes for user-defined encoding structures. These are encoding classes defined by the ECN user by specifying an encoding structure (see 9.3) as a structure made up of bit-fields and encoding constructors. These encoding structures are similar to the implicitly generated encoding structures, but the ECN user has full control of their structure. These classes enable complex encoding rules to be defined, and are important for the use of ASN.1 with ECN for specifying legacy protocols, where additional bit-fields are needed in the encoding for determinants.

9.2.2.4 Encoding classes for explicitly generated encoding structures. These are encoding classes produced from an implicitly generated encoding structure by selectively changing the names of certain classes in order to indicate places where specialized encodings are needed for optionality, sequence-of termination, etc.

9.3 Encoding structures

9.3.1 Encoding structure definitions have some similarity to ASN.1 type definitions, and have 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 D.2.8.4 for an example of an encoding structure definition).

9.3.2 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.

9.3.3 An encoding structure definition defines a structure-based encoding class. Such classes cannot have the same names as encoding classes that are imported into the module. (See ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.12, as modified by A.1 of the present document).

9.3.4 Encoding structure names can be exported and imported between Encoding Definition Modules and can be used whenever an encoding class name in the bit-field category is required.

9.3.5 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

9.4.1 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.

9.4.2 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.

9.4.3 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).

9.4.4 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 (see 18.2).

9.5 Encoding object sets

9.5.1 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 D.1.14 for an example).

9.5.2 A fundamental rule of encoding object set construction is that any set can contain only one encoding object of a given encoding class (see also 9.6.2). Thus there is no ambiguity when an encoding object set is applied to a type to define its encoding.

9.5.3 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

9.6.1 Those familiar with ASN.1 will be aware that a type assignment can be used to create new names (new types) from, for example, the types "INTEGER" or "BOOLEAN". The new names identify types that are the same as "INTEGER" or "BOOLEAN", but carry different semantics. This concept is extended in ECN to allow the creation (in a class assignment – see 16.1.1) of new names (new classes) for constructors such as #SEQUENCE. The new names identify classes that perform a similar function in structuring encodings (for example, concatenation), but which are to have different encoding objects applied to them. A new class name assigned for an old class retains certain characteristics of that old class. So an assignment such as "#My-Sequence ::= #SEQUENCE" creates the new class name #My-Sequence which is still an encoding class concerned with the concatenation of components. We say that such encoding classes are in the same category.

9.6.2 If a new encoding class is created from an existing encoding class, encoding objects of both the old encoding class and the new encoding class can appear in an encoding object set.

9.6.3 All built-in encoding classes are derived from one of a small number of primitive encoding classes. Thus #SEQUENCE and #SET are both derived from the #CONCATENATION class, #INTEGER and #ENUMERATED are both derived from the #INT class, and the classes for the different ASN.1 character string types are all derived from the #CHARS class. An encoding structure (for example, one implicitly generated from an ASN.1 type) can contain a mix of different classes all derived from the same primitive class, enabling different encodings to be applied to #SEQUENCE and #SET (for example).

9.6.4 It is often convenient to put encoding classes into categories, based on the primitive class they are derived from. Thus we say that #INTEGER, #ENUMERATED and #INT (and any class derived from them in a class assignment statement such as "#My-int ::= #INT") are in the integer category. There are also categories that group together a number of very different classes that share some characteristic. Thus any class that can have abstract values directly associated with it, and hence which produces bits in an encoding, is said to be in the bit-field category. Classes that are responsible for grouping or repeating encodings (for example classes in the alternatives or the repetition category) are in the encoding constructor category. There are also two classes whose encoding objects define procedures not directly related to constructing an encoding (#TRANSFORM and #OUTER): these are described as being in the encoding procedure category. Encoding structures are defined using classes in the bit-field category that are combined using classes in the encoding constructor category, together with classes in the optionality (representing encoding procedures for resolving optionality) and tag (representing encoding of tags) categories. All such classes are in the encoding structure category (and also in the bit-field category). In general, a class is in more than one category – if it is in the integer category or the encoding structure category, it is also in the bit-field category; if it is in the alternatives category it is also in the encoding constructor category, and so on.

9.6.5 For the primitive classes, the category is directly assigned. For classes created in an encoding class assignment statement, the category is determined by the notation to the right of the "::=" symbol. If that notation is an encoding structure definition, then the class is in both the encoding structure category and in the bit-field category. If the notation is a simple class reference name, then the category of the new class is the same as the category of the class being assigned.

9.6.6 The categories of encoding class (see 16.1.3) are:

9.6.6.1 The bit-field category (classes that correspond to actual fields in an encoding such as #INTEGER, or to more complex structures).

9.6.6.2 The alternatives category (classes that are derived by class assignment from #ALTERNATIVES).

9.6.6.3 The concatenation category (classes that are derived by class assignment from #CONCATENATION).

9.6.6.4 The repetition category (classes that are derived by class assignment from #REPETITION).

9.6.6.5 The optionality category (classes that are derived by class assignment from #OPTIONAL).

9.6.6.6 The encoding constructor category (classes that are in the alternatives, concatenation, or repetition categories).

9.6.6.7 The tag category (classes that are derived by class assignment from #TAG).

9.6.6.8 The encoding procedure category (classes not directly related to ASN.1 constructs, and which cannot be assigned new names - #OUTER, #TRANSFORM, #CONDITIONAL-INT, #CONDITIONAL-REPETITION).

9.6.6.9 Categories for classes derived from classes in the primitive bit-field category (the boolean, bitstring, characterstring, integer, null, objectidentifier, octetstring, opentype, pad, and real categories).

9.7 Defining encoding objects

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

9.7.1 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.

9.7.2 The second, available for a restricted set of encoding classes, is to use a defined syntax (see 17.2) 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 D.1.1.2 for an example of defining an encoding object of class #BOOLEAN which contains encodings for the ASN.1 type boolean).

9.7.3 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.

NOTE: This can often be useful, but requires knowledge of the encodings produced by standardized encoding rules.

9.7.4 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 the abstract values associated with "A" can now be defined as the application to the corresponding abstract values associated with "B" of the encoding object for "B". (See D.2.8.3 for an example). There are many variants of this (see 9.16).

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.

9.7.5 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 encoding class.

9.7.6 The sixth is to define an encoding object for differential encoding-decoding (see 9.8), using two separate encoding objects, one of which defines the encoder's behaviour, and the other of which tells a decoder what encoding should be assumed.

NOTE: An example would be to encode a field which is "reserved for future use" as all zeros, but to accept any value when decoding.

9.7.7 The seventh is to define an encoder's option encoding object, which contains a list of encoding objects of the same class. It is an encoder's choice which encoding object from the list is to be applied.

9.7.8 Finally, an encoding object can be defined using non-ECN notation. This is a facility to allow use of any desired notation (including natural language) to define the encoding object (see D.2.7.3).

NOTE: Non-ECN notation should be used with caution, as tool-support for implementation is generally not possible in this case.

9.8 Differential encoding-decoding

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

9.8.2 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).

9.8.3 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.

9.8.4 Support for differential encoding-decoding in ECN is provided by syntax that enables the definition of an encoding object (for any class) that encapsulates two encoding objects. Each encoding object defines rules for encoding. The first encoding object defines the rules that an encoder uses. The decoder uses the second encoding object as a specification of the way the encoding was done.

NOTE: In ECN, the rules a decoder uses (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 Encoders options in encodings

9.9.1 Encoders options in protocols are generally regarded today as something to be avoided, but ECN has to provide support for such options if a protocol designer decides (or has in the past decided) to include them.

9.9.2 When values are being encoded into an encoding space, it is possible to specify that the size of the encoding space (see 9.20) is an encoder's option, provided there is some form of length determinant associated with the encoding. (The extent of the encoder's options may be limited by the maximum value that can be encoded in the length determinant). This provides a detailed level of support for encoder's options.

9.9.3 A more global mechanism is similar to the support for differential encoding-decoding (see 9.8), but in this case an encoding object for a class can be defined as an encoder's choice of any encoding object from a list of defined encoding objects for that class. In addition to specifying the list of possible encodings, it is also necessary to provide the specification of an encoding object for a class in the alternatives category (see 9.6). This encoding object specifies the encodings and procedures needed to enable a decoder to determine which encoding object was used by the encoder.

9.10 Properties of encoding objects

9.10.1 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.

9.10.2 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).

NOTE: A somewhat different (and not standardized) external dependency would be the definition of a non-ECN encoding object for an #ALTERNATIVES encoding class which determines the selected alternative based on external data such as the channel the message is being sent on.

9.10.3 A third key feature is that an encoding object may exhibit an **identification handle** in its encodings. This is a part of all the encodings that it produces and distinguishes its encodings from encodings of other encoding objects (of any class) that exhibit the same identification handle. Identification handles have to be visible to decoders without knowledge of either the encoding class or the abstract value that was encoded (but with knowledge of the name of the identification handle that is being used). This concept models (and generalizes) the use of tags in BER encodings: the tag value in BER 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.

9.11 Parameterization

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

9.11.2 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 9.13.2). (See D.1.11.3 for an example of a parameterized ECN definition).

9.11.3 Another important use of parameterization is in the definition of an encoding structure that will be used to replace many different classes in an encoding (see also 9.16.5). For example, the mechanism used to handle optionality is often an immediately (mandatory) preceding "presence-bit" for each optional component. A parameterized structure can be defined consisting of a concatenation of a #BOOLEAN (used as a presence determinant) followed by an optional component defined as a dummy parameter (which will be instantiated with the element that the structure will replace), and whose presence is determined by the #BOOLEAN. The original #OPTIONAL encoding procedure is now defined as the replacement of the original component with this mandatory structure, using the original optional component as the actual parameter. (D.3.2 is a more complete example of this process).

9.11.4 Dummy parameters may be encoding objects, encoding object sets, encoding classes, references to encoding structure fields, and values of any of the ASN.1 types used in the built-in encoding classes defined in clause 23, as specified in ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by C.1 of the present document.

9.11.5 The modification of parameterization syntax that is specified in annex C requires the use of the symbol "<" (without spaces) instead of "{" to start a dummy or actual parameter list, and of ">" to end one.

NOTE: This was done to make parsing of ECN syntax easier for computers, and to avoid ambiguity when user-defined classes are used in structure definitions in place of #SEQUENCE, #CHOICE, #REPETITION, #SEQUENCE-OF, or #SET-OF.

9.12 Governors

9.12.1 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.

9.12.2 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.

9.12.3 The ECN syntax requires governors that are encoding classes to be class reference names, or parameterized class reference names.

9.12.4 If the governed notation is a reference name for an encoding object, then that encoding object is required to be of the same class as the governor (see 17.1.6).

9.13 General aspects of encodings

9.13.1 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 (see 9.13.3), or by reaching the end of a length-delimited container (or the end of the PDU) before the optional element appears.

9.13.2 Similarly, it recognizes that delimitation of repetitions can be done (for example) by:

- Some form of length count.
- Detecting the end of a container (or PDU) in which it is the last item.
- Use of an identification handle on each of the repetitions and on following encodings (see 9.13.3).
- Some terminating pattern that can never occur in an encoding in the repeated series. (A simple example is a null-terminated character string).
- Use of a "more bit" with each element, set to one to indicate that another repetition follows, and set to zero to indicate the end of the repetition.

ECN supports all these mechanisms for delimitation of repetitions, and similar mechanisms for identification of alternatives and for resolution of optionality.

9.13.3 In addition to terminating repetitions, the identification handle technique can also be used to determine the presence of optional elements or of alternatives. The mechanism is similar in all these cases. Encodings for all values of any given "possible next class" encoding will have the same bit-pattern (their identification) at some place in their encoding (the handle), but the identification for different "possible next class" encodings will be different for each one. All such encodings can be interpreted by a decoder as an encoding of any "possible next class", and the identification for the handle will determine which "possible next class" encoding is present. The concept is similar to that of using tags for such purposes in BER. Identification handles have names that are required to be unique within an ECN specification.

9.13.4 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 successfully 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 could also 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.

9.14 Identification of information elements

9.14.1 Many protocols have an encoding (usually of a fixed number of bits) to identify what are often called "information elements" or "data elements" in a protocol. These identifications correspond roughly to ASN.1 tags, but are usually less complex. They are often used as identification handles, but are not always so used.

9.14.2 ECN contains a #TAG class to support the definition of the encoding of information element identifiers through use of the ASN.1 tag notation. (It also supports the inclusion of such elements within an encoding structure with no reference to ASN.1 tags).

9.14.3 When an encoding structure is implicitly generated from an ASN.1 type (see clause 11), any **textually-present** ASN.1 tag notation generates an instance of the #TAG class, with the number of the ASN.1 tag associated with that instance of the #TAG class. An encoding for this encoding class can be defined in a similar way to an encoding for the #INTEGER class, and will encode the number in the tag notation.

9.14.4 The full ASN.1 tag-list (multiple tags each with a class and number) is notionally associated with all the abstract values of a tagged type, in accordance with the ASN.1 model. Such information is, however, only accessible in the current version of ECN through a non-ECN definition of an encoding object (see 9.7.8). The generation of a #TAG class is a separate mechanism, is simpler and more specific, and has full support within ECN.

9.14.5 It is, however, important to note that for the purposes of generating a #TAG class, it is only textually-present tag notation that is visible. Universal class tags and tags generated by automatic tagging are not visible. Similarly, the class of any textually present tag notation is ignored. Only the tag number is available to encoding objects of the #TAG class.

9.15 Reference parameters and determinants

9.15.1 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.

9.15.2 A new concept - a **reference parameter** - is introduced to satisfy the need for a dummy parameter that references an encoding structure field. The governor is the reserved word "REFERENCE", and the allowed notation for an actual parameter with this governor is any encoding structure field name within the encoding structure to which an encoding object or encoding object set with such a parameter is being applied (see 17.5.16). (See D.1.11.3 for an example of references to encoding structure fieldnames).

9.16 Replacement classes and structures

9.16.1 When writing ASN.1 specifications for legacy protocols (or in order to generate specialized encodings for new protocols), it is normal to ignore encoding issues and, in particular, determinant fields that are present solely to support decoding. Only fields of relevance to application code (carrying application semantics) are included in the ASN.1 specification.

9.16.2 When such protocols use more than one encoding mechanism to support (for example) "SEQUENCE OF" constructions in different places in the protocol, it is not possible (nor would it be appropriate) to formally specify this within the ASN.1 itself.

9.16.3 This means that the implicitly generated encoding structure will not distinguish between such constructions, nor will it contain encoding-related fields for determinants, and it is necessary to modify it to "correct" both problems before a structure is available that matches the encoding requirements.

9.16.4 The first and simplest modification is to replace some instances of a class (within the implicitly generated structure) with new class names that have been assigned the old class in a class assignment statement. This is done by creating an **explicitly generated structure** using a renames clause in an EDM. This clause imports an implicitly generated structure from an ASN.1 module and makes specified replacements of (textual) occurrences of named classes. The replacement can be of all occurrences textually within a list of implicitly generated classes (corresponding to the ASN.1 type definitions in a module), or within components of one of those classes, or "all occurrences except" those in a given definition or a given component (see 15.3). It is important here to note that these replacements are restricted to the use of classes that have been defined with an encoding class assignment statement that assigns the name of a replacement class to an old class (for example: "#Replacement-class ::= #Old-class"), so this mechanism is sometimes colloquially referred to as "coloring". The "coloring" identifies those parts of the specification that require different encodings from other parts. (An example of "coloring" is given in D.3.8).

9.16.5 Even with "coloring", the explicitly generated encoding structure, like the implicitly generated encoding structure, contains only fields corresponding to the fields in the ASN.1 specification, and it is usually necessary to modify the generated structures to add fields for determinants, etc. A new **replacement structure** is needed (for all or part of the original structure), with added fields. It is also important to identify (for each field in the original structure) which fields of the replacement structure (and what abstract values of that field) are used to carry the semantics of the original abstract values. We talk about mapping the abstract values from the original structure to the replacement structure.

9.16.6 There are many mechanisms for defining an encoding object for an existing structure as an encoding object for a totally different replacement structure, with defined **value mappings** between the old structure and the replacement structure. These mechanisms are described in 9.17.

9.16.7 A simpler situation frequently occurs, however, in which the designer requires the old structure to form (in its entirety) a single component of the replacement structure, with all abstract values being mapped from the old structure to the corresponding value of that component of the replacement structure. For this mechanism to be of general use, the replacement structure needs to have a dummy parameter for this single component, and for it to be instantiated with the actual parameter set to the old structure. This was described in 9.11.3.

9.16.8 When defining encoding objects for a class (any class), it is always possible to specify that the first action of that encoding object is to replace the class it is encoding with a parameterized replacement structure, instantiated as described in 9.16.7, and with abstract values mapped from the old class to the component.

9.16.9 It is also possible to define encoding objects for the #OPTIONAL class (or for any class of the optional category) that replace the optional element with a parameterized replacement structure (frequently one containing a #BOOLEAN field as a presence determinant). (An example of this is given in D.3.2.3).

9.16.10 For constructor classes such as #CONCATENATION, #REPETITION, and so on, it is also possible to define encoding objects that replace not the entire structure, but each component separately (or just mandatory, or just optional, components).

9.16.11 A more advanced, but powerful, mechanism is to require the replacement action to also include the insertion of a specified field at the head or at the end of a #CONCATENATION (or similar structure). In this case the replacement structure will contain a further dummy parameter which is a "REFERENCE" parameter. The instantiation of the replacement structure has the corresponding actual parameter set to a pointer to the inserted structure. (An example of this is given in D.3.1.5).

9.17 Mapping abstract values onto fields of encoding structures

9.17.1 There are six mechanisms provided for this.

9.17.2 The first is to map specified abstract values associated with one simple encoding class to specified abstract values associated with another simple encoding class. This can be used in many ways. For example, values of a character string (of digits) can be mapped to integer values (and hence encoded as integer values). Values of an enumerated type can be mapped to integer values, and so on (see 19.2). (See D.1.10.2 for an example).

9.17.3 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 19.3). (See D.2.8.3 for an example).

9.17.4 The third is to map by transforming all the abstract values associated with one encoding class into abstract values associated with a different (typically, but not necessarily) encoding class, using a transform function. With this mechanism, it is, for example, possible to map an #INTEGER into a #CHARS to obtain characters that can then be encoded in whatever way is desired (for example, Binary-Coded Decimal or ASCII). (See D.1.6.3 for an example). Transform functions are encoding objects of the class #TRANSFORM. They not only can transform between different encoding classes, but can also be used to define simple arithmetic functions such as multiplication by a fixed value, subtraction of a fixed value, and so on. When applied in succession, they enable general arithmetic to be specified (see 19.4). (See D.2.4.2 for an example).

9.17.5 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 19.5). (See D.1.4.2 for an example).

9.17.6 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 19.6 and D.2.1.3 for examples).

9.17.7 The final mechanism allows the ECN specifier to provide an explicit 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 E 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 (see 19.7).

9.18 Contents of Encoding Definition Modules

9.18.1 Encoding Definition Modules (or EDMs) 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 implicitly generated encoding structures).

9.18.2 An EDM can also contain a renames clause (see clause 15) which references implicitly generated encoding structures from one or more ASN.1 modules and generates, by "coloring" them (see 9.16.4), an explicitly generated encoding structure for each one. These explicitly generated encoding structures are available for use within the EDM, but are also automatically exported for possible import into the Encoding Link Module.

9.18.3 It would be unusual (but not illegal) for the same implicitly generated encoding structure to be referenced in both an imports clause and in a renames clause in an EDM. In this case, any reference to either of these classes in the body of the EDM requires the use of a fully-qualified name (see "ExternalEncodingClassReference" in 10.6).

9.18.4 The body of an EDM module contains:

- "EncodingObjectAssignment" statements that define and name an encoding object for some encoding class (there are seven forms of this statement, discussed in 9.7 and defined in clause 17).
- "EncodingObjectSetAssignment" statements that define sets of encoding objects (see clause 17).
- "EncodingClassAssignment" statements that define and name new encoding classes (see clause 15).

9.18.5 The EDM can also contain parameterized versions of these statements, as specified in clause 14 and in C.1.

9.18.6 Encoding objects can be defined for built-in encoding classes within any EDM module. Encoding objects can be defined for a generated encoding structure only in EDM modules that import the implicitly generated encoding structure from the ASN.1 module that defines the corresponding type (using either an imports or a renames clause), or that import the generated encoding structure from an EDM module that has exported it.

NOTE: If an implicitly generated encoding structure happens to have a name that is the same as a reserved encoding class name (see 8.5), it can still be imported into an EDM, but must be referenced in the body of the EDM using a fully-qualified name (see "ExternalEncodingClassReference" in 10.6).

9.19 Contents of the Encoding Link Module

9.19.1 All applications of the Encoding Control Notation require the identification of a single Encoding Link Module (or ELM).

9.19.2 The ELM module applies encoding object sets to ASN.1 types (formally, to a generated encoding structure corresponding to the ASN.1 type). These encoding object sets (or their constituent encoding objects) are imported into the ELM module from one or more EDM modules.

9.19.3 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 (see 12.2.5). For example, it is not permitted for an ELM to apply more than one encoding object set to a specific implicitly generated structure.

9.19.4 It is possible in simple cases for an ELM module to contain just a single statement (following an imports clause) that applies an encoding object set to the implicitly generated encoding structure corresponding to the single top-level type of an application. (See D.1.15 for an example).

9.20 Defining encodings for primitive encoding classes

9.20.1 Encoding rules for some primitive encoding classes can be defined using a user-friendly syntax which is specified in the "WITH SYNTAX" statements of encoding class definitions (see clauses 23 and 25). This syntax can also be used to define encoding rules for encoding classes derived from these primitive encoding classes (by encoding class assignment statements).

9.20.2 The notation used for the encoding class definitions in clauses 23 and 25 is based on the notation used for information object class definition. This syntax (and its associated semantics) is defined by reference to ITU-T Rec. X.681 | ISO/IEC 8824-2 as modified by annex B of the present document.

9.20.3 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.

9.20.4 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 encoding parameters which collectively define an encoding.

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

9.20.5 The meaning of the values of these encoding parameters is specified using an encoding model (see figure 1) where the value of each bit-field class produces a **value encoding** which is placed (left or right justified) into an **encoding space**.

9.20.6 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. If the size of the encoding space is variable, then either the value encoding has to be self-delimiting, or there has to be some external mechanism to enable a decoder to determine the size of the encoding space. Several mechanisms are available for this determination.

9.20.7 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.

9.20.8 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 required bit-reversals.

9.20.9 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.

9.20.10 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.

9.20.11 It is also possible to specify an arbitrary amount of encoder pre-padding (beyond that needed for alignment) that ends when the value of an earlier **start pointer** field identifies the start an element.

9.20.12 The steps in a definition of an encoding for a primitive bit-field 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). (See 22.2).
- Specify the form of any necessary padding to that point (encoding space pre-padding). (See 22.2).
- Specify (if necessary) a field that provides a pointer to the start-point of the encoding space. (See 22.3).
- 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). (See 22.4).
- Specify the size of the encoding space in these units. This may be 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. (See 22.4).

- Specify the alignment of the value encoding within the encoding space. (See 22.8).
- Specify the form of any necessary padding from the start of the encoding space to the start of the value encoding (value pre-padding). (See 22.8).
- Specify the form of any necessary padding between the end of the value encoding and the end of the encoding space (value post-padding). (See 22.8).
- Specify any necessary bit-reversals of the encoding space contents before adding the bits to the encoding done so far. (See 22.12).

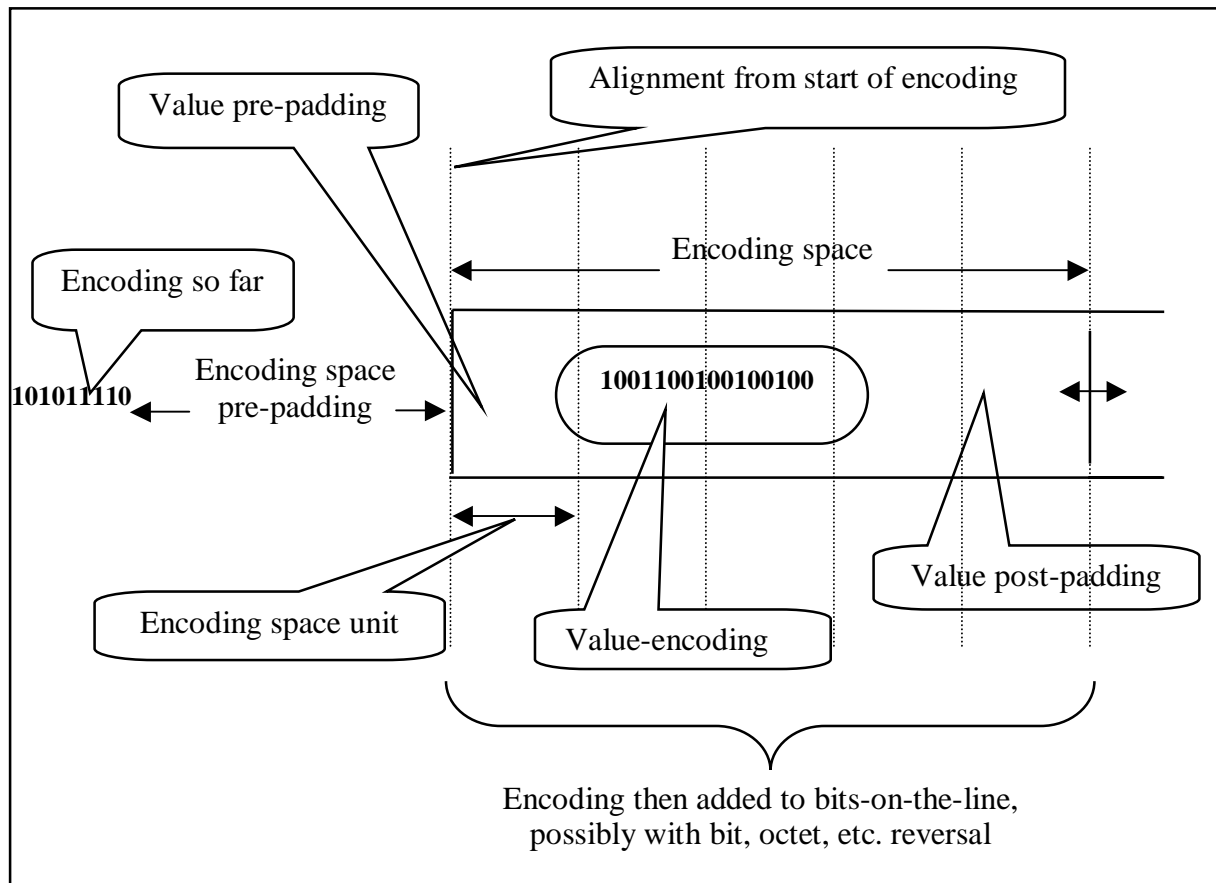


Figure 1: Encoding space, value-encoding and padding concepts

9.20.13 Encoding parameters are available to support the specification of the encoding rules for all these steps.

9.20.14 In real cases, only some (or none!) of these encoding parameters will have unusual values, and defaults operate if they are not specified. (See D.1.3 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).

9.21 Application of encodings

9.21.1 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 encodings (to an encoding structure generated from an ASN.1 type definition) only occurs within an Encoding Link Module, but application of encodings to fields of an encoding structure may be used in the definition of encodings for a larger encoding structure.

9.21.2 Encodings are applied by reference to an encoding object set (or to a single encoding object). Such application can occur in an EDM in the definition of encoding objects for any class (including encoding objects for a generated encoding structure and for a user-defined encoding structure). Such application in an EDM is merely the definition of more encoding objects for that encoding class: The definitive application to an actual type occurs only in the ELM.

9.21.3 When a set of encoding objects is being applied, it always results in a complete encoding specification for the encoding classes to which the objects are applied. If, in any given application, encodings are needed for encoding classes (present within an encoding structure being encoded) for which there are no encoding objects in the set being applied, then this is an error (see 13.2.11).

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

9.21.4 There are two exceptions to 9.21.3. 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 actual parameters. 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 implicitly generated encoding structure "#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 type "My-Type".

9.22 Combined encoding object set

9.22.1 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".

9.22.2 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 (see 13.2). A frequent set to use with "COMPLETED BY" is the built-in set "PER-BASIC-UNALIGNED". (See D.1.15 for an example of the application of a combined encoding object set).

9.22.3 While an encoding object set can contain only one encoding object for a class #SEQUENCE-OF (for example), it can also contain an encoding object for a class #Special-sequence-of (for example) which is defined as "#Special-sequence-of ::= #SEQUENCE-OF". An explicitly generated encoding structure can have both the #SEQUENCE-OF class and also the #Special-sequence-of class in its definition. In this way, a single combined encoding object set can be applied to produce standard encodings for some of the original "SEQUENCE OF" constructs, and specialized encodings for others.

9.23 Application point

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

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

9.23.3 If there is an encoding object in the combined encoding object set that matches a bit-field encoding class (initially a generated encoding structure) at the application point, it is applied and the process terminates. Otherwise the class at the application point is "expanded" by de-referencing. This expansion by de-referencing will continue until either an encoding object is found, or a primitive class is reached. If the class at the application point is an encoding constructor, and there is an encoding object for that encoding constructor (#CHOICE, #SEQUENCE, #SEQUENCE-OF, etc.), then it is applied, and the application point then passes to each component (as a parallel activity).

9.23.4 In a more complex case, there may be an #OPTIONAL class following a component class (and one or more #TAG classes preceding it). The application point passes first to the #OPTIONAL, and the encoding object for that class may replace the component (see 9.16.9). Then the application point passes to the tags, and finally to the component itself.

9.24 Conditional encodings

9.24.1 Mention has already been made of the #TRANSFORM encoding class as a means of performing simple arithmetic on integer values (see 9.17.3). This encoding class does, however, play a more fundamental role in the specification of encodings for some primitive classes. In general, the specification of encodings for many of the ASN.1 built-in 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.

9.24.2 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 clauses 24, 23.7 and 23.13 respectively, or by non-ECN definition of an encoding object, 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 (see 18.1.6).

9.24.3 Encoding specification for encoding classes in the integer category proceeds as follows: Encodings (of the #CONDITIONAL-INT encoding class) are defined for a particular **bounds condition**, specifying the container size (and how it is delimited), the transform of the integer to bits (using either two's complement or positive integer encodings), 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**. The encoding of the class in the integer category 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. (See D.1.5.4 for an example).

9.24.4 Encoding specification for encoding classes in the repetition category 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 encoding of classes in the integer category, the final encoding is defined as a list of conditional encodings.

9.24.5 Encoding specification for the encoding classes in the octets category proceeds as follows: First, #TRANSFORM encoding objects are defined to map a single octet to a self-delimiting bitstring. Second, one or more #CONDITIONAL-REPETITION encoding objects (for specific size-range conditions) are defined to take each of the bitstrings (transformed from an octet in the octet string) and to concatenate them into a delimited container (the definition of such encoding objects is not specific to encoding #OCTETS). The final encoding of the class in the octets category is defined as a list of #CONDITIONAL-REPETITION encoding objects. (See D.1.8.2 for an example).

9.24.6 Encoding specifications for encoding classes in the bitstring category proceeds as follows: First, #TRANSFORM encoding objects are defined to map a single bit into a bitstring, 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 an encoding class in the repetition or octetstring categories). Finally, the encoding of the class in the bitstring category is defined as a list of the #CONDITIONAL-REPETITION encoding objects. (See D.1.7.3 for an example).

9.24.7 Encoding specifications for encoding classes in the characterstring category proceeds as follows: First, #TRANSFORM encoding objects are defined to map a single character to a self-delimiting bitstring, using several possible mechanisms for defining the encoding of the character, and using the effective alphabet constraint where it is available. Secondly, one or more #CONDITIONAL-REPETITION encoding objects are defined, and finally the encoding of the class in the characterstring category is defined as a list of these. (See D.1.9.2 for an example).

9.25 Changes to ASN.1 Recommendations | International Standards

9.25.1 The present document 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.

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

9.25.3 Finally, the rules for parameterization are relaxed to allow a dummy parameter of an encoding object reference (being assigned in an assignment statement) to be used as an actual parameter of the encoding class reference which governs the notation defining the encoding object reference name. In particular, a parameterized encoding class can be used as a governor in an encoding object assignment statement (see C.2/8.4), with the actual parameter being a dummy parameter of the encoding object that is being defined.

9.25.4 These modifications to other ASN.1 Recommendations | International Standards are specified in annexes A to C, and are solely for the purposes of the present document.

10 Identifying encoding classes, encoding objects, and encoding object sets

10.1 Many of the productions within the present document require that an encoding class, encoding object, or encoding object set be identified.

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

- a) using a simple reference name;
- b) using a built-in reference name (not applicable for encoding objects, as there are no built-in encoding objects);
- c) using an external reference (also called a fully-qualified name);
- d) using a parameterized reference;
- e) in-line definition.

NOTE: The parameterized reference form may be used with a simple reference name or with an external reference (see C.3).

10.3 There are productions (or lexical items) for all of these means of identification. There are also productions that allow several alternatives. These lexical items or production names are used where appropriate in other productions, and are defined in the remainder of this clause.

10.4 The lexical items for use of a simple reference name are:

encoding class	"encodingclassreference" (see 8.3)
encoding object	"encodingobjectreference" (see 8.1)
encoding object set	"encodingobjectsetreference" (see 8.2)

10.4.1 An "encodingclassreference" is a name which is either:

- a) assigned an encoding class in an "EncodingClassAssignment" (see clause 16); or is
- b) imported into an EDM from some other EDM from which it has been exported; or is
- c) imported as the name of an implicitly generated encoding structure from an ASN.1 module (see 14.11), or from an EDM module into which it was imported; or is
- d) generated by a renames clause in the EDM (see clause 15).

NOTE: Only classes that are generated encoding structures can be imported into an EDM (see 12.1.8).

10.4.2 An "encodingclassreference" shall not be imported from an EDM module (as specified in 10.4.1) unless either:

- a) it is defined in or imported into the referenced module, and that module has no exports clause; or

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

- b) it is defined in or imported into the referenced module, and appears as a symbol in the exports clause of that module; or
- c) it is one of the reference names explicitly generated by a renames clause in the module from which it is being imported.

10.4.3 An implicitly generated encoding structure reference never appears in the exports clause of any ASN.1 module, but can always be imported from any ASN.1 module in which the corresponding type is defined and exported.

10.4.4 An implicitly generated encoding structure reference shall not appear in the exports clause of the EDM module in which it is generated, but any use of it in another EDM or the ELM requires its importation.

10.4.5 An "encodingobjectreference" is a name which is either:

- a) assigned an encoding object in an "EncodingObjectAssignment" (see clause 17) 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.

10.4.6 An "encodingobjectreference" shall not be imported from an EDM if 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.

10.4.7 An "encodingobjectsetreference" is a name which is either:

- a) assigned an encoding object set in an "EncodingObjectSetAssignment" (see clause 18) 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 set or is imported.

10.4.8 An "encodingobjectsetreference" shall not be imported from an EDM if the referenced module has an exports clause and the "encodingobjectsetreference" 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.

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

encoding class	"BuiltinEncodingClassReference" (see 16.1.6)
encoding object set	"BuiltinEncodingObjectSetReference" (see 18.2.1)

10.6 The productions for use of an external reference name are:

```

ExternalEncodingClassReference ::=
    modulereference "." encodingclassreference      |
    modulereference "." BuiltinEncodingClassReference

ExternalEncodingObjectReference ::=
    modulereference "." encodingobjectreference

ExternalEncodingObjectSetReference ::=
    modulereference "." encodingobjectsetreference

```

10.6.1 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.

10.6.2 The "ExternalEncodingClassReference" alternative that includes a "BuiltinEncodingClassReference" shall be used in the body of an EDM if and only if there is a generated encoding structure (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" (see 11.4.1); or
- b) imported into another EDM referenced by the "modulereference" and exported from that module; or
- c) generated in a renames clause of another EDM referenced by the "modulereference"; or
- d) generated in this EDM in a renames clause, in which case the "modulereference" shall refer to this EDM.

NOTE: The "BuiltinEncodingClassReference" name can appear as a "Symbol" in the imports clause (see A.1).

10.6.3 The productions defined in 10.6 (except as specified in 10.6.2) 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 have been generated in a renames clause in this EDM, or have been both imported and generated; or
- b) the simple reference name is a "BuiltinEncodingClassReference" (see 10.5); or
- c) both conditions hold.

10.7 A parameterized reference is a reference name defined in a "ParameterizedAssignment" (see C.1) and supplied with an actual parameter in accordance with the syntax of C.3. The productions involved are:

encoding classes	"ParameterizedEncodingClassAssignment" (see C.1) "ParameterizedEncodingClass" (see C.3)
encoding objects	"ParameterizedEncodingObjectAssignment" (See C.1) "ParameterizedEncodingObject" (See C.3)
encoding object sets	"ParameterizedEncodingObjectSetAssignment" (See C.1) "ParameterizedEncodingObjectSet" (See C.3)

10.8 The productions that allow all forms of identification are:

encoding classes	"EncodingClass" (See clause 16.1.4)
encoding objects	"EncodingObject" (See clause 17.1.4)
encoding object sets	"EncodingObjectSet" (See clause 19.1)

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

encoding classes	"DefinedEncodingClass" and "DefinedOrBuiltinEncodingClass"
encoding objects	"DefinedEncodingObject"
encoding object sets	"DefinedEncodingObjectSet" and "DefinedOrBuiltinEncodingObjectSet"

except that built-in encoding classes and built-in encoding object sets are not allowed by "DefinedEncodingClass" and "DefinedEncodingObjectSet".

NOTE: A further production "SimpleDefinedEncodingClass" is also used. This is defined in C.3 and allows only "encodingclassreference" and "ExternalEncodingClassReference".

10.9.1 The "DefinedEncodingClass" and "DefinedOrBuiltinEncodingClass" are:

DefinedEncodingClass ::=	
Encodingclassreference	
ExternalEncodingClassReference	
ParameterizedEncodingClass	
DefinedOrBuiltinEncodingClass ::=	
DefinedEncodingClass	
BuiltinEncodingClassReference	

10.9.2 The "DefinedEncodingObject" is:

DefinedEncodingObject ::=	
encodingobjectreference	
ExternalEncodingObjectReference	
ParameterizedEncodingObject	

10.9.3 The "DefinedEncodingObjectSet" and "DefinedOrBuiltinEncodingObjectSet" are:

DefinedEncodingObjectSet ::=	
Encodingobjectsetreference	
ExternalEncodingObjectSetReference	
ParameterizedEncodingObjectSet	
DefinedOrBuiltinEncodingObjectSet ::=	
DefinedEncodingObjectSet	
BuiltinEncodingObjectSetReference	

11 Encoding ASN.1 types

11.1 General

11.1.1 For all ASN.1 types, there is a corresponding implicitly generated 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, and is called an **implicitly generated encoding structure**.

11.1.2 There may also be one or more **explicitly generated encoding structures**. These are generated in an EDM using a `renames` clause.

11.1.3 The encoding of an ASN.1 type is formally defined as the result of encodings applied to precisely one of the encoding structures (implicitly or explicitly) generated from the ASN.1 type. The encodings are applied by statements in the ELM (see clause 12), using encoding objects in a combined encoding object set. An ELM shall apply encodings to at most one of the generated encoding structures corresponding to any given ASN.1 type.

11.1.4 The implicitly generated encoding structure is defined by first simplifying and expanding the ASN.1 notation (as specified in 11.3), and then by mapping ASN.1 types, type constructors and component names into corresponding built-in encoding classes, encoding constructors and encoding structure fieldnames.

11.1.5 An explicitly generated encoding structure is defined by making specified changes to the implicitly generated encoding structure using a `renames` clause.

11.1.6 Each field of a generated encoding structure has associated with it the abstract values of the corresponding type, and constraint-related information derived from the ASN.1 type definition (see 11.4.2). Encodings of the abstract values of the generated encoding structure are defined to be the encodings for the corresponding abstract values of the original ASN.1 type.

11.1.7 This clause 11 specifies:

- a) The built-in encoding classes that are used in defining the implicitly generated encoding structures corresponding to ASN.1 types (see 11.2).

NOTE: Clause 16.1.14 specifies additional classes that are used in the definition of user-defined encoding structures.

- b) Transformations of the ASN.1 syntax (simplification and expansion) before the implicitly generated structure are produced (see 11.3).
- c) The implicitly generated encoding structure for any ASN.1 type (see 11.4).

11.2 Built-in encoding classes used for implicitly generated encoding structures

11.2.1 The encoding classes used for implicitly generated encoding structures, and the ASN.1 types or constructors to which they correspond are listed in table 2.

11.2.2 Column 1 gives the ASN.1 notation which is replaced by an encoding class in the implicitly generated encoding structure. Column 2 gives the encoding class that replaces the column 1 notation. Column 3 gives the primitive class that the column 2 class is derived from.

Table 2: Encoding classes for ASN.1 notation

ASN.1 notation	Encoding Class	Primitive Class
BIT STRING	#BIT-STRING	#BITS
BOOLEAN	#BOOLEAN	#BOOL
CHARACTER STRING	#CHARACTER-STRING	#OPEN-TYPE
CHOICE	#CHOICE	#ALTERNATIVES
EMBEDDED PDV	#EMBEDDED-PDV	#OPEN-TYPE
ENUMERATED	#ENUMERATED	#INT
EXTERNAL	#EXTERNAL	#OPEN-TYPE
INTEGER	#INTEGER	#INT
NULL	#NULL	#NULL
OBJECT IDENTIFIER	#OBJECT-IDENTIFIER	#OBJECT-IDENTIFIER
OCTET STRING	#OCTET-STRING	#OCTETS
open type notation	#OPEN-TYPE	#OPEN-TYPE
OPTIONAL	#OPTIONAL	#OPTIONAL
REAL	#REAL	#REAL
RELATIVE-OID	#RELATIVE-OID	#OBJECT-IDENTIFIER
SEQUENCE	#SEQUENCE	#CONCATENATION
SEQUENCE OF	#SEQUENCE-OF	#REPETITION
SET	#SET	#CONCATENATION
SET OF	#SET-OF	#REPETITION
GeneralizedTime	#GeneralizedTime	#CHARS
UTCTime	#UTCTime	#CHARS
BMPString	#BMPString	#CHARS
GeneralString	#GeneralString	#CHARS
GraphicString	#GraphicString	#CHARS
IA5String	#IA5String	#CHARS
NumericString	#NumericString	#CHARS
PrintableString	#PrintableString	#CHARS
TeletexString	#TeletexString	#CHARS
UniversalString	#UniversalString	#CHARS
UTF8String	#UTF8String	#CHARS
VideotexString	#VideotexString	#CHARS
VisibleString	#VisibleString	#CHARS
Textually present tag notation	#TAG	#TAG

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

11.3.1 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 to the simpler constructions.

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

- fully-defined in clause 11.3.4 below; or
- referenced in those clauses as "See 11.3.2 b" and fully-defined in ITU-T Rec. X.680 | ISO/IEC 8824-1 (including annex F) with all published amendments and technical corrigenda; or
- referenced in those clauses as "See 11.3.2 c" and fully-defined in ITU-T Rec. X.681 | ISO/IEC 8824-2 with all published amendments and technical corrigenda.
- referenced in those clauses as "See 11.3.2 d" and fully-defined in ITU-T Rec. X.683 | ISO/IEC 8824-4 with all published amendments and technical corrigenda.

11.3.3 The ASN.1 syntactic constructs removed by the expansions and simplifications below are not referenced further in the present document.

11.3.4 The following expansions and simplifications shall be applied to all ASN.1 modules:

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

- a) All "ValueSetTypeAssignment"s shall be replaced by their equivalent "TypeAssignment"s with subtype constraints. (See 11.3.2 b).
- b) The ASN.1 "INSTANCE OF" construction shall be expanded into its equivalent sequence type. (See 11.3.2 c).
- c) "TypeFromObject" shall be replaced with the type that is referenced. (See 11.3.2 c).
- d) "ValueSetFromObjects" shall be replaced with the type that is referenced. (See 11.3.2 c).

11.3.4.2 The following transformations shall be applied recursively in the specified order, until a fixed-point is reached:

- a) All ASN.1 parameterization shall be fully resolved by the substitution of actual parameters for dummy parameters. (See 11.3.2 d).

NOTE: This means that where ASN.1 type notation contains an instantiation of an ASN.1 parameterized type, that instantiation becomes an inline definition.

- b) All "ComponentsOf"s shall be expanded to their full form. (See 11.3.2 b).
- c) All uses of "SelectionType" shall be resolved. (See 11.3.2 b).

11.3.4.3 The following transformations shall then be applied:

- a) Named number lists in integer type definitions shall be removed. Named numbers are not visible to ECN. ECN sees a single #INTEGER class (possibly with bounds as specified in 11.3.4.3 c).
- b) Named bit lists in bitstring definitions shall be removed. Named bits are not visible to ECN.
- c) 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).
- d) If there is a contents constraint with a "CONTAINING" construction, then the existence of the contents constraint, its contents type, and the presence or absence of an "ENCODED BY" clause become properties associated with the abstract values of such a constrained octetstring or bitstring type, and the constraint shall then be discarded. If there is a contents constraint with no "CONTAINING" construction, then it is not visible to ECN and shall 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 (see 11.3 and 13.2).

- e) All tagging which is not textually present in the ASN.1 notation shall be ignored in the mapping to encoding structures, but (in order to model BER encodings and PER procedures) the full tag-list of a type becomes a property of the field of the encoding structure to which the corresponding values are mapped.
- f) Textually present tag notation has the class of the tag removed.
- g) "DEFAULT Value" shall be replaced by "OPTIONAL-ENCODING #OPTIONAL" and the default value is associated with the field of the structure to which the ASN.1 component is mapped.
- h) "OPTIONAL" shall be replaced by "OPTIONAL-ENCODING #OPTIONAL".
- i) "T61String" shall be replaced by #TeletexString.
- j) "ISO646String" shall be replaced by #VisibleString.

11.3.4.4 Finally, the following transformations shall then be applied:

- a) 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. (See 11.3.4.3 c).

NOTE 1: The #ENUMERATED class de-references to the #INT class (see 11.2.2), and the enumerations map into bounded integer values of the class. The actual names of enumerations are not visible to ECN.

- b) All occurrences of "EncodingClassFieldType" that refer to a type field, a variable-type value field, or a variable-type value set field shall be replaced by the #OPEN-TYPE encoding class. (See 11.3.2 c).
- c) Extensibility markers and version brackets in sequence, set and choice constructions are removed, but (in order to model BER encodings and PER procedures) the identification of a component as part of the root or of version 1, version 2, etc becomes a property of the component, and the existence of the extensibility marker becomes a property of the class the construction maps to.
- d) The extensibility marker in constraints is removed, but the existence of the extensibility marker becomes a property of the class and whether an abstract value is in the root or is in an extension becomes a property of the abstract value.

NOTE 2: The properties referenced in items c) and d) above can only be interrogated through non-ECN definition of encoding objects in this version of the present document. Full support for extensibility is expected to be provided in a later version of the present document.

11.3.5 With these transformations, all ASN.1 type-related constructs have corresponding encoding classes, listed in table 2. The implicitly generated encoding structure shall be constructed by mapping the ASN.1 type-related constructs in column 1 to the classes in column 2 of table 2 (as specified in 11.4).

11.4 The implicitly generated encoding structure

11.4.1 There is an implicitly generated structure for each ASN.1 type definition with a name constructed from the ASN.1 type reference name by the pre-fixing of a "#" character. Where a fully-qualified name is required for an implicitly generated encoding structure, that fully-qualified name shall include the "ModuleIdentifier" of the ASN.1 module containing the type definition. (An example of an implicitly generated structure is given in D.1.9.2).

NOTE: An implicitly generated structure is generated and exported for each ASN.1 type in an ASN.1 module whether or not that type is listed in the "EXPORTS" clause.

11.4.2 The implicitly generated 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 notation in column 1 of table 2 is mapped to the built-in encoding classes in column 2 of table 2.

NOTE 1: Each textually present tag maps into a "[#TAG]" construction in the implicitly generated structure.

- c) ASN.1 "DefinedType"s are mapped to an encoding class name derived from the typereference by the addition of a character "#". If a type is imported into the ASN.1 module, any "ExternalEncodingClassReference" to the corresponding class in an implicitly generated structure shall reference the ASN.1 module that contains the definition of the referenced type.

NOTE 2: If the resulting class is the name of a built-in encoding class, then all references to it in either the renames clause, or in the ELM, will use the "ExternalEncodingClassReference" notation.

- d) Abstract values are mapped from a field of the type definition to the corresponding field of the encoding structure.
- 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.
- f) The tag number of textually present tags maps to the corresponding #TAG class.

11.4.3 All implicitly generated encoding structures can be encoded by the built-in encoding object sets (see 18.2), and will produce the same encodings as are specified by the corresponding The present document for those encodings when applied to ASN.1 types.

12 The Encoding Link Module (ELM)

NOTE: There are two top-level productions in ECN, the "ELMDefinition" specified in this clause and the "EDMDefinition" specified in clause 14. These specify the syntax for defining the ELM and EDMs respectively.

12.1 Structure of the ELM

12.1.1 The "ELMDefinition" is:

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

12.1.2 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".

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

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

12.1.5 The "ELModuleBody" is:

```
ELModuleBody ::=
  Imports ?
  EncodingApplicationList

EncodingApplicationList ::=
  EncodingApplication
  EncodingApplicationList ?
```

12.1.6 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 A.1 of the present document.

12.1.7 The "ExternalEncodingClassReference" shall not be used in the "Imports" unless required by 15.1.6.1.

12.1.8 The "Imports" makes available within the ELM:

12.1.9 implicitly generated encoding structures from an ASN.1 module;

12.1.10 explicitly generated encoding structures from an EDM module;

NOTE: When an ELM imports an explicitly generated encoding structure from an EDM, the renames clauses in other EDMs have no effect on the encoding of that structure (see 15.2.4).

12.1.11 objects and encoding object sets from an EDM module.

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

12.2 Encoding types

12.2.1 An "EncodingApplication" is:

```
EncodingApplication ::=
    ENCODE
    SimpleDefinedEncodingClass "," +
    CombinedEncodings
```

12.2.2 An "EncodingApplication" defines the encoding of the ASN.1 types corresponding to the "SimpleDefinedEncodingClass"es which shall be generated encoding structures. The encoding of the types is specified by the "CombinedEncodings" applied to the generated encoding structures as specified in 13.2.

NOTE: 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.

12.2.3 Encodings applied to a generated encoding structure corresponding to an ASN.1 type defined in some ASN.1 module 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 ASN.1 module and imported into a different ASN.1 module.

12.2.4 The encoding of the type in a content constraint is that specified by the encoding object applied to the containing class in the octetstring or bitstring category, and can be any combined encoding object set, or can be the combined encoding object set that was applied to the containing class in the octetstring or bitstring category.

12.2.5 An ELM shall not apply encodings more than once to the same ASN.1 type.

NOTE: The rules of application of encodings (specified in clause 13) mean that an "EncodingApplication" completely defines the encoding of a type unless it contains an instance of a contents constraint.

13 Application of encodings

13.1 General

13.1.1 Encodings are applied by the ELM to a generated structure (or independently to multiple generated structures) using a "CombinedEncodings" definition as specified in 13.1.3. This clause, together with 13.2, specifies the application of "CombinedEncodings" to a generated encoding structure.

13.1.2 In the ELM, the application is to the generated encoding structures identified in the "EncodingApplication". Later clauses also specify the application of encodings to all or part of an arbitrary encoding structure definition. This clause is applicable in both cases.

13.1.3 The "CombinedEncodings" is:

```
CombinedEncodings ::=
    WITH
    PrimaryEncodings
    CompletionClause ?

CompletionClause ::=
    COMPLETED BY
    SecondaryEncodings

PrimaryEncodings ::= EncodingObjectSet

SecondaryEncodings ::= EncodingObjectSet
```

13.1.4 "EncodingObjectSet" is defined in 18.1.1.

13.1.5 The use of "CombinedEncodings" is specified in 13.2.

13.2 The combined encoding object set and its application

13.2.1 A **combined encoding object set** is formed from the "CombinedEncodings" production (see 13.1.3) as follows:

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

13.2.3 Otherwise,

- a) all encoding objects in the "PrimaryEncodings" are placed in the combined encoding object set, then
- b) 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 that has the same encoding class (see 17.1.6 and 9.22.2).

13.2.4 Following this conceptual construction of the combined encoding object set, encoding commences with the "encodingclassreference" name of the encoding structures identified in the encoding application (see 13.1.2 and 17.5).

13.2.5 Where there are several encoding applications in the ELM, the rules of 12.2 ensure that applications are non-overlapping. They proceed independently. Similarly, the applications of encodings to encoding structures in EDMs (specified in 13.2.10) are always non-overlapping. The following clauses provide the rules for application to a single encoding structure.

13.2.6 Encoding objects from the combined encoding object set are applied at an **application point**. The application point is initially the "encodingclassreference" for a generated encoding structure (when application is in the ELM, as specified in 13.1.2) or is a component of an encoding structure (when application is in an EDM, as specified in 17.5).

13.2.7 Any encoding class in the alternatives, concatenation, and repetition categories (see 16.1.8, 16.1.9 and 16.1.10) is an encoding constructor.

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

- a) the alternatives of a constructor that is in the alternatives category;
- b) the field following a constructor that is in the repetition category;
- c) the components of a constructor that is in the concatenation category;
- d) a contained type (a type specified in a contents constraint);
- e) the type chosen (in an instance of communication) for use with a class in the open-type category.

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

- a) An encoding class name. This is completely encodable using the specification in an encoding object of the same class (see 17.1.6).
- b) An encoding constructor (see 16.2.12). 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. The specification of the encoding object that is applied may require that one or more of the components of the constructor are replaced by other (parameterized) structures before the application point passes to the components.
- c) A class in the bitstring or octetstring category that has a contained type as a property associated with the values (see 11.3.4.3 d). 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 (see 22.11).
- d) A component which is an encoding class (possibly preceded by one or more classes in the tag category), followed by an encoding class in the optionality category. The procedures and encodings for determining presence or absence are determined by the specification contained in an encoding object of the class in the optionality category. This encoding object may also require the replacement of the encoding class (together with all its preceding classes in the tag category) with a (parameterized) replacement structure before that class is encoded. The application point then passes to the first class in the tag category (if any), or to the component, or to its replacement.

- e) An encoding class preceded by one or more encoding classes in the tag category. The tag number associated with the class in the tag category is encoded using the specification in an encoding object of the class in the tag category, and the application point then passes to the next tag, if any, or to the tagged class.
- f) Any other built-in encoding class. This is completely encodable using the specification contained in an encoding object of that class.

13.2.10 Encoding proceeds as follows:

13.2.10.1 If the combined encoding object set contains an encoding object of the same class (see 17.1.6) as the current application point, then that encoding object is applied. This application may cause replacement of one or more components of the class to which the encoding is being applied. If the combined encoding object set does not contain such an encoding object, then either:

- a) the encoding class at the current application point is a reference to another encoding class; in this case it is de-referenced, and the procedures of 13.2.10 are recursively applied; or
- b) the encoding class at the current application point is not a reference to another encoding class; in this case the ECN specification is in error.

13.2.10.2 If an encoding has been applied at the application point to the encoding class, and it is not in the optionality or tag category and does not have any components (see 13.2.7), then that application completely determines the encoding of the class and terminates these procedures.

13.2.10.3 If an encoding has been applied at the application point to an encoding class that is in the optionality category then the application point passes to the (possibly tagged) optional element.

13.2.10.4 If an encoding has been applied at the application point to an encoding class that is in the tag category then the application point passes to the next tag class or to the tagged element, and the procedures of 13.2.10 are recursively applied.

13.2.10.5 If an encoding has been applied at the application point to an encoding class that has components which are not a contained type, then the procedures of 13.2.10 are applied recursively to each component.

NOTE: This implies that the current combined encoding object set is applied to the type chosen (in an instance of communication) for use with a class in the open-type category (see 13.2.8e).

13.2.10.6 If an encoding has been applied to an encoding class at the application point that has a component that is a class in the bitstring or octetstring category with a contained type associated with the values, then there are four cases that can occur:

- a) 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 override an "ENCODED BY" (see 22.11). 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.
- b) 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.
- c) 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.
- d) 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.

13.2.10.7 If there is no encoding object in the combined encoding object set of the same class (see 17.1.6) as the current application point, and the current application point is a reference name, then it is de-referenced and these procedures are applied recursively to the new encoding structure.

13.2.10.8 Otherwise the ECN specification is in error.

13.2.11 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 bit-field class is reached, the encoding terminates with the encoding of that class, except that if it has a contained type, encoding proceeds to the generated encoding structure corresponding to the contained type. When a type with components is reached, the process continues by applying the combined encoding object set to each component independently. When tags and optionality are involved, the optionality class is encoded first, then the first tag class, and finally the element. When encodings are applied to constructor classes they may cause replacement of one or more components. When they are applied to an optionality class they may cause replacement of the entire element (apart from the optionality class, but including any tag classes).

13.2.12 In the encoding process, encoding objects applied to encoding constructors (and to classes in the optionality category) 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)

NOTE: 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.

14.1 The "EDMDefinition" is:

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

14.2 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 encoding object sets can be used in the ELM.

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

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

14.5 The "EDMModuleBody" is:

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

EDMAssignmentList ::=
  EDMAssignment
  EDMAssignmentList ?

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


14.6 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 A.1 of the present document.

14.7 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.

NOTE: When the name of an imported implicitly generated encoding structure is a built-in encoding class reference, it can be used within the EDM with a fully-qualified name, but cannot be exported from the EDM (however, encoding structures defined using it can, of course, be exported).

14.8 The production "RenamesAndExports" is defined in clause 15.

14.9 The "RenamesAndExports" (called the renames clause) makes available (within the EDM) explicitly generated encoding structures derived from the implicitly generated encoding structures in specified ASN.1 modules. It also makes these explicitly generated encoding structures available for import into other EDMs (and the ELM). (See clause 15).

14.10 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.

14.11 All ASN.1 modules that define non-parameterized type reference names automatically produce and export an implicitly generated encoding structure 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, as specified in A.1, has to be used in the body of the importing module, and in any renames clause.

14.12 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.

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

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

EncodingClassAssignment	Clause 16
EncodingObjectAssignment	Clause 17
EncodingObjectSetAssignment	Clause 18
ParameterizedAssignment	Clause C.1

NOTE: The "ParameterizedAssignment" allows the parameterization of an "EncodingClassAssignment", an "EncodingObjectAssignment", and an "EncodingObjectSetAssignment", as specified in C.1.

15 The renames clause

15.1 Explicitly generated and exported structures

15.1.1 The production "RenamesAndExports" is:

```

RenamesAndExports ::=
    RENAMES
    ExplicitGenerationList ";"
ExplicitGenerationList ::=
    ExplicitGeneration
    ExplicitGenerationList ?
ExplicitGeneration ::=
    OptionalNameChanges
    FROM GlobalModuleReference
OptionalNameChanges ::=
    NameChanges | GENERATES

```

NOTE: An example of the use of the renames clause to produce explicitly generated encoding structures is given in D.3.8.

15.1.2 The production "GlobalModuleReference" is defined in ITU-T Rec. X.680 | ISO/IEC 8824-1, 12.1 and shall identify an ASN.1 module.

15.1.3 The "RenamesAndExports" is called a renames clause.

15.1.4 Each "ExplicitGeneration" generates, and exports from this module, an explicitly generated encoding structure for each of the implicitly generated encoding structures of the ASN.1 module referenced by "GlobalModuleReference". Each field of the explicitly generated encoding structure has associated with it the same abstract values as the corresponding field of the implicitly generated encoding structure (which are those associated with the corresponding field of the ASN.1 type from which it was generated).

15.1.5 These explicitly generated encoding structures have the same simple reference name as the implicitly generated encoding structure from which they were formed (but are distinct classes). Where a fully-qualified name is required for an explicitly generated encoding structure, that fully-qualified name shall include the "ModuleIdentifier" of the EDM module containing the renames clause, as specified in 15.1.6.

NOTE: The implicitly generated encoding structures used in their generation have the same simple reference name, but their fully-qualified name includes the "ModuleIdentifier" of the ASN.1 module in which the corresponding type was defined.

15.1.6 If an EDM produces explicitly generated encoding structures from more than one ASN.1 module, it is possible that some of these structures may have the same simple encoding class names. In this case, the restrictions of 15.1.6.1 and 15.1.6.2 shall apply.

15.1.6.1 If any of these structures are imported into another EDM or into the ELM, then the imports clause shall reference them using the "ExternalEncodingClassReference" containing the "modulereference" used as the ASN.1 module reference in the replaces clause of the EDM module which generated them.

15.1.6.2 If any of these structures are referenced in the body of this EDM, then the reference shall be an "ExternalEncodingClassReference" containing the "modulereference" used as the ASN.1 module reference in the replaces clause of this EDM module.

15.1.7 The "ExternalEncodingClassReference" notation shall not be used in an imports clause except where required by clause 15.1.6.

15.1.8 If a name which has been imported using an "ExternalEncodingClassReference" is used in the body of a module, then the simple "encodingclassreference" can be used unless an "ExternalEncodingClassReference" is required as specified in clause 15.1.6.2.

15.1.9 If the "OptionalNameChanges" is "GENERATES", then all the explicitly generated encoding structures are the same structure as the implicitly generated encoding structures used in their generation, except as specified in 15.1.11.

15.1.10 If "OptionalNameChanges" is "NameChanges", then 15.1.11 still applies, but the explicitly generated encoding structures are further modified as specified in 15.2.

15.1.11 Consider an implicitly generated encoding structure (A say) which contains an encoding class reference to some other implicitly generated encoding structure (B say). Then:

- a) If this renames clause (in any of its "ExplicitGeneration"s) produces an explicitly generated encoding structure corresponding to B (B1 say), then the corresponding reference in the explicitly generated encoding structure corresponding to A is a reference to B1.
- b) If there is no explicitly generated encoding structure corresponding to B, then the reference in the generated encoding structure corresponding to A is a reference to B.

15.2 Name changes

15.2.1 The "NameChanges" production is:

```

NameChanges ::=
    NameChange
    NameChanges ?

NameChange ::=
    OriginalClassName

```



```

AS
NewClassName
IN
NameChangeDomain

OriginalClassName ::= SimpleDefinedEncodingClass | BuiltinEncodingClassReference

NewClassName ::= encodingclassreference

```

15.2.2 Each "NameChanges" specifies that, in the generation of explicitly generated encoding structures, all occurrences of "OriginalClassName" within "NameChangeDomain" in the implicitly generated encoding structures are to be renamed as the class "NewClassName". "NameChangeDomain" is specified in 15.3, and identifies one or more implicitly generated encoding structures (or components of those structures) from the ASN.1 module referenced by the "GlobalModuleReference" in the "ExplicitGeneration".

NOTE 1: This enables different encodings to be applied to some occurrences of a class from that applied to other occurrences.

NOTE 2: This implies that "OriginalClassName" can only be a name implicitly generated from an ASN.1 type, that is, the name of a user-defined ASN.1 type (preceded by "#"), or one of the class names listed in column 2 of table 2.

15.2.3 References by "OriginalClassName" to fields of the implicitly generated encoding structure which correspond to use of "ExternalTypeReference" in the ASN.1 type definition shall use the "SimpleDefinedEncodingClass" notation with the same "modulereference" as the "ExternalTypeReference". Otherwise, if the "DefinedType" (preceded by a "#") is not a "BuiltinEncodingClassReference", a simple "encodingclassreference" shall be used. If a "typereference" (preceded by a "#") is a "BuiltinEncodingClassReference" then the "SimpleDefinedEncodingClass" notation shall be used with the same "modulereference" as the ASN.1 module that generated the implicitly generated encoding structure.

15.2.4 When an ELM imports an explicitly generated encoding structure from an EDM, renames clauses in other EDMs have no effect on the encoding of that structure.

NOTE: This means in practice that all the "coloring" (see 9.16.4) needed for any particular message, has to be done in a single EDM.

15.2.5 The "NewClassName" shall be defined in an encoding class assignment statement (see clause 16) of the form:

```
<NewClassName> ::= <OriginalClassName>
```

where "<NewClassName>" and "<OriginalClassName>" are the names of the new and original classes appearing in the "NameChanges" production. The assignment shall be in the EDM module with the renames clause.

15.3 Specifying the region for name changes

15.3.1 The production "NameChangeDomain" is:

```

NameChangeDomain ::=
    IncludedRegions
    Exception ?
Exception ::=
    EXCEPT
    ExcludedRegions
IncludedRegions ::=
    ALL | RegionList
ExcludedRegions ::= RegionList
RegionList ::=
    Region "," +
Region ::=
    SimpleDefinedEncodingClass |
    ComponentReference
ComponentReference ::=
    SimpleDefinedEncodingClass
    "."
    identifier

```

15.3.2 Each "SimpleDefinedEncodingClass" shall be the name of an implicitly generated encoding structure from the ASN.1 module referenced by the "GlobalModuleReference" in the "ExplicitGeneration". When used in "Region", it identifies the whole of that encoding structure definition.

NOTE: The "ExternalEncodingClassReference" form of "SimpleDefinedEncodingClass" is used if the referenced class is derived from a "typereference" name which (when preceded by "#") is a "BuiltinEncodingClassReference" (see 15.2.3).

15.3.3 Each "identifier" shall be the "identifier" in a "NamedField" of the implicitly generated encoding structure identified by the "encodingclassreference" in the "ComponentReference". The "ComponentReference" identifies the entire definition of that component of that encoding structure.

15.3.4 The definitions identified by different "Region"s in "RegionList" shall be disjoint. A definition is identified by "RegionList" if and only if it is identified by a "Region" in "RegionList".

15.3.5 If "IncludedRegions" is "ALL", it identifies all parts of all the implicitly generated encoding structures from the ASN.1 module referenced by the "GlobalModuleReference" in the "ExplicitGeneration".

15.3.6 The definitions identified by the "ExcludedRegions" shall be a proper subset of the definitions identified by the "IncludedRegions".

15.3.7 The "NameChangeDomain" specification identifies the definitions in which the name changes are to be made. The definitions in the "NameChangeDomain" are the definitions identified by the "IncludedRegions" which are not also identified by "ExcludedRegions".

16 Encoding class assignments

16.1 General

16.1.1 The "EncodingClassAssignment" is:

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

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

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

16.1.3 An encoding class is in one of the following categories:

- a) the bit-field category (see 16.1.7);
- b) the alternatives category (see 16.1.8);
- c) the concatenation category (see 16.1.9);
- d) the repetition category (see 16.1.10);
- e) the optionality category (see 16.1.11);
- f) the tag category (see 16.1.12);
- g) the encoding procedure category (see 16.1.13).

NOTE: There is also an encoding constructor category that consists of all classes in the alternatives, concatenation, and repetition categories.

16.1.4 The "EncodingClass" is:

```
EncodingClass ::=
    BuiltinEncodingClassReference |
    EncodingStructure
```

16.1.5 "BuiltinEncodingClassReference" is defined in 16.1.6. The category of each built-in encoding class is the category implied by the name of the production.

16.1.6 The "BuiltinEncodingClassReference" is:

```

BuiltinEncodingClassReference ::=
  BitfieldClassReference
  AlternativesClassReference
  ConcatenationClassReference
  RepetitionClassReference
  OptionalityClassReference
  TagClassReference
  EncodingProcedureClassReference

```

16.1.7 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
  #GraphicString
  #IA5String
  #NumericString
  #PrintableString
  #TeletexString
  #UniversalString
  #UTF8String
  #VideotexString
  #VisibleString

```

16.1.8 The "AlternativesClassReference" is:

```

AlternativesClassReference ::=
  #ALTERNATIVES
  #CHOICE

```

16.1.9 The "ConcatenationClassReference" is:

```

ConcatenationClassReference ::=
  #CONCATENATION
  #SEQUENCE
  #SET

```

16.1.10 The "RepetitionClassReference" is:

```

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

```

16.1.11 The "OptionalityClassReference" is:

```

OptionalityClassReference ::=
  #OPTIONAL

```

16.1.12 The "TagClassReference" is:

```

TagClassReference ::=
  #TAG

```


16.1.13 The "EncodingProcedureClassReference" is:

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

16.1.14 Some of these classes are defined to be primitive, and can only be encoded by encoding objects of their own class. Others are derived from a primitive class through class assignment statements, and can be de-referenced to these classes. The following are the primitive classes that each built-in class is derived from through class assignment statements. When defining encoding objects of derived classes, any syntax permitted for the corresponding primitive class can be used for the derived class:

Built-in class	Derived from
#ALTERNATIVES	(primitive constructor)
#BITS	(primitive bitstring)
#BIT-STRING	#BITS
#BOOL	(primitive bitstring)
#BOOLEAN	#BOOL
#CHARACTER-STRING	#CHARS
#CHARS	(primitive bitstring)
#CHOICE	#ALTERNATIVES
#CONCATENATION	(primitive constructor)
#CONDITIONAL-INT	(primitive, other procedures)
#CONDITIONAL-REPETITION	(primitive, other procedures)
#EMBEDDED-PDV	#OPEN-TYPE
#ENUMERATED	#INT
#EXTERNAL	#OPEN-TYPE
#INT	(primitive bitstring)
#INTEGER	#INT
#NUL	(primitive bitstring)
#NULL	#NUL
#OBJECT-IDENTIFIER	(primitive bitstring)
#OCTETS	(primitive bitstring)
#OCTET-STRING	#OCTETS
#OPEN-TYPE	(primitive bitstring)
#OPTIONAL	(primitive, other procedures)
#OUTER	(primitive, other procedures)
#PAD	(primitive bitstring)
#REAL	(primitive bitstring)
#RELATIVE-OID	#OBJECT-IDENTIFIER
#REPETITION	(primitive constructor)
#SEQUENCE	#CONCATENATION
#SEQUENCE-OF	#REPETITION
#SET	#CONCATENATION
#SET-OF	#REPETITION
#TAG	(primitive bitstring)
#TRANSFORM	(primitive, other procedures)
#GeneralizedTime	#CHARS
#UTCTime	#CHARS
#BMPString	#CHARS
#GeneralString	#CHARS
#GraphicString	#CHARS
#IA5String	#CHARS
#NumericString	#CHARS
#PrintableString	#CHARS
#TeletexString	#CHARS
#UniversalString	#CHARS
#UTF8String	#CHARS
#VideotexString	#CHARS
#VisibleString	#CHARS

16.2 Encoding structure definition

16.2.1 The "EncodingStructure" is:

```
EncodingStructure ::=
    TaggedStructure      |
    UntaggedStructure

TaggedStructure ::=
    "[ "
    TagClass
```



```

TagValue ?
 "]"
EncodingStructure

UntaggedStructure ::=
    DefinedEncodingClass      |
    EncodingStructureField    |
    EncodingStructureDefn

TagClass ::=
    DefinedEncodingClass      |
    TagClassReference

TagValue ::=
    "(" number ")"

```

16.2.2 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 generated encoding structures) for bit-fields, encoding constructors, and the encoding procedure classes in the optionality category. All classes defined by "EncodingStructure" are in the bit-field category. (Examples of an encoding structure assignment illustrating many of the syntactic structures is given in D.2.8.4 and D.2.2.3 is an example of the use of #TAG).

16.2.3 The "DefinedEncodingClass" is specified in 10.9.1 and shall be a bit-field class.

16.2.4 The "DefinedEncodingClass" in the "TagClass" shall be a class in the tag category (see 16.1.3).

16.2.5 The "number" in "TagValue" specifies a tag number which is associated with the class in the tag category.

16.2.6 The "EncodingStructureField" is:

```

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

```

16.2.7 The "EncodingStructureField"s represents all possible bitstring encodings for the corresponding ASN.1 types, and can be assigned values of those types in a value mapping (see clause 19).

16.2.8 The ASN.1 values which can be associated with each primitive field are as follows:

#NUL	The null value
#BOOL	The boolean values
#INT	The integer values
#BITS	Bitstring values

#OCTETS	Octetstring values
#CHARS	Character string values
#PAD	None
#OBJECT-IDENTIFIER	Object identifier values
#OPEN-TYPE	Open type values
#REAL	Real values
#TAG	Tag numbers

NOTE: The #PAD field cannot have associated ASN.1 values, and is never visible outside the encoding and decoding procedures.

16.2.9 The "Bounds" and "Size" specify the bounds or effective size constraint respectively on the abstract values that can be mapped to the field (see clause 19).

NOTE: Effective alphabet constraints cannot be assigned in an encoding structure definition. They can only be assigned through the value mappings of clause 19.

16.2.10 "Bounds" and "Size" are:

```

Bounds ::= "(" EffectiveRange ")"

EffectiveRange ::=
    MinMax |
    Fixed

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

SizeEffectiveRange ::=
    "(" EffectiveRange ")"

MinMax ::=
    ValueOrMin
    ".."
    ValueOrMax

ValueOrMin ::=
    SignedNumber |
    MIN

ValueOrMax ::=
    SignedNumber |
    MAX

Fixed ::= SignedNumber

```

16.2.11 "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.

16.2.12 The "EncodingStructureDefn" is:

```

EncodingStructureDefn ::=
    AlternativesStructure |
    RepetitionStructure |
    ConcatenationStructure

```

16.2.13 These encoding structures are defined in the following clauses:

AlternativesStructure	16.3
RepetitionStructure	16.4
ConcatenationStructure	16.5

16.3 Alternative encoding structure

16.3.1 The "AlternativesStructure" is:

```

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

```



```

AlternativesClass ::=
    DefinedEncodingClass
    AlternativesClassReference

NamedFields ::= NamedField "," +

NamedField ::=
    identifier
    EncodingStructure

```

16.3.2 The "AlternativesStructure" identifies the presence in an encoding of precisely one of the "EncodingStructure"s in its "NamedFields". The "DefinedEncodingClass" shall be a class in the alternatives category (see 16.1.8). The mechanisms used to identify which of the "EncodingStructure"s is present in an encoding are specified by an encoding object of the "AlternativesClass".

16.3.3 The "AlternativesStructure" is an encoding constructor: when an encoding object set is applied to this structure as specified in 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".

16.4 Repetition encoding structure

16.4.1 The "RepetitionStructure" is:

```

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

RepetitionClass ::=
    DefinedEncodingClass
    RepetitionClassReference

```

16.4.2 The "RepetitionStructure" identifies the presence in an encoding of repeated occurrences of the "EncodingStructure" in the production. The optional "Size" construction (see 16.2.9) 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 class in the repetition category (see 16.1.10).

16.4.3 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 repetitions, and the application point then proceeds to the "EncodingStructure" in the production.

NOTE: The characters "{" and "}" are used in this construction, but are not present in the related ASN.1 "SEQUENCE OF" construction. This was done to help avoid syntactic ambiguities in structure definition.

16.5 Concatenation encoding structure

16.5.1 The "ConcatenationStructure" is:

```

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

ConcatenationClass ::=
    DefinedEncodingClass
    ConcatenationClassReference

ConcatComponents ::=
    ConcatComponent "," *

ConcatComponent ::=
    NamedField
    ConcatComponentPresence ?

```



```
ConcatComponentPresence ::=
    OPTIONAL-ENCODING
    OptionalClass

OptionalClass ::=
    DefinedEncodingClass
    OptionalityClassReference
```

16.5.2 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 class in the concatenation category (see 16.1.9), and the "DefinedEncodingClass" in the "OptionalClass" shall be a class in the optionality category (see 16.1.3).

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

16.5.4 If "ConcatComponentPresence" 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".

16.5.5 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.

16.5.6 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.

17 Encoding object assignments

17.1 General

17.1.1 The "EncodingObjectAssignment" is:

```
EncodingObjectAssignment ::=
    encodingobjectreference
    DefinedOrBuiltinEncodingClass
    " : : ="
    EncodingObject
```

17.1.2 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 "DefinedOrBuiltinEncodingClass". (D.1.2.2, D.1.7.3 and D.1.8.2 provide examples of encoding object assignment for the different syntactic constructions for "EncodingObject" specified below).

17.1.3 The "DefinedOrBuiltinEncodingClass" 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.

17.1.4 The "EncodingObject" is:

```
EncodingObject ::=
    DefinedEncodingObject
    DefinedSyntax
    EncodeWith
    EncodeByValueMapping
    EncodeStructure
    EncodingOptionsEncodingObject
    DifferentialEncodeDecodeObject
    NonECNEncodingObject
```


17.1.5 "DefinedEncodingObject" identifies an encoding object and is specified in 10.9.2. The "DefinedEncodingObject" shall be of the same encoding class as the governor, or of a class which can be obtained from the governor by de-referencing. The "encodingobjectreference" being defined exhibits an identification handle if and only if the "DefinedEncodingObject" exhibits that identification handle.

17.1.6 In the present document, "the same encoding class" and "the same class" shall be interpreted as meaning that the notation used for defining the two classes shall be the same encoding class reference name.

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

DefinedSyntax	17.2 with clauses 20 to 25
EncodeWith	17.3
EncodeByValueMapping	17.4
EncodeStructure	17.5
DifferentialEncodeDecodeObject	17.6
EncodingOptionsEncodingObject	17.7
NonECNEncodingObject	17.8

17.2 Encoding with a defined syntax

17.2.1 The "DefinedSyntax" production is specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 11.5, as modified by B.15 of the present document, and is used for the definition of encoding objects for a governing encoding class. The detailed syntax for doing this is specified in clauses 23 to 25, and the semantics of the constructs is specified in clause 22.

17.2.2 This notation for defining encoding objects is only available for the governing encoding classes in the categories (or of the class) listed in table 3. The syntax to be used for each encoding object is the "DefinedSyntax" for the corresponding primitive encoding class (specified in clauses 23 to 25).

NOTE 1: 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, or for inclusion as encoding objects to be applied as part of a replacement action. They cannot be applied in the ELM.

NOTE 2: 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.

Table 3: Categories and classes supported by a defined syntax

```

null
boolean
integer
bitstring
octetstring
characterstring
pad
alternatives
repetition
concatenation
optionality
#CONDITIONAL-INT
#CONDITIONAL-REPETITION
tag
#TRANSFORM
#OUTER

```

17.2.3 The information required to specify an encoding object of one of these classes is specified by the definition of the encoding class in clauses 23 to 25.

17.2.4 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" (specified in B.16) shall be used. No other use shall be made of the "EncodingClassFieldType" notation.

17.2.5 Where the syntax defined in clause 23 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.

17.2.6 The defined syntax specifies whether the "encodingobjectreference" being defined exhibits an identification handle.

17.3 Encoding with encoding object sets

17.3.1 The "EncodeWith" is:

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

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

17.3.3 The encoding object defined by the "EncodeWith" is the application of the "CombinedEncodings" to the encoding class that is the governor (see 17.1.3) of the "EncodeWith" notation.

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

17.3.5 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.

17.3.6 In the application of encodings specified in clause 13, there is an encoding object (A say) which produces the first bit-field in the resulting encoding. The "encodingobjectreference" being defined exhibits an identification handle if and only if the encoding object A exhibits that identification handle.

17.4 Encoding using value mappings

17.4.1 The "EncodeByValueMapping" is:

```
EncodeByValueMapping ::=
    "{ "
    USE
    DefinedOrBuiltinEncodingClass
    MAPPING
    ValueMapping
    WITH
    ValueMappingEncodingObjects
    " } "

ValueMappingEncodingObjects ::=
    EncodingObject |
    DefinedOrBuiltinEncodingObjectSet
```

17.4.2 The production "DefinedOrBuiltinEncodingClass" and its semantics are defined in 10.9.1. It shall be a class in the bit-field category (see 16.1.7).

17.4.3 The production "ValueMapping" is specified in 19.1.5, and shall be a mapping of values associated with the governing encoding class to the class identified by the "DefinedOrBuiltinEncodingClass".

17.4.4 The "ValueMappingEncodingObjects" specifies the encoding of the "DefinedOrBuiltinEncodingClass". The "EncodingObject" shall define an encoding object using notation governed by that class, or by a class to which it can be de-referenced (see 17.1.3). The "DefinedOrBuiltinEncodingObjectSet" can alternatively be used to specify the encoding of the "DefinedOrBuiltinEncodingClass" and shall contain sufficient encoding objects to fully specify the encoding of that class.

17.4.5 The syntax for "EncodingObject" allows both in-line definition of encoding objects (recursive application of this clause) and the use of reference names. (D.2.9.3 gives an example of in-line definition to perform two value mappings in a single assignment).

17.4.6 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.

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

17.4.8 If the "EncodingObject" alternative of "ValueMappingEncodingObjects" is used, then the "encodingobjectreference" being defined exhibits an identification handle if and only if the "EncodingObject" exhibits that identification handle. If the "DefinedOrBuiltinEncodingObjectSet" alternative of "ValueMappingEncodingObjects" is used to define the encoding of the "DefinedOrBuiltinEncodingClass", then determination of whether the "encodingobjectreference" exhibits an identification handle is in accordance with 17.3.6.

17.5 Encoding an encoding structure

17.5.1 The "EncodeStructure" is:

```

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

StructureEncoding ::=
    STRUCTURED WITH
    TagAndStructureEncodingEncodingObject

TagAndStructureEncoding ::=
    EncodingOrUseSet
    TagEncoding TagAndStructureEncoding

TagEncoding ::= "[ " EncodingOrUseSet "]"

EncodingOrUseSet ::=
    EncodingObject
    USE-SET

```

17.5.2 The "EncodeStructure" can be used to define an encoding only if the governing encoding structure is a construction defined using a class in the alternatives, concatenation, or repetition category, or a class in one of these categories preceded by one or more classes in the tag category. This class is called the governing encoding constructor.

17.5.3 "StructureEncoding", if this production is present, shall define an encoding for the governing encoding constructor and for any preceding classes in the tag category. If the production is absent, the "CombinedEncodings" shall be present, and shall contain encoding objects which can encode the governing encoding constructor and any preceding classes in the tag category, otherwise the ECN specification is in error.

NOTE: "CombinedEncodings" has to be present if the "StructureEncoding" is absent, because a complete encoding has to be produced. If it is desired to defer the specification of part of an encoding, then a dummy parameter should be used.

17.5.4 The number of "TagEncoding"s in the "TagAndStructureEncoding" shall exactly equal the number of classes of the tag category preceding the governing encoding constructor, and shall correspond to those classes in textual order. They shall be governed by those classes.

NOTE: The number of classes in the tag category may be zero.

17.5.5 If the "EncodingOrUseSet" in the "TagAndStructureEncoding" is an "EncodingObject", it shall be governed by the governing encoding constructor.

17.5.6 If "USE-SET" is specified in any "EncodingOrUseSet", then the encoding of the corresponding class is obtained by applying the "CombinedEncodings", which shall be present, and shall be sufficient to encode the corresponding class, otherwise the ECN specification is in error.

17.5.7 The "ComponentEncodingList" is:

```

ComponentEncodingList ::=
    ComponentEncoding "," *

ComponentEncoding ::=
    NonOptionalComponentEncodingSpec
    OptionalComponentEncodingSpec

```


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

NOTE: The absence of "ComponentEncoding"s can be detected by following named fields, or by the end of the "ComponentEncodingList".

17.5.9 The "OptionalComponentEncodingSpec" shall be used if and only if the component is optional (i.e., is followed by an encoding class in the optionality category).

17.5.10 If the "ComponentEncodingList" is empty, then the "CombinedEncodings" must be present, and is required, on application to the component (see 13.2), to provide a complete encoding of that component, otherwise it is an error in the ECN specification.

```

NonOptionalComponentEncodingSpec ::=
    identifier ?
    TagAndElementEncoding

OptionalComponentEncodingSpec ::=
    identifier
    TagAndElementEncoding
    OPTIONAL-ENCODING
    OptionalEncoding

TagAndElementEncoding ::=
    EncodingOrUseSet
    TagEncoding TagAndElementEncoding

OptionalEncoding ::= EncodingOrWith

```

17.5.11 The "identifier" 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 a class in the repetition category.

17.5.12 "TagAndElementEncoding" in the "ComponentEncoding" shall provide a complete encoding for that component (including any classes in the tag category that are prefixed to the element).

17.5.13 The number of "TagEncoding"s in the "TagAndElementEncoding" shall exactly equal the number of classes of the tag category at the start of the element, and correspond to those classes in textual order.

NOTE: This may be zero.

17.5.14 The "EncodingObject"s in the "EncodingOrUseSet"s in the "TagAndElementEncoding" shall be governed by the corresponding encoding class in the component. If an "EncodingOrUseSet" is "USE-SET" then the encoding of the corresponding class is obtained by applying the "CombinedEncodings" which shall be present.

17.5.15 The "EncodingOrUseSet" in the "OptionalEncoding" shall completely encode the optionality class of that component. If an "EncodingOrUseSet" is "USE-SET" then the encoding of the corresponding class is obtained by applying the "CombinedEncodings" which shall be present.

17.5.16 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 "identifier"s that are textually present in the construction. If the "REFERENCE" is required to identify a container, it can also be supplied as:

- a) "STRUCTURE" (provided the constructor for the structure being encoded is not an alternative category) when it refers to that structure;
- b) "OUTER" when it refers to the container of the complete encoding.

NOTE: The "EncodeStructure" is the only production in which "REFERENCE"s can be supplied, except through the use of dummy parameters.

17.5.17 Determination of whether the "encodingobjectreference" being defined exhibits an identification handle is in accordance with 17.3.6.

17.6 Differential encoding-decoding

17.6.1 The "DifferentialEncodeDecodeObject" is:

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

SpecForEncoding ::= EncodingObject

SpecForDecoders ::= EncodingObject
```

17.6.2 The "DifferentialEncodingObject" specifies rules for encoding abstract values associated with the class of the governor of this notation, and (separately) 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.

17.6.3 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 the present document.

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.

17.6.4 The "SpecForEncoding" and the "SpecForDecoders" 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".

NOTE: This restriction is present because otherwise specification of the meaning of the encode/decode construction would become more complex with no added functionality.

17.6.5 The "encodingobjectreference" being defined exhibits an identification handle if and only if the same identification handle is being exhibited by the "SpecForEncoding" and by the "SpecForDecoders".

17.7 Encoding with encoder's options

17.7.1 The "EncodingOptionsEncodingObject" is:

```
EncodingOptionsEncodingObject ::=
  "{ "
    OPTIONS
    EncodingOptionsList
    WITH AlternativesEncodingObject
  "}"

EncodingOptionsList ::= EncodingObjectList

AlternativesEncodingObject ::= EncodingObject
```

17.7.2 The "EncodingOptionsEncodingObject" specifies that the encoder may encode (subject to 17.7.5) using any of the "EncodingObject"s in the "EncodingOptionsList". These "EncodingObject"s shall all be encoding objects of the governing class.

NOTE: New implementations are strongly recommended to encode using the earliest "EncodingObject" in the list that is capable of encoding the abstract value to be encoded (see 17.7.5). The encoder's options specification is provided only because it is necessary to reflect options provided in legacy protocols. All the encoder's options encodings can, of course, occur when decoding.

17.7.3 The "AlternativesEncodingObject" shall be an encoding object of any class in the alternatives category, and encoders and decoders shall use the encodings and procedures specified by that encoding object as if the encoder's options were encodings for components of an instance of that class.

NOTE: If the "AlternativesEncodingObject" is parameterized with a reference parameter, then the "encodingobjectreference" being defined has to be parameterized with a dummy reference parameter that is used as the actual parameter for the "AlternativesEncodingObject".

17.7.4 If the "AlternativesEncodingObject" specifies determination by use of a specified identification handle, then all "EncodingObject"s in the "EncodingOptionsList" shall exhibit that identification handle.

17.7.5 The encoder shall restrict its choice of "EncodingObject"s in the "EncodingOptionsList" to those that provide encodings for the actual abstract value being encoded. It is an ECN specification or application error if there is not at least one such "EncodingObject" for any abstract value that is to be encoded.

NOTE 1: It is possible that the sets of abstract values encoded by the "EncodingObject"s in the "EncodingOptionsList" are disjoint. This is not an error, and can be a convenient way of specifying different structures for encoding different ranges of abstract values of the governing class, for example short form and long form encodings where the short form is mandatory for small values.

NOTE 2: It is possible to use an encoder's options encoding object as the "SpecForDecoders" (see 17.6), where the "SpecForEncoding" is an encoder's options encoding object that contains exactly one of the options in the "SpecForDecoders". This is another approach to extensibility.

17.8 Non-ECN definition of encoding objects

17.8.1 The "NonECNEncodingObject" is:

```
NonECNEncodingObject ::=
    NON-ECN-BEGIN
    AssignedIdentifier
    anystringexceptnonecnend
    NON-ECN-END
```

17.8.2 The "NonECNEncodingObject" shall specify an encoding object of the governor class (see 17.1.3). The notation used to do this is contained in "anystringexceptnonecnend" and is not standardized.

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

17.8.4 If the "empty" alternative of "AssignedIdentifier" is used, then the notation is determined by means outside of the present document.

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

17.8.6 An identification handle is exhibited by the "encodingobjectreference" being defined if and only if the "anystringexceptnonecnend" specifies that it does so. The means of such specification is not defined in this Recommendation | International Standard.

18 Encoding object set assignments

18.1 General

18.1.1 The "EncodingObjectSetAssignment" is:

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

EncodingObjectSet ::=
    DefinedEncodingObjectSet |
    EncodingObjectSetSpec
```


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

18.1.3 "DefinedEncodingObjectSet" is defined in 10.9.3.

18.1.4 The "EncodingObjectSetSpec" is:

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

EncodingObjects ::=
    DefinedEncodingObject |
    DefinedEncodingObjectSet

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

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

18.1.6 Encoding objects forming an encoding object set shall all be of distinct encoding classes, and shall not be classes in the encoding procedure category unless they are of the #OUTER class (see 16.1.13).

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.

18.1.7 If "CompletionClause" is present, then the encoding object set defined by "EncodingObjectSetSpec" is considered to be "PrimaryEncodings" (see 13.2), and the encoding object set assigned to the "encodingobjectsetreference" is the combined encoding object set formed as specified in 13.2.

18.2 Built-in encoding object sets

18.2.1 The "BuiltinEncodingObjectSetReference" is:

```
BuiltinEncodingObjectSetReference ::=
    PER-BASIC-ALIGNED
    PER-BASIC-UNALIGNED
    PER-CANONICAL-ALIGNED
    PER-CANONICAL-UNALIGNED
    BER
    CER
    DER
```

18.2.2 These encoding object set names reference 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.

NOTE: These Recommendations | International Standards were written before this ECN, the present document, and do not use the encoding object terminology. They define, for example, the way an ASN.1 "INTEGER" or "BOOLEAN" type is to be encoded. This should be interpreted as the definition of an encoding object of class #INTEGER or class #BOOLEAN.

Table 4: Built-in 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)}

18.2.3 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 defined by the user) to specify the corresponding BER or PER encodings.

18.2.4 The above sets all contain encoding objects for the classes used in implicitly generated encoding structures (see 11.2) which are different for each set of encoding rules. They also each contain identical encoding objects for the classes #INT, #BOOL, #NUL, #CHARS, #OCTETS, #BITS, #CONCATENATION. They do **not** contain encoding objects for #ALTERNATIVES, #REPETITION, and #PAD.

18.2.5 These encoding classes represent basic building blocks of encodings, and are encoded simply by all the above built-in encoding object sets. The encoding objects for these classes specify encodings as follows:

18.2.5.1 #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 when this encoding object is applied to the #INT.

18.2.5.2 #BOOL and #NUL are encoded as PER-BASIC-UNALIGNED #BOOLEAN and #NULL respectively.

18.2.5.3 #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.

18.2.5.4 #CONCATENATION is encoded as a PER-BASIC-UNALIGNED encoding of a #SEQUENCE with no optional elements. If these encoding objects are applied to a #CONCATENATION with optional elements, then it is an ECN specification error.

18.2.6 The #OPEN-TYPE encoding objects in the BER, CER, and DER built-in encoding object sets produce no additional encoding for the #OPEN-TYPE class. When these encoding objects are applied to a class in the open-type category, it is an ECN specification error if the encodings of the values of the type chosen (in an instance of communication) for use with the #OPEN-TYPE class are not self-delimiting.

NOTE: The combined encoding object set applied to the type chosen for use with the #OPEN-TYPE class is always the same as the combined encoding object set applied to the #OPEN-TYPE class (see 13.2.10.5).

19 Mapping values

19.1 General

19.1.1 This clause specifies the syntax for mapping values to be encoded by the fields of one encoding structure (which may be a generated encoding structure or any other encoding structure) to the fields of another encoding structure.

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 17.4 and the example in D.1.10.2). These mapping mechanisms can be extended and generalized, but this will not be done unless further user requirements are identified.

19.1.2 In specifying the "EncodeByValueMapping" notation (see 17.4.1) the "DefinedOrBuiltinEncodingClass" in the "EncodingObjectAssignment" (see 17.1.1), of which it is a part, is called the source governor or the source encoding class (depending on context). The "DefinedOrBuiltinEncodingClass" in the "EncodeByValueMapping" itself is called the target governor or the target encoding class (depending on context).

19.1.3 Both the source governor and the target governor may be encoding classes preceded by classes in the tag category. Where the following text requires that the source or target class be in a specified category, this includes the case where these classes are preceded by classes in the tag category.

19.1.4 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 ECN 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 the present document, and responsibility rests with the ECN user.

19.1.5 The "ValueMapping" is:

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

NOTE: All occurrences of this syntax are preceded by the reserved word "MAPPING". (D.1.2.2, D.1.4.2, D.1.10.2, and D.2.1.3 and annex E give examples of the definition of encodings using each of these value mappings).

19.1.6 The "ValueMapping" productions are specified as follows:

MappingByExplicitValues	19.2
MappingByMatchingFields	19.3
MappingByTransformEncodingObjects	19.4
MappingByAbstractValueOrdering	19.5
MappingByValueDistribution	19.6
MappingIntToBits	19.7

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.

19.2 Mapping by explicit values

19.2.1 This clause provides notation for specifying the mapping of values between different primitive bit-field encoding classes. (D.1.10.2 gives an example).

19.2.2 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.

19.2.3 Table 5 specifies the ASN.1 value notation to be used with each governing encoding class. In each case the class may or may not have an associated size or value range constraint.

19.2.4 ECN supports mapping by explicit values (either to or from the encoding class) for all encoding classes in the categories 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 an encoding class in the category listed 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: Categories of encoding classes and value notation used in mapping by explicit values

Category of governing encoding class	ASN.1 value notation
bitstring	"bstring" or "hstring" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 11.9 and 11.10)
boolean	"BooleanValue" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 17.3)
characterstring	"RestrictedCharacterStringValue" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 36.7)

integer	"SignedNumber" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 18.1)
null	"NullValue" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 23.3)
objectidentifier	"DefinitiveIdentifier" (see A.1)
octetstring	"bstring" or "hstring" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 11.9 and 11.10)
real	"RealValue" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 20.6)
tag	"number" (see ITU-T Rec. X.680 ISO/IEC 8824-1, 11.8)

19.2.5 The "MappingByExplicitValues" is:

```

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

MappedValues ::=
    MappedValue1
    TO
    MappedValue2

MappedValue1 ::= Value
MappedValue2 ::= Value

```

19.2.6 The "MappedValue1" shall be value notation governed by the source governor and "MappedValue2" shall be value notation governed by the target governor (see 19.1.2). The value in the source specified by "MappedValue1" is mapped to the value in the target specified by "MappedValue2".

19.2.7 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.

19.3 Mapping by matching fields

19.3.1 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.

19.3.2 The "MappingByMatchingFields" is:

```

MappingByMatchingFields ::=
    FIELDS

```

19.3.3 If either the source or the target encoding classes are user-defined encoding structures (see 9.2.2.3) or generated encoding structures, then these references are resolved (but references within the resulting structures are not). After resolution, the source and the target encoding classes shall start with the same encoding constructor (of any category), and the resulting encoding structures are called the source and target encoding structures respectively.

19.3.4 No further resolution of references takes place during these procedures.

19.3.5 All fieldnames that are visible (after the de-referencing specified in 19.3.3) in the source encoding structure shall be distinct, and all fieldnames that are visible in the target encoding structure shall be distinct.

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

19.3.6 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 same encoding class (see 17.1.6).

19.3.7 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.

19.3.8 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.

19.4 Mapping by #TRANSFORM encoding objects

19.4.1 This mapping permits one or more #TRANSFORM encoding objects to be applied to produce the mapping.

19.4.2 The #TRANSFORM encoding class is defined in clause 24. It enables encoding objects to be specified which will transform source abstract values into result abstract values. The rules for forming an ordered list of transforms (for "TransformList") are specified in clause 24. The complete list is defined to transform from a source to a result.

NOTE: Examples of mappings defined with these transforms are given in D.1.2.2 and D.2.4.2. The example in D.1.6.3 shows the use of this production to define BCD encodings of an ASN.1 integer.

19.4.3 The "MappingByTransformEncodingObjects" is:

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

TransformList ::= Transform "," +
Transform ::= EncodingObject
```

19.4.4 All the "EncodingObject"s in the "TransformList" shall be governed by the encoding class #TRANSFORM.

19.4.5 The target and source classes for this mapping (see 19.1.2) shall be of the bitstring, boolean, characterstring, integer, or octetstring category. The source of the first transform in the list and the result of the last transform in the list shall agree with the category of the source and target categories as specified in 24.2.7.

19.4.6 All "Transform"s in the "TransformList" shall be reversible transforms.

NOTE: Clause 24 specifies, for each transform, the conditions under which it is defined to be reversible.

19.4.7 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 specification or in the ECN specification.

19.5 Mapping by abstract value ordering

19.5.1 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.

19.5.2 The "MappingByAbstractValueOrdering" is:

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

19.5.3 For this mapping, all encoding class names are de-referenced (recursively), and the result shall be a class in the null, boolean, integer or real category, or shall be a construction defined using a class in the alternatives category.

19.5.4 An ordering of abstract values is defined for encoding classes in the following categories: null, boolean, integer (but only if it has a lower bound), real (but only if it is constrained to a finite number of values), and for any encoding structure defined using a class in the alternatives category whose alternatives all have a defined ordering.

NOTE: The tag numbers associated with classes in the tag category are not abstract values.

19.5.5 Classes in the null category have a single abstract value. Classes in the boolean category are defined to have "TRUE" before "FALSE". Classes in the integer category are defined to have higher integer values following lower integer values. Classes in the real category are defined to have higher values following lower values.

NOTE: The number of abstract values associated with a class in the integer category is not necessarily finite.

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

19.5.7 The ordering of the abstract values associated with a class in the alternatives category (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.

19.5.8 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.

19.5.9 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).

19.5.10 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 specification or the ECN specification.

19.5.11 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.

19.5.12 This mapping can also be applied in all cases where the only abstract values in the target structure are those associated with a single instance of the same class as the source structure.

NOTE: This case would occur if the target structure were the same as the source structure preceded by one or more instances of classes in the tag category.

19.6 Mapping by value distribution

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

19.6.2 All encoding structure names are de-referenced (recursively) before the application of this mapping.

19.6.3 The source encoding class shall then be a class in the integer category.

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

19.6.5 The "MappingByValueDistribution" is:

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

Distribution ::=
    SelectedValues
    TO
    identifier

SelectedValues ::=
    SelectedValue      |
    DistributionRange  |
    REMAINDER

DistributionRange ::=
    DistributionRangeValue1
    ".."
    DistributionRangeValue2
```



```

SelectedValue ::= SignedNumber

DistributionRangeValue1 ::= SignedNumber
DistributionRangeValue2 ::= SignedNumber

```

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

19.6.7 "DistributionRangeValue1" shall be less than "DistributionRangeValue2".

19.6.8 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".

19.6.9 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".

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

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

19.7 Mapping integer values to bits

19.7.1 This mapping takes single values or ranges of values from an encoding class in the integer category, 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 E for further discussion and examples of Huffman encodings).

19.7.2 The source encoding class shall be a class in the integer category.

19.7.3 The destination encoding class shall be a class in the bitstring category.

19.7.4 The "MappingIntToBits" is:

```

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

MappedIntToBits ::=
    SingleIntValMap |
    IntValRangeMap

```

19.7.5 Each "SingleIntValMap" maps a single integer value to a single bitstring value.

19.7.6 Each "IntValRangeMap" maps a range of contiguous and increasing integer values to a range of contiguous and increasing bitstring values.

19.7.7 Bitstring 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.

19.7.8 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 ECN specification.

19.7.9 The "SingleIntValMap" is:

```
SingleIntValMap ::=
    IntValue
    TO
    BitValue

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

19.7.10 The "SignedNumber", "bstring", and "hstring" are specified in ITU-T Rec. X.680 | ISO/IEC 8824-1, 18.1, 11.9, and 11.10, respectively.

19.7.11 The "SingleIntValMap" maps the specified integer value to the specified bitstring value.

19.7.12 The "IntValRangeMap" is:

```
IntValRangeMap ::=
    IntRange
    TO
    BitRange

IntRange ::=
    IntRangeValue1
    ".."
    IntRangeValue2

BitRange ::=
    BitRangeValue1
    ".."
    BitRangeValue2

IntRangeValue1 ::= SignedNumber
IntRangeValue2 ::= SignedNumber

BitRangeValue1 ::=
    bstring |
    hstring

BitRangeValue2 ::=
    bstring |
    hstring
```

19.7.13 The bitstrings "BitRangeValue1" and "BitRangeValue2" shall be the same number of bits.

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

19.7.15 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".

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

19.7.17 The "IntValRangeMap" maps each of the integers in the specified range to the corresponding bitstring value in the "BitRange". (Annex E gives examples of an "IntValRangeMap").

20 Defining encoding objects using defined syntax

20.1 Clauses 21 to 25 specify the information needed to define encoding objects for each encoding class category, and the syntax to be used. This syntax is called the defined syntax, and is specified using the information object class notation of ITU-T Rec. X.681 | ISO/IEC 8824-2 as modified by annex B of the present document.

20.2 The defined syntax for each category can also be used to define encoding objects for structures which are classes of that category, preceded by one or more instances of a class in the tag category. Where the following text requires that a class be in a specified category, this includes the case where the class is preceded by classes in the tag category.

20.3 The use of the modified information object class notation is solely for use within the present document.

20.4 The use of the defined syntax notation to define encoding objects is specified in 17.2. The defined syntax for defining encoding objects shall be the syntax specified by the "WITH SYNTAX" statements in clauses 23 to 25.

20.5 The "WITH SYNTAX" statements impose constraints on the setting of some encoding parameters, in conjunction with other encoding parameters, to enforce some (but not all) semantic constraints. Other constraints on the use of the "WITH SYNTAX" statements are specified in text.

20.6 The defined syntax for each encoding class specifies a number of encoding parameters which can be supplied with values of the ASN.1 types defined in clause 21 (or in some cases with other encoding classes and encoding objects) in order to provide the information needed to specify an encoding object of that class. The information needed to define an encoding object is in general a combination of encoding parameter values, together with the particular instance of defined syntax used to specify those values.

NOTE: This differs from the use of a "WITH SYNTAX" statement in normal information object definition, where the semantics associated with the information object depends solely on the values set for the fields of the information object class, not on the form of the "WITH SYNTAX" statement used to set those values (see B.14).

20.7 The encoding parameters specified in clauses 23 to 25 operate together in encoding parameter groups and use values of ASN.1 types for their definition. Clause 21 specifies the meaning of values of the types commonly used in the specification of these encoding parameters.

20.8 Some definitive text in clauses 21 and 22 is copied into clauses 22 to 25. Where this occurs, the copied text is "grayed-out", and a reference is given to the definitive text.

20.9 Clause 25 specifies a number of transforms that can be applied to abstract values. Several encoding parameter groups require a list of transforms that are to be applied by an encoder. For decoding to be possible, the transforms applied by an encoder have to be reversible by a decoder in order to recover the original abstract values. Clauses 23 and 24 specify when transforms have to be reversible, and clause 25 specifies the conditions under which any given transform is reversible.

21 Types used in defined syntax specification

NOTE: All ASN.1 type definitions given here assume automatic tags and no extensibility.

21.1 The Unit type

21.1.1 The "Unit" type is:

```
Unit ::= INTEGER {repetitions(0), bit(1), nibble(4), octet(8), word16(16), dword32(32)} (0..256)
```

21.1.2 The default value for this type is always "bit".

21.1.3 An encoding parameter of this type specifies the unit in which other encoding parameters or determinant fields are counting.

21.1.4 The value of an encoding parameter of this type is restricted in all cases but one to the non-zero values. In these cases the encoding parameter specifies a number of bits. That number of bits determines the unit in which other encoding parameters or determinant fields are counting.

21.1.5 When used in the definition of an encoding object of a class in the repetition category, the value "repetitions" is also allowed, and specifies that the associated count gives the number of repetitions in the encoding.

21.2 The EncodingSpaceSize type

21.2.1 The "EncodingSpaceSize" type is:

```
EncodingSpaceSize ::= INTEGER
    { encoder-option-with-determinant(-3),
      variable-with-determinant(-2),
      self-delimiting-values(-1),
      fixed-to-max(0) } (-2..MAX)
```

21.2.2 The default value for this type is always "self-delimiting-values".

21.2.3 An encoding parameter of this type specifies the size of the encoding space (see 9.20.5).

21.2.4 Positive (non-zero) values specify a fixed size for the encoding space, as the value of type "Unit" multiplied by the value of type "EncodingSpaceSize", in bits. If the value of type "Unit" is "repetitions", then the encoding space size may be variable, but is always an integral number of repetitions.

21.2.5 The value "encoder-option-with-determinant" specifies that the size of the encoding space may vary according to the abstract value being encoded, **and** that the encoder shall choose the encoding space size, recording the chosen size in the associated determinant. In this case, a value of type "EncodingSpaceDetermination" is required.

NOTE: A value of type "EncodingSpaceDetermination" (to determine the encoding space size) is required in this case (and in the case of 21.2.6), but the provision of a determinant is allowed in all the other cases, to support encodings (similar to BER) that use length determinants even when they are redundant. Any difference between the two determinations is an error. It may, however, not always be possible to determine whether this is an ECN specification error or is an application error, but conforming encoders are required not to transmit such encodings.

21.2.6 The value "variable-with-determinant" specifies that the size of the encoding space may 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 the "EncodingSpaceDetermination" type (see 21.3). In this case, a value of type "EncodingSpaceDetermination" is required.

21.2.7 The value "self-delimiting-values" specifies that the value encoding is self-delimiting, that is, each value encodes into a multiple of the specified value of type "Unit". There shall be no pair of abstract values for which the encoding of one abstract value is the first part of the encoding of the other abstract value.

NOTE: A decoder can (after possible determination of unused bits and justification) determine the end of the encoding space by matching the encoding of each possible abstract value with the encoding that is being examined. Precisely one will match in encodings produced by a conforming encoder. Decoders may develop more efficient but equivalent approaches.

21.2.8 The value "fixed-to-max" specifies that the encoding space is to be the same for the encoding of all abstract values. It specifies that the size of the encoding space is to be the smallest multiple of "Unit" that can contain the specified encoding of any one (all) of the abstract values.

NOTE 1: A special case is when there is a single abstract value whose value encoding is zero bits. This results in an empty encoding space (zero bits).

NOTE 2: If such a specification is applied when a maximum size cannot be determined (for example, for encoding an unbounded integer), this is an ECN specification error, but conforming encoders are required to refuse to generate encodings in such cases.

21.3 The EncodingSpaceDetermination type

21.3.1 The "EncodingSpaceDetermination" type is:

```
EncodingSpaceDetermination ::= ENUMERATED {added-field, asn1-field, container}
```

21.3.2 The default value for this type is always "added-field".

21.3.3 An encoding parameter of this type specifies the way in which the encoding space is determined when an encoding parameter of type "EncodingSpaceSize" is set to "variable-with-determinant-mechanism".

21.3.4 The value "added-field" requires the specification of a "REFERENCE" to another field that does not carry application semantics (i.e., does not appear within the ASN.1 specification). The encoding specification determines how an encoder is to set the value of this field from the size (in encoding space units) of the encoding space.

21.3.5 The value "asn1-field" requires the specification of a "REFERENCE" to another field whose value is set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification). The encoding specification determines how a decoder is to obtain the size of the encoding space from the value of this field. A conforming encoder shall not produce encodings in which the decoder's transforms of this field do not correctly identify the end of the encoding space.

21.3.6 The value "container" requires either the specification of a "REFERENCE" to another field whose encoding class (the container) has a length determinant and whose contents include this encoding space, or of a specification that the end of the PDU determines the end of the encoding space (using "OUTER"). The encoding space terminates when the specified container terminates or when the end of the PDU is encountered. The present document can only be used if the encoding space of the element being encoded is the last encoding to be placed in the container.

NOTE: It is an ECN encoder's error (possibly resulting from an ECN specification or application error) if additional encodings are placed in the container.

21.4 The UnusedBitsDetermination type

21.4.1 The "UnusedBitsDetermination" type is:

```
UnusedBitsDetermination ::= ENUMERATED {added-field, asn1-field, not-needed}
```

21.4.2 The default value for this type is always "added-field".

21.4.3 An encoding parameter of this type specifies the way in which a decoder can determine the unused bits when a value encoding is left or right justified in an encoding space.

21.4.4 The value "added-field" requires the specification of a "REFERENCE" to another field that does not carry application semantics (i.e. does not appear within the ASN.1 specification). The encoding specification determines how an encoder is to determine the number of unused bits, and how to set the value of this field from the number of unused bits.

21.4.5 The value "asn1-field" requires the specification of a "REFERENCE" to another field whose value is set from the abstract syntax (i.e. a corresponding field appears within the ASN.1 specification). The encoding specification determines how a decoder is to determine the number of unused bits from the value of this field. A conforming encoder shall not produce encodings in which the decoder's transforms of this field do not correctly identify the number of unused bits.

21.4.6 The value "not-needed" identifies that a decoder does not require an explicit determinant in order to discover the number of unused bits. The number of unused bits will be deducible from the encoding specification without knowledge of the actual abstract value that has been encoded. This determination is described for each value encoding.

21.5 The OptionalityDetermination type

21.5.1 The "OptionalityDetermination" type is:

```
OptionalityDetermination ::= ENUMERATED  
{added-field, asn1-field, container, handle, pointer}
```

21.5.2 The default value for this type is always "added-field".

21.5.3 An encoding parameter of this type specifies the way in which the presence or absence of an optional element is determined.

21.5.4 The value "added-field" requires the specification of a "REFERENCE" to another field that does not carry application semantics (i.e., does not appear within the ASN.1 specification). The ECN specification will also include an encoding parameter that specifies how an encoder is to set the value of this field from a conceptual boolean value which it true if the optional element is present and false if the optional element is absent.

21.5.5 The value "asn1-field" requires the specification of a "REFERENCE" to another field whose value is set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification). The specification will also include an encoding parameter that specifies how a decoder is to determine the presence or absence of the optional element from the value of this field. A conforming encoder shall ensure that the value of this field correctly determines the presence or absence of the optional field.

21.5.6 The value "container" requires either the specification of a "REFERENCE" to another field whose encoding class (the container) has a length determinant and whose contents include this optional element, or of a specification that the container is the end of the PDU (using "OUTER"). If the container end is present when a decoder is looking for the start of this optional element, then the decoder shall determine that this optional element is absent.

NOTE: The present document can only be used if the abstract values being encoded are such that no further encodings are to be placed in the container. This may require restrictions to be placed on the abstract values of the ASN.1 type, for example, to prohibit the inclusion of a later optional element unless all earlier optional elements are present. It is either an ECN specification error or an application error if additional encodings are to be placed in the container following a component whose optimality is determined in this way, but a conforming encoder shall not generate such encodings.

21.5.7 The value "handle" requires that an identification handle be specified. This identification handle shall be exhibited by the optional element and by any possible alternative encoding that can follow if this optional element is absent. This handle value specifies that a decoder shall determine that the element is present if and only if decoding the remaining parts of the encoding produces a value for the specified identification handle which matches that of the optional element. If the end of any open container (or the end of the PDU) is reached before the identification handle can be decoded, then this is an encoding error. It is an ECN specification error if this does not result in correct identification of the presence or absence of an encoding of the optional element, but conforming encoders shall not generate such encodings.

21.5.8 The value "pointer" requires the specification of a start-of-encoding "REFERENCE" to another field. If that field is zero, then this element is absent. If it is non-zero, then the rules for a start-of-encoding pointer apply (see 22.3).

21.6 The AlternativeDetermination type

21.6.1 The "AlternativeDetermination" type is:

```
AlternativeDetermination ::= ENUMERATED {added-field, asn1-field, handle}
```

21.6.2 The default value for this type is always "added-field".

21.6.3 An encoding parameter of this type specifies the way in which a decoder determines which alternative is present in an encoding of a class in the alternatives category.

21.6.4 The value "added-field" requires the specification of a "REFERENCE" to another field that does not carry application semantics (i.e., does not appear within the ASN.1 specification). The specification will also include an encoding parameter that specifies how an encoder is to set the value of this field from a conceptual integer value that identifies each alternative (using an order specified in other encoding parameters).

21.6.5 The value "asn1-field" requires the specification of a "REFERENCE" to another field whose value is set from the abstract syntax (i.e., appears within the ASN.1 specification). The specification will also include an encoding parameter that specifies how a decoder is to determine (from the value of the referenced field) a conceptual integer value which identifies the alternative (using an order specified in other encoding parameters).

21.6.6 The value "handle" requires that an identification handle be specified. This identification handle shall be exhibited by all of the alternatives in the class, and each alternative shall have a different value for this alternative. (Violation of this rule is an ECN specification error, but conforming encoders are required not to generate encodings where this rule is violated). This value specifies that a decoder shall determine the alternative that is present by decoding the remaining parts of the encoding to produce a value for the specified identification handle. The alternative whose identification handle matches this value is the alternative that is present. If the end of any open container (or the end of the PDU) is reached before the identification handle can be decoded, or if the value of the identification handle does not match that of any alternative, then this is an encoding error.

21.7 The RepetitionSpaceDetermination type

21.7.1 The "RepetitionSpaceDetermination" type is:

```
RepetitionSpaceDetermination ::= ENUMERATED
    {added-field, asn1-field, container, pattern, handle, not-needed}
```

21.7.2 The default value for this type is always "added-field".

21.7.3 An encoding parameter of this type specifies the way in which a decoder determines the end of the encoding space in an encoding of a class in the repetition category. It replaces use of an encoding parameter of type "EncodingSpaceDetermination" in the encoding of repetitions.

21.7.4 The value "added-field" requires the specification of a "REFERENCE" to another field that does not carry application semantics (i.e., does not appear within the ASN.1 specification). The encoding specification determines how an encoder is to set the value of this field from the size (in repetition space units) of the repetition space.

21.7.5 The value "asn1-field" requires the specification of a "REFERENCE" to another field whose value is set by the application (i.e., appears within the ASN.1 specification). The encoding specification determines how a decoder is to obtain the size (in repetition space units) of the encoding space from the value of this field. A conforming encoder shall not produce encodings in which the decoder's transforms of this field do not correctly identify the end of the encoding space.

21.7.6 The value "container" requires either the specification of a "REFERENCE" to another field whose encoding class (the container) has a length determinant and whose contents include the encoding class in the repetition category, or of a specification (using "OUTER") that the end of the PDU determines the end of the repetitions. The repetitions terminate when the specified container terminates or when, following the complete encoding of one repetition, the end of the PDU is encountered.

NOTE: The present document can only be used if the encoding of the (repetition category) class is the last encoding to be placed in the container. It is an ECN specification error if additional encodings are placed in the container, but conforming encoders shall not generate such encodings.

21.7.7 The value "pattern" specifies that some specified pattern of bits (see 21.10) will terminate the repetitions. In this case additional encoding parameters will require the insertion by an encoder of a specified pattern, and the detection of this pattern by a decoder. It is an ECN specification error if the encoding of the pattern can be the initial part of the encoding of an abstract value of a repetition. A conforming encoder shall detect such errors and shall not generate encodings that violate this rule.

NOTE: An example is a null-terminated character string whose contents are not allowed to include a null character.

21.7.8 The value "handle" requires that an identification handle be specified. This identification handle shall be exhibited by the element being repeated, and by all possible (taking account of optionality) following elements.

21.7.9 The value "not-needed" specifies that the number of repetitions is fixed in the abstract syntax.

NOTE: It is an ECN specification error (which shall be detected and blocked by encoders) if this encoding is specified and the number of repetitions are not so restricted, or if the application violates that restriction.

21.8 The Justification type

21.8.1 The "Justification" type is:

```
Justification ::= CHOICE
    { left      INTEGER (0..MAX),
      right     INTEGER (0..MAX) }
```

21.8.2 The default value for this type is always "right:0".

21.8.3 An encoding parameter of this type specifies right or left justification of the encoding of a value within the encoding space, with an offset in bits from the ends of the encoding space.

21.8.4 The "left" alternative specifies that the leading bit of the value encoding is positioned relative to the leading edge of the encoding space. The integer value specifies the number of bits between the leading edge of the encoding space and the leading bit of the value encoding.

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.

21.8.5 The "right" alternative specifies that the trailing bit of the value encoding is positioned relative to the trailing edge of the encoding space. The integer value specifies the number of bits between the trailing bit of the value encoding and the trailing edge of the encoding space.

21.8.6 The setting of the bits (if any) before or after the value encoding is determined by encoding parameters of type "Padding" and "Pattern" (see 21.9 and 21.10).

21.9 The Padding type

21.9.1 The "Padding" type is:

```
Padding ::= ENUMERATED {zero, one, pattern, encoder-option}
```

21.9.2 The default value for an encoding parameter of this type is always "zero".

21.9.3 An encoding parameter of this type specifies details of the padding for pre-padding, for classes in the pad category, and for the post-padding of a PDU specified in the #OUTER encoding class.

21.9.4 If the value is "zero", then the padding is with zero bit.

21.9.5 If the value is "one", then the padding is with one bit.

21.9.6 If the value is "pattern" then the bits are set according to the encoding parameter of type "Pattern" (see 21.10).

21.9.7 If the value is "encoder-option", then the encoder freely chooses the bit values.

21.10 The Pattern and Non-Null-Pattern types

21.10.1 The "Pattern" type is:

```
Pattern ::= CHOICE
{bits          BIT STRING,
 octets        OCTET STRING,
 char8         IA5String,
 char16        BMPString,
 char32        UniversalString,
 any-of-length (1..MAX),
 different     ENUMERATED {any} }
```

21.10.2 The "Non-Null-Pattern" type is:

```
Non-Null-Pattern ::= Pattern (ALL EXCEPT (bits:''B | octets:''H | char8:'' | char16:'' | char32:''))
```

21.10.3 The default value for an encoding parameter of this type is always "bits:'0'B".

21.10.4 The "bits" or "octets" alternative specifies a pattern of length and value equal to the given bitstring or octet string respectively.

21.10.5 The "char8" alternative specifies a (multiple of 8-bits) pattern where each character in the given string is converted to its ISO/IEC 10646-1 value as an 8-bit value.

21.10.6 The "char16" alternative specifies a (multiple of 16-bits) pattern where each character in the given string is converted to its ISO/IEC 10646-1 value as a 16-bit value.

21.10.7 The "char32" alternative specifies a (multiple of 32-bits) pattern where each character in the given string is converted to its ISO/IEC 10646-1 value as a 32-bit value.

21.10.8 The "any-of-length" alternative specifies a size for the pattern, but leaves the actual value of the pattern as an encoder's option.

21.10.9 The "different:any" value is permitted only when there is another encoding parameter of type "Pattern" in the same parameter group. In this case, either (but not both) of the encoding parameters of type "Pattern" can be set to "different:any". The "different:any" value specifies that the length of the pattern shall be the same as the length of the pattern specified for the other encoding parameter. It also specifies that its value is an encoder's option, provided that the value is different from the value of the pattern specified for the other encoding parameter.

21.10.10 When used for pre-padding and for justification (but not for other uses), the "Non-Null-Pattern" is used, and the pattern is truncated and/or replicated as necessary to provide sufficient bits for the pre-padding, value pre-padding, or value post-padding.

21.10.11 The "different:any" value of type "Pattern" is excluded from most uses of this type. When a parameter of type "Pattern" is used to specify the pattern for a boolean value ("TRUE", say), then the value "different:any" can be used to specify the pattern for the other boolean value ("FALSE" in this case). When used in this way, "different:any" 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.

21.11 The RangeCondition type

21.11.1 The "RangeCondition" type is:

```
RangeCondition ::= ENUMERATED
{unbounded-or-no-lower-bound,
 semi-bounded-with-negatives,
 bounded-with-negatives,
 semi-bounded-without-negatives,
 bounded-without-negatives}
```

21.11.2 The default value for an encoding parameter of this type is always "unbounded-or-no-lower-bound".

21.11.3 An encoding parameter of type "RangeCondition" is used in the specification of a predicate which tests the existence and nature of bounds on the integer values associated with an encoding class in the integer category.

21.11.4 The predicate is satisfied for each enumeration value if and only if the following conditions are satisfied by the bounds on the encoding class in the integer category:

- a) **unbounded-or-no-lower-bound:** either there are no bounds, or else there is only an upper bound but no lower bound;
- b) **semi-bounded-with-negatives:** there is a lower bound that is less than zero, but no upper bound;
- c) **bounded-with-negatives:** there is a lower bound that is less than zero, and an upper bound;
- d) **semi-bounded-without-negatives:** there is a lower bound that is greater than or equal to zero, but no upper bound;
- e) **bounded-without-negatives:** there is a lower bound that is greater than or equal to zero, and an upper bound.

NOTE: For any given set of bounds, exactly one predicate will be satisfied.

21.12 The SizeRangeCondition type

21.12.1 The "SizeRangeCondition" type is:

```
SizeRangeCondition ::= ENUMERATED
{no-ub-with-zero-lb,
 ub-with-zero-lb,
 no-ub-with-non-zero-lb,
 ub-with-non-zero-lb,
 fixed-size}
```

21.12.2 The default value for an encoding parameter of this type is always "no-ub-with-zero-lb".

21.12.3 An encoding parameter of type "SizeRangeCondition" is used to test properties of the bounds in an effective size constraint associated with a class in the repetition or characterstring category.

21.12.4 The predicate is satisfied for each enumeration value if and only if the effective size constraint satisfies the following conditions:

- a) **no-ub-with-zero-lb**: there is no upper bound on the size and the lower bound is zero;
- b) **ub-with-zero-lb**: there is an upper bound on the size and the lower bound is zero;
- c) **no-ub-with-non-zero-lb**: there is no upper bound on the size and the lower bound is non-zero;
- d) **ub-with-non-zero-lb**: there is an upper bound on the size and the lower bound is non-zero;
- e) **fixed-size**: the lower bound and the upper bound on the size are the same value.

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

21.13 The ReversalSpecification type

21.13.1 The "ReversalSpecification" type is:

```
ReversalSpecification ::= ENUMERATED
    {no-reversal,
     reverse-bits-in-units,
     reverse-half-units,
     reverse-bits-in-half-units}
```

21.13.2 The default value for an encoding parameter of this type is always "no-reversal".

21.13.3 An encoding parameter of type "ReversalSpecification" is used in the final transform of bits from an encoding space into an output buffer for transmission (with the reverse transform being applied for decoding).

NOTE: Bits inserted as a result of pre-padding specified by an encoding object do not form part of the encoding to which bit-reversal specified by that encoding object, but may be subject to bit-reversal specified by an encoding object for a container in which the complete encoding is embedded.

21.13.4 Values of this type are always used in conjunction with an encoding parameter of type "Unit" that specifies a unit size in bits (see 21.1).

21.13.5 It is an ECN specification error if the values "reverse-half-units" and "reverse-bits-in-half-units" are used when the encoding parameter of type "Unit" is not an even number of bits.

21.13.6 The enumerations specify (in the order of enumerations listed above) either:

- a) no reversal of bits, or
- b) reversal of the order of half-units (without changing the order of bits in each half unit), or
- c) reversal of the order of bits in each half-unit but without reversing the order of the half-units, or
- d) reversal of the order of the bits in each unit.

21.13.7 It is an ECN specification error if the number of bits in an encoding to which bit-reversal is applied is not an integral multiple of "Unit".

21.13.8 Bit-reversal can be specified for the encoding of all classes that can appear as fields of encoding structures, except an encoding class of the alternatives category, which does not use the encoding space concept.

21.14 The ResultSize type

21.14.1 The "ResultSize" type is:

```
ResultSize ::= INTEGER {variable(-1), fixed-to-max(0)} (-1..MAX)
```

21.14.2 The default value for an encoding parameter of this type is always "variable".

21.14.3 An encoding parameter of this type specifies the size of the result in a #TRANSFORM class.

21.14.4 The value "variable" specifies that the size of the #TRANSFORM result will vary for different abstract values, and is determined by the detailed specification of the transform.

21.14.5 The value "fixed-to-max" specifies that the size of the #TRANSFORM result is to be the same for the transform of all abstract values. It specifies that the target size is to be the smallest size that can contain the specified encoding of any one (all) of the abstract values. The precise details of the present document are defined for each transform in which values of this type are used.

21.14.6 A positive value of type "ResultSize" specifies that the size of the #TRANSFORM result is fixed. This value is used in the specification of the actual transform.

22 Commonly used encoding parameter groups

This clause specifies groups of encoding parameters that are commonly used in the defined syntax (see clause 20). The purpose of each group, the restrictions on both the values of encoding parameters and the syntax that can be used, as well as the encoder and decoder actions for each group are also specified.

22.1 Replacement specification

There are three variants of replacement specification:

- a) Full replacement specification: this is used for classes in the concatenation category, where replacement can be of the entire structure, or can selectively replace optional and non-optional components.
- b) Structure or component replacement specification: this is used for classes in the alternatives category and for the #CONDITIONAL-REPETITION encoding class, where replacement can be of the entire structure or of the component.

NOTE: When an encoding object of the #CONDITIONAL-REPETITION class is used to define encodings for a class in the bitstring, characterstring, or octetstring category, it can only perform structure-only replacement.

- c) Structure-only replacement specification: this is used for classes that do not have components.

22.1.1 Encoding parameters, syntax, and purpose

22.1.1.1 Full replacement specification uses the following encoding parameters:

&#Replacement-structure		OPTIONAL,
&#Replacement-structure2		OPTIONAL,
&replacement-structure-encoding-object	&#Replacement-structure	OPTIONAL,
&replacement-structure-encoding-object2	&#Replacement-structure2	OPTIONAL,
&#Head-end-structure		OPTIONAL,
&#Head-end-structure2		OPTIONAL

22.1.1.2 The syntax to be used for full replacement specification shall be:

```
[REPLACE
  [STRUCTURE]
  [COMPONENT]
  [ALL COMPONENTS]
  [OPTIONALS]
  [NON-OPTIONALS]
  WITH &#Replacement-structure
    [ENCODED BY &replacement-structure-encoding-object
      [INSERT AT HEAD &#Head-end-structure]]
  [AND OPTIONALS WITH &#Replacement-structure2
    [ENCODED BY &replacement-structure-encoding-object2
      [INSERT AT HEAD &#Head-end-structure2]]] ]
```


22.1.1.3 Structure or component replacement specification uses the following encoding parameters:

<code>&#Replacement-structure</code>		OPTIONAL,
<code>&replacement-structure-encoding-object</code>	<code>&#Replacement-structure</code>	OPTIONAL,
<code>&#Head-end-structure</code>		OPTIONAL

22.1.1.4 The syntax to be used for structure or component replacement specification shall be:

```
[REPLACE
  [STRUCTURE]
  [COMPONENT]
  [ALL COMPONENTS]
  WITH &Replacement-structure
  [ENCODED BY &replacement-structure-encoding-object
    [INSERT AT HEAD &#Head-end-structure]]]
```

22.1.1.5 Structure-only replacement specification uses the following encoding parameters:

<code>&#Replacement-structure</code>	OPTIONAL,
<code>&replacement-structure-encoding-object</code>	<code>&#Replacement-structure</code> OPTIONAL

22.1.1.6 The syntax to be used for structure-only replacement specification shall be:

```
[REPLACE
  [STRUCTURE]
  WITH &#Replacement-structure
  [ENCODED BY &replacement-structure-encoding-object]]
```

22.1.1.7 Use of the "WITH SYNTAX" for these encoding parameter groups specifies that either:

- the encoding class to which this encoding object is applied is to be replaced completely ("REPLACE STRUCTURE"); in the case of an encoding class in the optionality category, the entire element is replaced; in the case of a #CONDITIONAL-REPETITION encoding object used in defining an encoding object for a class in the bitstring, characterstring, octetstring or repetition category, then (if the range condition is satisfied), the entire bitstring, characterstring, octetstring or repetition structure is replaced; or
- all its components (except for the structure-only specification) are to be replaced (with the same replacement action for all components) ("REPLACE COMPONENT" or "REPLACE ALL COMPONENTS"); or
- all its optional components (only for full replacement specification) are to be replaced ("REPLACE OPTIONALS"); or
- all its non-optional components (only for full replacement specification) are to be replaced ("REPLACE NON-OPTIONALS"); or
- all its components (only for full replacement specification) are to be replaced, with different replacement actions for optionals and for non-optionals ("REPLACE NON-OPTIONALS AND OPTIONALS").

22.1.1.8 "REPLACE COMPONENT" is a synonym for "REPLACE ALL COMPONENTS". It would be normal but not required to use this if there is only a single component.

22.1.1.9 The optional "ENCODED BY"s specify an encoding object for the replacement structure.

22.1.1.10 The optional "INSERT AT HEAD"s specify an encoding structure (the head-end insertion) to be inserted before all components of the (constructor) class performing the replacement. There is one head-end insertion for each component that is replaced, and they are inserted in the order of the original components.

22.1.2 Specification restrictions

22.1.2.1 Exactly one of the permitted syntaxes between "REPLACE" and "WITH" shall be used.

22.1.2.2 The "WITH" replacement structures shall be parameterized encoding structures with a single encoding class parameter. When they are specified in the above defined syntax, the class reference name only of the structure shall be given. It shall not have any parameter list in this use of the names.

22.1.2.3 These parameterized structures are instantiated during the replacement action with an actual parameter as specified in 22.1.3. The use of the dummy parameter in the replacement parameterized structures shall be consistent with the class of the actual parameter that will be supplied in the replacement action.

NOTE: In particular, if "REPLACE STRUCTURE" is used for an encoding class in the tag category, the dummy parameter can only occur in the replacement structure where an encoding class in the tag category is permitted.

22.1.2.4 The "ENCODED BY" encoding objects shall be parameterized encoding objects for the "WITH" encoding structures. They shall have a dummy parameter (#D, say) that is an encoding class, and they shall have been defined in a parameterized encoding object assignment in which the governor was the corresponding "WITH" parameterized encoding structure, instantiated with #D. When they are specified in the above defined syntax, the encoding object reference name only shall be given. They shall not have any parameter list in this use of the names.

22.1.2.5 They are instantiated during the replacement action with an actual parameter which is the same as the actual parameter used to instantiate the corresponding "WITH" replacement encoding structures. They may also have:

- (optionally) another (but only one) dummy parameter that is an encoding object set; when they are instantiated during the replacement action, the actual parameter for this dummy parameter is the current combined encoding object set;
- (conditionally) another (but only one) dummy parameter that is a "REFERENCE" parameter. This parameter shall be present if and only if "INSERT AT HEAD" is specified. When the encoding objects are instantiated during the replacement action, the actual parameter for this dummy parameter is a reference to the corresponding "INSERT AT HEAD" structure.

22.1.2.6 All fields of the replacement structure that are not part of the encoding class parameter are auxiliary fields, and shall be set by the encoding of the replacement structure.

22.1.2.7 The "INSERT AT HEAD" encoding structures shall not have dummy parameters. All their fields are auxiliary fields, and shall be set by the "ENCODED BY" encoding object through its "REFERENCE" parameter.

22.1.2.8 If an encoding object has a "REPLACE STRUCTURE" clause, it shall not have an "INSERT AT HEAD" clause.

22.1.3 Encoder actions

22.1.3.1 If an encoding object of a class in the bit-field or tag categories specifies "REPLACE STRUCTURE", then an encoder shall replace the structure with an instantiation of the replacement structure, using the name of the original structure as the actual parameter.

22.1.3.2 If an encoding object of a class in the encoding constructor category specifies "REPLACE STRUCTURE", then an encoder shall replace the entire construction with an instantiation of the replacement structure, using the entire original construction as the actual parameter.

22.1.3.3 If an encoding object of a class in the optionality category specifies "REPLACE STRUCTURE", then an encoder shall replace the entire optional component with a non-optional instantiation of the replacement structure. The actual parameter shall be a hidden structure name (which matches no other structure, and which can never have encoding objects). This hidden structure name shall de-reference to the entire original optional component (including any classes in the tag category) except for the class in the optionality category.

22.1.3.4 If an encoding object of any class specifies "REPLACE COMPONENT", "REPLACE ALL COMPONENTS", "REPLACE OPTIONAL COMPONENTS", or "REPLACE NON-OPTIONAL COMPONENTS", then an encoder shall replace the entire specified component(s) with a non-optional instantiation of the replacement structure. The actual parameter shall be a hidden structure name (which matches no other structure, and which can never have encoding objects). This hidden structure name shall de-reference to the entire original optional component (including any classes in the tag category) except for any class in the optionality category.

22.1.3.5 All abstract values and tag values of the original structure or component shall be mapped to corresponding abstract values and tag numbers in the actual parameter of the replacement structure. Values of other fields in the replacement structure shall be set according to the specification in the replacement structure encoding object.

22.1.3.6 If a head-end insertion is specified, then the encoder shall insert the head-end structure before all components of the structure whose encoding object is performing the replacement. Head-end insertions shall be inserted in the same textual order as the components being replaced. The values of fields of this structure shall be set in accordance with the specification in the replacement structure encoding object.

NOTE: These structures will normally be a simple integer field providing a location determinant for the field being replaced.

22.1.3.7 The encoder shall instantiate the replacement structure encoding-object(s) with actual parameters as follows:

- a) The dummy parameter that is an encoding class shall be given an actual parameter that is the same as the actual parameter of the instantiation of the replacement structure.
- b) The dummy parameter (if any) that is a "REFERENCE" parameter shall be given an actual parameter that is a reference to the inserted head-end structure.
- c) The dummy parameter (if any) that is an encoding object set (whose governor is #ENCODINGS) shall be given an actual parameter that is the current combined encoding object set.

22.1.3.8 The encoder shall then use this instantiated encoding object to encode the corresponding replacement structure instead of the combined encoding object set.

NOTE: The encoding of the head-end insertions is determined by the application of the current combined encoding object set.

22.1.4 Decoder actions

A decoder shall generate (for an application) the abstract values of the original structure that was being encoded, hiding any replacement activity (even if performed by repeated application of replacements).

22.2 Pre-alignment and padding specification

22.2.1 Encoding parameters, syntax, and purpose

22.2.1.1 Pre-alignment and padding specification uses the following encoding parameters:

&encoding-space-pre-alignment-unit	Unit (ALL EXCEPT repetitions) DEFAULT bit,
&encoding-space-pre-padding	Padding DEFAULT zero,
&encoding-space-pre-pattern	Non-Null-Pattern (ALL EXCEPT different:any) DEFAULT bits:'0'B

22.2.1.2 The syntax to be used for pre-alignment and padding specification shall be:

```
[ALIGNED TO
  [NEXT]
  [ANY]
  &encoding-space-pre-alignment-unit
  [PADDING &encoding-space-pre-padding
  [PATTERN &encoding-space-pre-pattern]]
```

22.2.1.3 The definition of types used in pre-alignment and padding specification is:

```
Unit ::= INTEGER      {repetitions(0), bit(1), nibble(4), octet(8),
                        word16(16), dword32(32)} (0..256)      -- (see 21.1)
Padding ::= ENUMERATED {zero, one, pattern, encoder-option} -- (see 21.9)
Pattern ::= CHOICE
{bits          BIT STRING,
 octets        OCTET STRING,
 char8         IA5String,
 char16        BMPString,
 char32        UniversalString,
 any-of-length INTEGER (1..MAX),
 different     ENUMERATED {any} }

Non-Null-Pattern ::= Pattern (ALL EXCEPT (bits:''B | octets:''H | char8:'' |
                                char16:'' | char32:'')) -- (see 21.10)
```


22.2.1.4 The pre-alignment encoding parameters use a value of type "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 (see clause 25). Encoding parameters of type "Padding" and "Pattern" are used to control the bits that provide padding to the required alignment. Specification of "ALIGNED TO NEXT" produces the minimum number of inserted bits. Specification of "ALIGNED TO ANY" leaves the actual number of inserted bits (subject to the above restriction to a multiple of "Unit") as an encoders option, and requires the specification of a start pointer.

22.2.2 Specification constraints

22.2.2.1 At most one of "NEXT" and "ANY" shall be specified. When not specified, "NEXT" is assumed.

22.2.2.2 If "ALIGNED TO ANY" is specified, then the encoding object specification shall include the "START-POINTER" clause.

22.2.3 Encoder actions

22.2.3.1 If "NEXT" is specified (or is defaulted), the encoder shall insert the minimum number of bits necessary to ensure that the total number of bits in the encoding (from the alignment point) is a multiple of the encoding parameter of type "Unit".

22.2.3.2 If "ANY" is specified, the encoder shall insert an encoder-dependent number of bits, provided that the total number of bits in the encoding (from the alignment point) is a multiple of the encoding parameter of type "Unit".

22.2.3.3 The inserted bits shall be set so that the first inserted bit is the leading bit of "Pattern", and so on. If more bits are needed than are present in the encoding parameter of type "Pattern", then the pattern shall be re-used, most significant bit first.

22.2.4 Decoder actions

22.2.4.1 The decoder shall determine the number of inserted bits from the encoder actions if "NEXT" is specified.

22.2.4.2 The decoder shall determine the number of inserted bits from the start pointer specification if "ANY" is specified.

22.2.4.3 In all cases, the decoder shall discard the inserted bits transparently to the application. It shall not diagnose an encoder or a specification error if the bits are not in agreement with the specified encoders actions.

22.3 Start pointer specification

22.3.1 Encoding parameters, syntax, and purpose

22.3.1.1 Start pointer specification uses the following encoding parameters:

<code>&start-pointer</code>	REFERENCE OPTIONAL,
<code>&start-pointer-unit</code>	Unit (ALL EXCEPT repetitions) DEFAULT bit,
<code>&Start-pointer-encoder-transforms</code>	#TRANSFORM ORDERED OPTIONAL

22.3.1.2 The syntax to be used for start pointer specification shall be:

```
[START-POINTER      &start-pointer
 [MULTIPLE OF      &start-pointer-unit]
 [ENCODER-TRANSFORMS  &Start-pointer-encoder-transforms]]
```

22.3.1.3 The definition of the type used in start pointer specification is:

```
Unit ::= INTEGER {repetitions(0), bit(1), nibble(4), octet(8),
                  word16(16), dword32(32)} (0..256) -- (see 21.1)
```


22.3.1.4 The present document identifies the start of the encoding space for an element. If the start of the encoding space for the element is an offset of "n" "MULTIPLE OF" units, then the value placed in the "START-POINTER" reference field is the value obtained by applying "ENCODER-TRANSFORMS" to "n".

NOTE 1: If "MULTIPLE OF" is not "bits", this implies that that offset from the start of the "START-POINTER" reference field to the start of the encoding space is required to be an integral multiple of "MULTIPLE OF" units.

NOTE 2: There will in general be encodings of other elements, and perhaps of other start-pointers between the "START-POINTER" reference field and the start of the encoding of this element.

22.3.2 Specification constraints

22.3.2.1 If "ENCODER-TRANSFORMS" is not present, then "START-POINTER" shall be a class in the integer category.

22.3.2.2 If "ENCODER-TRANSFORMS" is present, then "START-POINTER" shall be a class with a category that can encode a value of the result of the final transform in "ENCODER-TRANSFORMS".

22.3.2.3 All transforms in the "ENCODER-TRANSFORMS" shall be reversible, and the first transform shall have a source which is integer.

22.3.3 Encoder actions

22.3.3.1 The encoder shall determine the number "n" of "MULTIPLE OF" units from the start of the encoding of the "START-POINTER" field (after any pre-alignment of that field) to the start of the encoding of the element with the start-pointer specification (after any pre-alignment of that element). It is an ECN specification error if "n" is not integral. If the element being encoded is optional, and is absent, then "n" shall be set to zero.

22.3.3.2 The value "n" shall be transformed using the "ENCODER-TRANSFORMS" (if present) to produce a conceptual value "m". If this resulting value "m" is not an abstract value that can be associated with the encoding class of the "START-POINTER", then it is an ECN specification error, and encoding shall not proceed. Otherwise the value "m" shall be the value encoded in the field referenced by "START-POINTER".

NOTE: The encoding object applied to the field referenced by "START-POINTER" will determine the encoding of the value "m".

22.3.4 Decoder actions

22.3.4.1 The decoder shall determine the conceptual value "m" in the field referenced by "START-POINTER", and shall use knowledge of the encoder's actions to reverse the transforms (if any) to produce the integer value "n".

22.3.4.2 If "n" is zero, then the decoder shall diagnose an encoder's error if the element being decoded is not an optional element with an optionality specification determining optionality by the start pointer. If "n" is zero, and the element being decoded is an optional element with an optionality specification determining optionality by the start pointer, then the decoder shall determine that the element is absent.

22.3.4.3 The value "n" is multiplied by "MULTIPLE OF", and the start of the encoding of the "START-POINTER" field is added to produce a position "p". If "p" is a position in the encoding that is earlier than the current decoding point, then the decoder shall diagnose an encoding error.

22.3.4.4 If "p" is a position in the encoding that is equal to or beyond the current decoding point, then the decoder shall silently ignore all bits up to position "p", and shall continue decoding of this element from position "p".

22.4 Encoding space specification

22.4.1 Encoding parameters, syntax, and purpose

22.4.1.1 Encoding space specification uses the following encoding parameters:

&encoding-space-size	EncodingSpaceSize DEFAULT self-delimiting-values,
&encoding-space-unit	Unit (ALL EXCEPT repetitions) DEFAULT bit,
&encoding-space-determination	EncodingSpaceDetermination DEFAULT added-field,
&encoding-space-reference	REFERENCE OPTIONAL,
&Encoder-transforms	#TRANSFORM ORDERED OPTIONAL,
&Decoder-transforms	#TRANSFORM ORDERED OPTIONAL

22.4.1.2 The syntax to be used for encoding space specification shall be:

```
ENCODING-SPACE
[SIZE &encoding-space-size
  [MULTIPLE OF &encoding-space-unit]]
[DETERMINED BY &encoding-space-determination]
[USING &encoding-space-reference
  [ENCODER-TRANSFORMS &Encoder-transforms]
  [DECODER-TRANSFORMS &Decoder-transforms]]
```

22.4.1.3 The definition of types used in the present document is:

```
EncodingSpaceSize ::= INTEGER
  {encoder-option-with-determinant(-3),
   variable-with-determinant(-2),
   self-delimiting-values(-1),
   fixed-to-max(0)} (-2..MAX) -- (see 21.2)

Unit ::= INTEGER
  {repetitions(0), bit(1), nibble(4), octet(8), word16(16),
   dword32(32)} (0..256) -- (see 21.1)

EncodingSpaceDetermination ::= ENUMERATED
  {added-field, asnl-field, container} -- (see 21.3)
```

22.4.1.4 The purpose of the present document is to determine encoder and decoder actions to ensure that a decoder can correctly determine the end of an encoding space.

NOTE: An actual value encoding does not necessarily fill the entire encoding space, and recovery of the value encoding by a decoder will in general also require actions specified for value padding and justification (see 22.8).

22.4.1.5 The meaning of the encoding parameters of type "Unit", "EncodingSpaceSize", and "EncodingSpaceDetermination" were given in 21.1, 21.2, and 21.3. Together these specify the way in which the end of the encoding space for this element is determined.

NOTE: "Variable-with-determinant" can be specified even if the encoding space is fixed size, if the ECN specifier requires that a length determinant is to be included, even if not needed.

22.4.1.6 The "USING" specification is a reference which enables a decoder to determine the end of the encoding space. It is a reference to an auxiliary field or to a field carrying abstract values, or to a container, depending on the value of "DETERMINED BY".

22.4.2 Specification restrictions

22.4.2.1 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is not present, then the default value ("added-field") is assumed.

22.4.2.2 "USING" shall be specified if and only if "SIZE" is "variable-with-determinant".

22.4.2.3 "ENCODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to (or defaults to) "added-field". The "USING" reference in this case shall be an auxiliary field of category bitstring, characterstring or integer.

22.4.2.4 All transforms specified in "ENCODER-TRANSFORMS" shall be reversible transforms. The first transform shall have a source which is integer and the last transform shall have a result which can be encoded by the class of the field referenced by "USING".

22.4.2.5 "DECODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to "asn1-field". The first transform shall have a source which is the same as the category of the field referenced by "USING" which shall not be an auxiliary field. The last transform shall have a result which is integer.

22.4.2.6 The "USING" reference, if present, shall be a field that is present in the encoding earlier than the field being encoded. It is an application or an ECN specification error if, in an instance of encoding, the field being encoded is present but the "USING" reference field is absent (through the exercise of optionality).

22.4.2.7 If "DETERMINED BY" is "container", the "USING" reference shall be to a concatenation or to a repetition (or to a bitstring or octetstring with a contained type) in which the element being encoded is a component (or a component of a component, to any depth). It is an application or an ECN specification error if, in an instance of encoding, later elements within the same concatenation or repetition are to be encoded.

22.4.2.8 The present document is considered set if the "ENCODING-SPACE" keyword is used, and it is mandatory for it to be set in all places in the defined syntax where it is allowed. Defaulting all encoding parameters of this group (e.g., use of "ENCODING-SPACE" alone) would not satisfy the above constraints.

22.4.3 Encoder actions

22.4.3.1 Encoders shall not generate encodings if the conditions of 22.4.2 are not satisfied.

22.4.3.2 If "SIZE" is a positive value, then the encoding space is that multiple of "MULTIPLE OF" units and there is no further encoder action.

22.4.3.3 If "SIZE" is not set to a positive value, then the encoder shall determine the size ("s", say) of the encoding space in "MULTIPLE OF" units from the value encoding specification. This determination is specified in the clauses on value encoding specification.

22.4.3.4 If "SIZE" is "encoder-option-with-determinant" then the encoder (as an encoder's option) may increase the size "s" (as determined in 22.4.3.3) in "MULTIPLE OF" units from that determined from the value encoding specification to any value which can be encoded in the associated determinant.

22.4.3.5 If "SIZE" is "fixed-to-max" or to "self-delimiting-values", then there is no further encoder action.

22.4.3.6 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is "container", then there is no further encoder action.

22.4.3.7 If "DETERMINED BY" is "added-field", then the encoder shall apply the transforms specified by "ENCODER-TRANSFORMS" (if any) to the value "s" to produce a value that shall be encoded in the "USING" reference.

NOTE: The encoding of the "USING" reference (bit-field "A", say) in this case appears earlier in the encoding than the encoding of this field (bit-field "B", say), and an encoder will need to defer the encoding of bit-field "A" until the value to be encoded has been determined by the encoding of bit-field "B".

22.4.3.8 If "DETERMINED BY" is "asn1-field" then the encoder shall check that the value in the "USING" reference when transformed by the "DECODER-TRANSFORMS" (if any) is equal to "s". It is an application error if this condition is not met, and encoding shall not proceed.

22.4.4 Decoder actions

22.4.4.1 If "SIZE" is a positive value, then the decoder determines the encoding space as that multiple of "MULTIPLE OF" units.

22.4.4.2 If "SIZE" is "fixed-to-max" or to "self-delimiting-values", then the decoder shall determine the end of the encoding space in accordance with the specification of the value encoding. This determination is specified in the clauses on value encoding specification.

22.4.4.3 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is set to "container", then the decoder shall use the end of the container specified by "USING" as the end of the encoding space.

22.4.4.4 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is set to (or defaults to) "added-field", then the decoder shall recover the value "s" by applying the reversal of the "ENCODER-TRANSFORMS" (if any) to the value of the "USING" reference.

22.4.4.5 If "DETERMINED BY" is "asn1-field" then the decoder shall recover the value "s" by applying the "DECODER-TRANSFORMS" (if any) to the value of that field.

22.5 Optionality determination

22.5.1 Encoding parameters, syntax, and purpose

22.5.1.1 Optionality determination uses the following encoding parameters:

<code>&optionality-determination</code>	OptionalityDetermination
<code>&optionality-reference</code>	DEFAULT added-field,
<code>&Encoder-transforms</code>	REFERENCE OPTIONAL,
<code>&Decoder-transforms</code>	#TRANSFORM ORDERED OPTIONAL,
<code>&handle-id</code>	#TRANSFORM ORDERED OPTIONAL,
	PrintableString
	DEFAULT "default-handle"

22.5.1.2 The syntax to be used for optionality determination shall be:

```

PRESENCE
  [DETERMINED BY &optionality-determination
    [HANDLE &handle-id]]
  [USING &optionality-reference
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [DECODER-TRANSFORMS &Decoder-transforms]]

```

22.5.1.3 The definition of types used in optionality determination is:

```

OptionalityDetermination ::= ENUMERATED
  {added-field, asn1-field, container, handle, pointer} -- (see 21.5)

```

22.5.1.4 The purpose of the present document is to specify rules that ensure that a decoder can correctly determine whether an encoder has encoded a value of an optional element. Where a pointer is used to determine optionality, pre-alignment and start pointer specification is also required.

22.5.1.5 An encoder will encode the value of an optional element if required to do so by the application, unless such an encoding would be in violation of rules governing the presence of optional elements.

NOTE: An example of violation of such a rule would be where the presence of an (absent) optional element was to be determined by the end of a container, and the application requested that later optional elements in the same container be encoded.

22.5.1.6 The present document is considered set if the "PRESENCE" keyword is used, and it is mandatory for it to be set in all places in the defined syntax where it is allowed. Defaulting all other parts of this defined syntax (e.g., use of "PRESENCE" alone) would not satisfy the above constraints.

22.5.2 Specification restrictions

22.5.2.1 If "DETERMINED BY" is not present, then the default value ("added-field") is assumed.

22.5.2.2 "HANDLE" shall not be specified unless "DETERMINED BY" is "handle".

22.5.2.3 "USING" shall not be specified if "DETERMINED BY" is "handle" or "pointer".

22.5.2.4 If "DETERMINED BY" is "pointer", there shall be a "START-POINTER" specification in the same encoding object (see 22.3).

NOTE: A start pointer specification normally also needs a pre-alignment specification with "ALIGNED TO ANY" (see 22.2).

22.5.2.5 If "HANDLE" is specified, then the element whose presence is being determined, together with all following optional and the next mandatory element (if any) shall all be encoded by encoding objects whose specification exhibits and defines an identification handle with the same name as "HANDLE", and with the same set of bits forming the identification handle. The value of the bits forming the identification handle shall be different for all these elements.

NOTE: It is a requirement that the bits that form an identification handle shall have the same value for all abstract values encoded by an encoding object exhibiting that identification handle.

22.5.2.6 "ENCODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to (or defaults to) "added-field". The "USING" reference in this case shall be an auxiliary field of category bitstring, boolean, characterstring or integer.

22.5.2.7 All transforms specified in "ENCODER-TRANSFORMS" shall be reversible transforms. The first transform shall have a source which is boolean and the last transform shall have a result which can be encoded by the class of the field referenced by "USING".

22.5.2.8 "DECODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to "asn1-field". The first transform shall have a source which is the same as the category of the field referenced by "USING" which shall not be an auxiliary field. The last transform shall have a result which is boolean.

22.5.2.9 The "USING" reference, if present, shall be a field that is present in the encoding earlier than the field whose presence is being determined. It is an application or an ECN specification error if, in an instance of encoding, the "USING" reference field is required by a decoder but is absent (through the exercise of optionality).

22.5.2.10 If "DETERMINED BY" is "container", the "USING" reference shall be to a concatenation or to a repetition (or to a bitstring or octetstring with a contained type) in which the element being encoded is a component (or a component of a component, to any depth). It is an application or an ECN specification error if, in an instance of encoding, later elements within the same concatenation or repetition are to be encoded when the element whose optionality is being determined is absent.

22.5.2.11 If "DETERMINED BY" is "container", then it is an ECN specification error if any of the abstract values of the optional element have an encoding that is zero bits.

22.5.3 Encoder actions

22.5.3.1 Encoders shall not generate encodings if the conditions of 22.5.2 are not satisfied.

22.5.3.2 An encoder shall determine whether the application wishes the optional element to be encoded, and shall create a conceptual boolean value "element-is-present" set to "TRUE" if a value of the element is to be encoded, and to "FALSE" otherwise.

22.5.3.3 If "DETERMINED BY" is "added-field", then the encoder shall apply the transforms specified by "ENCODER-TRANSFORMS" (if any) to the conceptual boolean value "element-is-present" to produce a value that shall be encoded in the "USING" reference.

NOTE: The encoding of the "USING" reference in this case appears earlier in the encoding than the encoding of this field, and an encoder will need to suspend the encoding of that field until the value to be encoded has been determined by the encoding of this field.

22.5.3.4 If "DETERMINED BY" is "asn1-field" then the encoder shall check that the value in the USING reference when transformed by the "DECODER-TRANSFORMS" (if any) is a boolean value equal to the conceptual value "element-is-present". It is an application error if this condition is not met, and encoding shall not proceed.

22.5.3.5 If "DETERMINED BY" is "container" there is no further action needed by the encoder, except to detect an error and to cease encoding if the application requests the encoding of further elements in the "USING" container when the conceptual value "element-is-present" is false for this optional element.

22.5.3.6 If "DETERMINED BY" is "handle" there is no further action needed by the encoder.

22.5.3.7 If "DETERMINED BY" is "pointer" then there are no encoder actions needed except those of the accompanying pre-alignment (if any) and start pointer specifications.

22.5.4 Decoder actions

22.5.4.1 If "DETERMINED BY" is set to (or defaults to) "added-field", then the decoder shall recover the value "element-is-present" by applying the reversal of the "ENCODER-TRANSFORMS" (if any) to the value of the USING reference.

22.5.4.2 If "DETERMINED BY" is "asn1-field" then the decoder shall recover the conceptual value "element-is-present" by applying the "DECODER-TRANSFORMS" (if any) to the value of that field.

22.5.4.3 If "DETERMINED BY" is "container" then the decoder shall set the conceptual value "element-is-present" to "TRUE" if and only if there is at least one bit remaining in the "USING" container.

22.5.4.4 If "DETERMINED BY" is "handle", then the decoder shall determine the value of the bits associated with the specified "HANDLE". If these bits match those encoded by (all) abstract values of the optional element, then the decoder shall set the conceptual value "element-is-present" to "TRUE", otherwise the decoder shall set it to "FALSE".

22.5.4.5 If "DETERMINED BY" is "pointer" then the decoder shall proceed as specified in 22.3 in order to determine the conceptual value of "element-is-present".

22.5.4.6 If the decoder determines (by any of the above means) that the conceptual value "element-is-present" is "FALSE", then decoding proceeds to the next element, otherwise the decoder expects an encoding of a value of the optional element and will diagnose an encoding error if one is not present.

22.6 Alternative determination

22.6.1 Encoding parameters, syntax, and purpose

22.6.1.1 Alternative determination uses the following encoding parameters:

<code>&alternative-determination</code>	AlternativeDetermination
<code>&alternative-reference</code>	DEFAULT added-field,
<code>&Encoder-transforms</code>	REFERENCE OPTIONAL,
<code>&Decoder-transforms</code>	#TRANSFORM ORDERED OPTIONAL,
<code>&handle-id</code>	#TRANSFORM ORDERED OPTIONAL,
	PrintableString
<code>&alternative-ordering</code>	DEFAULT "default-handle",
	ENUMERATED {textual, tag}
	DEFAULT textual

22.6.1.2 The syntax to be used for alternative determination shall be:

```

ALTERNATIVE
  [DETERMINED BY &alternative-determination
    [HANDLE &handle-id]]
  [USING &alternative-reference
    [ORDER &alternative-ordering]
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [DECODER-TRANSFORMS &Decoder-transforms]]

```

22.6.1.3 The definition of types used for alternative determination is:

```

AlternativeDetermination ::= ENUMERATED {added-field, asn1-field, handle} -- (see 21.6)

```

22.6.1.4 The purpose of the present document is to determine the rules that ensure that a decoder can correctly identify which component of an encoding class in the alternatives category has been encoded.

22.6.2 Specification restrictions

22.6.2.1 If "DETERMINED BY" is not present, then the default value ("added-field") is assumed.

22.6.2.2 "HANDLE" shall not be specified unless "DETERMINED BY" is "handle".

22.6.2.3 "USING" shall not be specified if "DETERMINED BY" is "handle".

22.6.2.4 If "HANDLE" is specified, then all the alternatives of the encoding class in the alternatives category shall be encoded by encoding objects whose specification exhibits and defines an identification handle with the same name as "HANDLE", and with the same set of bits forming the identification handle. The value of the bits forming the identification handle shall be different for all these alternatives.

NOTE: It is a requirement that the bits that form an identification handle shall have the same value for all abstract values encoded by an encoding object exhibiting that identification handle.

22.6.2.5 "ENCODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to (or defaults to) "added-field". The first transform shall have a source which is integer and the last transform shall have a result which can be encoded by the class of the field referenced by "USING".

22.6.2.6 All transforms specified in "ENCODER-TRANSFORMS" shall be reversible transforms.

22.6.2.7 "DECODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to "asn1-field". The first transform shall have a source which is the same as the category of the field referenced by "USING" which shall not be an auxiliary field. The last transform shall have a result which is integer.

22.6.2.8 The "USING" reference, if present, shall be a field that is present in the encoding earlier than the encoding of the alternative. It is an application or an ECN specification error if, in an instance of encoding, the "USING" reference field is required by a decoder but is absent (through the exercise of optionality).

22.6.2.9 The present document is considered set if the "ALTERNATIVE" keyword is used, and it is mandatory for it to be set in all places in the defined syntax where it is allowed. Defaulting all other parts of this defined syntax (e.g., use of "ALTERNATIVE" alone) would not satisfy the above constraints.

22.6.2.10 If "ORDER" is "tag", then every alternative shall start with an encoding class in the tag category. The tag number associated with this class is called the component-tag.

22.6.2.11 The component-tags of each alternative shall be distinct.

22.6.3 Encoder actions

22.6.3.1 Encoders shall not generate encodings if the conditions of 22.6.2 are not satisfied.

22.6.3.2 An encoder shall determine which alternative the application wishes to be encoded, and shall create a conceptual integer value "alternative-index" to identify that alternative.

22.6.3.3 The value "alternative-index" shall be zero for the first alternative, one for the next, and so on, where the order of the alternatives is determined by "ORDER".

22.6.3.4 If "ORDER" is "textual", the textual order in the ASN.1 type specification or the ECN structure definition shall be used. If "ORDER" is "tag", then the order shall be that of the tag numbers in the component-tags (lowest tag number first).

22.6.3.5 If "DETERMINED BY" is "added-field", then the encoder shall apply the transforms specified by "ENCODER-TRANSFORMS" (if any) to the conceptual value "alternative-index" to produce a value that shall be encoded in the "USING" reference.

NOTE: The encoding of the "USING" reference in this case appears earlier in the encoding than the encoding of the alternative, and an encoder will need to suspend the encoding of that field until the alternative to be encoded has been determined.

22.6.3.6 If "DETERMINED BY" is "asn1-field" then the encoder shall check that the value in the USING reference when transformed by the "DECODER-TRANSFORMS" (if any) is an integer value equal to the conceptual value "alternative-index". It is an application error if this condition is not met, and encoding shall not proceed.

22.6.3.7 If "DETERMINED BY" is "handle" there is no further action needed by the encoder.

22.6.4 Decoder actions

22.6.4.1 The decoder shall use "ORDER" as specified for encoder actions to determine the alternative-index value that is associated with each alternative, and shall assume the presence of an encoding of the associated alternative once an "alternative-index" conceptual value has been determined.

22.6.4.2 If "DETERMINED BY" is set to (or defaults to) "added-field", then the decoder shall recover the value "alternative-index" by applying the reversal of the "ENCODER-TRANSFORMS" (if any) to the value of the "USING" reference.

22.6.4.3 If "DETERMINED BY" is "asn1-field" then the decoder shall recover the conceptual value "alternative-index" by applying the "DECODER-TRANSFORMS" (if any) to the value of that field.

22.6.4.4 If "DETERMINED BY" is "handle", then the decoder shall determine the value of the bits associated with the specified "HANDLE". These bits shall be compared to those encoded by each of the alternatives. If none match, then the decoder shall diagnose an encoder's error. Otherwise the conceptual value "alternative-index" shall be set to the matching alternative.

22.7 Repetition space specification

22.7.1 Encoding parameters, syntax, and purpose

22.7.1.1 Repetition space specification uses the following encoding parameters:

&repetition-space-size	EncodingSpaceSize
&repetition-space-unit	DEFAULT self-delimiting-values, Unit
&repetition-space-determination	RepetitionSpaceDetermination DEFAULT added-field,
&main-reference	REFERENCE OPTIONAL,
&Encoder-transforms	#TRANSFORM ORDERED OPTIONAL,
&Decoder-transforms	#TRANSFORM ORDERED OPTIONAL,
&handle-id	PrintableString DEFAULT "default-handle",
&termination-pattern	Non-Null-Pattern (ALL EXCEPT different:any) DEFAULT '0'B

22.7.1.2 The syntax to be used for repetition space specification shall be:

```

REpetition-SPACE
  [SIZE &repetition-space-size
    [MULTIPLE OF &repetition-space-unit]]
  [DETERMINED BY &repetition-space-determination
    [HANDLE &handle-id]]
  [USING &main-reference
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [DECODER-TRANSFORMS &Decoder-transforms]]
  [PATTERN &termination-pattern]

```

22.7.1.3 The definition of types used in the present document is:

```

EncodingSpaceSize ::= INTEGER
  {encoder-option-with-determinant(-3),
   variable-with-determinant(-2),
   self-delimiting-values(-1),
   fixed-to-max(0)} (-2..MAX) -- (see 21.2)

Unit ::= INTEGER
  {repetitions(0), bit(1), nibble(4), octet(8), word16(16),
   dword32(32)} (0..256) -- (see 21.1)

RepetitionSpaceDetermination ::= ENUMERATED
  {added-field, asn1-field, container, pattern, handle, not-needed} -- (see 21.7)

Non-Null-Pattern ::= Pattern
  (ALL EXCEPT (bits:''B | octets:''H | char8:'' | char16:'' | char32:'')) -- (see 21.10)

```

22.7.1.4 The purpose of the present document is to determine encoder and decoder actions to ensure that a decoder can correctly determine the end of the encoding space occupied by a repetition.

NOTE: An actual repetition encoding does not necessarily fill the entire encoding space, and recovery of the repetition encoding by a decoder will in general also require actions specified for value padding and justification (see 22.8).

22.7.1.5 The meaning of the encoding parameters of type "Unit", "EncodingSpaceSize", and "RepetitionSpaceDetermination" were given in 21.1, 21.2, and 21.7. Together these specify the way in which the end of the encoding space for repetitions is determined.

NOTE: If the ECN specifier requires that a length determinant is to be included, the value "variable-with-determinant" of "SIZE" can be specified even if the repetition space is fixed size.

22.7.1.6 The "USING" specification is a reference to an auxiliary field or to a field carrying abstract values, or to a container, depending on the value of "DETERMINED BY".

22.7.2 Specification constraints

22.7.2.1 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is not present, then the default value ("added-field") is assumed.

22.7.2.2 "USING" shall be specified if and only if "SIZE" is "variable-with-determinant" and "DETERMINED BY" is "added-field" or "asn1-field", or "container".

22.7.2.3 "ENCODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to (or defaults to) "added-field". The first transform shall have a source which is integer and the last transform shall have a result which can be encoded by the class of the field referenced by "USING".

22.7.2.4 All transforms specified in "ENCODER-TRANSFORMS" shall be reversible transforms.

22.7.2.5 "DECODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to "asn1-field". The first transform shall have a source which is the same as the category of the field referenced by "USING" which shall not be an auxiliary field. The last transform shall have a result which is integer.

22.7.2.6 The "USING" reference, if present, shall be a field that is present in the encoding earlier than the field being encoded. It is an application or an ECN specification error if, in an instance of encoding, the field being encoded is present but the "USING" reference field is absent (through the exercise of optionality).

22.7.2.7 If "DETERMINED BY" is "container", the "USING" reference shall be to a concatenation or to a repetition (or to a bitstring or octetstring with a contained type) in which the repetition being encoded is a component (or a component of a component, to any depth). It is an application or an ECN specification error if, in an instance of encoding, later elements within the same concatenation or repetition are to be encoded.

22.7.2.8 "HANDLE" shall be specified only if "SIZE" is "variable-with-determinant" and "DETERMINED BY" is "handle".

22.7.2.9 If "HANDLE" is specified, then the repeated element, together with any element which (through the use of optionality) may follow the repeated element shall all be encoded by encoding objects whose specification exhibits and defines an identification handle with the same name as "HANDLE", and with the same set of bits forming the identification handle. The value of the bits forming the identification handle in the repeating element shall be different from those of any possible following element.

NOTE: It is a requirement that the bits that form an identification handle shall have the same value for all abstract values encoded by an encoding object exhibiting that identification handle.

22.7.2.10 "PATTERN" shall be specified only if "SIZE" is "variable-with-determinant" and "DETERMINED BY" is "pattern".

22.7.2.11 "PATTERN" shall not be the initial sub-string of the encoding of any value of the repeated element.

NOTE: There is no prohibition on the occurrence of "PATTERN" within an encoding of the repeated element other than at its start.

22.7.2.12 The present document is considered set if the "REPETITION-SPACE" keyword is used, and it is mandatory for it to be set in all places in the defined syntax where it is allowed. Defaulting all other parts of this defined syntax (e.g., use of "REPETITION-SPACE" alone) would not satisfy the above constraints.

22.7.3 Encoder actions

22.7.3.1 Encoders shall not generate encodings if the conditions of 22.7.2 are not satisfied.

22.7.3.2 If "SIZE" is a positive value, then the encoding space is that multiple of "MULTIPLE OF" units. If "MULTIPLE OF" is repetitions, then the encoder shall cease encoding if the abstract value to be encoded is not "SIZE" repetitions, diagnosing a specification or application error.

22.7.3.3 If "SIZE" is not set to a positive value, then the encoder shall determine the size "s" of the repetition space in "MULTIPLE OF" units from the value encoding specification. This determination is specified in the clauses on value encoding specification.

22.7.3.4 If "SIZE" is "encoder-option-with-determinant" then the encoder (as an encoder's option) may increase the size "s" (as determined in 22.7.3.3) in "MULTIPLE OF" units from that determined from the value encoding specification to any value which can be encoded in the associated determinant.

22.7.3.5 If "SIZE" is "fixed-to-max" or to "self-delimiting-values", then there is no further encoder action.

22.7.3.6 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is "container", then there is no further encoder action.

22.7.3.7 If "DETERMINED BY" is "added-field", then the encoder shall apply the transforms specified by "ENCODER-TRANSFORMS" (if any) to the value "s" to produce a value that shall be encoded in the "USING" reference.

NOTE: The encoding of the "USING" reference in this case appears earlier in the encoding than the encoding of the repetition, and an encoder will need to suspend the encoding of that field until the repetition to be encoded has been determined.

22.7.3.8 If "DETERMINED BY" is "asn1-field" then the encoder shall check that the value in the "USING" reference when transformed by the "DECODER-TRANSFORMS" (if any) is equal to "s". It is an application error if this condition is not met, and encoding shall not proceed.

22.7.3.9 If "DETERMINED BY" is "handle" there is no further action needed by the encoder.

22.7.3.10 If "DETERMINED BY" is "pattern", then the encoder shall check that the specified pattern is not an initial substring of any of the encodings of the repeated element, and shall cease encoding if this check fails, diagnosing a specification or application error. The encoder shall add the pattern "PATTERN" to the end of the encoding of the repetition.

22.7.4 Decoder actions

22.7.4.1 If "SIZE" is a positive value, then the decoder determines the encoding space as that multiple of "MULTIPLE OF" units. If "MULTIPLE OF" is repetitions, then the actual end of the repetition space is determined by decoding and counting repetitions.

22.7.4.2 If "SIZE" is not set to a positive value, then the encoder shall determine the size "s" of the repetition space in "MULTIPLE OF" units from the value encoding specification. This determination is specified in the clauses on value encoding specification.

22.7.4.3 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is set to "container", then the decoder shall use the end of the container specified by "USING" as the end of the encoding space.

22.7.4.4 If "SIZE" is "variable-with-determinant" and "DETERMINED BY" is set to (or defaults to) "added-field", then the decoder shall recover the value "s" by applying the reversal of the "ENCODER-TRANSFORMS" (if any) to the value of the "USING" reference.

22.7.4.5 If "DETERMINED BY" is "asn1-field" then the decoder shall recover the value "s" by applying the "DECODER-TRANSFORMS" (if any) to the value of that field.

22.7.4.6 If "DETERMINED BY" is "handle", then the decoder shall determine the bits associated with the identification handle and attempt to decode the following element (in parallel) as either a further repetition or as a following element, using the value of the bits in the identification handle to distinguish these alternatives. If decoding succeeds for more than one of these, it is an encoding error. If it succeeds for none of these it is an encoding or a specification error.

22.7.4.7 If "DETERMINED BY" is "pattern" then the decoder shall, at the start of decoding each repetition, check whether "PATTERN" is present. If "PATTERN" is present, the bits of pattern shall be discarded, and the repetition terminated.

22.8 Value padding and justification

22.8.1 Encoding parameters, syntax, and purpose

22.8.1.1 Value padding and justification uses the following encoding parameters:

&value-justification	Justification DEFAULT right:0,
&value-pre-padding	Padding DEFAULT zero,
&value-pre-pattern	Non-Null-Pattern DEFAULT bits:'0'B
&value-post-padding	Padding DEFAULT zero,
&value-post-pattern	Non-Null-Pattern DEFAULT bits:'0'B
&unused-bits-determination	UnusedBitsDetermination DEFAULT added-field,
&unused-bits-reference	REFERENCE OPTIONAL,
&Encoder-unused-bits-transforms	#TRANSFORM ORDERED OPTIONAL,
&Decoder-unused-bits-transforms	#TRANSFORM ORDERED OPTIONAL

22.8.1.2 The syntax to be used for value padding and justification shall be:

```
[VALUE-PADDING
  [JUSTIFIED &value-justification]
  [PRE-PADDING &value-pre-padding
    [PATTERN &value-pre-pattern]]
  [POST-PADDING &value-post-padding
    [PATTERN &value-post-pattern]]
  [UNUSED BITS
    [DETERMINED BY &unused-bits-determination]
    [USING &unused-bits-reference
      [ENCODER-TRANSFORMS &Encoder-unused-bits-transforms]
      [DECODER-TRANSFORMS &Decoder-unused-bits-transforms]]]]
```

22.8.1.3 The definition of types used in justification is:

```
Justification ::= CHOICE
  { left          INTEGER (0..MAX),
    right         INTEGER (0..MAX)}          -- (see 21.8)

Padding ::= ENUMERATED {zero, one, pattern, encoder-option}  -- (see 21.9)

Pattern ::= CHOICE
  {bits          BIT STRING,
   octets        OCTET STRING,
   char8         IA5String,
   char16        BMPString,
   char32        UniversalString,
   any-of-length INTEGER (1..MAX),
   different     ENUMERATED {any} }

Non-Null-Pattern ::= Pattern
  (ALL EXCEPT (bits:'B | octets:''H | char8:'' |
    char16:'' | char32:''))          -- (see 21.10)

UnusedBitsDetermination ::= ENUMERATED
  {added-field, asn1-field, not-needed}  -- (see 21.4)
```

22.8.1.4 The purpose of the present document is to determine the way in which an encoder places a value encoding in an encoding space, and enables a decoder to determine the position of the value encoding.

22.8.1.5 The precise number of bits to be added by an encoder depends on both the encoding space specification and on the value encoding specification, and is specified for each instance of value encoding.

22.8.1.6 "USING" is a reference that enables a decoder to determine the number of padding bits inserted. It is a reference to an auxiliary field or to a field carrying abstract values, depending on "DETERMINED BY".

22.8.2 Specification restrictions

22.8.2.1 The number of bits specified in justification shall be less than or equal to the total number of padding bits "b" (see below).

22.8.2.2 "USING" shall be specified if and only if "DETERMINED BY" is not "not-needed".

22.8.2.3 "ENCODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to (or defaults to) "added-field". The first transform shall have a source which is integer and the last transform shall have a result which can be encoded by the class of the field referenced by "USING".

22.8.2.4 All transforms specified in "ENCODER-TRANSFORMS" shall be reversible transforms.

22.8.2.5 "DECODER-TRANSFORMS" shall be present only if "DETERMINED BY" is set to "asn1-field". The first transform shall have a source which is the same as the category of the field referenced by "USING" which shall not be an auxiliary field. The last transform shall have a result which is integer.

22.8.2.6 The "USING" reference, if present, shall be a field that is present in the encoding earlier than the field being encoded. It is an application or an ECN specification error if, in an instance of encoding, the field being encoded is present but the "USING" reference field is absent (through the exercise of optionality).

22.8.2.7 The present document is considered set if the "VALUE" keyword is used. Actions if it is not set are specified in all places where that syntax is permitted.

22.8.3 Encoder actions

22.8.3.1 Encoders shall not generate encodings if the conditions of 22.8.2 are not satisfied.

22.8.3.2 The present document is applied if and only if the encoding space or the repetition space encoding specification, together with the value encoding specification, determine that there may be added padding bits around the value or repetition encoding within the encoding or repetition space. Let the determined number of added padding bits in an instance of encoding be "b" (where "b" is greater than or equal to 0).

22.8.3.3 If "JUSTIFIED" is "right:n", then "b"- "n" bits shall be added as pre-padding before the value or repetition encoding, and "n" bits shall be added as post-padding after it.

22.8.3.4 If "JUSTIFIED" is "left:n", then "n" bits shall be added as pre-padding before the value or repetition encoding, and "b"- "n" bits shall be added as post-padding after it.

22.8.3.5 The padding bits shall be set in accordance with the "PRE-PADDING" and "POST-PADDING" specifications, with the leading bit of the pattern as the first inserted bit in each case.

22.8.3.6 If "DETERMINED BY" is "not-needed" then this completes the encoders actions.

22.8.3.7 If "DETERMINED BY" is "added-field", then the encoder shall apply the transforms specified by "ENCODER-TRANSFORMS" (if any) to the value "b" to produce a value that shall be encoded in the "USING" reference.

NOTE: The encoding of the "USING" reference in this case appears earlier in the encoding than the encoding of this field, and an encoder will need to suspend the encoding of that field until the value to be encoded has been determined by the encoding of this field.

22.8.3.8 If "DETERMINED BY" is "asn1-field" then the encoder shall check that the value in the "USING" reference when transformed by the "DECODER-TRANSFORMS" (if any) is equal to "b". It is an application error if this condition is not met, and encoding shall not proceed.

22.8.4 Decoder actions

22.8.4.1 If "DETERMINED BY" is "not-needed", then the decoder shall determine the value of "b" as determined by the specification of value encoding and encoding space or repetition determination.

22.8.4.2 If "DETERMINED BY" is set to (or defaults to) "added-field", then the decoder shall recover the value "b" by applying the reversal of the "ENCODER-TRANSFORMS" (if any) to the value of the "USING" reference.

22.8.4.3 If "DETERMINED BY" is "asn1-field" then the decoder shall recover the value "b" by applying the "DECODER-TRANSFORMS" (if any) to the value of that field.

22.8.4.4 The decoder shall use the "JUSTIFIED" and the value of "b" to determine the position of the value encoding within the encoding space, and shall ignore the value of all padding bits.

22.9 Identification handle specification

22.9.1 Encoding parameters, syntax, and purpose

22.9.1.1 Identification handle specification uses the following encoding parameters:

<code>&exhibited-handle</code>	PrintableString OPTIONAL,
<code>&Handle-positions</code>	INTEGER (0..MAX) OPTIONAL

22.9.1.2 The syntax to be used for identification handle specification shall be:

```
[EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
```

22.9.1.3 The present document is used to identify that an encoding object exhibits an identification handle within all its encodings. The name of the identification handle is specified, and the bits that are associated with that identification handle, but the value of those bits is determined by other parts of the encoding specification.

22.9.1.4 The list of positions in "AT" shall be the positions of the bits forming the identification handle in the final encoding, after any encoder bit-reversal actions have occurred except those bit-reversals that result from the specification of an encoding object in the #OUTER class.

22.9.2 Specification constraints

22.9.2.1 In any application of ECN, all identification handles with the same name shall specify the same set of bits for the location of the identification handle.

NOTE: There is no general requirement that the value of the bits for an identification handle (exhibited by different classes) should be distinct, but distinct values are required when the identification handle is used to resolve optionality, alternative selection, or repetition termination.

22.9.2.2 The ECN specifier shall ensure that any encoding object exhibiting an identification handle produces the same value for the bits in the identification handle for every abstract value that is encoded.

22.9.2.3 If an encoding class in the repetition category exhibits an identification handle, then that identification handle shall also be exhibited by the repeated element.

22.9.2.4 If an encoding class in the alternatives category exhibits an identification handle, then that identification handle shall also be exhibited by all alternatives.

22.9.2.5 If an encoding class in the concatenation category exhibits an identification handle, then the bits contributing to that identification handle are not required to be part of the encoding of the first component, but may be part of the first and subsequent component encodings.

22.9.2.6 The present document is considered set if the "EXHIBITS-HANDLE" keyword is used. If it is not set then there is no identification handle exhibited.

22.9.3 Encoders actions

22.9.3.1 If an encoding object exhibits an identification handle, the encoder shall check that the encoding of all abstract values have the same value for the bits in the identification handle, and shall diagnose a specification or application error otherwise.

22.9.4 Decoders actions

22.9.4.1 There are no decoders actions directly resulting from the exhibition of an identification handle. Decoder actions only result from use of the identification handle to determine optionality, end of repetitions, or choice of alternatives.

22.10 Concatenation specification

22.10.1 Encoding parameters, syntax, and purpose

22.10.1.1 Concatenation specification uses the following encoding parameters:

<code>&concatenation-order</code>	ENUMERATED {textual, tag, random} DEFAULT textual,
<code>&concatenation-alignment</code>	ENUMERATED {none, aligned} DEFAULT aligned,
<code>&concatenation-handle</code>	PrintableString DEFAULT "default-handle"

22.10.1.2 The syntax to be used for concatenation specification shall be:

```
[CONCATENATION
  [ORDER &concatenation-order]
  [ALIGNMENT &concatenation-alignment]
  [HANDLE &concatenation-handle]]
```

22.10.1.3 The present document determines the order in which the elements of an encoding class in the concatenation category are encoded, the means an encoder uses to identify each component, and any pre-alignment padding that is to be provided between components.

22.10.2 Specification constraints

22.10.2.1 If "ORDER" is "random", then "HANDLE" assumes the default value of "default-handle" if not set, and all components shall exhibit "HANDLE" with distinct values for the bits in the identification handle.

22.10.2.2 If "ALIGNMENT" is "aligned", then the pre-alignment specification assumes the default value unless set.

22.10.2.3 If a component has its own explicit pre-alignment, this is applied after any pre-alignment of the component resulting from the setting of "ALIGNMENT" in the encoding class of the concatenation category.

NOTE: The equivalent function is not provided for repetitions, as it can be achieved more simply by pre-alignment of the single component.

22.10.2.4 If "ORDER" is "tag", then every component shall start with an encoding class in the tag category. The tag number associated with this class is called the component-tag.

22.10.2.5 The component-tags of each alternative shall be distinct.

22.10.2.6 The present document is considered set if the "CONCATENATION" keyword is used. If it is not set then encoders and decoders act as if it was set with each encoding parameter taking its default value.

22.10.3 Encoder actions

22.10.3.1 If "ORDER" is "textual", the textual order in the ASN.1 type specification or the ECN structure definition shall be used.

22.10.3.2 If "ORDER" is "tag", then the order shall be that of the tag numbers in the component-tags (lowest tag number first).

22.10.3.3 If "ORDER" is "random", then the encoder shall determine the order of concatenation without constraint.

22.10.3.4 If "ALIGNMENT" is "none", the encoder shall juxtapose the encodings of components with no inserted bits.

22.10.3.5 If "ALIGNMENT" is "aligned", then the encoder shall apply the pre-alignment specification of the class in the concatenation category before encoding each component, except that a pre-alignment specification of "ALIGNED TO ANY" shall be interpreted as a specification of "ALIGNED TO NEXT" (see 22.2).

NOTE 1: This is because there can only be a single start pointer for "ALIGNED TO ANY".

NOTE 2: Any pre-alignment specified for a component (including "ALIGNED TO ANY") is applied after the above actions.

22.10.4 Decoder actions

22.10.4.1 When decoding a component, a decoder shall first perform the decoder actions associated with the pre-alignment specification for "ALIGNMENT" if it is set to "aligned", treating "ALIGNED TO ANY" as "ALIGNED TO NEXT" (see 22.2). If "ALIGNMENT" is set to "none", then the decoder shall proceed directly to decoding the component.

22.10.4.2 The decoder shall determine the order of the components from the defined order for the encoder if "ORDER" is "textual" or "tag".

22.10.4.3 If "ORDER" is "random", the decoder shall determine the order of the components by examining the value of the bits associated with "HANDLE".

22.10.4.4 Each component has a distinct value for the bits associated with "HANDLE" that enables the component to be identified. Decoding shall proceed until an abstract value for every component has been obtained, and an encoder shall diagnose an encoder's error if more than one encoding is identified for a component, or if unexpected values appear for identification handles during the decoding.

NOTE: Unexpected values can occur as part of extensibility provision, but this is not supported in this version of the present document, and such occurrences shall be treated as encoder errors.

22.11 Contained type encoding specification

22.11.1 Encoding parameters, syntax, and purpose

22.11.1.1 The contained type encoding specification uses the following encoding parameters:

&Primary-encoding-object-set	#ENCODINGS OPTIONAL,
&Secondary-encoding-object-set	#ENCODINGS OPTIONAL,
&over-ride-encoded-by	BOOLEAN DEFAULT FALSE

22.11.1.2 The syntax to be used for contained type encoding specification shall be:

```
[CONTENTS-ENCODING &Primary-encoding-object-set
  [COMPLETED BY &Secondary-encoding-object-set]
  [OVERRIDE &over-ride-encoded-by]]
```

22.11.1.3 The purpose of the present document is to determine the encoding of a contained type, and whether an ASN.1 "ENCODED BY" contents constraint associated with that contained type shall be overridden.

22.11.1.4 The present document 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.

22.11.1.5 The present document is considered set if the "CONTENTS-ENCODING" keyword is used.

22.11.2 Encoder actions

22.11.2.1 If "CONTENTS-ENCODING" is not set, then a contained type shall be encoded using the combined encoding object set applied to the container if "ENCODED BY" is not present in the ASN.1 contents constraint, otherwise with the encoding rules specified by the "ENCODED BY" statement.

22.11.2.2 If "CONTENTS-ENCODING" is set, the combined encoding object set formed from "COMPLETED BY" shall be applied to the contained type if "ENCODED BY" is not present in the ASN.1 contents constraint, or if "ENCODED BY" is present and "OVERRIDE" is "TRUE". Otherwise the combined encoding set applied to the containing type shall be applied to the contained type.

22.11.3 Decoder actions

22.11.3.1 A decoder shall decode the contained type in accordance with the encoding applied by the encoder, as specified above.

22.12 Bit reversal specification

22.12.1 Encoding parameters, syntax, and purpose

22.12.1.1 Bit reversal specification uses the following encoding parameter:

```
&bit-reversal          ReversalSpecification
                        DEFAULT no-reversal
```

22.12.1.2 The syntax to be used for bit reversal specification shall be:

```
[BIT-REVERSAL &bit-reversal]
```

22.12.1.3 The definition of types used in this group is:

```
ReversalSpecification ::= ENUMERATED
    {no-reversal,
     reverse-bits-in-units,
     reverse-half-units,
     reverse-bits-in-half-units} -- (see 21.13)
```

22.12.1.4 The purpose of the present document is to enable the order of bits in the final encoding to be different from those bits generated as part of an encoding-space or repetition-space, or in the complete encoding of a PDU (see clause 25).

NOTE 1: Bit reversal can be specified for individual bit-field encodings and also for the results of concatenation or repetition. Care should be taken to ensure that one reversal does not negate the other.

NOTE 2: Bit reversal applies to the contents of an encoding space or repetition space (including any value pre-padding or post-padding), but does not apply to any pre-alignment padding.

22.12.2 Specification constraints

22.12.2.1 The present document is only available when an encoding space or repetition space encoding is required, and within #OUTER.

22.12.2.2 "BIT-REVERSAL" shall not be "reverse-half-units" or "reverse-bits-in-half-units" unless "MULTIPLE OF" is set to an even number of bits for the encoding space or repetition space or #OUTER reversal. (This requirement means that a value of "repetitions" for "MULTIPLE OF" is not allowed in this case).

22.12.2.3 "BIT-REVERSAL" shall not be set unless "MULTIPLE-OF" is repetitions or is greater than one bit.

22.12.2.4 The present document is considered set if the "BIT-REVERSAL" keyword is used. If it is not set then encoders and decoders act as if it was set with the encoding parameter taking its default value.

22.12.3 Encoder actions

22.12.3.1 Except when performing #OUTER actions, an encoder shall divide the contents of the encoding space or repetitions space into "MULTIPLE OF" units unless "MULTIPLE OF" is "repetitions". If "MULTIPLE OF" is "repetitions", then the entire encoding space shall be treated as a single unit. When performing bit-reversal for #OUTER, the entire encoding (after any "PADDING" has been applied) shall be divided into "MULTIPLE OF" units. It is an ECN specification error if the entire encoding is not an integral multiple of "MULTIPLE OF" units.

22.12.3.2 The encoder shall do no reversal (the default value), or shall reverse the bits in each unit, or shall reverse the half-units (without changing the order of bits in each half-unit) or shall reverse the bits within each half-unit, as specified by the value of "BIT-REVERSAL".

22.12.4 Decoder actions

22.12.4.1 The decoder shall first determine (see encoding space and repetition space specification) the end of the encoding space or repetition space or (for bit-reversal specification within #OUTER) the end of the entire encoding, and shall then perform the reversal actions specified for the encoder before continuing with decoding.

NOTE: Performing the same reversals will recover the original bit-order.

23 Defined syntax specification for bitfield and constructor classes

This clause provides the full syntax for defining encoding objects of each encoding class in the different categories.

NOTE: Encoder and decoder actions are specified in the following clauses as conditional on a parameter group being set. A group is set if and only if the initial keyword of the group is present in the specification of the encoding object.

23.1 Defining encoding objects for classes in the alternatives category

23.1.1 The defined syntax

The syntax for defining encoding objects for classes in the alternatives category is defined as:

```
#ALTERNATIVES ::= ENCODING-CLASS {

    -- Structure or component replacement specification (see 22.1)
    &#Replacement-structure                                OPTIONAL,
    &#Replacement-structure2                              OPTIONAL,
    &replacement-structure-encoding-object &#Replacement-structure OPTIONAL,
    &replacement-structure-encoding-object2 &#Replacement-structure2 OPTIONAL,
    &#Head-end-structure                                OPTIONAL,
    &#Head-end-structure2                              OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding             Padding DEFAULT zero,
    &encoding-space-pre-pattern             Non-Null-Pattern (ALL EXCEPT different:any)
                                           DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                          REFERENCE OPTIONAL,
    &start-pointer-unit                     Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms       #TRANSFORM ORDERED OPTIONAL,

    -- Alternative determination (see 22.6)
    &alternative-determination              AlternativeDetermination
                                           DEFAULT added-field,
    &alternative-reference                  REFERENCE OPTIONAL,
    &Encoder-transforms                     #TRANSFORM ORDERED OPTIONAL,
    &Decoder-transforms                     #TRANSFORM ORDERED OPTIONAL,
    &handle-id                             PrintableString
                                           DEFAULT "default-handle",
    &alternative-ordering                   ENUMERATED {textual, tag}
                                           DEFAULT textual,

    -- Identification handle specification (see 22.9)
    &exhibited-handle                       PrintableString OPTIONAL,
    &Handle-positions                       INTEGER (0..MAX) OPTIONAL
```



```

} WITH SYNTAX {
  [REPLACE
    [STRUCTURE]
    [COMPONENT]
    [ALL COMPONENTS]
    [OPTIONALS]
    [NON-OPTIONALS]
    WITH &#Replacement-structure
      [ENCODED BY &replacement-structure-encoding-object
        [INSERT AT HEAD &#Head-end-structure]]
      [AND OPTIONALS WITH &#Replacement-structure2
        [ENCODED BY &replacement-structure-encoding-object2
          [INSERT AT HEAD &#Head-end-structure2]]] ]
  [ALIGNED TO
    [NEXT]
    [ANY]
    &encoding-space-pre-alignment-unit
    [PADDING &encoding-space-pre-padding
    [PATTERN &encoding-space-pre-pattern]]
  [START-POINTER      &start-pointer
    [MULTIPLE OF      &start-pointer-unit]
    [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
  ALTERNATIVE
    [DETERMINED BY &alternative-determination
      [HANDLE &handle-id]]
    [USING &alternative-reference
      [ORDER &alternative-ordering]
      [ENCODER-TRANSFORMS &Encoder-transforms]
      [DECODER-TRANSFORMS &Decoder-transforms]]
  [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
}

```

23.1.2 Purpose and restrictions

23.1.2.1 This syntax is used to define the start of the encoding space for an encoding class in the alternatives category, the determination of the alternative that has been encoded, and an optional declaration that all encodings exhibit a specified identification handle.

23.1.2.2 If "REPLACE STRUCTURE" is set, then no other encoding parameter groups shall be set.

23.1.2.3 Encodings of this class do not exhibit an identification handle unless "EXHIBITS HANDLE" is set (even if all components exhibit an identification handle, that may or may not be the same).

23.1.2.4 If "EXHIBITS HANDLE" is set, then encodings of all the alternatives of this class are required to exhibit the defined identification handle.

NOTE: This would normally require that every component had a "EXHIBITS HANDLE" set to the same value, unless a head-end insertion exhibited the identification handle (see 9.10.3).

23.1.3 Encoder actions

23.1.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) alternative determination;
- e) identification handle.

23.1.4 Decoder actions

23.1.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) alternative determination.

23.2 Defining encoding objects for classes in the bitstring category

23.2.1 The defined syntax

The syntax for defining encoding objects for classes in the bitstring category is defined as:

```
#BITS ::= ENCODING-CLASS {

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding             Padding DEFAULT zero,
    &encoding-space-pre-pattern              Non-Null-Pattern (ALL EXCEPT different:any)
                                           DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                           REFERENCE OPTIONAL,
    &start-pointer-unit                      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms        #TRANSFORM ORDERED OPTIONAL,

    -- Bits value encoding
    &value-reversal                          BOOLEAN DEFAULT FALSE,
    &Encoder-transforms                      #TRANSFORM ORDERED OPTIONAL,
    &Bits-repetition-encodings               #CONDITIONAL-REPETITION ORDERED OPTIONAL,
    &bits-repetition-encoding                #CONDITIONAL-REPETITION OPTIONAL

    -- Identification handle specification (see 22.9)
    &exhibited-handle                       PrintableString OPTIONAL,
    &Handle-positions                       INTEGER (0..MAX) OPTIONAL,

    -- Contained type encoding specification (see 22.11)
    &Primary-encoding-object-set             #ENCODINGS OPTIONAL,
    &Secondary-encoding-object-set           #ENCODINGS OPTIONAL,
    &over-ride-encoded-by                    BOOLEAN DEFAULT FALSE

} WITH SYNTAX {
    [ALIGNED TO
        [NEXT]
        [ANY]
        &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
        [PATTERN &encoding-space-pre-pattern]]
    [START-POINTER &start-pointer
        [MULTIPLE OF &start-pointer-unit]
        [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
    [VALUE-REVERSAL &value-reversal]
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [REPETITION-ENCODINGS &Bits-repetition-encodings]
    [REPETITION-ENCODING &bits-repetition-encoding]
    [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
    [CONTENTS-ENCODING &Primary-encoding-object-set
        [COMPLETED BY &Secondary-encoding-object-set]
        [OVERRIDE &over-ride-encoded-by]]
}
```


23.2.2 Model for the encoding of classes in the bitstring category

23.2.2.1 The model of bits encodings is:

- a) the order of bits in the bitstring can be reversed;
- b) the bits are then considered as a repetition of bit;
- c) there is an optional transform (specified by "ENCODER-TRANSFORMS") in which each bit is transformed into a (self-delimiting) bitstring;
- d) either "REPETITION-ENCODING" or "REPETITION-ENCODINGS" specify how the repetition of the sequences of bits (or of the original bits, if "ENCODER-TRANSFORMS" is not set) are to be encoded.

NOTE: The sole purpose of allowing "REPETITION-ENCODING" as well as "REPETITION-ENCODINGS" is to provide a syntax that does not contain a double curly-bracket ("{{") in the common case of a single conditional encoding. Use of "REPETITION-ENCODINGS" when there is a single conditional encoding is deprecated but is allowed.

23.2.2.2 Bounds (if present) on the class being encoded (a class in the bitstring category) are bounds on the number of bits in the bitstring forming each abstract value.

23.2.2.3 When considered as a repetition of a bit, these bounds shall be interpreted as bounds on the number of repetitions, and can be used in the specification of the encoding objects of class #CONDITIONAL-REPETITION that are used in the specification of this encoding object.

23.2.3 Purpose and restrictions

23.2.3.1 This syntax is used to define the start of the encoding space for a class in the bitstring category, the encoding of the abstract values of that class, an optional declaration that all bits encodings exhibit a specified identification handle, a specification of how to encode a contained type.

23.2.3.2 If "REPLACE" is set, then no other encoding parameter groups shall be set.

23.2.3.3 The #CONDITIONAL-REPETITION that is applied by this encoding object shall not specify "REPLACE" unless it is "REPLACE STRUCTURE".

23.2.3.4 The first transform in "ENCODER-TRANSFORMS" (if any) shall have a source that is a single bit and the last transform shall have a result that is bitstring. The bitstrings produced for a one-bit and for a zero-bit shall form a self-delimiting set (see 3.2.39).

NOTE: This means that the final transform is required to be self-delimiting.

23.2.3.5 The "ENCODER-TRANSFORMS" shall be reversible transforms.

23.2.3.6 Exactly one of "REPETITION-ENCODING" and "REPETITION-ENCODINGS" shall be set.

23.2.3.7 If an encoding object in the "REPETITION-ENCODINGS" list is defined using "IF", then all preceding encoding objects in that list shall be defined using "IF".

23.2.3.8 If "DETERMINED BY" is "not-needed" in one or more of the "REPETITION-ENCODING(S)" specifications, then the abstract values of the original bitstring to which that encoding object is applied shall be constrained to a finite self-delimiting set that can be identified from the ECN specification.

NOTE: This would be the case if the bitstring values resulted from a Huffman-style encoding (see annex E) specified by mapping integer values to bits (see 19.7), or if the bitstring values had an ECN-visible bound restricting them to a fixed number of bits.

23.2.3.9 If "EXHIBITS HANDLE" is set, then all encodings of values associated with this class shall exhibit the specified identification handle.

NOTE: This will in general require restrictions on the abstract values of the associated type or the addition of redundant bits in the transform into bits, or both.

23.2.4 Encoder actions

23.2.4.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) bits value encoding (see 23.2.4.2);
- d) identification handle;
- e) contained type encoding.

23.2.4.2 For bits value encoding, the encoder shall:

- a) reverse the order of bits in the entire bitstring abstract value if "VALUE-REVERSAL" is set to "TRUE";
- b) treat the bitstring value as a repetition of a bit;
- c) apply the specified "ENCODER-TRANSFORMS" (if any) to each bit to produce a repetition of bits;
- d) encode the repetition by applying the first "REPETITION-ENCODING(S)" whose condition is satisfied.

23.2.4.3 It is an ECN specification error if there is no "REPETITION-ENCODING(S)" whose condition is satisfied.

23.2.5 Decoder actions

23.2.5.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) bits value decoding (see 23.2.5.2);
- d) contained type decoding.

23.2.5.2 For bits value decoding, the decoder shall use the "REPETITION-ENCODING(S)" to determine the repetition space and to recover the original bit order using the "BIT-REVERSAL" specification.

23.2.5.3 If "ENCODER-TRANSFORMS" is set, then the decoder shall use the self-delimiting property of the encoding of each bit to determine the end of each repetition, and shall reverse the transforms to recover the original bitstring value.

23.2.5.4 If "VALUE-REVERSAL" is set to "TRUE", then the final order of the bits in the bitstring abstract value shall be reversed.

23.3 Defining encoding objects for classes in the boolean category

23.3.1 The defined syntax

The syntax for defining encoding objects for classes in the boolean category is defined as:

```
#BOOL ::= ENCODING-CLASS {
    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                                OPTIONAL,
    &replacement-structure-encoding-object &#Replacement-structure OPTIONAL,
```



```

-- Pre-alignment and padding specification (see 22.2)
&encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
&encoding-space-pre-padding              Padding DEFAULT zero,
&encoding-space-pre-pattern              Non-Null-Pattern (ALL EXCEPT different:any)
                                          DEFAULT bits:'0'B,

-- Start pointer specification (see 22.3)
&start-pointer                           REFERENCE                                OPTIONAL,
&start-pointer-unit                      Unit (ALL EXCEPT repetitions) DEFAULT bit,
&Start-pointer-encoder-transforms        #TRANSFORM ORDERED                      OPTIONAL,

-- Encoding space specification (see 22.4)
&encoding-space-size                     EncodingSpaceSize
                                          DEFAULT self-delimiting-values,
&encoding-space-unit                     Unit (ALL EXCEPT repetitions)
                                          DEFAULT bit,
&encoding-space-determination            EncodingSpaceDetermination
                                          DEFAULT added-field,
&encoding-space-reference                REFERENCE OPTIONAL,
&Encoder-transforms                      #TRANSFORM ORDERED OPTIONAL,
&Decoder-transforms                      #TRANSFORM ORDERED OPTIONAL,

-- Boolean value encoding
&value-true-pattern                      Pattern DEFAULT bits:'1'B,
&value-false-pattern                     Pattern DEFAULT bits:'0'B,

-- Value padding and justification (see 22.8)
&value-justification                     Justification DEFAULT right:0,
&value-pre-padding                       Padding DEFAULT zero,
&value-pre-pattern                       Non-Null-Pattern DEFAULT bits:'0'B
&value-post-padding                      Padding DEFAULT zero,
&value-post-pattern                      Non-Null-Pattern DEFAULT bits:'0'B
&unused-bits-determination               UnusedBitsDetermination
                                          DEFAULT added-field,
&unused-bits-reference                   REFERENCE OPTIONAL,
&Encoder-unused-bits-transforms          #TRANSFORM ORDERED OPTIONAL,
&Decoder-unused-bits-transforms          #TRANSFORM ORDERED OPTIONAL,

-- Identification handle specification (see 22.9)
&exhibited-handle                        PrintableString OPTIONAL,
&Handle-positions                        INTEGER (0..MAX) OPTIONAL,

-- Bit reversal specification (see 22.12)
&bit-reversal                            ReversalSpecification
                                          DEFAULT no-reversal

} WITH SYNTAX {
  [REPLACE
    [STRUCTURE]
    WITH &#Replacement-structure
    [ENCODED BY &replacement-structure-encoding-object]]
  [ALIGNED TO
    [NEXT]
    [ANY]
    &encoding-space-pre-alignment-unit
    [PADDING &encoding-space-pre-padding
    [PATTERN &encoding-space-pre-pattern]]
  [START-POINTER &start-pointer
    [MULTIPLE OF &start-pointer-unit]
    [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
  ENCODING-SPACE
    [SIZE &encoding-space-size
    [MULTIPLE OF &encoding-space-unit]]
    [DETERMINED BY &encoding-space-determination]
    [USING &encoding-space-reference
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [DECODER-TRANSFORMS &Decoder-transforms]]
  [TRUE-PATTERN &value-true-pattern]
  [FALSE-PATTERN &value-false-pattern]
  [VALUE-PADDING
    [JUSTIFIED &value-justification]
    [PRE-PADDING &value-pre-padding
    [PATTERN &value-pre-pattern]]
    [POST-PADDING &value-post-padding
    [PATTERN &value-post-pattern]]

```



```

    [UNUSED BITS
      [DETERMINED BY &unused-bits-determination]
      [USING &unused-bits-reference
        [ENCODER-TRANSFORMS &Encoder-unused-bits-transforms]
        [DECODER-TRANSFORMS &Decoder-unused-bits-transforms]]]
    [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
    [BIT-REVERSAL &bit-reversal]
  }

```

23.3.2 Purpose and restrictions

23.3.2.1 This syntax is used to define the start of the encoding space for a class in the boolean category, the encoding of the abstract values of that class, their positioning within the encoding space, an optional declaration that all bits encodings exhibit a specified identification handle, and possible bit-reversal of the encoding space for the boolean.

23.3.2.2 If "REPLACE" is set, then no other encoding parameter groups shall be set.

23.3.2.3 At most one of "TRUE-PATTERN" and "FALSE-PATTERN" shall be set to "different:any".

23.3.2.4 If the alternative "any-of-length" is selected for either pattern (or both), then the length in bits of the two patterns shall be different.

23.3.2.5 If "ENCODING-SPACE SIZE" is "self-delimiting", then "TRUE-PATTERN" and "FALSE-PATTERN" shall form a self-delimiting set (see 3.2.39).

23.3.2.6 "UNUSED BITS DETERMINED BY" shall not be "not-needed" unless:

- a) both patterns are integral multiples of "ENCODING-SPACE MULTIPLE OF" units and "ENCODING SPACE SIZE" is "variable-with-determinant"; or
- b) both patterns are the same length; or
- c) "JUSTIFIED" is "left" and the patterns form a self-delimiting set; or
- d) "JUSTIFIED" is "right" and the reverse of the patterns form a self-delimiting set (see 3.2.39).

23.3.3 Encoder actions

23.3.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) encoding space (see 23.3.3.2);
- e) value encoding (see 23.3.3.3);
- f) value padding and justification;
- g) identification handle;
- h) bit reversal.

23.3.3.2 If "ENCODING-SPACE SIZE" is not set to a positive value, then the encoding space size "s" is the smallest number of "MULTIPLE OF" units (subject to 23.3.3.3) that can accommodate the pattern of the value that is to be encoded.

23.3.3.3 An encoder (as an encoder's option) may increase the encoding space size "s" (as determined in 23.3.3.2) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "added-field" or "asn1-field" imposes) if the "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

23.3.3.4 The number of unused bits can be determined from the value "s" and from the pattern of the value to be encoded.

23.3.3.5 If the number of unused bits is non-zero, then "VALUE-JUSTIFICATION" shall be applied, using either the set values or the default values.

23.3.4 Decoder actions

23.3.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) encoding space;
- d) bit reversal;
- e) value padding and justification;
- f) value decoding (see 23.3.4.2).

23.3.4.2 Value decoding shall be performed by identifying the "TRUE-PATTERN" or the "FALSE-PATTERN" by:

- a) using an "UNUSED BITS" determination, if any; or
- b) using the self-delimiting property of the patterns or their reversals.

23.4 Defining encoding objects for classes in the characterstring category

23.4.1 The defined syntax

The syntax for defining encoding objects for classes in the characterstring category is defined as:

```
#CHARS ::= ENCODING-CLASS {

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding           Padding DEFAULT zero,
    &encoding-space-pre-pattern           Non-Null-Pattern (ALL EXCEPT different:any)
                                         DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                        REFERENCE                                OPTIONAL,
    &start-pointer-unit                   Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms     #TRANSFORM ORDERED                      OPTIONAL,

    -- Chars value encoding
    &value-reversal                       BOOLEAN DEFAULT FALSE,
    &Encoder-transforms                   #TRANSFORM ORDERED,
    &Chars-repetition-encodings           #CONDITIONAL-REPETITION ORDERED OPTIONAL,
    &chars-repetition-encoding           #CONDITIONAL-REPETITION OPTIONAL,

    -- Identification handle specification (see 22.9)
    &exhibited-handle                     PrintableString OPTIONAL,
    &Handle-positions                     INTEGER (0..MAX) OPTIONAL,

} WITH SYNTAX {
    [ALIGNED TO
        [NEXT]
        [ANY]
        &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
        [PATTERN &encoding-space-pre-pattern]]
    [START-POINTER      &start-pointer
        [MULTIPLE OF    &start-pointer-unit]
        [ENCODER-TRANSFORMS  &Start-pointer-encoder-transforms]]
```



```

[VALUE-REVERSAL          &value-reversal]
ENCODER-TRANSFORMS      &Encoder-transforms
[REPETITION-ENCODINGS    &Chars-repetition-encodings]
[REPETITION-ENCODING     &chars-repetition-encoding]
[EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
}

```

23.4.2 Model for the encoding of classes in the characterstring category

23.4.2.1 The model of characterstring encodings is:

- a) the order of characters in the character string can be reversed;
- b) the chars are considered as a repetition of a char;
- c) there is a required transform (specified by "ENCODER-TRANSFORMS") in which each character is transformed into a self-delimiting bitstring;
- d) either "REPETITION-ENCODING" or "REPETITION-ENCODINGS" specify how the repetition of bitstring is to be encoded.

NOTE: The sole purpose of allowing "REPETITION-ENCODING" as well as "REPETITION-ENCODINGS" is to provide a syntax that does not contain a double curly-bracket ("{"") in the common case of a single conditional encoding. Use of "REPETITION-ENCODINGS" when there is a single conditional encoding is deprecated but is allowed.

23.4.2.2 Bounds (if present) on the class being encoded (a class in the characterstring category) are bounds on the number of chars in the character string forming each abstract value.

23.4.2.3 When considered as a repetition of chars, these bounds shall be interpreted as bounds on the number of repetitions, and can be used in the specification of the encoding objects of class #REPETITION-ENCODING that are used in the specification of this encoding object.

23.4.3 Purpose and restrictions

23.4.3.1 This syntax is used to define the start of the encoding space for a class in the characterstring category, the encoding of the abstract values associated with that class, an optional declaration that all chars encodings exhibit a specified identification handle.

23.4.3.2 The #CONDITIONAL-REPETITION that is applied by this encoding object shall not specify "REPLACE" unless it is "REPLACE STRUCTURE".

23.4.3.3 The first transform of "ENCODER-TRANSFORMS" (if any) shall have a source that is a single character and the last transform shall have a result that is bitstring. The bitstrings produced for the set of all characters to be encoded shall form a self-delimiting set (see 3.2.39).

NOTE: This means that the final transform is required to be self-delimiting.

23.4.3.4 The "ENCODER-TRANSFORMS" shall be reversible transforms.

23.4.3.5 Exactly one of "REPETITION-ENCODING" and "REPETITION-ENCODINGS" shall be set.

23.4.3.6 If an encoding object in the "REPETITION-ENCODINGS" list is defined using "IF", then all preceding encoding objects in that list shall be defined using "IF".

23.4.3.7 If "EXHIBITS HANDLE" is set, then all encodings of values associated with this class shall exhibit the specified identification handle.

NOTE: This will in general require restrictions on the abstract values of the associated type, or the inclusion of redundant bits in the encoding of each character, or both.

23.4.4 Encoder actions

23.4.4.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) chars value encoding (see 23.4.4.3);
- d) repetition encoding as specified by the first "REPETITION-ENCODING(S)" whose condition is satisfied;
- e) identification handle specification.

23.4.4.2 It is an ECN specification error if there is no "REPETITION-ENCODING(S)" whose condition is satisfied.

23.4.4.3 For characterstring value encoding, the encoder shall:

- a) reverse the order of characters in the entire character string abstract value if "VALUE-REVERSAL" is set to TRUE;
- b) treat the characterstring value of chars as a repetition of char;
- c) apply the specified "ENCODER TRANSFORMS" (if any) to each char to produce a repetition of bits;
- d) encode the repetition by applying the "REPETITION-ENCODING(S)".

23.4.5 Decoder actions

23.4.5.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) repetition decoding as specified by the first "REPETITION-ENCODING(S)" whose condition is satisfied;
- d) characterstring value decoding (see 23.4.5.2).

23.4.5.2 For characterstring value decoding, the decoder shall use the "REPETITION-ENCODING(S)" to determine the repetition space and to recover the original characters. If "ENCODER-TRANSFORMS" is set, then the decoder shall use the self-delimiting (which includes a possible fixed length) property of the encoding of each character to determine the end of each repetition, and shall reverse the transforms to recover a characterstring value.

23.4.5.3 If "VALUE-REVERSAL" is set to "TRUE", then the final order of the characters in the characterstring abstract value shall be reversed.

23.5 Defining encoding objects for classes in the concatenation category

23.5.1 The defined syntax

The syntax for defining encoding objects for classes in the concatenation category is defined as:

```
#CONCATENATION ::= ENCODING-CLASS {
    -- Full replacement specification (see 22.1)
    &#Replacement-structure                                OPTIONAL,
    &#Replacement-structure2                                OPTIONAL,
    &replacement-structure-encoding-object &#Replacement-structure  OPTIONAL,
    &replacement-structure-encoding-object2 &#Replacement-structure2  OPTIONAL,
    &#Head-end-structure                                    OPTIONAL,
    &#Head-end-structure2                                    OPTIONAL,
```



```

-- Pre-alignment and padding specification (see 22.2)
&encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
&encoding-space-pre-padding             Padding DEFAULT zero,
&encoding-space-pre-pattern              Non-Null-Pattern (ALL EXCEPT different:any)
                                         DEFAULT bits:'0'B,

-- Start pointer specification (see 22.3)
&start-pointer                          REFERENCE OPTIONAL,
&start-pointer-unit                     Unit (ALL EXCEPT repetitions) DEFAULT bit,
&Start-pointer-encoder-transforms        #TRANSFORM ORDERED OPTIONAL,

-- Encoding space specification (see 22.4)
&encoding-space-size                    EncodingSpaceSize
                                         DEFAULT self-delimiting-values,
&encoding-space-unit                    Unit (ALL EXCEPT repetitions)
                                         DEFAULT bit,
&encoding-space-determination           EncodingSpaceDetermination
                                         DEFAULT added-field,
&encoding-space-reference                REFERENCE OPTIONAL,
&Encoder-transforms                     #TRANSFORM ORDERED OPTIONAL,
&Decoder-transforms                     #TRANSFORM ORDERED OPTIONAL,

-- Concatenation specification (see 22.10)
&concatenation-order                    ENUMERATED {textual, tag, random}
                                         DEFAULT textual,
&concatenation-alignment                 ENUMERATED {none, aligned}
                                         DEFAULT aligned,
&concatenation-handle                   PrintableString
                                         DEFAULT "default-handle",

-- Value padding and justification (see 22.8)
&value-justification                    Justification DEFAULT right:0,
&value-pre-padding                      Padding DEFAULT zero,
&value-pre-pattern                      Non-Null-Pattern DEFAULT bits:'0'B
&value-post-padding                     Padding DEFAULT zero,
&value-post-pattern                     Non-Null-Pattern DEFAULT bits:'0'B
&unused-bits-determination              UnusedBitsDetermination
                                         DEFAULT added-field,
&unused-bits-reference                  REFERENCE OPTIONAL,
&Encoder-unused-bits-transforms          #TRANSFORM ORDERED OPTIONAL,
&Decoder-unused-bits-transforms          #TRANSFORM ORDERED OPTIONAL,

-- Identification handle specification (see 22.9)
&exhibited-handle                       PrintableString OPTIONAL,
&Handle-positions                       INTEGER (0..MAX) OPTIONAL,

-- Bit reversal specification (see 22.12)
&bit-reversal                           ReversalSpecification
                                         DEFAULT no-reversal

} WITH SYNTAX {
  [REPLACE
    [STRUCTURE]
    [COMPONENT]
    [ALL COMPONENTS]
    [OPTIONALS]
    [NON-OPTIONALS]
    WITH &#Replacement-structure
      [ENCODED BY &replacement-structure-encoding-object
        [INSERT AT HEAD &#Head-end-structure]]
      [AND OPTIONALS WITH &#Replacement-structure2
        [ENCODED BY &replacement-structure-encoding-object2
          [INSERT AT HEAD &#Head-end-structure2]]] ]
  [ALIGNED TO
    [NEXT]
    [ANY]
    &encoding-space-pre-alignment-unit
    [PADDING &encoding-space-pre-padding
    [PATTERN &encoding-space-pre-pattern]]
  [START-POINTER &start-pointer
    [MULTIPLE OF &start-pointer-unit]
    [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
  ENCODING-SPACE
    [SIZE &encoding-space-size
    [MULTIPLE OF &encoding-space-unit]]

```



```

[DETERMINED BY &encoding-space-determination]
[USING &encoding-space-reference
  [ENCODER-TRANSFORMS &Encoder-transforms]
  [DECODER-TRANSFORMS &Decoder-transforms]]
[CONCATENATION
  [ORDER &concatenation-order]
  [ALIGNMENT &concatenation-alignment]
  [HANDLE &concatenation-handle]]
[VALUE-PADDING
  [JUSTIFIED &value-justification]
  [PRE-PADDING &value-pre-padding
    [PATTERN &value-pre-pattern]]
  [POST-PADDING &value-post-padding
    [PATTERN &value-post-pattern]]
  [UNUSED BITS
    [DETERMINED BY &unused-bits-determination]
    [USING &unused-bits-reference
      [ENCODER-TRANSFORMS &Encoder-unused-bits-transforms]
      [DECODER-TRANSFORMS &Decoder-unused-bits-transforms]]]]
[EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
[BIT-REVERSAL &bit-reversal]
}

```

23.5.2 Purpose and restrictions

23.5.2.1 This syntax is used to define the start of the encoding space for a class in the concatenation category, the way in which the encodings of the components are to be combined, their positioning within the encoding space, an optional declaration that all encodings exhibit a specified identification handle, and possible bit-reversal of the encoding space.

23.5.2.2 If "REPLACE STRUCTURE" is set, then no other encoder parameter groups shall be set.

23.5.2.3 "ENCODING-SPACE SIZE" shall be either "variable-with-determinant" or "self-delimiting-values".

23.5.2.4 If "EXHIBITS HANDLE" is set then the encoding of all possible abstract values associated with this class shall exhibit the defined identification handle

NOTE: This would often be achieved by ensuring that the first component of the concatenation, or a head-end insert, exhibited the identification handle.

23.5.3 Encoder actions

23.5.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) encoding space (see 23.5.3.2);
- e) concatenation;
- f) value padding and justification;
- g) identification handle specification;
- h) bit reversal.

23.5.3.2 If "ENCODING SPACE" is "variable-with-determinant", it shall be the minimum number of "MULTIPLE OF" units needed to contain the concatenation.

23.5.4 Decoder actions

23.5.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) encoding space;
- d) bit reversal;
- e) value padding and justification;
- f) concatenation.

23.6 Defining encoding objects for classes in the integer category

23.6.1 The defined syntax

The syntax for defining encoding objects for classes in the integer category is defined as:

```
#INT ::= ENCODING-CLASS {
    -- Integer encoding
    &Integer-encodings      #CONDITIONAL-INT ORDERED OPTIONAL,
    &integer-encoding       #CONDITIONAL-INT OPTIONAL
} WITH SYNTAX {
    [ENCODINGS &Integer-encodings]
    [ENCODING &integer-encoding]
}
```

23.6.2 Purpose and restrictions

23.6.2.1 This syntax is used to define the encoding of a class in the integer category by specifying one or more encodings of the #CONDITIONAL-INT class.

23.6.2.2 Exactly one of "ENCODING" and "ENCODINGS" shall be set.

NOTE: The sole purpose of allowing "ENCODING" as well as "ENCODINGS" is to provide a syntax that does not contain a double curly-bracket ("{"") in the common case of a single encoding object. Use of "ENCODINGS" when there is a single encoding object is deprecated but is allowed.

23.6.2.3 If an encoding object in the "ENCODINGS" list is defined using "IF", then all preceding encoding objects in that list shall be defined using "IF".

23.6.3 Encoder actions

23.6.3.1 The encoder shall select and apply the first #CONDITIONAL-INT encoding object in "ENCODING(S)" whose conditions are satisfied. It is an ECN specification error if none of the conditional encodings have conditions that are satisfied.

NOTE: It would be unusual but not illegal if there were #CONDITIONAL-INT encoding objects present that could never be used because the conditions on use of earlier encoding objects would always be satisfied.

23.6.4 Decoder actions

23.6.4.1 The decoder shall select and use the first #CONDITIONAL-INT encoding object in "ENCODING(S)" whose conditions are satisfied.

23.7 Defining encoding objects for the #CONDITIONAL-INT class

23.7.1 The defined syntax

The syntax for defining encoding objects for the #CONDITIONAL-INT class is defined as:

```
#CONDITIONAL-INT ::= ENCODING-CLASS {

    -- Condition (see 21.11)
    &range-condition                               RangeCondition OPTIONAL,

    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                         OPTIONAL,
    &replacement-structure-encoding-object &#Replacement-structure OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit             Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding                    Padding DEFAULT zero,
    &encoding-space-pre-pattern                    Non-Null-Pattern (ALL EXCEPT different:any)
                                                    DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                                 REFERENCE OPTIONAL,
    &start-pointer-unit                            Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms              #TRANSFORM ORDERED OPTIONAL,

    -- Encoding space specification (see 22.4)
    &encoding-space-size                          EncodingSpaceSize
                                                    DEFAULT self-delimiting-values,
    &encoding-space-unit                          Unit (ALL EXCEPT repetitions)
                                                    DEFAULT bit,
    &encoding-space-determination                 EncodingSpaceDetermination
                                                    DEFAULT added-field,
    &encoding-space-reference                     REFERENCE OPTIONAL,
    &Encoder-transforms                           #TRANSFORM ORDERED OPTIONAL,
    &Decoder-transforms                           #TRANSFORM ORDERED OPTIONAL,

    -- Value encoding
    &Transform                                     #TRANSFORM ORDERED OPTIONAL,
    &encoding                                     ENUMERATED
                                                    {positive-int, twos-complement,
reverse-positive-int, reverse-twos-complement}
                                                    DEFAULT twos-complement,

    -- Value padding and justification (see 22.8)
    &value-justification                          Justification DEFAULT right:0,
    &value-pre-padding                            Padding DEFAULT zero,
    &value-pre-pattern                            Non-Null-Pattern DEFAULT bits:'0'B
    &value-post-padding                           Padding DEFAULT zero,
    &value-post-pattern                           Non-Null-Pattern DEFAULT bits:'0'B
    &unused-bits-determination                   UnusedBitsDetermination
                                                    DEFAULT added-field,
    &unused-bits-reference                       REFERENCE OPTIONAL,
    &Encoder-unused-bits-transforms               #TRANSFORM ORDERED OPTIONAL,
    &Decoder-unused-bits-transforms               #TRANSFORM ORDERED OPTIONAL,

    -- Identification handle specification (see 22.9)
    &exhibited-handle                             PrintableString OPTIONAL,
    &Handle-positions                             INTEGER (0..MAX) OPTIONAL,

    -- Bit reversal specification (see 22.12)
    &bit-reversal                                 ReversalSpecification
                                                    DEFAULT no-reversal

} WITH SYNTAX {
    [IF &range-condition] [ELSE]
    [REPLACE
        [STRUCTURE]
        WITH &#Replacement-structure
        [ENCODED BY &replacement-structure-encoding-object]]
}
```



```

[ALIGNED TO
  [NEXT]
  [ANY]
    &encoding-space-pre-alignment-unit
    [PADDING &encoding-space-pre-padding
    [PATTERN &encoding-space-pre-pattern]]
[START-POINTER      &start-pointer
  [MULTIPLE OF      &start-pointer-unit]
  [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
ENCODING-SPACE
  [SIZE &encoding-space-size
    [MULTIPLE OF &encoding-space-unit]]
  [DETERMINED BY &encoding-space-determination]
  [USING &encoding-space-reference
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [DECODER-TRANSFORMS &Decoder-transforms]]
[TRANSFORMS      &Transforms]
[ENCODING        &encoding]
[VALUE-PADDING
  [JUSTIFIED &value-justification]
  [PRE-PADDING &value-pre-padding
    [PATTERN &value-pre-pattern]]
  [POST-PADDING &value-post-padding
    [PATTERN &value-post-pattern]]
  [UNUSED BITS
    [DETERMINED BY &unused-bits-determination]
    [USING &unused-bits-reference
      [ENCODER-TRANSFORMS &Encoder-unused-bits-transforms]
      [DECODER-TRANSFORMS &Decoder-unused-bits-transforms]]]]
[EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
[BIT-REVERSAL &bit-reversal]
}

```

23.7.2 Purpose and restrictions

23.7.2.1 This syntax is used to define a #CONDITIONAL-INT encoding object. The only use of such an encoding object is in the specification of an encoding object of a class in the integer category.

23.7.2.2 The syntax allows the specification of a single condition on the bounds of the integer for this encoding to be applied (use of "IF"). It also allows the specification that there is no condition. The use of "ELSE", or omission of both "IF" and "ELSE" specifies that there is no condition.

23.7.2.3 Using this syntax the ECN specifier can define the start of the encoding space for the encoding of a class in the integer category, the encoding of the abstract values associated with that class, their positioning within the encoding space, and possible bit-reversal of the encoding space.

23.7.2.4 At most one of "IF" and "ELSE" shall be present.

23.7.2.5 If "REPLACE" is set, then no other encoding parameter groups shall be set.

23.7.2.6 The first transform of "TRANSFORMS", if present, shall have a source that is integer and the last transform shall have a result that is integer. All transforms in the list shall be reversible.

NOTE: The test for the "IF" condition takes place on the bounds of the original value, and is not affected by these transforms.

23.7.2.7 The "INT-TO-INT" transform with the value "subtract:lower-bound" shall be included only if the "IF" condition restricts the application of this encoding to classes of the integer category with a lower bound, and (if present) shall be the first transform in the list.

23.7.2.8 The "ENCODING-SPACE SIZE" shall not be "fixed-to-max" unless the "IF" condition restricts the encoding to a class with both an upper and a lower bound.

23.7.2.9 "ENCODING-SPACE SIZE" shall not be set to "self-delimiting-values".

23.7.2.10 If "EXHIBITS HANDLE" is set, then the specifier asserts that the encoding of all values exhibits the identification handle.

NOTE: This will normally require use of "VALUE-PADDING" with justification from the left to allow the padding to exhibit the identification handle.

23.7.3 Encoder actions

23.7.3.1 The encoder shall detect an ECN specification or application error if any of the restrictions in 23.7.2 are violated.

23.7.3.2 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) encoding space;
- e) value encoding (see below);
- f) value padding and justification;
- g) identification handle;
- h) bit reversal.

23.7.3.3 The encoder shall apply the "TRANSFORMS", if any to the value being encoded.

23.7.3.4 The encoder shall use the following table giving the range of integer values that can be encoded in "n" bits:

"ENCODING"	Min value	Max value
"positive-int"	0	$2^n - 1$
"reverse-positive-int"	0	$2^n - 1$
"twos-complement"	-2^{n-1}	$2^{n-1} - 1$
"reverse-twos-complement"	-2^{n-1}	$2^{n-1} - 1$

23.7.3.5 The "ENCODING" parameter selects the encoding as 2's-complement encoding or as a positive integer encoding, or as the reversal of one of these. The specification of 2's-complement encoding and positive integer encoding is given in ITU-T Rec. X.690 | ISO/IEC 8825-1, 8.3.2 and 8.3.3. A reversal of these encodings is an encoding in which, following production of the "n" bits, the order of the "n" bits is reversed.

23.7.3.6 An encoder shall detect an ECN specification or an application error if a value is to be encoded into a number of bits which is insufficient, as specified in 23.7.3.4.

23.7.3.7 If the "ENCODING-SPACE SIZE" is a positive integer, then its size in bits is calculated as "SIZE" multiplied by "MULTIPLE OF" units. If "VALUE-PADDING" is not set, then this shall be the number of bits "n" that the integer shall encode into and there are no unused bits. If "VALUE-PADDING" is set, then the number of bits that the integer shall encode into is reduced by the integer value "m" specified for "JUSTIFIED", and there will be "m" unused bits.

23.7.3.8 If the "ENCODING-SPACE SIZE" is "fixed-to-max", then the encoder shall determine the minimum number of "MULTIPLE OF" units that has sufficient bits to encode any of the values of the class, and shall proceed (as specified above) as if "SIZE" were a positive integer set to that value.

23.7.3.9 If the "ENCODING-SPACE SIZE" is "variable-with-determinant", then the encoder shall determine the minimum number of "MULTIPLE OF" units ("s", say) that has sufficient bits to encode the actual abstract value being encoded, and shall proceed (as specified above) as if "SIZE" were a positive integer set to that value.

23.7.3.10 The encoder (as an encoder's option) may increase "s" (as determined in 23.7.3.9) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "added-field" or "asn1-field" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

23.7.3.11 The encoder shall then proceed (as specified above) as if "SIZE" were a positive integer set to "s".

23.7.4 Decoder actions

23.7.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) encoding space;
- d) bit reversal;
- e) value padding and justification;
- f) value decoding (see 23.7.4.2).

23.7.4.2 The decoder shall recover the integer value from the bits used to encode it, decoding according to the specified encoding, and shall then reverse the "TRANSFORMS" (if specified) to recover the original abstract value.

23.8 Defining encoding objects for classes in the null category

23.8.1 The defined syntax

The syntax for defining encoding objects for classes in the null category is defined as:

```
#NUL ::= ENCODING-CLASS {

    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                                OPTIONAL,
    &replacement-structure-encoding-object    &#Replacement-structure    OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit          Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding                Padding DEFAULT zero,
    &encoding-space-pre-pattern                Non-Null-Pattern (ALL EXCEPT different:any)
                                              DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                                REFERENCE                                OPTIONAL,
    &start-pointer-unit                          Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms            #TRANSFORM ORDERED OPTIONAL,

    -- Encoding space specification (see 22.4)
    &encoding-space-size                        EncodingSpaceSize
                                              DEFAULT self-delimiting-values,
    &encoding-space-unit                        Unit (ALL EXCEPT repetitions)
                                              DEFAULT bit,
    &encoding-space-determination              EncodingSpaceDetermination
                                              DEFAULT added-field,
    &encoding-space-reference                  REFERENCE OPTIONAL,
    &Encoder-transforms                        #TRANSFORM ORDERED OPTIONAL,
    &Decoder-transforms                       #TRANSFORM ORDERED OPTIONAL,

    -- Value pattern
    &value-pattern                              Pattern (ALL EXCEPT different:any)
                                              DEFAULT bits:''B,

    -- Value padding and justification (see 22.8)
    &value-justification                        Justification DEFAULT right:0,
    &value-pre-padding                          Padding DEFAULT zero,
    &value-pre-pattern                          Non-Null-Pattern DEFAULT bits:'0'B
    &value-post-padding                        Padding DEFAULT zero,
    &value-post-pattern                          Non-Null-Pattern DEFAULT bits:'0'B
    &unused-bits-determination                UnusedBitsDetermination
                                              DEFAULT added-field,
    &unused-bits-reference                    REFERENCE OPTIONAL,
    &Encoder-unused-bits-transforms            #TRANSFORM ORDERED OPTIONAL,
    &Decoder-unused-bits-transforms           #TRANSFORM ORDERED OPTIONAL,
```



```

-- Identification handle specification (see 22.9)
&exhibited-handle      PrintableString OPTIONAL,
&Handle-positions      INTEGER (0..MAX) OPTIONAL,

-- Bit reversal specification (see 22.12)
&bit-reversal          ReversalSpecification
                        DEFAULT no-reversal

} WITH SYNTAX {
  [REPLACE
    [STRUCTURE]
    WITH &#Replacement-structure
    [ENCODED BY &replacement-structure-encoding-object]]
  [ALIGNED TO
    [NEXT]
    [ANY]
    &encoding-space-pre-alignment-unit
    [PADDING &encoding-space-pre-padding
    [PATTERN &encoding-space-pre-pattern]]]
  [START-POINTER      &start-pointer
    [MULTIPLE OF      &start-pointer-unit]
    [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]]
  ENCODING-SPACE
    [SIZE &encoding-space-size
    [MULTIPLE OF &encoding-space-unit]]
    [DETERMINED BY &encoding-space-determination]
    [USING &encoding-space-reference
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [DECODER-TRANSFORMS &Decoder-transforms]]]
  [PATTERN &value-pattern]
  [VALUE-PADDING
    [JUSTIFIED &value-justification]
  [PRE-PADDING &value-pre-padding
    [PATTERN &value-pre-pattern]]]
  [POST-PADDING &value-post-padding
    [PATTERN &value-post-pattern]]]
  [UNUSED BITS
    [DETERMINED BY &unused-bits-determination]
    [USING &unused-bits-reference
    [ENCODER-TRANSFORMS &Encoder-unused-bits-transforms]
    [DECODER-TRANSFORMS &Decoder-unused-bits-transforms]]]]]
  [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
  [BIT-REVERSAL &bit-reversal]
}

```

23.8.2 Purpose and restrictions

23.8.2.1 This syntax is used to define the encoding of a class in the null category.

23.8.2.2 If "REPLACE STRUCTURE" is set, then no other encoding parameter groups shall be set.

23.8.2.3 If the "ENCODING-SPACE SIZE" is positive, it shall be sufficient to hold the size of the "PATTERN" together with any bits added as a result of a "VALUE-PADDING" specification.

23.8.2.4 If "PATTERN" is not an integral multiple of the "ENCODING-SPACE MULTIPLE OF" unit, then "VALUE-PADDING" shall be set.

23.8.3 Encoder actions

23.8.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) encoding space;
- e) value encoding (see 23.8.3.2);

- f) value padding and justification;
- g) identification handle;
- h) bit reversal.

23.8.3.2 The value encoding shall be the bits of the "PATTERN". If the "ENCODING-SPACE SIZE" is "variable-with-determinant", it shall be the minimum number of "MULTIPLE OF" units needed to contain the pattern.

23.8.3.3 If "ENCODING-SPACE SIZE" is "variable-with-determinant" or "encoder-option-with-determinant", it shall be the minimum number of "MULTIPLE OF" units needed to contain the pattern ("s", say), subject to 23.8.3.4.

23.8.3.4 An encoder (as an encoder's option) may increase "s" (as determined in 23.8.3.3) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "added-field" or "asn1-field" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

23.8.4 Decoder actions

23.8.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) encoding space;
- d) bit reversal;
- e) value padding and justification.

23.8.4.2 The decoder shall determine the size of the null pattern, and identify those bits in the encoding, but shall silently accept any value for those bits.

23.9 Defining encoding objects for classes in the octetstring category

23.9.1 The defined syntax

The syntax for defining encoding objects for classes in the octetstring category is defined as:

```
#OCTETS ::= ENCODING-CLASS {

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding              Padding DEFAULT zero,
    &encoding-space-pre-pattern              Non-Null-Pattern (ALL EXCEPT different:any)
                                           DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                           REFERENCE OPTIONAL,
    &start-pointer-unit                      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms        #TRANSFORM ORDERED OPTIONAL,

    -- Octets value encoding
    &value-reversal                          BOOLEAN DEFAULT FALSE,
    &Encoder-transforms                      #TRANSFORM ORDERED OPTIONAL,
    &Octets-repetition-encodings             #CONDITIONAL-REPETITION ORDERED
                                           OPTIONAL,
    &octets-repetition-encoding              #CONDITIONAL-REPETITION OPTIONAL

    -- Identification handle specification (see 22.9)
    &exhibited-handle                        PrintableString OPTIONAL,
    &Handle-positions                        INTEGER (0..MAX) OPTIONAL,
```



```

-- Contained type encoding specification (see 22.11)
&Primary-encoding-object-set          #ENCODINGS OPTIONAL,
&Secondary-encoding-object-set        #ENCODINGS OPTIONAL,
&over-ride-encoded-by                 BOOLEAN DEFAULT FALSE

} WITH SYNTAX {
    [ALIGNED TO
        [NEXT]
        [ANY]
        &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
        [PATTERN &encoding-space-pre-pattern]]
    [START-POINTER &start-pointer
        [MULTIPLE OF &start-pointer-unit]
        [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
    [VALUE-REVERSAL &value-reversal]
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [REPETITION-ENCODINGS &Octets-repetition-encodings]
    [REPETITION-ENCODING &octets-repetition-encoding]
    [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
    [CONTENTS-ENCODING &Primary-encoding-object-set
        [COMPLETED BY &Secondary-encoding-object-set]
        [OVERRIDE &over-ride-encoded-by]]
}

```

23.9.2 Model for the encoding of classes in the octetstring category

23.9.2.1 The model of octetstring encoding is:

- a) The order of octets in the octetstring can be reversed.
- b) The octets are then considered as a repetition of an octet.
- c) There is an optional transform (specified by "ENCODER-TRANSFORMS") in which each octet is transformed into a self-delimiting bitstring.
- d) Either "REPETITION-ENCODING" or "REPETITION-ENCODINGS" specify how the repetition of octet is to be encoded.

NOTE: The sole purpose of allowing "REPETITION-ENCODING" as well as "REPETITION-ENCODINGS" is to provide a syntax that does not contain a double curly-bracket ("{"") in the common case of a single conditional encoding. Use of "REPETITION-ENCODINGS" when there is a single conditional encoding is deprecated but is allowed.

23.9.2.2 Bounds (if present) on the class being encoded (a class in the octetstring category) are bounds on the number of octets in the octetstring forming each abstract value.

23.9.2.3 When considered as a repetition of an octet, these bounds shall be interpreted as bounds on the number of repetitions, and can be used in the specification of the encoding objects of class #CONDITIONAL-REPETITION that are used in the specification of this encoding object.

23.9.3 Purpose and restrictions

23.9.3.1 This syntax is used to define the start of the encoding space for a class in the octetstring category, the encoding of the abstract values associated with that class, an optional declaration that all octetstring encodings exhibit a specified identification handle, a specification of how to encode a contained type.

23.9.3.2 If "REPLACE" is set, then no other encoding parameter groups shall be set.

23.9.3.3 The #CONDITIONAL-REPETITION that is applied by this encoding object shall not specify "REPLACE" unless it is "REPLACE STRUCTURE".

23.9.3.4 The first transform of "ENCODER-TRANSFORMS" (if any) shall have a source that is bitstring and the last transform shall have a result that is a self-delimiting bitstring (see 3.2.39).

23.9.3.5 The "ENCODER-TRANSFORMS" shall be reversible transforms.

23.9.3.6 Exactly one of "REPETITION-ENCODING" and "REPETITION-ENCODINGS" shall be set.

23.9.3.7 If an encoding object in the "REPETITION-ENCODINGS" list is defined using "IF", then all preceding encoding objects in that list shall be defined using "IF".

23.9.3.8 If "EXHIBITS HANDLE" is set, then all encodings of values of this class shall exhibit the specified identification handle.

NOTE: This will in general require restrictions on the abstract values of the associated type.

23.9.4 Encoder actions

23.9.4.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) value encoding as specified below;
- d) repetition encoding as specified by the first "REPETITION-ENCODING(S)" whose condition is satisfied;
- e) identification handle;
- f) contained type encoding.

23.9.4.2 For value encoding, the encoder shall:

- a) reverse the order of octets in the entire octetstring abstract value if "VALUE-REVERSAL" is set to "TRUE";
- b) treat the octetstring value as a repetition of octet;
- c) apply the "ENCODER-TRANSFORMS" (if any) to each octet to produce a repetition of bitstring;

NOTE: If there are no transforms, each octet forms a bitstring.

- d) encode the repetition by applying the first "REPETITION-ENCODING(S)" whose condition is satisfied.

23.9.4.3 It is an ECN specification error if there is no "REPETITION-ENCODING(S)" whose condition is satisfied.

23.9.5 Decoder actions

23.9.5.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) value decoding (see 23.9.5.2);
- d) contained type decoding.

23.9.5.2 The decoder shall reverse the "ENCODER TRANSFORMS" (if any) to recover the original octets.

23.9.5.3 If "VALUE-REVERSAL" is set to "TRUE", then the final order of the octets in the octetstring abstract value shall be reversed.

23.10 Defining encoding objects for classes in the optionality category

23.10.1 The defined syntax

The syntax for defining encoding objects for classes in the optionality category is defined as:

```
#OPTIONAL ::= ENCODING-CLASS {

    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                                OPTIONAL,
    &replacement-structure-encoding-object  &#Replacement-structure  OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding             Padding DEFAULT zero,
    &encoding-space-pre-pattern             Non-Null-Pattern (ALL EXCEPT different:any)
                                           DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                            REFERENCE                OPTIONAL,
    &start-pointer-unit                      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms        #TRANSFORM ORDERED OPTIONAL,

    -- Optionality determination (see 22.5)
    &optionality-determination              OptionalityDetermination
                                           DEFAULT added-field,
    &optionality-reference                  REFERENCE OPTIONAL,
    &Encoder-transforms                     #TRANSFORM ORDERED OPTIONAL,
    &Decoder-transforms                     #TRANSFORM ORDERED OPTIONAL,
    &handle-id                             PrintableString
                                           DEFAULT "default-handle"
} WITH SYNTAX {
    [REPLACE
        [STRUCTURE]
        WITH &#Replacement-structure
        [ENCODED BY &replacement-structure-encoding-object]]
    [ALIGNED TO
        [NEXT]
        [ANY]
        &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
        [PATTERN &encoding-space-pre-pattern]]
    [START-POINTER      &start-pointer
        [MULTIPLE OF    &start-pointer-unit]
        [ENCODER-TRANSFORMS      &Start-pointer-encoder-transforms]]
    PRESENCE
        [DETERMINED BY &optionality-determination
        [HANDLE &handle-id]]
        [USING &optionality-reference
        [ENCODER-TRANSFORMS &Encoder-transforms]
        [DECODER-TRANSFORMS &Decoder-transforms]]
    }
```

23.10.2 Purpose and restrictions

23.10.2.1 This syntax is used to define the encoding of a class in the optionality category.

23.10.2.2 If "REPLACE STRUCTURE" is set, then no other encoding parameter groups shall be set.

23.10.3 Encoder actions

23.10.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement (see 23.10.3.2);
- b) pre-alignment and padding;
- c) start pointer;
- d) optionality determination.

23.10.3.2 If "REPLACE STRUCTURE" is set then the entire element (including any classes in the tag category, but excluding the optionality category class) is provided as the actual parameter for the replacement structure, which becomes a mandatory element.

23.10.4 Decoder actions

23.10.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) optionality determination.

23.11 Defining encoding objects for classes in the pad category

23.11.1 The defined syntax

The syntax for defining encoding objects for classes in the pad category is defined as:

```
#PAD ::= ENCODING-CLASS {

    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                OPTIONAL,
    &replacement-structure-encoding-object  &#Replacement-structure  OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit      Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding             Padding DEFAULT zero,
    &encoding-space-pre-pattern             Non-Null-Pattern (ALL EXCEPT different:any)
                                           DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                          REFERENCE                OPTIONAL,
    &start-pointer-unit                     Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms        #TRANSFORM ORDERED OPTIONAL,

    -- Encoding space specification (see 22.4)
    &encoding-space-size                    EncodingSpaceSize
                                           DEFAULT self-delimiting-values,
    &encoding-space-unit                    Unit (ALL EXCEPT repetitions)
                                           DEFAULT bit,
    &encoding-space-determination           EncodingSpaceDetermination
                                           DEFAULT added-field,
    &encoding-space-reference               REFERENCE OPTIONAL,
    &Encoder-transforms                     #TRANSFORM ORDERED OPTIONAL,
    &Decoder-transforms                     #TRANSFORM ORDERED OPTIONAL,

    -- Value encoding
    &pad-pattern                            Pattern (ALL EXCEPT different:any)
                                           DEFAULT bits:''B

    -- Identification handle specification (see 22.9)
    &exhibited-handle                       PrintableString            OPTIONAL,
    &Handle-positions                       INTEGER (0..MAX)           OPTIONAL,
```



```

-- Bit reversal specification (see 22.12)
&bit-reversal                                ReversalSpecification
                                              DEFAULT no-reversal

} WITH SYNTAX {
  [ALIGNED TO
    [NEXT]
    [ANY]
    &encoding-space-pre-alignment-unit
    [PADDING &encoding-space-pre-padding
    [PATTERN &encoding-space-pre-pattern]]
  [START-POINTER      &start-pointer
    [MULTIPLE OF      &start-pointer-unit]
    [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
  ENCODING-SPACE
    [SIZE &encoding-space-size
    [MULTIPLE OF &encoding-space-unit]]
    [DETERMINED BY &encoding-space-determination]
    [USING &encoding-space-reference
    [ENCODER-TRANSFORMS &Encoder-transforms]
    [DECODER-TRANSFORMS &Decoder-transforms]]
  [PATTERN &pad-pattern]
  [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
  [BIT-REVERSAL &bit-reversal]
}

```

23.11.2 Purpose and restrictions

23.11.2.1 This syntax is used to define the encoding of a class in the pad category.

23.11.2.2 If "ENCODING-SPACE SIZE" is positive, "PATTERN" shall not be of zero length, and is replicated and truncated to fill the encoding space.

23.11.2.3 If "REPLACE STRUCTURE" is set, then no other encoding parameter group shall be set.

23.11.3 Encoder actions

23.11.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) encoding space;
- e) value encoding (see below);
- f) identification handle;
- g) bit reversal.

23.11.3.2 If "ENCODING-SPACE SIZE" is positive, the value shall be the "PATTERN", replicated and truncated to fill the encoding space.

23.11.3.3 If "ENCODING-SPACE SIZE" is "fixed-to-max", or is "variable-with-determinant" or is "encoder-option-with-determinant", then the encoding space shall be the smallest number of "MULTIPLE OF" units that is greater than the size of "PATTERN" ("s", say), and the "PATTERN" shall then be replicated and truncated to fill that space (but see 23.11.3.4).

NOTE: This will be an empty encoding space if the "PATTERN" is null.

23.11.3.4 An encoder (as an encoder's option) may increase "s" (as determined in 23.11.3.3) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "added-field" or "asn1-field" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

23.11.4 Decoder actions

23.11.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) bit reversal;
- d) encoding space.

23.11.4.2 The decoder shall determine the size of the pad value encoding, and identify those bits in the encoding, but shall silently accept any value for those bits.

23.12 Defining encoding objects for classes in the repetition category

23.12.1 The defined syntax

The syntax for defining encoding objects for classes in the repetition category is defined as:

```
#REPETITION ::= ENCODING-CLASS {
    -- Repetition encoding
    &Repetition-encodings          #CONDITIONAL-REPETITION ORDERED OPTIONAL,
    &repetition-encoding           #CONDITIONAL-REPETITION OPTIONAL
} WITH SYNTAX {
    [REPETITION-ENCODINGS &Repetition-encodings]
    [REPETITION-ENCODING &repetition-encoding]
}
```

23.12.2 Purpose and restrictions

23.12.2.1 This syntax is used to define the encoding of a class in the repetition category by specifying one or more encodings of the #CONDITIONAL-REPETITION class.

23.12.2.2 Exactly one of "REPETITION-ENCODING" and "REPETITION-ENCODINGS" shall be set.

NOTE: The sole purpose of allowing "REPETITION-ENCODING" as well as "REPETITION-ENCODINGS" is to provide a syntax that does not contain a double curly-bracket ("{{") in the common case of a single encoding object. Use of "REPETITION-ENCODINGS" when there is a single encoding object is deprecated but is allowed.

23.12.2.3 If an encoding object in the "REPETITION-ENCODINGS" list is defined using "IF", then all preceding encoding objects in that list shall be defined using "IF".

23.12.3 Encoder actions

23.12.3.1 The encoder shall select and apply the first #CONDITIONAL-REPETITION encoding object in "ENCODING(S)" whose conditions are satisfied. It is an ECN specification error if none of the conditional encodings have conditions that are satisfied.

NOTE: It would be unusual but not illegal if there were #CONDITIONAL-REPETITION encoding objects present that could never be used because the conditions on use of earlier encoding objects would always be satisfied.

23.12.4 Decoder actions

23.12.4.1 The decoder shall select and use the first #CONDITIONAL-REPETITION encoding object in "ENCODING(S)" whose conditions are satisfied.

23.13 Defining encoding objects for the #CONDITIONAL-REPETITION class

23.13.1 The defined syntax

The syntax for defining encoding objects for the #CONDITIONAL- REPETITION class is defined as:

```
#CONDITIONAL-REPETITION ::= ENCODING-CLASS {

    -- Condition (see 21.12)
    &size-range-condition                               SizeRangeCondition           OPTIONAL,

    -- Structure or component replacement specification (see 22.1)
    &#Replacement-structure                             OPTIONAL,
    &replacement-structure-encoding-object &#Replacement-structure    OPTIONAL,
    &#Head-end-structure                               OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit                 Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding                         Padding DEFAULT zero,
    &encoding-space-pre-pattern                         Non-Null-Pattern (ALL EXCEPT different:any)
                                                        DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                                     REFERENCE                     OPTIONAL,
    &start-pointer-unit                                 Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms                   #TRANSFORM ORDERED           OPTIONAL,

    -- Repetition space specification (see 22.7)
    &repetition-space-size                             EncodingSpaceSize
                                                        DEFAULT self-delimiting-values,
    &repetition-space-unit                             Unit
                                                        DEFAULT bit,
    &repetition-space-determination                    RepetitionSpaceDetermination
                                                        DEFAULT added-field,
    &main-reference                                     REFERENCE OPTIONAL,
    &Encoder-transforms                                #TRANSFORM ORDERED OPTIONAL,
    &Decoder-transforms                                #TRANSFORM ORDERED OPTIONAL,
    &handle-id                                           PrintableString
                                                        DEFAULT "default-handle",
    &termination-pattern                               Non-Null-Pattern (ALL EXCEPT
                                                        different:any) DEFAULT '0'B,

    -- Repetition alignment
    &repetition-alignment                               ENUMERATED {none, aligned}
                                                        DEFAULT none,

    -- Value padding and justification (see 22.8)
    &value-justification                               Justification DEFAULT right:0,
    &value-pre-padding                                 Padding DEFAULT zero,
    &value-pre-pattern                                 Non-Null-Pattern DEFAULT bits:'0'B
    &value-post-padding                                Padding DEFAULT zero,
    &value-post-pattern                                Non-Null-Pattern DEFAULT bits:'0'B
    &unused-bits-determination                         UnusedBitsDetermination
                                                        DEFAULT added-field,
    &unused-bits-reference                             REFERENCE OPTIONAL,
    &Encoder-unused-bits-transforms                     #TRANSFORM ORDERED OPTIONAL,
    &Decoder-unused-bits-transforms                     #TRANSFORM ORDERED OPTIONAL,

    -- Identification handle specification (see 22.9)
    &exhibited-handle                                   PrintableString OPTIONAL,
    &Handle-positions                                  INTEGER (0..MAX) OPTIONAL,

    -- Bit reversal specification (see 22.12)
    &bit-reversal                                       ReversalSpecification
                                                        DEFAULT no-reversal
}
```



```

} WITH SYNTAX {
  [IF &size-range-condition] [ELSE]
  [REPLACE
    [STRUCTURE]
    [COMPONENT]
    [ALL COMPONENTS]
    WITH &Replacement-structure
    [ENCODED BY &replacement-structure-encoding-object
      [INSERT AT HEAD &#Head-end-structure]]]
  [ALIGNED TO
    [NEXT]
    [ANY]
    &encoding-space-pre-alignment-unit
    [PADDING &encoding-space-pre-padding
      [PATTERN &encoding-space-pre-pattern]]]
  [START-POINTER &start-pointer
    [MULTIPLE OF &start-pointer-unit]
    [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
  REPETITION-SPACE
    [SIZE &repetition-space-size
      [MULTIPLE OF &repetition-space-unit]]
    [DETERMINED BY &repetition-space-determination
      [HANDLE &handle-id]]
    [USING &main-reference
      [ENCODER-TRANSFORMS &Encoder-transforms]
      [DECODER-TRANSFORMS &Decoder-transforms]]
    [PATTERN &termination-pattern]
  [ALIGNMENT &repetition-alignment]
  [VALUE-PADDING
    [JUSTIFIED &value-justification]
    [PRE-PADDING &value-pre-padding
      [PATTERN &value-pre-pattern]]
    [POST-PADDING &value-post-padding
      [PATTERN &value-post-pattern]]
    [UNUSED BITS
      [DETERMINED BY &unused-bits-determination]
      [USING &unused-bits-reference
        [ENCODER-TRANSFORMS &Encoder-unused-bits-transforms]
        [DECODER-TRANSFORMS &Decoder-unused-bits-transforms]]]]
  [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
  [BIT-REVERSAL &bit-reversal]
}

```

23.13.2 Purpose and restrictions

23.13.2.1 This syntax is used to define the encoding of a class in the repetition category subject to satisfaction of a condition based on the bounds of the repetition (use of "IF"). It also allows the specification that there is no condition. The use of "ELSE", or omission of both "IF" and "ELSE" specifies that there is no condition.

23.13.2.2 At most one of "IF" and "ELSE" shall be present.

23.13.2.3 If "REPLACE STRUCTURE" is set, then no other encoding parameter groups shall be set.

23.13.2.4 If "EXHIBITS HANDLE" is set, this asserts that all encodings of this class exhibit the specified identification handle.

NOTE: This would normally, but not necessarily, mean that every instance of the component exhibited the entire identification handle.

23.13.2.5 If the "REPETITION-SPACE SIZE" is "self-delimiting-values", then the number of repetitions shall be constrained by bounds to a single value.

23.13.3 Encoder actions

23.13.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) repetition space;
- e) repetition encoding (see 23.13.3.4);
- f) value padding and justification;
- g) identification handle;
- h) bit reversal.

23.13.3.2 If "ALIGNMENT" is set to "aligned", then the settings of pre-alignment and padding shall be used to pre-align each encoding of the component.

NOTE: This is performed before any pre-alignment specified by the component.

23.13.3.3 The complete encodings of the components (with any pre-alignment however specified) shall be concatenated to form the bits for the value of the repetition.

23.13.3.4 If the "REPETITION-SPACE SIZE" is "variable-with-determinant" or "encoder-option-with-determinant", then the size shall be the smallest multiple of "MULTIPLE OF" units ("s", say) that will contain the value of the repetition (but see 23.13.3.5).

23.13.3.5 An encoder (as an encoder's option) may increase "s" (as determined in 23.13.3.4) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "added-field" or "asn1-field" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

23.13.3.6 The repetition value is then placed in the encoding space in using "VALUE-PADDING" (or its default value if it is not set) if there are any unused bits.

23.13.4 Decoder actions

23.13.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) repetition space;
- d) bit reversal;
- e) value padding and justification;
- f) repetition decoding (see 23.13.4.2).

23.13.4.2 Each repetition shall be extracted, and decoded in accordance with the encoding specification of the component of the repetition class.

23.14 Defining encoding objects for classes in the tag category

23.14.1 The defined syntax

The syntax for defining encoding objects for classes in the tag category is defined as:

```
#TAG ::= ENCODING-CLASS {

    -- Structure-only replacement specification (see 22.1)
    &#Replacement-structure                                OPTIONAL,
    &replacement-structure-encoding-object &#Replacement-structure OPTIONAL,

    -- Pre-alignment and padding specification (see 22.2)
    &encoding-space-pre-alignment-unit    Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &encoding-space-pre-padding           Padding DEFAULT zero,
    &encoding-space-pre-pattern           Non-Null-Pattern (ALL EXCEPT different:any)
                                         DEFAULT bits:'0'B,

    -- Start pointer specification (see 22.3)
    &start-pointer                        REFERENCE                                OPTIONAL,
    &start-pointer-unit                  Unit (ALL EXCEPT repetitions) DEFAULT bit,
    &Start-pointer-encoder-transforms    #TRANSFORM ORDERED                    OPTIONAL,

    -- Encoding space specification (see 22.4)
    &encoding-space-size                 EncodingSpaceSize
                                         DEFAULT self-delimiting-values,
    &encoding-space-unit                 Unit (ALL EXCEPT repetitions)
                                         DEFAULT bit,
    &encoding-space-determination        EncodingSpaceDetermination
                                         DEFAULT added-field,
    &encoding-space-reference            REFERENCE OPTIONAL,
    &Encoder-transforms                  #TRANSFORM ORDERED                    OPTIONAL,
    &Decoder-transforms                  #TRANSFORM ORDERED                    OPTIONAL,

    -- Value padding and justification (see 22.8)
    &value-justification                 Justification DEFAULT right:0,
    &value-pre-padding                   Padding DEFAULT zero,
    &value-pre-pattern                   Non-Null-Pattern DEFAULT bits:'0'B
    &value-post-padding                  Padding DEFAULT zero,
    &value-post-pattern                  Non-Null-Pattern DEFAULT bits:'0'B
    &unused-bits-determination          UnusedBitsDetermination
                                         DEFAULT added-field,
    &unused-bits-reference               REFERENCE OPTIONAL,
    &Encoder-unused-bits-transforms      #TRANSFORM ORDERED                    OPTIONAL,
    &Decoder-unused-bits-transforms      #TRANSFORM ORDERED                    OPTIONAL,

    -- Identification handle specification (see 22.9)
    &exhibited-handle                    PrintableString                        OPTIONAL,
    &Handle-positions                    INTEGER (0..MAX)                      OPTIONAL,

    -- Bit reversal specification (see 22.12)
    &bit-reversal                        ReversalSpecification
                                         DEFAULT no-reversal

} WITH SYNTAX {
    [REPLACE
        [STRUCTURE]
        WITH &#Replacement-structure
        [ENCODED BY &replacement-structure-encoding-object]]
    [ALIGNED TO
        [NEXT]
        [ANY]
        &encoding-space-pre-alignment-unit
        [PADDING &encoding-space-pre-padding
        [PATTERN &encoding-space-pre-pattern]]]
    [START-POINTER &start-pointer
        [MULTIPLE OF &start-pointer-unit]
        [ENCODER-TRANSFORMS &Start-pointer-encoder-transforms]]
    ENCODING-SPACE
        [SIZE &encoding-space-size
        [MULTIPLE OF &encoding-space-unit]]
        [DETERMINED BY &encoding-space-determination]
        [USING &encoding-space-reference
        [ENCODER-TRANSFORMS &Encoder-transforms]
        [DECODER-TRANSFORMS &Decoder-transforms]]
    [VALUE-PADDING
```



```

    [JUSTIFIED &value-justification]
    [PRE-PADDING &value-pre-padding
      [PATTERN &value-pre-pattern]]
    [POST-PADDING &value-post-padding
      [PATTERN &value-post-pattern]]
    [UNUSED BITS
      [DETERMINED BY &unused-bits-determination]
      [USING &unused-bits-reference
        [ENCODER-TRANSFORMS &Encoder-unused-bits-transforms]
        [DECODER-TRANSFORMS &Decoder-unused-bits-transforms]]]]
    [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions]
    [BIT-REVERSAL &bit-reversal]
  }

```

23.14.2 Purpose and restrictions

23.14.2.1 This syntax is used to define the encoding of a class in the tag category.

23.14.2.2 If "REPLACE STRUCTURE" is set, then no other specifications shall be set.

23.14.2.3 The "ENCODING-SPACE SIZE" shall not be "fixed-to-max" or "self-delimiting-values".

23.14.3 Encoder actions

23.14.3.1 For any encoding parameter group that is set, the encoder shall perform the encoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) replacement;
- b) pre-alignment and padding;
- c) start pointer;
- d) encoding space;
- e) value encoding (see 23.14.3.3);
- f) value padding and justification;
- g) identification handle;
- h) bit reversal.

23.14.3.2 The encoder shall determine the minimum number of bits "n" needed to encode the tag number as the smallest value of "n" such that $2^n - 1$ is greater than or equal to the tag number. If "n" is zero, it shall be increased to 1.

23.14.3.3 The encoding shall be a positive integer encoding. The specification of a positive integer encoding is given in ITU-T Rec. X.690 | ISO/IEC 8825-1, 8.3.2 and 8.3.3.

23.14.3.4 An encoder shall detect an ECN specification error if a tag number is to be encoded into a number of bits which is insufficient, as specified above.

23.14.3.5 If "ENCODING-SPACE SIZE" is a positive integer, then its size in bits is calculated as "SIZE" multiplied by "MULTIPLE OF" units. If "VALUE-PADDING" is not set, then this shall be the number of bits "n" that the tag number shall encode into and there are no unused bits. If "VALUE-PADDING" is set, then the number of bits that the tag number shall encode into is reduced by the integer value "m" specified for "JUSTIFIED", and there will be "m" unused bits.

23.14.3.6 If "ENCODING-SPACE SIZE" is "variable-with-determinant" or "encoder-option-with-determinant", then the encoder shall determine the minimum number of "MULTIPLE OF" units that has sufficient bits to encode the tag number ("s", say), and shall proceed (as specified above) as if "SIZE" were a positive integer set to that value (but see 23.14.3.7).

23.14.3.7 An encoder (as an encoder's option) may increase "s" (as determined in 23.14.3.6) in "MULTIPLE OF" units (subject to any restrictions that the range of values of any "added-field" or "asn1-field" imposes) if "ENCODING-SPACE SIZE" is set to "encoder-option-with-determinant".

23.14.4 Decoder actions

23.14.4.1 For any encoding parameter group that is set, the decoder shall perform the decoder actions specified in clause 22, in the following order and in accordance with the encoding object definition:

- a) pre-alignment and padding;
- b) start pointer;
- c) encoding space;
- d) bit reversal;
- e) value padding and justification;
- f) value decoding.

23.14.4.2 The decoder shall recover the tag number from the bits used to encode it, decoding from a positive integer encoding.

23.15 Defining encoding objects for classes in the other categories

In this version of the present document there is no defined syntax for classes in the following categories:

```
object-identifier
open-type
real
```

24 Defined syntax specification for the #TRANSFORM encoding class

24.1 Summary of encoding parameters and defined syntax

24.1.1 The syntax for defining encoding objects for the #TRANSFORM class shall be:

```
#TRANSFORM ::= ENCODING-CLASS {

    -- int-to-int (see 24.3)
    &int-to-int

    CHOICE
    {increment  INTEGER (1..MAX),
    decrement  INTEGER (1..MAX),
    multiply    INTEGER (2..MAX),
    divide      INTEGER (2..MAX),
    negate      ENUMERATED{value},
    modulo      INTEGER (2..MAX),
    subtract    ENUMERATED{lower-bound}
    } OPTIONAL,

    -- bool-to-bool (see 24.4)
    &bool-to-bool

    CHOICE
    {logical    ENUMERATED{not}}
    DEFAULT logical:not,

    -- bool-to-int (see 24.5)
    &bool-to-int

    ENUMERATED {true-zero, true-one}
    DEFAULT true-one,

    -- int-to-bool (see 24.6)
    &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 (see 24.7)
&int-to-chars-size      ResultSize DEFAULT variable,
&int-to-chars-plus      BOOLEAN DEFAULT FALSE,
&int-to-chars-pad        ENUMERATED
                          {spaces, zeros} DEFAULT zeros,

-- int-to-bits (see 24.8)
&int-to-bits-encoded-as  ENUMERATED
                          {positive-int, twos-complement}
                          DEFAULT twos-complement,
&int-to-bits-unit        Unit (1..MAX) DEFAULT bit,
&int-to-bits-size        ResultSize DEFAULT variable,

-- bits-to-int (see 24.9)
&bits-to-int-decoded-assuming  ENUMERATED
                                {positive-int, twos-complement}
                                DEFAULT twos-complement,

-- char-to-bits (see 24.10)
&char-to-bits-encoded-as  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        ResultSize DEFAULT variable,

-- bits-to-char (see 24.11)
&bits-to-char-decoded-assuming  ENUMERATED
                                {iso10646, mapped}
                                DEFAULT iso10646,
&Bits-to-char-values          BIT STRING ORDERED OPTIONAL,
&Bits-to-char-chars           UniversalString (SIZE(1))
                                ORDERED OPTIONAL,

-- bit-to-bits (see 24.12)
&bit-to-bits-one              Non-Null-Pattern DEFAULT bits:'1'B,
&bit-to-bits-zero             Non-Null-Pattern DEFAULT bits:'0'B,

-- bits-to-bits (see 24.13)
&Source-values                BIT STRING ORDERED,
&Result-values                BIT STRING ORDERED
} WITH SYNTAX {

  -- Only one of the following clauses can be used.

  [INT-TO-INT &int-to-int]

  [BOOL-TO-BOOL [AS &bool-to-bool]]

  [BOOL-TO-INT AS &bool-to-int]

  [INT-TO-BOOL
    [AS &int-to-bool]
    [TRUE-IS &Int-to-bool-true-is]
    [FALSE-IS &Int-to-bool-false-is]]

  [INT-TO-CHARS
    [SIZE &int-to-chars-size]
    [PLUS-SIGN &int-to-chars-plus]
    [PADDING &int-to-chars-pad]]

  [INT-TO-BITS
    [AS &int-to-bits-encoded-as]
    [SIZE &int-to-bits-size]
    [MULTIPLE OF &int-to-bits-unit]]

  [BITS-TO-INT
    [AS &bits-to-int-decoded-assuming]]

  [CHAR-TO-BITS
    [AS &char-to-bits-encoded-as]
    [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
  [AS &bits-to-char-decoded-assuming]
  [BITS-LIST &Bits-to-char-values]
  [CHAR-LIST &Bits-to-char-chars]]

[BIT-TO-BITS
  [ZERO-PATTERN &bit-to-bits-zero]
  [ONE-PATTERN &bit-to-bits-one]]

[BITS-TO-BITS
  SOURCE-LIST &Source-values
  RESULT-LIST &Result-values]
}

```

24.2 Source and target of transforms

24.2.1 The #TRANSFORM encoding class allows the specification of procedures which transform input abstract values (the source) into output abstract values of the same or a different type (the result). The source is either the result of a previous transform, or is obtained from a source class (see 19.4). The result is either the source for a following transform, or becomes associated with a target class (see 19.4).

NOTE: Clause 23 also uses transforms whose source is a single bit and a single character.

24.2.2 These transforms are used in the definition of value mappings and in the definition of encoding objects for bit-field encoding classes (see clauses 20 to 23).

24.2.3 The source and result are indicated by words ("INT-TO-INT", "BOOL-TO-BOOL", etc) in the specification of a #TRANSFORM encoding object, and are defined in the associated text.

24.2.4 Clauses 24.2.4.1 to 24.2.4.3 specify rules for using transforms in succession, and for the source and target classes of a list of transforms.

24.2.4.1 When encoding objects of the class #TRANSFORM are specified in an ordered list, the source of a following #TRANSFORM encoding object shall be the result of the preceding #TRANSFORM encoding object.

24.2.4.2 For the first and last of an ordered list of transforms used in the definition of encoding objects in clauses 22 and 23, text in those clauses specifies the source for the first transform and the required result for the last transform.

24.2.4.3 For the first and last of an ordered list of transforms used in the specification of value mapping by transforms in 19.4, text in that clause specifies a source class and a target class, both of which will be of the bitstring, boolean, characterstring, integer or octetstring category (see 19.4.2). The required source for the first transform and the required result of the last transform (for each of these categories) are specified in 24.2.7.

24.2.5 Text in this clause specifies the source of a transform and the result of a transform as a single bit, bitstring, boolean, single character, characterstring or integer.

24.2.6 A source or target that is a single bit or a single character occurs only when successive transforms have these as output and input, or as specified in clauses 22 and 23. The first transform of the ordered list referenced in 19.4 shall not have a source which is a single bit or a single character. The last transform of the ordered list referenced in 19.4 shall not have a target which is a single bit or a single character.

24.2.7 When used in 19.4, the source for the first transform and the target for the last transform shall be the same as the category of the source encoding class and target encoding class (respectively), with the following exceptions. When the category of the source encoding class is octetstring, the source for the first transform shall be bitstring (treating each octetstring value as a bitstring value). When the last transform is "BITS-TO-BITS" with "MULTIPLE OF" set to 8, the target class may be octetstring.

24.3 The int-to-int transform

NOTE: Examples of this transform are given in D.1.2.2.

24.3.1 The int-to-int transform uses the following encoding parameter:

```
&int-to-int      CHOICE
                  {increment  INTEGER (1..MAX),
                   decrement  INTEGER (1..MAX),
                   multiply   INTEGER (2..MAX),
                   divide     INTEGER (2..MAX),
                   negate     ENUMERATED{value},
                   modulo     INTEGER (2..MAX),
                   subtract   ENUMERATED{lower-bound}
                  } OPTIONAL
```

24.3.2 The syntax for the int-to-int transform shall be:

```
[INT-TO-INT &int-to-int]
```

24.3.3 Both the source and result of this transform are integer. There are no bounds associated with the result unless this is the last transform in a mapping by transforms (see 19.4) and the target class of the mapping by transforms has bounds. In that case, the transform can only be applied to source integer values that map into the bounds of the target class.

24.3.4 An int-to-int transform is defined by giving a value to "INT-TO-INT", permitting any given encoding object to specify precisely one arithmetic operation. General arithmetic can, however, be defined by the use of a list of transforms (this is permitted wherever transforms involving integers are allowed).

24.3.5 The values "increment:n", "decrement:n", "multiply:n", "negate:n" have their normal mathematical meaning.

24.3.6 The value "divide:n" is defined to produce an integer result which is the integer value that is closest to the mathematical result, but is no further from zero than that result. In programming terms, "divide:n" truncates towards zero, so a value of -1 with "divide:2" will give zero.

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

24.3.8 The transform for the value "subtract:lower-bound" shall only be used as the first of an ordered list of transforms. The source shall have a lower bound.

24.3.9 This transform is defined to be reversible for the following values: "increment:n", "decrement:n", "multiply:n", "negate:n", "subtract:lower-bound". The values "divide:n" and "modulo:n" shall not be used where reversible transforms are required.

24.4 The bool-to-bool transform

24.4.1 The bool-to-bool transform uses the following encoding parameter:

```
&bool-to-bool    CHOICE
                  {logical      ENUMERATED{not}}
                  DEFAULT logical:not
```

24.4.2 The syntax for the bool-to-bool transform shall be:

```
[BOOL-TO-BOOL [AS &bool-to-bool]]
```

24.4.3 Both the source and result of this transform are boolean.

24.4.4 There is only one value for "BOOL-TO-BOOL", "AS logical:not", which may be omitted. This transform converts boolean "TRUE" to "FALSE", and vice-versa.

24.4.5 This transform is defined to be reversible.

24.5 The bool-to-int transform

24.5.1 The bool-to-int transform uses the following encoding parameter:

```
&bool-to-int          ENUMERATED {true-zero, true-one}
                      DEFAULT true-one
```

24.5.2 The syntax for the bool-to-int transform shall be:

```
[BOOL-TO-INT AS &bool-to-int]
```

24.5.3 The source for this transform is boolean and the result is integer with the value zero or one. The result has no associated bounds.

24.5.4 The value "true-zero" of "BOOL-TO-INT" produces integer 0 for "TRUE" and integer 1 for "FALSE". The value "true-one" produces integer 1 for "TRUE" and integer 0 for "FALSE".

24.5.5 This transform is defined to be reversible.

24.6 The int-to-bool transform

24.6.1 The int-to-bool transform uses the following encoding parameters:

```
&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
```

24.6.2 The syntax for the int-to-bool transform shall be:

```
[INT-TO-BOOL
 [AS &int-to-bool]
 [TRUE-IS &Int-to-bool-true-is]
 [FALSE-IS &Int-to-bool-false-is]]
```

24.6.3 The source for this transform is integer and the result is boolean.

24.6.4 At most one of "AS", "TRUE-IS" and "FALSE-IS" can be set. If none are set, then the default value for "AS" is assumed.

24.6.5 If "AS" is set (or is defaulted), then the value "zero-true" produces "TRUE" for the value zero and "FALSE" for all non-zero values, and the value "zero-false" produces "FALSE" for the value zero and "TRUE" for all non-zero values.

24.6.6 If "TRUE-IS" is set, the list of integer values for "TRUE-IS" produces "TRUE" and all other integer values produce "FALSE".

24.6.7 If "FALSE-IS" is set, the list of integer values for "FALSE-IS" produces "FALSE" and all other integer values produce "TRUE".

24.6.8 This transform shall not be used when reversible transforms are required.

24.7 The int-to-chars transform

24.7.1 The int-to-chars transform uses the following encoding parameters:

```
&int-to-chars-size    ResultSize DEFAULT variable,
&int-to-chars-plus    BOOLEAN DEFAULT FALSE,
&int-to-chars-pad     ENUMERATED
                      {spaces, zeros} DEFAULT zeros
```

24.7.2 The syntax for the int-to-chars transform shall be:

```
[INT-TO-CHARS
 [SIZE &int-to-chars-size]
 [PLUS-SIGN &int-to-chars-plus]
 [PADDING &int-to-chars-pad]]
```


24.7.3 The source for this transform is integer, and the result is characterstring.

24.7.4 "SIZE", "PLUS-SIGN", and "PADDING" all have default values and can be omitted.

24.7.5 "SIZE" specifies either:

- a) a fixed size in characters for the resulting size (a positive value of "SIZE"); or
- b) that a variable length string of characters is to be produced (the value "variable" of "SIZE"); or
- c) a fixed-size just large enough to contain the transform of all abstract values in the source class (the value "fixed-to-max" of "SIZE").

24.7.6 "SIZE" shall not be set to "fixed-to-max" unless this is the first transform in an ordered set, and the source class has both lower and upper bounds. This is synonymous with the specification of a positive value equal to the smallest value needed to contain the transform of every abstract value within the bounds.

24.7.7 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, and only if, "PLUS-SIGN" is set to true, positive values have a "+" (PLUS SIGN) pre-fixed to the digits.

24.7.8 The most significant digit shall be at the leading end of the characterstring.

24.7.9 If "SIZE" is "variable", then this is the resulting string of characters. In this case it is not an error to specify a value for "PADDING", but the value is ignored.

24.7.10 If "SIZE" is a positive value or "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 or application error.

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

24.7.11 If "SIZE" is a positive value or "fixed-to-max", and the string is smaller than the fixed size, then it is padded with either " " (SPACE) or "0" (DIGIT ZERO), determined by the value of "PADDING", pre-fixed to produce the specified size.

24.7.12 This transform is defined to be reversible.

24.8 The int-to-bits transform

NOTE: An example of this transform is given in D.1.5.5.

24.8.1 The int-to-bits transform uses the following encoding parameters:

```
&int-to-bits-encoded-as      ENUMERATED
                             {positive-int, twos-complement}
                             DEFAULT twos-complement,
&int-to-bits-unit           Unit (1..MAX) DEFAULT bit,
&int-to-bits-size           ResultSize DEFAULT variable
```

24.8.2 The syntax for the int-to-chars transform shall be:

```
[INT-TO-BITS
 [AS &int-to-bits-encoded-as]
 [SIZE &int-to-bits-size]
 [MULTIPLE OF &int-to-bits-unit]]
```

24.8.3 The source for this transform is integer and the result is bitstring. There are no bounds associated with the result. The following clauses use the term resulting bitstring for the result of this transform.

24.8.4 "AS", "SIZE", and "MULTIPLE OF" have default values and need not be set.

24.8.5 "SIZE" shall not be set to "fixed-to-max" unless this is the first transform in an ordered set in the syntax defined in 19.4, and the source class has both lower and upper bounds. This is synonymous with the specification of a positive value equal to the smallest value needed to contain the transform of every abstract value within the bounds.

24.8.6 "AS" 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.

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

24.8.8 The integer shall first be encoded into the minimum number of bits necessary to produce an initial bitstring. 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 bits or two successive leading one bits.

24.8.9 If "AS" is set to "positive-int", and the value to be transformed is negative, this is an ECN specification or an application 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.

24.8.10 If "SIZE" is "variable", then the initial bitstring becomes the resulting bitstring. In this case it is not an error to specify a value for "MULTIPLE OF", but the value is ignored.

24.8.11 If "SIZE" is a positive value, the size of the resulting bitstring shall be "MULTIPLE OF" multiplied by "SIZE".

24.8.12 If "SIZE" is "fixed-to-max", then the size of the resulting bitstring shall be the smallest multiple of "MULTIPLE OF" that is large enough to receive the encoding of any abstract value of the class to which the transform is applied.

24.8.13 If the initial bitstring (in an instance of application of this transform during encoding) is too large for the fixed size, then this is an ECN specification or an application 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.

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

24.8.15 This transform is defined to be reversible. This transform produces a self-delimiting bitstring if and only if "SIZE" is not "variable".

24.9 The bits-to-int transform

24.9.1 The bits-to-int transform uses the following encoding parameter:

```
&bits-to-int-decoded-assuming  ENUMERATED
                                {positive-int, twos-complement}
                                DEFAULT twos-complement
```

24.9.2 The syntax for the bits-to-int transform shall be:

```
[BITS-TO-INT
 [AS &bits-to-int-decoded-assuming]]
```

24.9.3 The source for this transform is bitstring and the result is integer. There are no bounds associated with the result.

24.9.4 The integer value shall be produced by interpreting the bits as 2's-complement or as a positive integer encoding, as specified in ITU-T Rec. X.690 | ISO/IEC 8825-1, 8.3.2 and 8.3.3. The value of "AS" (or its default value if not set) determines the encoding to be assumed.

24.9.5 This transform shall not be used where reversible transforms are required.

24.10 The char-to-bits transform

24.10.1 The char-to-bits transform uses the following encoding parameters:

<code>&char-to-bits-encoded-as</code>	ENUMERATED {iso10646, compact, mapped} DEFAULT compact,
<code>&Char-to-bits-chars</code>	UniversalString (SIZE(1)) ORDERED OPTIONAL,
<code>&Char-to-bits-values</code>	BIT STRING ORDERED OPTIONAL,
<code>&char-to-bits-unit</code>	Unit (1..MAX) DEFAULT bit,
<code>&char-to-bits-size</code>	ResultSize DEFAULT variable

24.10.2 The syntax for the char-to-bits transform shall be:

```
[CHAR-TO-BITS
  [AS &char-to-bits-encoded-as]
  [CHAR-LIST &Char-to-bits-chars]
  [BITS-LIST &Char-to-bits-values]
  [SIZE &char-to-bits-size]
  [MULTIPLE OF &char-to-bits-unit]]
```

24.10.3 The source for this transform is a single character and the result is bitstring.

24.10.4 Where the following text refers to a possible "effective permitted alphabet constraint", such a constraint exists if and only if the transform is the first in an ordered list used in 23.4 and the class to which the encoding object is applied has an effective permitted alphabet constraint.

24.10.5 "AS", "SIZE" and "MULTIPLE OF" all have default values and need not be set. "CHAR-LIST" and "BITS-LIST" are only used if "AS" is set to "mapped", in which case their presence is mandatory, and they shall then contain at least one element in the ordered list.

24.10.6 ECN supports only characters in the ISO/IEC 10646-1 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 transform.

24.10.7 If "AS" is "mapped", then the transform is specified by the values of "CHAR-LIST" and "BITS-LIST", both of which shall be specified, and the values of "MULTIPLE OF" and "SIZE" are ignored. The transform is specified in 24.10.7.1 to 24.10.7.5.

24.10.7.1 "CHAR-LIST" and "BITS-LIST" are respectively a list of single characters and of bitstring values. (These parameters are ignored if "AS" is not set to "mapped").

24.10.7.2 There shall be an equal number of values in each list, and all character values in "CHAR-LIST" shall be distinct.

24.10.7.3 The encoding of a character in "CHAR-LIST" is the bitstring specified in the corresponding position in "BITS-LIST".

24.10.7.4 If in an instance of application of this transform a character is to be transformed that is not in the "CHAR-LIST", this is an ECN specification or an application 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.

24.10.7.5 In this case ("AS" set to "mapped"), the transform is defined to be reversible if and only if the set of all bitstring values in "BITS-LIST" are distinct. The result is self-delimiting if the bitstring values in "BITS-LIST" are self-delimiting (see 3.2.41).

24.10.8 If "AS" is "iso10646", the transform is specified in 24.10.8.1 to 24.10.8.5.

24.10.8.1 The character is first converted to an integer with the numerical value specified in ISO/IEC 10646-1.

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

24.10.8.2 If the character is from a character string that has an associated effective alphabet constraint (see 24.10.4), then the integer has effective size constraints just sufficient to contain the numerical values of all characters in the effective permitted alphabet.

24.10.8.3 If there is no effective alphabet constraint, then the integer has an associated effective size constraint of 0..32 767.

24.10.8.4 This integer value is then converted to bits using the transform:

```
INT-TO-BITS -- (see 24.8)
  AS positive-int
  SIZE <size>
  MULTIPLE OF <multiple-of>
```

where "<size>" is the value of "SIZE" and "<multiple-of>" is the value of "MULTIPLE OF" for the char-to-bits transform. ("SIZE" and "MULTIPLE OF" take their default values if not set).

24.10.8.5 In this case ("AS" set to "iso10646"), the transform is defined to be reversible. It produces a self-delimiting string of bits if and only if "SIZE" is not "variable".

24.10.9 If "AS" is "compact", then it is an ECN specification error if there is no effective permitted alphabet constraint, otherwise the transform is specified in 24.10.9.1 to 24.10.9.4.

24.10.9.1 All characters in the effective permitted alphabet are placed in canonical order using their ISO/IEC 10646-1 value, lowest value first. The first in the list is then assigned the integer value zero, the next one, and so on.

24.10.9.2 If the effective permitted alphabet contains "n" characters, then the integer has an effective size constraint of 0..n-1.

24.10.9.3 This integer is then converted to bits using the transform:

```
INT-TO-BITS -- (see 24.8)
  AS positive-int
  SIZE <size>
  MULTIPLE OF <multiple-of>
```

where "<size>" is the value of "SIZE" and "<multiple-of>" is the value of "MULTIPLE OF" for the char-to-bits transform. ("SIZE" and "MULTIPLE OF" take their default values if not set).

NOTE: The PER encoding of character string types uses the equivalent of "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 present document.

24.10.9.4 In this case ("AS" set to "compact"), the transform is defined to be reversible. It produces a self-delimiting string of bits if and only if "SIZE" is not "variable".

24.11 The bits-to-char transform

24.11.1 The bits-to-char transform uses the following encoding parameters:

```
&bits-to-char-decoded-assuming  ENUMERATED
                                {iso10646, mapped}
                                DEFAULT iso10646,
&Bits-to-char-values             BIT STRING ORDERED OPTIONAL,
&Bits-to-char-chars             UniversalString (SIZE(1))
                                ORDERED OPTIONAL
```

24.11.2 The syntax for the bits-to-char transform shall be:

```
[BITS-TO-CHAR
  [AS &bits-to-char-decoded-assuming]
  [BITS-LIST &Bits-to-char-values]
  [CHAR-LIST &Bits-to-char-chars]]
```

24.11.3 The source for this transform is bitstring and the result is a single character.

NOTE: This transform cannot be used as the last transform in an ordered list.

24.11.4 If "AS" is "iso10646", then the bitstring shall be interpreted as a positive integer encoding which contains the ISO/IEC 10646-1 numerical value of a character. It is an ECN specification error if the integer value exceeds 32 767.

24.11.5 If "AS" is "mapped", then the transform is specified by the values of "CHAR-LIST" and "BITS-LIST". The transform is defined in 24.11.5.1 to 24.11.5.5.

24.11.5.1 "CHAR-LIST" and "BITS-LIST" are respectively a list of single characters and of bitstring values. (These parameters are ignored if "AS" is not set to "mapped").

24.11.5.2 There shall be an equal number of values in each list, and all character values and all bitstring values in the list shall be distinct.

24.11.5.3 The encoding of a character in the "CHAR-LIST" list is the bitstring specified in the corresponding position in the "BITS-LIST".

24.11.5.4 If in an instance of application of this transform a bitstring is to be transformed that is not in the "BITS-LIST", this is an ECN specification or an application 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.

24.11.5.5 The transform is defined to be reversible.

24.12 The bit-to-bits transform

24.12.1 The bit-to-bits transform uses the following encoding parameters:

<code>&bit-to-bits-one</code>	Non-Null-Pattern DEFAULT bits:'1'B,
<code>&bit-to-bits-zero</code>	Non-Null-Pattern DEFAULT bits:'0'B

24.12.2 The syntax for the bit-to-bits transform shall be:

```
[BIT-TO-BITS
  [ZERO-PATTERN &bit-to-bits-zero]
  [ONE-PATTERN &bit-to-bits-one]]
```

24.12.3 The source for this transform is a single bit from either:

- a) a previous transform that produced a bit; or
- b) the specification of an encoding for the bitstring category (see 23.2);

and the result is a bitstring.

24.12.4 At most one of "ZERO-PATTERN" and "ONE-PATTERN" shall be "different:any".

NOTE: A value of "different:any" here means a pattern that is not the same as the other pattern, but is the same length.

24.12.5 The "any-of-length" alternative shall not be used for either "ZERO-PATTERN" or "ONE-PATTERN".

24.12.6 If the bit is set to zero, the result is the "ZERO-PATTERN". If the bit is set to one, the result is the "ONE-PATTERN".

24.12.7 It is an ECN specification error if "ZERO-PATTERN" and "ONE-PATTERN" are the same, or if one is an initial sub-string of the other.

24.12.8 This transform is defined to be reversible and the result is self-delimiting.

24.13 The bits-to-bits transform

24.13.1 The bits-to-bits transform uses the following encoding parameters:

<code>&Source-values</code>	BIT STRING ORDERED,
<code>&Result-values</code>	BIT STRING ORDERED

24.13.2 The syntax for the bits-to-bits transform shall be:

```
[BITS-TO-BITS
  SOURCE-LIST &Source-values
  RESULT-LIST &Result-values]
```

24.13.3 The source for this transform is bitstring and the result is bitstring.

24.13.4 "SIZE" and "MULTIPLE OF" both have default values and need not be set. "SOURCE-LIST" and "RESULT-LIST" are required, and shall contain at least one element in the ordered list.

24.13.5 The transform is specified by the values of "SOURCE-LIST" and "RESULT-LIST".

24.13.6 There shall be an equal number of bitstring values in each list, and all bitstring values in "SOURCE-LIST" shall be distinct.

24.13.7 The transform of a bitstring in "SOURCE-LIST" is the bitstring specified in the corresponding position in "RESULT-LIST".

24.13.8 If, in an instance of application of this transform, a source bitstring is not in the "SOURCE-LIST", this is an ECN specification or an application 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.

24.13.9 The transform is defined to be reversible if and only if the set of all bitstring values in "RESULT-LIST" are distinct. The result is self-delimiting if the bitstring values in "RESULT-LIST" are distinct and self-delimiting (see 3.2.41).

25 Complete encodings and the #OUTER class

25.1 General rules for encoding and decoding

25.1.1 If there is no encoding object of the #OUTER class (see 25.2) in the combined encoding object set being to a type in the ELM, then the encoder and decoder shall assume encoding object of this class in which all encoding parameters have their default values.

25.1.2 It is an encoder error (which shall be detected by a decoder) if the rules for determining the size of a container result in the end of the container being detected when there are still elements within the container for which encodings are expected but have not yet occurred, unless all such elements are optional with optionality determined by the end of the container.

NOTE: The container may be a concatenation or a repetition, or may be the end of the PDU.

25.2 Encoding parameters, syntax, and purpose for the #OUTER class

25.2.1 The syntax for defining encoding objects of the #OUTER class is defined as:

```
#OUTER ::= ENCODING-CLASS {

    -- Alignment point
    &alignment-point                               ENUMERATED
                                                    {unchanged, reset } DEFAULT reset,

    -- Padding
    &post-padding-unit                             Unit (1..MAX) DEFAULT octet,
    &post-padding                                   Padding DEFAULT zero,
    &post-padding-pattern                         Non-Null-Pattern (ALL EXCEPT other)
                                                    DEFAULT bits:'0'B,

    -- Bit reversal specification (see 22.12)
    &bit-reversal                                   ReversalSpecification
                                                    DEFAULT no-reversal,

    -- Added bits action
    &added-bits                                     ENUMERATED
                                                    {hard-error, signal-application,
                                                    silently-ignore, next-value}
                                                    DEFAULT hard-error

} WITH SYNTAX {
```



```

[ALIGNMENT &alignment-point]
[PADDING
  [MULTIPLE OF &post-padding-unit]
  [POST-PADDING &post-padding
    [PATTERN &post-padding-pattern]]]
[BIT-REVERSAL &bit-reversal]
[ADDED BITS DECODING          &added-bits]
}

```

25.2.2 Encoding objects of the #OUTER class specify encoder and decoder actions in relation 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.

25.2.3 Three independent specifications can be made (see 25.2.4 to 25.2.6).

25.2.4 The "ALIGNMENT" specification is applicable only for a contained type, and determines whether the alignment point is to be reset to the head of the container or is to be the same as that in use for the encoding of the container.

25.2.5 The "PADDING" specification determines that the entire encoding is to be padded with trailing bits to make the number of bits from the alignment point an integral multiple of some unit.

25.2.6 The "ADDED BITS DECODING" specification is applicable only to encoders, and determines the action to be taken if there are further bits in the PDU after decoding according to encoding specifications has been completed.

NOTE: This provision is primarily to provide a simple mechanism for extensibility without use of the ASN.1 extensibility marker. A later version of the present document is expected to give enhanced support for extensibility.

25.2.7 "ALIGNMENT", "PADDING", and "ADDED BITS DECODING" all take their default values if not set or if there is no encoding object of class #OUTER in the combined encoding object set.

NOTE: The default values are those used by the encoding object of class #OUTER for PER basic unaligned.

25.3 Encoder actions for #OUTER

25.3.1 If "ALIGNMENT" is "unchanged", then the alignment point used in encoding a contained type shall be the alignment point used in encoding the container.

25.3.2 If "ALIGNMENT" is "reset", then the alignment point used in encoding a contained type shall be the start of the encoding of that type.

25.3.3 If "PADDING" is set, then the encoder shall add bits in accordance with the value of "PADDING" and "PATTERN" to make the number of bits from the alignment point a multiple of "MULTIPLE OF" units. "PATTERN" shall be replicated and truncated as necessary.

25.3.4 The encoder shall diagnose an ECN specification or application error if the encoding is for a type in a contents constraint on an octetstring, and the encoding of the type (after all specified "PADDING" actions) is not an integral multiple of eight bits.

25.3.5 If bit-reversal is set, the encoder actions specified in 22.12 shall be applied using the value of "MULTIPLE OF" specified for (or defaulted in) "PADDING".

25.3.6 The encoder shall ignore "ADDED BITS DECODING".

25.4 Decoder actions for #OUTER

25.4.1 If bit-reversal is set, the decoder actions specified in 22.12 shall be applied using the value of "MULTIPLE OF" specified for (or defaulted in) "PADDING".

25.4.2 If "ALIGNMENT" is "unchanged", then the alignment point used in encoding a contained type shall be the alignment point used in encoding the container.

25.4.3 If "ALIGNMENT" is "reset", then the alignment point used in encoding a contained type shall be the start of the encoding of that type.

25.4.4 The decoder shall determine the bits added by "PADDING" (if any), and shall silently ignore the added bits, no matter what their value.

25.4.5 If the PDU (or the container of a contained type) contains further bits after the end of the encoding, then the decoder shall take the following actions:

- a) if "ADDED BITS DECODING" is "hard-error": diagnose an encoder error;
- b) if "ADDED BITS DECODING" is "signal-application": ignore all further bits and signal the application that there may be critical extensions to the protocol;
- c) if "ADDED BITS DECODING" is "silently-ignore": ignore all further bits;
- d) if "ADDED BITS DECODING" is "next-value": cease decoding and expect the application to initiate decoding of a new value from the remaining bits.

Annex A (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 the present document.

A.1 Exports and imports clauses

The productions "AssignedIdentifier", "Symbol" and "Reference" of 12.1, as well as clauses 12.12, 12.15, and 12.19 of ITU-T Rec. X.680 | ISO/IEC 8824-1 are modified as follows:

12.1 AssignedIdentifier ::= DefinitiveIdentifier

```

Symbol ::=
  Reference
  BuiltinEncodingClassReference
  ParameterizedReference

Reference ::=
  encodingclassreference
  ExternalEncodingClassReference
  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: "BuiltinEncodingClassReference" can only be used as a "Symbol" in an imports clause. The use of production "ExternalEncodingClassReference" in "Reference" is explained in 14.11.

12.12 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.

12.15 When the "SymbolsImported" alternative of "Imports" is selected:

- a) each "Symbol" in "SymbolsFromModule" shall either
 - 1) be defined in the body of the module denoted by the "GlobalModuleReference" in "SymbolsFromModule", or
 - 2) 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, 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.

A.2 Addition of "REFERENCE"

NOTE: This modification is introduced for the sole purpose of clause 23.

The production "Type" in ITU-T Rec. X.680 | ISO/IEC 8824-1, 16.1, is modified as follows:

```
Type ::=
    BuiltinType           |
    ReferencedType        |
    ConstrainedType       |
    REFERENCE
```

A.3 Notation for character string values

The production "CharsDefn" of ITU-T Rec. X.680 | ISO/IEC 8824-1, 36.7, is modified as follows:

```
CharsDefn ::=
    cstring           |
    Quadruple         |
    Tuple             |
    AbsoluteCharReference

AbsoluteCharReference ::=
    ModuleIdentifier
    "."
    valuereference
```

The "AbsoluteCharReference" is a fully-qualified name which 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 B (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 the present document.

B.1 Definitions

The following definitions are added to ITU-T Rec. X.681 | ISO/IEC 8824-2, 3.4:

encoding class field type: type specified by reference to some type field of an encoding object class

encoding class field: field which contains an arbitrary encoding class

encoding object field: field which contains an encoding object of some specified encoding class

Such a field is either of fixed-class or of variable-class. In the former case, the class of the encoding object is fixed by the field specification. In the latter case, the class of the encoding object is contained in some (specific) encoding class field of the same encoding object.

encoding object list field: field which contains an ordered non-empty list of encoding objects of some specified encoding class

encoding object set field: field which contains a set of encoding objects of some specified encoding class

value list field: field which contains an ordered (possibly empty) list of values of some specified type

B.2 Additional lexical items

NOTE: This modification is introduced for the sole purpose of clause 23.

The following definitions are added to ITU-T Rec. X.681 | ISO/IEC 8824-2, 7:

B.2.1 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.

B.2.2 Encoding class field references

Name of item – `encodingclassfieldreference`

An "encodingclassfieldreference" shall consist of an ampersand ("&") immediately followed by a number sign ("#") which itself is immediately followed by a sequence of characters as specified for an "encodingclassreference" in 8.3.

B.3 Addition of "ENCODING-CLASS"

NOTE: This modification is introduced for the sole purpose of clause 23.

Replace the reserved word "CLASS" with "ENCODING-CLASS" in ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.3.

B.4 FieldSpec additions

NOTE: This modification is introduced for the sole purpose of clause 23.

ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.4, is modified as follows:

```
FieldSpec ::=
    FixedTypeValueFieldSpec
    FixedTypeValueSetFieldSpec
    FixedTypeValueListFieldSpec
    FixedClassEncodingObjectFieldSpec
    VariableClassEncodingObjectFieldSpec
    FixedClassEncodingObjectSetFieldSpec
    FixedClassEncodingObjectListFieldSpec
    EncodingClassFieldSpec
```

B.5 Fixed-type value list field spec

NOTE: This modification is introduced for the sole purpose of clause 23.

A "FixedTypeValueListFieldSpec" specifies that the field is a fixed-type value list field (see B.1 of the present document):

```
FixedTypeValueListFieldSpec ::=
    valuelistfieldreference
    DefinedType
    ORDERED
    FixedTypeValueListOptionalitySpec ?

FixedTypeValueListOptionalitySpec ::= OPTIONAL | DEFAULT ValueList
```

The name of the field is "valuelistfieldreference". The "DefinedType" references the type of values contained in the field. The "FixedTypeValueListOptionalitySpec", if present, specifies that the field may be unspecified in an encoding object definition, or, in the "DEFAULT" case, that omission produces the following "ValueList" (see ITU-T Rec. X.680 | ISO/IEC 8824-1, 25.3), all of whose values shall be of "DefinedType".

B.6 Fixed-class encoding object field spec

NOTE: This modification is introduced for the sole purpose of clause 23.

A "FixedClassEncodingObjectFieldSpec" specifies that the field is a fixed-class encoding object field (see B.1 of the present document):

```
FixedClassEncodingObjectFieldSpec ::=
    objectfieldreference
    DefinedOrBuiltinEncodingClass
    EncodingObjectOptionalitySpec?

EncodingObjectOptionalitySpec ::= OPTIONAL | DEFAULT EncodingObject
```

The name of the field is "objectfieldreference". The "DefinedOrBuiltinEncodingClass" references the encoding class of the encoding object contained in the field (which may be the "EncodingClass" 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" (see 17.1.4 of the present document) which shall be of the "DefinedOrBuiltinEncodingClass".

B.7 Variable-class encoding object field spec

A "VariableClassEncodingObjectFieldSpec" specifies that the field is a variable-class encoding object field (see B.1 of the present document):

```
VariableClassEncodingObjectFieldSpec ::=
    objectfieldreference
    encodingclassfieldreference
    EncodingObjectOptionalitySpec?
```

The name of the field is "objectfieldreference". The "encodingclassfieldreference" references an encoding class field of the encoding class being specified. The "EncodingObjectOptionalitySpec", if present, specifies that the encoding object may be omitted in an encoding object definition, or, in the "DEFAULT" case, that omission produces the following "EncodingObject". The "EncodingObjectOptionalitySpec" shall be such that:

- a) if the type field denoted by the "encodingclassfieldreference" has an "EncodingClassOptionalitySpec" of "OPTIONAL", then the "EncodingObjectOptionalitySpec" shall also be "OPTIONAL"; and
- b) if the "EncodingObjectOptionalitySpec" is "DEFAULT EncodingObject", then the encoding class field denoted by the "encodingclassfieldreference" shall have an "EncodingClassOptionalitySpec" of "DEFAULT DefinedOrBuiltinEncodingClass", and "EncodingObject" shall be an encoding object of that class.

B.8 Fixed-class encoding object set field spec

NOTE: This modification is introduced for the sole purpose of clause 23.

A "FixedClassEncodingObjectSetFieldSpec" specifies that the field is a fixed-class encoding object set field (see B.1 of the present document):

```
FixedClassEncodingObjectSetFieldSpec ::=
    objectsetfieldreference
    DefinedOrBuiltinEncodingClass
    EncodingObjectSetOptionalitySpec?

EncodingObjectSetOptionalitySpec ::= OPTIONAL | DEFAULT EncodingObjectSet
```

The name of the field is "objectsetfieldreference". The "DefinedOrBuiltinEncodingClass" 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" (see clause 18), all of whose objects shall be of "DefinedOrBuiltinEncodingClass".

B.9 Fixed-class encoding object list field spec

NOTE: This modification is introduced for the sole purpose of clause 23.

A "FixedClassEncodingObjectListFieldSpec" specifies that the field is a fixed-class encoding object list field (see B.1 of the present document):

```
FixedClassEncodingObjectListFieldSpec ::=
    encodingobjectlistfieldreference
    DefinedOrBuiltinEncodingClass
    ORDERED
    EncodingObjectListOptionalitySpec?

EncodingObjectListOptionalitySpec ::= OPTIONAL | DEFAULT EncodingObjectList
```

The name of the field is "encodingobjectlistfieldreference". The "DefinedOrBuiltinEncodingClass" 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" (see B.11 of the present document), all of whose objects shall be of "DefinedOrBuiltinEncodingClass".

B.10 Encoding class field spec

NOTE: This modification is introduced for the sole purpose of clause 23.

An "EncodingClassFieldSpec" specifies that the field is an encoding class field (see B.1 of the present document):

```
EncodingClassFieldSpec ::=
    encodingclassfieldreference
    EncodingClassOptionalitySpec?

EncodingClassOptionalitySpec ::= OPTIONAL | DEFAULT DefinedOrBuiltinEncodingClass
```

The name of the field is "encodingclassfieldreference". If the "EncodingClassOptionalitySpec" is absent, all encoding object definitions for that class are required to include a specification of an encoding class for that field. If "OPTIONAL" is present, then the field can be left undefined. If "DEFAULT" is present, then the following "DefinedOrBuiltinEncodingClass" provides the default setting for the field if it is omitted in a definition.

B.11 Encoding object list notation

```
EncodingObjectList ::= "{" EncodingObject "," + "}"
```

The "EncodingObjectList" is an ordered list of one or more encoding objects of the governing class. 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.

B.12 Primitive field names

ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.13, is modified as follows:

9.13 The construct "PrimitiveFieldName" is used to identify a field relative to the encoding class containing its specification:

```
PrimitiveFieldName ::=
    valuefieldreference           |
    valuesetfieldreference        |
    valuelistfieldreference
```

B.13 Additional reserved words

ITU-T Rec. X.681 | ISO/IEC 8824-2, 10.6, is modified as follows:

10.6 A "word" lexical item used as a "Literal" cannot be one of the following:

```
BEGIN
BER
CER
DER
ENCODE
ENCODE-DECODE
END
FALSE
MINUS-INFINITY
NON-ECN-BEGIN
NULL
OUTER
PER-BASIC-ALIGNED
PER-BASIC-UNALIGNED
PER-CANONICAL-UNALIGNED
PER-CANONICAL-UNALIGNED
```


PLUS-INFINITY
TRUE
UNION
USE
USE-SET

NOTE: This list comprises only those ASN.1 reserved words which can appear as the first item of a "Value", "EncodingObject", or "EncodingObjectSet", 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.

B.14 Definition of encoding objects

The restriction imposed by ITU-T Rec. X.681 | ISO/IEC 8824-2, 10.12.d, is removed.

NOTE: This affects the defined syntax for defining encoding objects of some classes (see clauses 23 and 24). It means, for example, that, for a defined syntax such as:

```
[BOOL-TO-INT [AS &bool-to-int]]
```

the user is 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 transform "BOOL-TO-INT".

B.15 Additions to "Setting"

ITU-T Rec. X.681 | ISO/IEC 8824-2, 11.6, is modified as follows:

11.6 A "Setting" specifies the setting of some field within an encoding object being defined:

```
Setting ::=
    Value           |
    ValueSet        |
    ValueList       |
    EncodingObject  |
    EncodingObjectSet |
    EncodingObjectList |
    DefinedOrBuiltinEncodingClass |
    OUTER
```

If the field is:

- a) a value field, the "Value" alternative;
- b) a value set field, the "ValueSet" alternative;
- c) a value list field, the "ValueList" alternative;
- d) an encoding object field, the "EncodingObject" alternative;
- e) an encoding object set field, the "EncodingObjectSet" alternative;
- f) an encoding object list field, the "EncodingObjectList" alternative;

- g) an encoding class field, the "DefinedOrBuiltinEncodingClass" alternative;
- h) a reference field, the "Value" or the "OUTER" alternative;

shall be selected. For a reference field specified using the syntax of clauses 20 to 25, the "Value" shall be a dummy parameter. "OUTER" can be used whenever a reference is required and identifies a container which is the entire encoding.

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.

B.16 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, B.16.2 to B.16.4 below specify the type that is referenced.

B.16.1 The notation for an encoding class field type shall be "EncodingClassFieldType":

```
EncodingClassFieldType ::=
    DefinedOrBuiltinEncodingClass
    "."
    FieldName
```

where the "FieldName" is as specified in ITU-T Rec. X.681 | ISO/IEC 8824-2, 9.14, relative to the encoding class identified by the "DefinedOrBuiltinEncodingClass".

B.16.2 For a fixed-type value, a fixed type value set field, or a fixed type value list field, the notation denotes the "Type" that appears in the specification of that field in the definition of the encoding object class.

B.16.3 This notation is not permitted if the field is an encoding object, an encoding object set or an encoding object list field.

B.16.4 The notation for defining a value of this type shall be "FixedTypeFieldVal" as defined in ITU-T Rec. X.681 | ISO/IEC 8824-2, 14.6.

Annex C (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 the present document.

C.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 the present document. The "ParameterizedAssignment" construct is:

```
ParameterizedAssignment ::=
    ParameterizedEncodingObjectAssignment      |
    ParameterizedEncodingClassAssignment       |
    ParameterizedEncodingObjectSetAssignment
```

8.3 ParameterList ::= "<" Parameter "," + ">"

```
Governor ::=
    EncodingClassFieldType                     |
    REFERENCE                                |
    DefinedOrBuiltinEncodingClass             |
    #ENCODINGS
```

A "DummyReference" in "Parameter" may stand for:

- an encoding class, in which case there shall be no "ParamGovernor";
- an ASN.1 value, value set, or value list, 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 encoding object, or an encoding object list, in which case the "ParamGovernor" shall be present as a "Governor" that is an encoding class ("DefinedOrBuiltinEncodingClass");
- an encoding object set in which case the "ParamGovernor" shall be present as a "Governor" that is "#ENCODINGS".

NOTE: "DummyGovernor"s are not allowed in ECN.

C.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
    DefinedOrBuiltinEncodingClass
    " :="
    EncodingObject
```



```

ParameterizedEncodingObjectSetAssignment ::=
  encodingobjectsetreference
  ParameterList
  #ENCODINGS
  " ::= "
  EncodingObjectSet

```

ITU-T Rec. X.683 | ISO/IEC 8824-4, 8.4, is modified as follows:

8.4 The scope of a "DummyReference" appearing in a "ParameterList" is the "ParameterList" itself, together with that part of the "ParameterizedAssignment" which follows the "::<=".

In case of a "ParameterizedEncodingObjectAssignment", the scope extends to the "DefinedOrBuiltinEncodingClass" which precedes the "::<=".

The "DummyReference" hides any other "Reference" with the same name in that scope.

NOTE: The special case for "ParameterizedEncodingObjectAssignment" is intended to be used in common with renames clauses (see D.3.3.3). It allows to write an assignment such as the following in which the dummy parameter "#Any-Class" of the encoding object "new-component-encoding" is used as an actual parameter for the encoding class "#New-component":

```

new-component-encoding {< #Any-class >} #New-component {< #Any-class >} ::=
  { -- encoding object definition -- }

```

C.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

```


C.4 Actual parameter list

ITU-T Rec. X.683 | ISO/IEC 8824-4, 9.5, is modified as follows:

9.5 The "ActualParameterList" is:

```
ActualParameterList ::=
    "{<" ActualParameter "," + ">}"

ActualParameter ::=
    Value
    ValueSet
    ValueList
    DefinedOrBuiltinEncodingClass
    EncodingObject
    EncodingObjectSet
    EncodingObjectList
    identifier
    STRUCTURE
    OUTER
```

If the corresponding dummy parameter is:

- a) a value, the "Value" alternative;
- b) a value set, the "ValueSet" alternative;
- c) a value list, the "ValueList" alternative;
- d) an encoding class, the "DefinedOrBuiltinEncodingClass" alternative;
- e) an encoding object, the "EncodingObject" alternative;
- f) an encoding object set, the "EncodingObjectSet" alternative;
- g) an encoding object list, the "EncodingObjectList" alternative;
- h) a reference, the "identifier", "STRUCTURE" or "OUTER" alternative;

shall be selected. "STRUCTURE" shall only be selected when the actual parameter is used as specified in 17.5.16. "OUTER" can be used whenever a reference is required to identify a container, and identifies the container of the entire encoding.

Annex D (informative): Examples

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

- General examples, which show the look-and-feel of ECN definitions (D.1.1 to D.1.12.4).
- Specialization examples, which show how to modify some parts of a standardized encoding. Each example has a description of the requirements for the encoding and a description of the selected solution and possible alternative solutions (D.1.1 to D.2.11).
- Explicitly generated structure examples, which show the use of explicitly generated structures when the same specialized encoding is used several times (D.3.1 to D.3.4).
- Legacy protocol examples, which show how to construct ECN definitions for a protocol whose message encodings have been specified using a tabular notation (D.4.1 to D.4.8)

D.1 General examples

The examples described in D.1.1 to D.1.13 are part of a complete ECN specification whose ELM, ASN.1, and EDM modules are given in outline in D.1.14, D.1.16 and D.1.17, and are given completely in a copy of this annex which is available from the website cited in annex F.

D.1.1 An encoding object for a boolean type

D.1.1.1 The ASN.1 assignment is:

```
Married ::= BOOLEAN
```

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

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

marriedEncoding-1 #MarriedEncoding ::= booleanEncoding
```

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

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

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

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

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

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

D.1.2 An encoding object for an integer type

D.1.2.1 The ASN.1 assignments are:

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

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

D.1.2.2 The encoding object assignments are:

```
evenPositiveIntegerEncoding #EvenPositiveInteger ::= {
  USE #INT (0..MAX)
  MAPPING TRANSFORMS {{INT-TO-INT divide:2}}
  WITH PER-BASIC-UNALIGNED}

evenNegativeIntegerEncoding #EvenNegativeInteger ::= {
  USE #INT (MIN..0)
  MAPPING TRANSFORMS {{INT-TO-INT divide:2
    -- Note: -1 / 2 = 0 - see clause 24.3.6 -- }}
  WITH PER-BASIC-UNALIGNED}
```

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

D.1.3 Another encoding object for an integer type

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

D.1.3.2 The ASN.1 assignment is:

```
Integer ::= INTEGER(0..65 535)
```

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

```
integerRightAlignedEncoding #Integer ::= {
  ENCODING {
    ALIGNED TO NEXT octet
    ENCODING-SPACE
    SIZE 16
    VALUE-PADDING
    JUSTIFIED right: 0}}
```

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

D.1.4.1 The ASN.1 assignment is:

```
IntegerWithHole ::= INTEGER (-256..-1 | 32..1 056)
```

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

```
integerWithHoleEncoding #IntegerWithHole ::= {
  USE #INT (0..1 280)
  MAPPING ORDERED VALUES
  WITH PER-BASIC-UNALIGNED}
```

D.1.4.3 "IntegerWithHole" is encoded as a positive integer. Values in the range -256..-1 are mapped to values in the range 0..255 and values in the range 32..1 056 are mapped to 256..1 280.

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

D.1.5.1 The ASN.1 assignments are:

```
PositiveInteger ::= INTEGER (1..MAX)

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


D.1.5.2 The encoding object assignments are:

```
positiveIntegerEncoding #PositiveInteger ::=
    integerEncoding

negativeIntegerEncoding #NegativeInteger ::=
    integerEncoding
```

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

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

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

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

D.1.6 Positive integers encoded in BCD

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

D.1.6.2 The ASN.1 assignment is:

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

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

```
positiveIntegerBCDEncoding #PositiveIntegerBCD ::= {
    USE #CHARS
    MAPPING TRANSFORMS{{
        INT-TO-CHARS
        -- We convert to characters (e.g., integer 42
        -- becomes character string "42") and encode the characters
        -- with the encoding object "numeric-chars-to-bcdEncoding"
        SIZE variable
        PLUS-SIGN FALSE}}
    WITH numeric-chars-to-bcdEncoding }
```



```

numeric-chars-to-bcdEncoding #CHARS ::= {
    ALIGNED TO NEXT nibble
    ENCODER-TRANSFORMS {{
        CHAR-TO-BITS
        -- We convert each character to a bitstring
        --(e.g., character "4" becomes '0100'B and "2" becomes '0010'B)
        AS mapped
        CHAR-LIST    { "0","1","2","3",
                       "4","5","6","7",
                       "8","9"}
        BITS-LIST    { '0000'B, '0001'B, '0010'B, '0011'B,
                       '0100'B, '0101'B, '0110'B, '0111'B,
                       '1000'B, '1001'B }}}
    REPETITION-ENCODING {
        REPETITION-SPACE
        -- We determine the concatenation of the bitstrings for the
        -- characters and add a terminator (e.g.,
        -- '0100'B + '0010'B becomes '0100 0010 1111'B)
        SIZE variable-with-determinant
        DETERMINED BY pattern
        PATTERN bits:'1111'B}}

```

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

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

D.1.7 An encoding object of class #BITS

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

D.1.7.2 The ASN.1 assignment is:

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

D.1.7.3 The encoding object assignment (see 23.2.1, 23.12.1 and 23.13.1) is:

```

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

```

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

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

D.1.8 An encoding object for an octetstring type

D.1.8.1 The ASN.1 assignment is:

```
OctetString ::= OCTET STRING
```


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

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

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

D.1.9 An encoding object for a character string type

D.1.9.1 The ASN.1 assignment is:

```
CharacterString ::= PrintableString
```

D.1.9.2 The encoding object assignment (see 23.4.1 and 23.13.1) is:

```
characterStringEncoding #CharacterString ::= {
    ALIGNED TO NEXT octet
    ENCODER-TRANSFORMS {{CHAR-TO-BITS AS compact}}
    REPETITION-ENCODING {
        REPETITION-SPACE
        SIZE variable-with-determinant
        DETERMINED BY container
        USING OUTER}}
```

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

D.1.10 Mapping character values to bit values

D.1.10.1 The ASN.1 assignment is:

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

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

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

where "integerEncoding" is defined in D.1.5.5.

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

D.1.11 An encoding object for a sequence type

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

D.1.11.2 The ASN.1 assignment is:

```
Sequence1 ::= SEQUENCE {
    a          BOOLEAN,
    b          INTEGER OPTIONAL,
    c          OCTET STRING}
```

D.1.11.3 The ECN assignments (see 17.5 and 23.10.1) are:

```
sequenceEncoding #Sequence1 ::= {
    ENCODE STRUCTURE {
        b USE-SET OPTIONAL-ENCODING parameterizedPresenceEncoding {< a >},
        c octetStringEncoding}
    WITH PER-BASIC-UNALIGNED}

parameterizedPresenceEncoding {< REFERENCE:reference >} #OPTIONAL ::= {
    PRESENCE
        DETERMINED BY asn1-field
        USING reference}
```

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

D.1.12 An encoding object for a choice type

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

D.1.12.2 The ASN.1 assignment is:

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

D.1.12.3 The ECN assignments (see 23.1.1 and 23.14.1) are:

```
choiceEncoding #Choice ::= {
    ENCODE STRUCTURE {
        boolean [tagEncoding] USE-SET,
        integer [tagEncoding] USE-SET,
        string [tagEncoding] USE-SET
        STRUCTURED WITH {
            ALTERNATIVE
                DETERMINED BY handle
                HANDLE "Tag"}}
    WITH PER-BASIC-UNALIGNED}

tagEncoding #TAG ::= {
    ENCODING-SPACE {
        SIZE 3
        MULTIPLE OF bit
        EXHIBITS HANDLE "Tag" AT {0 | 1 | 2}}
```

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

```
MyEncodings #ENCODINGS ::= { tagEncoding } COMPLETED BY PER-BASIC-UNALIGNED

choiceEncoding #Choice ::= {
    ENCODE STRUCTURE {
        STRUCTURED WITH {
            ALTERNATIVE
                DETERMINED BY handle
                HANDLE "Tag"}}
    WITH MyEncodings}
```


D.1.13 Encoding a bitstring containing another encoding

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

D.1.13.2 The ASN.1 assignments are:

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

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

D.1.13.3 The ECN assignments (see 25.2) are:

```
sequence2Encoding #Sequence2 ::= {
    ENCODE STRUCTURE {
        b      containerEncoding }
    WITH PER-BASIC-UNALIGNED}

containerEncoding #OUTER ::= {
    PADDING
    MULTIPLE OF octet}
```

D.1.14 An encoding object set

This encoding object set contains encoding definitions for some types specified in the ASN.1 module of D.1.16.

```
Example1Encodings #ENCODINGS ::= {
    marriedEncoding-1
    evenPositiveIntegerEncoding
    evenNegativeIntegerEncoding
    integerRightAlignedEncoding
    integerWithHoleEncoding
    positiveIntegerEncoding
    negativeIntegerEncoding
    positiveIntegerBCDEncoding
    bitStringEncoding
    octetStringEncoding
    characterStringEncoding
    characterStringToBitEncoding
    sequence1Encoding
    choiceEncoding
    sequence2Encoding}
```

D.1.15 ELM definitions

The following ELM encodes the ASN.1 module defined in D.1.16, using objects specified in the EDM defined in D.1.17.

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

IMPORTS Example1Encodings FROM Example-EDM
    {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(3)}
#Married, #EvenPositiveInteger, #EvenNegativeInteger, #IntegerRightAligned,
#IntegerWithHole, #PositiveInteger, #NegativeInteger, #PositiveIntegerBCD,
#BitString, #OctetString, #CharacterString, #CharacterStringToBit, #Sequence1,
#Choice, #Sequence2
FROM Example1-ASN1-Module
    {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(2)};
ENCODE #Married,
    #EvenPositiveInteger,
    #EvenNegativeInteger,
    #IntegerRightAligned,
    #IntegerWithHole, .#PositiveInteger,
    #NegativeInteger, .#PositiveIntegerBCD,
    #BitString, .#OctetString,
```



```

        #CharacterString,
        #CharacterStringToBit,
        #Sequence1, #Choice,
        #Sequence2
WITH Example1Encodings
COMPLETED BY PER-BASIC-UNALIGNED
END

```

D.1.16 ASN.1 definitions

D.1.16.1 This ASN.1 module groups all the ASN.1 definitions from D.1.1 to D.1.12.4 together. They will be encoded according to the encoding objects defined in the EDM of D.1.17, together with the PER basic unaligned encoding rules.

```

Example1-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(2)}
DEFINITIONS AUTOMATIC TAGS ::=
BEGIN

    Married ::= BOOLEAN

    -- etc.

END

```

D.1.17 EDM definitions

D.1.17.1 This EDM module groups all the ECN definitions from D.1.1 to D.1.12.4 together.

```

Example1-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(3)}
ENCODING-DEFINITIONS ::=
BEGIN

    EXPORTS Example1Encodings;
    IMPORTS #Married, #EvenPositiveInteger, #EvenNegativeInteger, #IntegerRightAligned,
    #IntegerWithHole, #PositiveInteger, #NegativeInteger, #PositiveIntegerBCD, #BitString,
    #OctetString, #CharacterString, #CharacterStringToBit, #Sequence1, #Choice, #Sequence2
    FROM Example1-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(2)
    };

    Example1Encodings #ENCODINGS ::= {
        marriedEncoding-1 |
        -- etc
        sequence2Encoding}

    -- etc

END

```

D.2 Specialization examples

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

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

The examples are presented using the following format:

- the ASN.1 assignment, which shows the original ASN.1 type definition;
- the requirement, which lists the requirement and the problem with the encoding provided by PER basic unaligned;
- the encoding object assignment, which shows the resulting encoding specification;
- the encoding structure assignment, if any;
- discussion on how the specialization has been achieved, and other options that might be used.

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

D.2.1.1 The ASN.1 assignment is:

```
NormallySmallValues ::= INTEGER (0..1 024)
-- Usually values are in the range 0..63, but sometimes the whole value range is used.
```

D.2.1.2 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 in E.1.

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

```
normallySmallValuesEncoding-1 #NormallySmallValues ::= {
  USE      #NormallySmallValuesStruct
  MAPPING DISTRIBUTION {
    0..63      TO small,
    REMAINDER  TO large }
  WITH PER-BASIC-UNALIGNED}
```

D.2.1.4 The encoding structure assignment is:

```
#NormallySmallValuesStruct ::= #CHOICE {
  small  #INT (0..63),
  large  #INT (64..1 024)}
```

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

D.2.1.6 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 the field "large" (whose lower bound would then be zero), or by transforming values of the field "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.

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

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

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

```
normallySmallValuesEncoding-2 #NormallySmallValues ::= {
    USE      #NormallySmallValuesStruct2
    MAPPING  ORDERED VALUES
    WITH     normallySmallValuesTag-encoding}

normallySmallValuesTag-encoding #TAG ::= {
    SIZE 1}
```

D.2.2.3 The encoding structure assignment is:

```
#NormallySmallValuesStruct2 ::= #CHOICE {
    small  [#TAG(0)]  #INT (0..63),
    large  [#TAG(1)]  #INT }
```

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

D.2.3 Compression of non-continuous value ranges

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

D.2.3.2 The ASN.1 assignment is:

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

D.2.3.3 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.

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

```
sparseEvenlyDistributedValueSetEncoding #SparseEvenlyDistributedValueSet ::= {
    USE      #INT (0..7)
    MAPPING  ORDERED VALUES
    WITH     PER-BASIC-UNALIGNED}
```

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

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

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

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

```
sparseEvenlyDistributedValueSetEncoding-2 #SparseEvenlyDistributedValueSet ::= {
    USE      #INT (0..7)
    MAPPING  TRANSFORMS {{INT-TO-INT divide: 2}, {INT-TO-INT decrement:1}}
    WITH     PER-BASIC-UNALIGNED}
```

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

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

D.2.5.1 The ASN.1 assignment is:

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

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

D.2.5.3 The encoding object assignment is:

```
sparseUnevenlyDistributedValueSetEncoding #SparseUnevenlyDistributedValueSet ::= {
  USE      #INT (0..5)
  MAPPING ORDERED VALUES
  WITH PER-BASIC-UNALIGNED}
```

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

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

D.2.6.1 The ASN.1 assignment is:

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

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

D.2.6.3 The encoding object assignment is:

```
conditionalPresenceOnValueEncoding #ConditionalPresenceOnValue ::= {
  ENCODE STRUCTURE {
    c  USE-SET OPTIONAL-ENCODING is-c-present{< a >},
    d  USE-SET OPTIONAL-ENCODING is-d-present{< a >}}
  WITH PER-BASIC-UNALIGNED}

is-c-present {< REFERENCE : a >} #OPTIONAL ::= {
  PRESENCE
    DETERMINED BY asn1-field
    USING a
    DECODER-TRANSFORMS {{INT-TO-BOOL TRUE-IS {0}}}}

is-d-present {< REFERENCE : a >} #OPTIONAL ::= {
  PRESENCE
    DETERMINED BY asn1-field
    USING a
    DECODER-TRANSFORMS {{INT-TO-BOOL TRUE-IS {0 | 2 | 3 | 4}}}}
```

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

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

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

D.2.7.1 The ASN.1 assignment is:

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

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

D.2.7.3 The encoding object assignment is:

```
conditionalPresenceOnExternalConditionEncoding #ConditionalPresenceOnExternalCondition ::= {
    ENCODE STRUCTURE {
        a    USE-SET OPTIONAL-ENCODING is-a-present}
    WITH PER-BASIC-UNALIGNED}

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

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

D.2.8 A variable length list

D.2.8.1 The ASN.1 assignment is:

```
EnclosingStructureForList ::= SEQUENCE {
    list    VariableLengthList}

VariableLengthList ::= SEQUENCE (SIZE (0..1 023) ) OF INTEGER (1..2)
    -- Normally the list contains only a few elements (0..31), but it might contain many.
```

D.2.8.2 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.

D.2.8.3 The encoding object assignment is:

```
enclosingStructureForListEncoding #EnclosingStructureForList ::= {
    USE #EnclosingStructureForListStruct
    MAPPING FIELDS
    WITH {
        ENCODE STRUCTURE {
            aux-length list-lengthEncoding,
            list {
                ENCODING {
                    REPETITION -SPACE
```



```

        DETERMINED BY added-field
        USING aux-length}}}}
        WITH PER-BASIC-UNALIGNED}
    -- First mapping: use of an encoding structure with an
    -- explicit length determinant.

list-lengthEncoding #AuxVariableListLength ::= {
    USE #AuxVariableListLengthStruct      -- See D.2.8.4.
    MAPPING ORDERED VALUES
    WITH PER-BASIC-UNALIGNED}
    -- Second mapping: list length is encoded as a choice between a short form "normally"
and
    -- a long form "sometimes".

```

D.2.8.4 The encoding structure assignments are:

```

#EnclosingStructureForListStruct ::= #CONCATENATION {
    aux-length      #AuxVariableListLength,
    list            #VariableLengthList}

#AuxVariableListLength ::= #INT (0..1 023)

#AuxVariableListLengthStruct ::= #ALTERNATIVES {
    normally        #INT (0..31),
    sometimes       #INT (32..1 023)}

```

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

D.2.9 Equal length lists

D.2.9.1 The ASN.1 assignment is:

```

EqualLengthLists ::= SEQUENCE {
    list1 List1,
    list2 List2}
(CONSTRAINED BY {
    -- "list1" and "list2" always have the same number of elements. --
})
List1 ::= SEQUENCE (SIZE (0..1 023)) OF BOOLEAN

List2 ::= SEQUENCE (SIZE (0..1 023)) OF INTEGER (1..2)

```

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

D.2.9.3 The encoding object 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 {
        REPETITION-SPACE
        SIZE variable-with-determinant
        MULTIPLE OF repetitions
        DETERMINED BY added-field
        USING length-determinant}}

```

D.2.9.4 The encoding structure assignments are:

```

#EqualLengthListsStruct ::= #CONCATENATION {
    aux-length      #AuxListLength,
    list1           #List1,
    list2           #List2}

#AuxListLength ::= #INT (0..1 023)

```


D.2.10 Uneven choice alternative probabilities

D.2.10.1 The ASN.1 assignment is:

```
EnclosingStructureForChoice ::= SEQUENCE {
    choice      UnevenChoiceProbability }

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

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

D.2.10.3 The encoding structure assignments are:

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

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

D.2.10.4 The encoding object assignments are:

```
enclosingStructureForChoiceEncoding #EnclosingStructureForChoice ::= {
    USE #EnclosingStructureForChoiceStruct
    MAPPING FIELDS
    WITH {
        ENCODE STRUCTURE {
            aux-selector      auxSelectorEncoding,
            choice {
                ALTERNATIVE
                DETERMINED BY added-field
                USING aux-selector}}
        WITH PER-BASIC-UNALIGNED} }
-- First mapping: inserts an explicit auxiliary alternative determinant.
-- This encoding object specifies that an auxiliary determinant is used
-- as an alternative determinant.

auxSelectorEncoding #AuxSelector ::= {
    USE #BITS
    -- ECN Huffman
    -- RANGE (0..7)
    -- (0..1) IS 60%
    -- (2..4) IS 30%
    -- (5..7) IS 10%
    -- End Definition
    -- Mappings produced by "ECN Public Domain Software for Huffman encodings, version 1"
    -- (see E.8)
    MAPPING TO BITS {
        0 .. 1 TO '10'B .. '11'B,
        2 .. 4 TO '001'B .. '011'B,
        5 TO '0001'B,
        6 .. 7 TO '00000'B .. '00001'B}
    -- Second mapping: Map determinant indexes to bitstrings
```

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

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

D.2.11 A version 1 message

D.2.11.1 ASN.1 assignment:

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

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.

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

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

D.2.11.3 The encoding object assignments are:

```
version1MessageEncoding #Version1Message ::= {
    ENCODE-DECODE
    {ENCODE WITH PER-BASIC-UNALIGNED }
    DECODE AS IF decodingSpecification}

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

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

D.2.12 The encoding object set

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

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


D.2.13 ELM definitions

The following ELM is associated with the ASN.1 module defined in D.2.14, and the EDM defined in D.2.15.

```
Example2-ELM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) elm-module(4)}
LINK-DEFINITIONS ::=
BEGIN
IMPORTS Example2Encodings FROM Example2-EDM
{joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(6)}
#NormallySmallValues, #SparseEvenlyDistributedValueSet,
#SparseUnevenlyDistributedValueSet, #ConditionalPresenceOnValue,
#ConditionalPresenceOnExternalCondition, #EnclosingStructureForList,
#EqualLengthLists, #EnclosingStructureForChoice, #Version1Message
FROM Example2-ASN1-Module
{joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(5)};
ENCODE NormallySmallValues, SparseEvenlyDistributedValueSet,
SparseUnevenlyDistributedValueSet, ConditionalPresenceOnValue,
ConditionalPresenceOnExternalCondition, EnclosingStructureForList,
EqualLengthLists, EnclosingStructureForChoice, Version1Message
WITH Example2Encodings
COMPLETED BY PER-BASIC-UNALIGNED

END
```

D.2.14 ASN.1 definitions

This module groups together all the ASN.1 definitions from D.1.1 to D.2.11 that will be encoded according to the encoding objects defined in the EDM, and also lists the other ASN.1 definitions that will be encoded with the PER basic unaligned encoding rules.

```
Example2-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(5)}
DEFINITIONS AUTOMATIC TAGS ::=
BEGIN
NormallySmallValues ::= INTEGER (0..1024)

-- etc

END
```

D.2.15 EDM definitions

```
Example2-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(6)}
ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS Example2Encodings;
IMPORTS #NormallySmallValues, #SparseEvenlyDistributedValue,
#SparseUnevenlyDistributedValueSet, #ConditionalPresenceOnValueSet,
#ConditionalPresenceOnExternalCondition,
#EnclosingStructureForList, #EqualLengthLists, #EnclosingStructureForChoice,
#Version1Message
FROM Example2-ASN1-Module
{joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(5)};

Example2Encodings #ENCODINGS ::= {
NormallySmallValuesEncoding |
-- etc
extensibleMessageEncoding}

-- etc

END
```


D.3 Explicitly generated structure examples

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

The examples are presented using the following format:

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

D.3.1 Sequence with optional components defined by a pointer

D.3.1.1 The ASN.1 assignment is:

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

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

D.3.1.3 The encoding class #INTEGER is replaced with "#Integer-with-pointer" in the encoding object of "sequence1-encoding".

D.3.1.4 Then an encoding object of class "#Integer-with-pointer" is defined; that encoding object specifies the alignment on an octet boundary and the pointer (see 22.1 and 22.3).

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

```
sequence1-encoding #SEQUENCE ::= {
    REPLACE OPTIONALS
    WITH #Integer-with-pointer
    ENCODED BY integer-with-pointer-encoding
    INSERTED AT HEAD #Pointer}

#Pointer ::= #INTEGER

#Integer-with-pointer {< #Element >} ::= #Element

integer-with-pointer-encoding {< #Element, REFERENCE:pointer >}
    #Integer-with-pointer{< #Element >} ::= {
    ENCODING {
        ALIGNED TO ANY octet
        START-POINTER pointer}}
```

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

D.3.2.1 The ASN.1 assignment is:

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

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

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

The encoding class #OPTIONAL is renamed as #Sequence2-optional in the "RENAMES" clause (see D.3.8). Therefore the "#Sequence2" class is implicitly replaced with:

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

where:

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

Then an encoding object of class "#Sequence2-optional" is defined; that object, using the replacement group, replaces the component encoding definition (see 23.10.3.2) with the class "Optional-with-determinant".

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

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

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

where:

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

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

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

where:

```
Sequence2-combined-encoding-object-set #ENCODINGS ::= PER-BASIC-UNALIGNED
```

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

if-component-present-encoding {<REFERENCE:presence-bit>} #Presence-determinant ::= {

```
    PRESENCE
    DETERMINED BY presence-bit}

determinant-encoding #BOOLEAN ::= {
    TRUE-PATTERN bits:'0'B}
```

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

D.3.3.1 The ASN.1 assignments are:

```
Octets3 ::= OCTET STRING(CONTAINING Sequence3)

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


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

D.3.3.3 The encoding classes #OCTETS, #OPTIONAL and #TAG are renamed respectively as #Octets3, #Sequence3-optional and #TAG-4-bits in the "RENAMES" clause (see D.3.8). Then encoding objects of the new encoding classes are defined.

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

```
#Octets3 ::= #OCTETS

octets3-encoding #Octets3 ::= {
    REPLACE STRUCTURE
    WITH #Octets-with-length
    ENCODED BY octets-with-length-encoding}

#Octets-with-length{< #Element >} ::= CONCATENATION {
    length #INT(0..2047),
    octets #Element}

octets-with-length-encoding{< #Element >} #Octets-with-length{< #Element >} ::= {
    ENCODE STRUCTURE {
        octets octets-encoding{< length >}}
    WITH PER-BASIC-UNALIGNED}

octets-encoding{< REFERENCE:length >} #OCTETS ::= {
    REPETITION-SPACE
    DETERMINED BY added-field
    USING length}
```

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

```
#Sequence3-optional ::= #OPTIONAL

sequence3-optional-encoding #Sequence3-optional ::= {
    PRESENCE
    DETERMINED BY container
    USING OUTER}

#TAG-4-bits ::= #TAG

tag-4-bits-encoding #TAG-4-bits ::= {
    ENCODING-SPACE
    SIZE(4)}
```

D.3.4 Sequence-of type with a count

D.3.4.1 The ASN.1 assignment is:

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

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

D.3.4.3 The encoding class #SEQUENCE-OF is renamed as #SequenceOf in the "RENAMES" clause (see D.3.8). An encoding object of the new encoding class is defined. The encoding class and encoding object assignments are:

```
#SequenceOf ::= #REPETITION
sequenceOf-encoding #SequenceOf {
    ENCODING
    REPLACE STRUCTURE
    WITH #SequenceOf-with-count
    ENCODED BY sequenceOf-with-count-encoding}}

#SequenceOf-with-count{< #Element >} ::= #CONCATENATION {
    count #INT(0..63),
    elements #Element }
```



```
sequenceOf-with-count-encoding{< #Element >} #Sequence-with-count{< #Element >} ::= {
  ENCODE STRUCTURE {
    elements elements-encoding{< count >}}
  WITH PER-BASIC-UNALIGNED}

elements-encoding{< REFERENCE:count >} #REPETITION {
  ENCODING
    REPETITION-SPACE
      DETERMINED BY added-field,
      USING count}
```

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

D.3.5 Encoding object set

The encoding object set contains encoding objects of classes defined in the EDM module.

```
Example3Encodings #ENCODINGS ::= {
  sequence1-encoding
  sequence2-optional-encoding
  octets-encoding
  sequence3-optional-encoding
  tag-4-bits-encoding
  sequenceOf-encoding }
```

D.3.6 ELM definitions

The following ELM is associated with the ASN.1 module defined in D.3.7 and the EDM defined in D.3.8.

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

  IMPORTS Example3Encodings, Sequence2-combined-encoding-object-set
  FROM Example3-EDM { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-modules(10) }
  #Sequence1, #Sequence2, #Octet3, #Sequence3, #SequenceOfIntegers
  FROM Example3-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4)
  examples(5) asn1-modules(9);

  ENCODE #Sequence1, #Sequence2, #Octet3, #Sequence3, #SequenceOfIntegers
  WITH Example3Encodings
  COMPLETED BY PER-BASIC-UNALIGNED

  END
```

D.3.7 ASN.1 definitions

This module groups together the ASN.1 definitions from D.3.1 to D.3.4 that will be encoded according to the encoding objects defined in the EDM of D.3.8.

```
Example3-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(9)}
DEFINITIONS
  AUTOMATIC TAGS ::=
  BEGIN

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

    -- etc

  END
```


D.3.8 EDM definitions

```

Example3-EDM { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(10) }
ENCODING-DEFINITIONS ::=
BEGIN

EXPORTS Example3Encodings;
RENAMES
    #OPTIONAL AS #Sequence2-optional
        IN #Sequence2
    #OCTET-STRING AS #Octets3
        IN ALL
    #OPTIONAL AS #Sequence3-optional
        IN #Sequence3
    #TAG AS #TAG-4-bits
        IN #Sequence3
FROM Example3-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(9) };

IMPORTS #Sequence1, #Sequence2, #Sequence3, #SequenceOfIntegers
FROM Example3-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(9) };

Example3Encodings #ENCODINGS ::= {
    sequence1-optional-encoding |
    -- etc
    sequenceOf-encoding }
-- etc
END

```

D.4 Legacy protocol example

D.4.1 Introduction

D.4.1.1 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 3	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 2...	Something – 1							

Message 3:

	8	7	6	5	4	3	2	1
Octet 1	Message id							
Octet 2...	Something – 2							

D.4.1.2 All the messages have a common heading part (shown in gray in the tables). In this example it is used only for message identification.

D.4.1.3 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").

D.4.1.4 The fields "b", "c" and "d" are all required to start on an octet boundary.

D.4.1.5 The fields "b", "c" and "d" are composed of sub-fields ("b1", "b2", "b3", "c1", etc.) of fixed length. In addition fields "c" and "d" may appear multiple times (but only one occurrence is shown above). The field "b2" is required to start on a nibble boundary.

D.4.1.6 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 "d" is present if there are octets left in the message.

D.4.1.7 The length of a field that can appear multiple times is determined using different methods:

- the number of repetitions of the field "c" is governed by the determinant field "c-len";
- the number of repetitions of the field "d" is determined by the end of message.

D.4.1.8 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.

D.4.1.9 The ASN.1 module is:

```
LegacyProtocol-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module(11)}
  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,                -- 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 treat 5 to 7 as 4.

    B ::= SEQUENCE {
      b1      ENUMERATED { e0, e1, e2, e3 },
      b2      BOOLEAN,
      b3      INTEGER (0..3) }
```



```

C ::= SEQUENCE (SIZE (0..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 1 -- }

Message3 ::= SEQUENCE {
    -- something 2 -- }

END

```

D.4.1.10 The EDM module in D.4.8 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.

D.4.2 Encoding definition for the top-level message structure

D.4.2.1 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".

D.4.2.2 The encoding object "legacyProtocolMessagesEncoding" is:

```

legacyProtocolMessagesEncoding #LegacyProtocolMessages ::= {
    ENCODE STRUCTURE {
        message-id {
            ENCODING {
                ENCODING SPACE
                SIZE 8}},
        messages USE-SET OPTIONAL-ENCODING {
            ALTERNATIVE
                DETERMINED BY asn1-field
                USING message-id}}
    WITH PER-BASIC-UNALIGNED}

```

D.4.3 Encoding definition for a message structure

D.4.3.1 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 "TRUE";
- 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.

D.4.3.2 The encoding object for "Message1" is:

```
message1Encoding #Message1 ::= {
  ENCODE STRUCTURE {
    b  b-encoding
      OPTIONAL-ENCODING {
        PRESENCE
          DETERMINED BY asn1-field
          USING b-flag},
    c  octet-aligned-seq-of-with-ext-determinant{< c-len >},
    d  octet-aligned-seq-of-until-end-of-container
  }
  WITH PER-BASIC-UNALIGNED}
```

D.4.4 Encoding for the sequence type "B"

D.4.4.1 Padding of one bit is inserted between the fields "b2" and "b3" ("aux-reserved"). The encoding of "B" is octet-aligned.

D.4.4.2 The encoding for "B" is:

```
b-encoding #B ::= {
  ENCODE STRUCTURE {
    -- Components
    b3 {
      ALIGNED TO NEXT nibble}
    -- Structure
    STRUCTURED WITH {
      ALIGNED TO NEXT octet }
    -- The rest
  }
  WITH PER-BASIC-UNALIGNED}
```

D.4.5 Encoding for an octet-aligned sequence-of type with a length determinant

D.4.5.1 One of the sequence-of types used in the legacy protocol has an explicit length determinant.

D.4.5.2 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 >} # REPETITION ::= {
  ENCODING {
    ALIGNED TO NEXT octet
    REPETITION-SPACE
    DETERMINED BY asn1-field
    USING len}}
```

D.4.6 Encoding for an octet-aligned sequence-of type which continues to the end of the PDU

D.4.6.1 The encoding is octet-aligned. The number of elements is determined by the end of the PDU.

D.4.6.2 The encoding object is:

```
octet-aligned-seq-of-until-end-of-container # REPETITION ::= {
  ENCODING {
    REPETITION-SPACE
    ALIGNED TO NEXT octet
    DETERMINED BY container
    USING OUTER}}
```


D.4.7 ELM definitions

The ELM for the legacy protocol is:

```
LegacyProtocol-ELM-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) elm-module(12) }
LINK-DEFINITIONS ::=
BEGIN
IMPORTS LegacyProtocolEncodings
FROM LegacyProtocol-EDM-Module
{ joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module(13) }

#LegacyProtocolMessages
FROM LegacyProtocol-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4)
examples(5) asn1-module(11) };
ENCODE LegacyProtocolMessages
WITH LegacyProtocolEncodings
COMPLETED BY PER-BASIC-UNALIGNED
END
```

D.4.8 EDM definitions

The EDM definitions are:

```
LegacyProtocol-EDM-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) module(13) }
ENCODING-DEFINITIONS ::=
BEGIN
EXPORTS LegacyProtocolEncodings;

IMPORTS #LegacyProtocolMessages
FROM LegacyProtocol-ASN1-Module
{ joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) module(11) };

LegacyProtocolEncodings #ENCODINGS ::= {
    legacyProtocolMessagesEncoding |
    message1Encoding }

-- etc

END
```

Annex E (informative): Support for Huffman encodings

E.1 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.

E.2 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.

E.3 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.

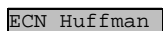
E.4 For Huffman encodings to be efficient for decoders, it is desirable that where successive integer values encode into the same number of bits, those bits should define successive integer values when interpreted as a positive integer encoding.

E.5 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 (see 19.7) which is both optimal and easy to decode.

E.6 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 F).

E.7 The following text contains three examples of ECN Huffman specification.

E.8 In the version with the macro, double clicking the button below:

The image shows a rectangular button with the text "ECN Huffman" inside. The button has a thin border and a slightly shaded background, typical of a graphical user interface element from the late 1990s.

will add the ECN Huffman mapping specifications to the text.

E.9 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.

E.10 The informal syntax for an ECN Huffman specification should be clear from the following examples. All lines start with an ASN.1 comment marker ("--").

E.11 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.

E.12 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 1 000, but can include negative values. Not all values in the range need to be mapped.

E.13 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.

E.14 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.

E.15 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.

E.15.1 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 }

```

E.15.2 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 }

```

E.15.3 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 '00000000000000000000'B .. '00000001101010001'B
  }
  WITH my-self-delim-bits-encoding }

```

Annex F (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 G (informative): Summary of the ECN notation

G.1 Terminal symbols

The following terminal symbols are used in the present document.

G.1.1 The following items are defined in clause 8:

anystringexceptuserfunctionend
encodingobjectreference
encodingobjectsetreference
encodingclassreference
"::="

"::"

".."

"{"

"}"

"("

")"

" "

"."

"|"

ALL

AS

BEGIN

BER

BITS

BY

CER

COMPLETED

DECODE

DER

DISTRIBUTION

ENCODE

ENCODE-DECODE

ENCODING-CLASS

ENCODING-DEFINITIONS

END

EXCEPT

EXPORTS

FALSE

FIELDS

FROM

GENERATES

IF

IMPORTS

IN

LINK-DEFINITIONS

MAPPING

MAX

MIN

MINUS-INFINITY

NON-ECN-BEGIN

NON-ECN-END

NULL

OPTIONAL-ENCODING

ORDERED

OUTER

PER-BASIC-ALIGNED

PER-BASIC-UNALIGNED

PER-CANONICAL-ALIGNED

PER-CANONICAL-UNALIGNED

PLUS-INFINITY

REFERENCE

REMAINDER

RENAMES

SIZE

STRUCTURE

STRUCTURED

TO

TRANSFORMS

TRUE

UNION

USE

USE-SET

VALUES

WITH

G.1.2 The following item is defined in annex A:

REFERENCE

G.1.3 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

G.1.4 The following items are defined in ITU-T Rec. X.681 | ISO/IEC 8824-2:

word
valuefieldreference
valuesetfieldreference

G.1.5 The following items are defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

"{<"
">}"

G.2 Productions

G.2.1 The following productions are used in the present document, with the items defined in G.1 as terminal symbols:

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

ELModuleBody ::=
  Imports ?
  EncodingApplicationList

EncodingApplicationList ::=
  EncodingApplication
  EncodingApplicationList ?

EncodingApplication ::=
  ENCODE
  SimpleDefinedEncodingClass " , " +
  CombinedEncodings
```



```

CombinedEncodings ::=
    WITH
    PrimaryEncodings
    CompletionClause ?

CompletionClause ::=
    COMPLETED BY
    SecondaryEncodings

PrimaryEncodings ::= EncodingObjectSet

SecondaryEncodings ::= EncodingObjectSet

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

EDMModuleBody ::=
    Exports ?
    RenamesAndExports ?
    Imports ?
    EDMAssignmentList ?

EDMAssignmentList ::=
    EDMAssignment
    EDMAssignmentList ?

EDMAssignment ::=
    EncodingClassAssignment
    EncodingObjectAssignment
    EncodingObjectSetAssignment
    ParameterizedAssignment

RenamesAndExports ::=
    RENAMES
    ExplicitGenerationList ";"

ExplicitGenerationList ::=
    ExplicitGeneration
    ExplicitGenerationList ?

ExplicitGeneration ::=
    OptionalNameChanges
    FROM GlobalModuleReference

OptionalNameChanges ::=
    NameChanges | GENERATES

NameChanges ::=
    OriginalClassName
    AS
    NewClassName
    IN
    NameChangeDomain

OriginalClassName ::= SimpleDefinedEncodingClass | BuiltinEncodingClassReference

NewClassName ::= encodingclassreference

NameChangeDomain ::=
    IncludedRegions
    Exception ?

Exception ::=
    EXCEPT
    ExcludedRegions

IncludedRegions ::=
    ALL | RegionList

ExcludedRegions ::= RegionList

RegionList ::=
    Region "," +

```



```

Region ::=
    SimpleDefinedEncodingClass      |
    ComponentReference

ComponentReference ::=
    SimpleDefinedEncodingClass
    "."
    identifier

EncodingClassAssignment ::=
    encodingclassreference
    " : : ="
    EncodingClass

EncodingClass ::=
    BuiltinEncodingClassReference  |
    EncodingStructure

EncodingObjectAssignment ::=
    encodingobjectreference
    DefinedOrBuiltinEncodingClass
    " : : ="
    EncodingObject

EncodingObjectSetAssignment ::=
    encodingobjectsetreference
    #ENCODINGS
    " : : ="
    EncodingObjectSet
    CompletionClause ?

EncodingObjectSet ::=
    DefinedEncodingObjectSet      |
    EncodingObjectSetSpec

EncodingStructure ::=
    TaggedStructure               |
    UntaggedStructure

TaggedStructure ::=
    "["
    TagClass
    TagValue ?
    "]"
    EncodingStructure

UntaggedStructure ::=
    DefinedEncodingClass          |
    EncodingStructureField        |
    EncodingStructureDefn

TagClass ::=
    DefinedEncodingClass          |
    TagClassReference

TagValue ::=
    "(" number ")"

EncodingStructureDefn ::=
    AlternativesStructure         |
    RepetitionStructure           |
    ConcatenationStructure

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

AlternativesClass ::=
    DefinedEncodingClass          |
    AlternativesClassReference

```



```

NamedFields ::= NamedField "," +

NamedField ::=
    identifier
    EncodingStructure

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

RepetitionClass ::=
    DefinedEncodingClass |
    RepetitionClassReference

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

ConcatenationClass ::=
    DefinedEncodingClass |
    ConcatenationClassReference

ConcatComponents ::=
    ConcatComponent "," *

ConcatComponent ::=
    NamedField
    ConcatComponentPresence ?

ConcatComponentPresence ::=
    OPTIONAL-ENCODING
    OptionalClass

OptionalClass ::=
    DefinedEncodingClass |
    OptionalityClassReference

DefinedEncodingClass ::=
    Encodingclassreference |
    ExternalEncodingClassReference |
    ParameterizedEncodingClass

DefinedOrBuiltinEncodingClass ::=
    DefinedEncodingClass |
    BuiltinEncodingClassReference

DefinedEncodingObject ::=
    Encodingobjectreference |
    ExternalEncodingObjectReference |
    ParameterizedEncodingObject

DefinedEncodingObjectSet ::=
    Encodingobjectsetreference |
    ExternalEncodingObjectSetReference |
    ParameterizedEncodingObjectSet

DefinedOrBuiltinEncodingObjectSet ::=
    DefinedEncodingObjectSet |
    BuiltinEncodingObjectSetReference

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

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

EncodingObjects ::=
    DefinedEncodingObject |
    DefinedEncodingObjectSet

UnionMark ::=
    "| " |
    UNION

EncodingObject ::=
    DefinedEncodingObject |
    DefinedSyntax |
    EncodeWith |
    EncodeByValueMapping |
    EncodeStructure |
    DifferentialEncodeDecodeObject |
    NonECNEncodingObject

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

EncodeByValueMapping ::=
    "{ "
    USE
    DefinedOrBuiltinEncodingClass
    MAPPING
    ValueMapping
    WITH
    ValueMappingEncodingObjects
    }"

ValueMappingEncodingObjects ::=
    EncodingObject |
    DefinedOrBuiltinEncodingObjectSet

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

SpecForEncoding ::= EncodingObject

SpecForDecoders ::= EncodingObject

NonECNEncodingObject ::=
    NON-ECN-BEGIN
    AssignedIdentifier
    anystringexceptnonecnend
    NON-ECN-END

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

StructureEncoding ::=
    STRUCTURED WITH
    EncodingObject

```



```

ComponentEncodingList ::=
    ComponentEncoding "," *

ComponentEncoding ::=
    NonOptionalComponentEncodingSpec |
    OptionalComponentEncodingSpec

NonOptionalComponentEncodingSpec ::=
    identifier ?
    TagAndElementEncoding

OptionalComponentEncodingSpec ::=
    identifier
    TagAndElementEncoding
    OPTIONAL-ENCODING
    OptionalEncoding

TagAndElementEncoding ::=
    EncodingOrUseSet |
    TagEncoding TagAndElementEncoding

TagEncoding ::= "[" EncodingOrUseSet "]"

OptionalEncoding ::= EncodingOrUseSet

EncodingOrUseSet ::=
    EncodingObject |
    USE-SET

BuiltinEncodingClassReference ::=
    BitfieldClassReference
    AlternativesClassReference
    ConcatenationClassReference
    RepetitionClassReference
    OptionalityClassReference
    TagClassReference
    EncodingProcedureClassReference

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
    #GraphicString
    #IA5String
    #NumericString
    #PrintableString
    #TeletexString
    #UniversalString
    #UTF8String
    #VideotexString
    #VisibleString

AlternativesClassReference ::=
    #ALTERNATIVES
    #CHOICE

```



```

ConcatenationClassReference ::=
    #CONCATENATION
    #SEQUENCE
    #SET

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

OptionalityClassReference ::=
    #OPTIONAL

TagClassReference ::=
    #TAG

EncodingProcedureClassReference ::=
    #TRANSFORM
    #CONDITIONAL-INT
    #CONDITIONAL-REPETITION
    #OUTER

EncodingStructureField ::=
    #NUL
    #BOOL
    #INT          Bounds?
    #BITS         Size?
    #OCTETS       Size?
    #CHARS        Size?
    #PAD
    #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 ")"

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

```

G.2.2 The following productions are defined ITU-T Rec. X.680 | ISO/IEC 8824-1, as modified by annex A, with the items defined in G.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
    ReferencedValue
    ObjectClassFieldValue
```

```
BuiltinValue ::=
    BitStringValue
    BooleanValue
    CharacterStringValue
    ChoiceValue
    EmbeddedPDVValue
    EnumeratedValue
    ExternalValue
    InstanceOfValue
    IntegerValue
    NullValue
    ObjectIdentifierValue
    OctetStringValue
    RealValue
    SequenceValue
    SequenceOfValue
    SetValue
    SetOfValue
    TaggedValue
```

```
BitStringValue ::=
    bstring
    hstring
    "{" IdentifierList "}"
    "{" "}"
```



```

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
identifier

SignedNumber ::=
    number
    "-" number

NullValue ::= NULL

ObjectIdentifierValue ::=
    "{" ObjIdComponentList "}"
"{" DefinedValue ObjIdComponentList "}"

ObjIdComponentList ::=
    ObjIdComponent
    ObjIdComponent ObjIdComponentList

ObjIdComponent ::=
    NameForm
    NumberForm
    NameAndNumberForm

NameForm ::= identifier

NumberForm ::=
    number
DefinedValue

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 "," "..."~~
~~"..." "," AdditionalElementSetSpec~~
~~RootElementSetSpec "," "...", " AdditionalElementSetSpec~~

RootElementSetSpec ::= ElementSetSpec

ElementSetSpec ::=

Unions
ALL Exclusions

Exclusions ::= EXCEPT Elements

Unions ::=

Intersections
UElems UnionMark Intersections

UElems ::= Unions

Intersections ::=

~~IntersectionElements~~
~~UElems IntersectionMark IntersectionElements~~

IntersectionElements ::= Elements ~~+UElems Exclusions~~

UnionMark ::=

"|"
UNION


```

Elements ::=
  SubtypeElements
  ObjectSetElements
  "(" ElementSetSpec ")"

SubtypeElements ::=
  SingleValue
  ContainedSubtype
  ValueRange
  PermittedAlphabet
  SizeConstraint
  TypeConstraint
  InnerTypeConstraints

SingleValue ::= Value

```

G.2.3 The following productions are defined ITU-T Rec. X.681 | ISO/IEC 8824-2, as modified by annex B, with the items defined in G.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

```

G.2.4 The following productions are defined ITU-T Rec. X.683 | ISO/IEC 8824-4 as modified by annex C, with the items defined in G.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

```

History

Document history		
V1.1.1	May 2001	Publication