International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.692

(02/2021)

SERIES X: DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

OSI networking and system aspects – Abstract Syntax Notation One (ASN.1)

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

Recommendation ITU-T X.692

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

PUBLIC DATA NETWORKS	
Services and facilities	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalling and switching	X.50–X.89
Network aspects	X.90–X.149
Maintenance	X.150–X.179
Administrative arrangements	X.180–X.199
OPEN SYSTEMS INTERCONNECTION	
Model and notation	X.200–X.209
Service definitions	X.210–X.219
Connection-mode protocol specifications	X.220–X.229
Connectionless-mode protocol specifications	X.230–X.239
PICS proformas	X.240–X.259
Protocol Identification	X.260–X.269
Security Protocols	X.270–X.279
Layer Managed Objects	X.280–X.289
Conformance testing	X.290–X.299
INTERWORKING BETWEEN NETWORKS	
General	X.300–X.349
Satellite data transmission systems	X.350–X.369
IP-based networks	X.370–X.379
MESSAGE HANDLING SYSTEMS	X.400–X.499
DIRECTORY	X.500–X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600–X.629
Efficiency	X.630–X.639
Quality of service	X.640–X.649
Naming, Addressing and Registration	X.650–X.679
Abstract Syntax Notation One (ASN.1)	**X.680–X.699**
OSI MANAGEMENT	
Systems management framework and architecture	X.700–X.709
Management communication service and protocol	X.710–X.719
Structure of management information	X.720–X.729
Management functions and ODMA functions	X.730–X.799
SECURITY	X.800–X.849
OSI APPLICATIONS	
Commitment, concurrency and recovery	X.850–X.859
Transaction processing	X.860–X.879
Remote operations	X.880–X.889
Generic applications of ASN.1	X.890–X.899
OPEN DISTRIBUTED PROCESSING	X.900–X.999
INFORMATION AND NETWORK SECURITY	X.1000–X.1099
SECURE APPLICATIONS AND SERVICES (1)	X.1100–X.1199
CYBERSPACE SECURITY	X.1200–X.1299
SECURE APPLICATIONS AND SERVICES (2)	X.1300–X.1499
CYBERSECURITY INFORMATION EXCHANGE	X.1500–X.1599
CLOUD COMPUTING SECURITY	X.1600–X.1699
QUANTUM COMMUNICATION	X.1700–X.1729
DATA SECURITY	X.1750–X.1799
5G SECURITY	X.1800–X.1819

For further details, please refer to the list of ITU-T Recommendations.

INTERNATIONAL STANDARD ISO/IEC 8825-3
RECOMMENDATION ITU-T X.692

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

Summary

Recommendation ITU-T X.692 | ISO/IEC 8825-3 defines the Encoding Control Notation (ECN) used to specify encodings (of ASN.1 types) that differ from those provided by standardized encoding rules such as the Basic Encoding Rules (BER) and the Packed Encoding Rules (PER).

History

Edition	Recommendation	Approval	Study Group	Unique ID[*]
1.0	ITU-T X.692	2002-03-08	17	11.1002/1000/5647
1.1	ITU-T X.692 (2002) Annex E	2002-03-08	17	11.1002/1000/6361
1.2	ITU-T X.692 (2002) Amd. 1	2004-08-29	17	11.1002/1000/7292
1.3	ITU-T X.692 (2002) Technical Cor. 1	2005-05-14	17	11.1002/1000/8514
1.4	ITU-T X.692 (2002) Amd. 2	2006-06-13	17	11.1002/1000/8840
2.0	ITU-T X.692	2008-11-13	17	11.1002/1000/9610
2.1	ITU-T X.692 (2008) Cor. 1	2011-10-14	17	11.1002/1000/11380
3.0	ITU-T X.692	2015-08-13	17	11.1002/1000/12485
4.0	ITU-T X.692	2021-02-13	17	11.1002/1000/14474

[*] To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, http://handle.itu.int/11.1002/1000/11830-en.

CONTENTS

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 (Rec. ITU-T X.690 | ISO/IEC 8825-1 and Rec. ITU-T 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 support only binary-based encodings, but could be extended in the future to cover character-based encodings.

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

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

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

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 clause 5 and 9.28).

Annex D does not form an integral part of this Recommendation | International Standard, and contains examples of the use of ECN.

Annex E does not form an integral part of this Recommendation | International Standard and provides more detail on the support for Huffman encodings in ECN.

Annex F does not form an integral part of this Recommendation | International Standard, and identifies a Web site providing access to further information and links relevant to ECN.

Annex G does not form an integral part of this Recommendation | International Standard, and provides a summary of ECN using the notation of clause 5.

INTERNATIONAL STANDARD
ITU-T RECOMMENDATION

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

1 Scope

This Recommendation | International Standard 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.

ECN does not currently provide any support for specifications using the OID internationalized resource identifier type or the relative OID internationalized resource identifier type (see Rec. ITU-T X.680 | ISO/IEC 8824-1), and these are not referred to further in this Standard.

2 Normative references

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

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

2.1 Identical Recommendations | International Standards

– Recommendation ITU-T X.660 (2011) | ISO/IEC 9834-1:2012, *Information technology – Open Systems Interconnection – Procedures for the operation of OSI Registration Authorities: General procedures and top arcs of the international object identifier tree.*
– Recommendation ITU-T X.680 (2021) | ISO/IEC 8824-1:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*
– Recommendation ITU-T X.681 (2021) | ISO/IEC 8824-2:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*
– Recommendation ITU-T X.682 (2021) | ISO/IEC 8824-3:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
– Recommendation ITU-T X.683 (2021) | ISO/IEC 8824-4:2021, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*
– Recommendation ITU-T X.690 (2021) | ISO/IEC 8825-1:2021, *Information technology – ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).*
– Recommendation ITU-T X.691 (2021) | ISO/IEC 8825-2:2021, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER).*

NOTE – 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 10646:2003*, Information technology – Universal Multiple-Octet Coded Character Set (UCS)*.

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

3 Definitions

For the purposes of this Recommendation | International Standard, the following definitions apply.

3.1 ASN.1 definitions

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

3.2 ECN-specific definitions

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

3.2.2 auxiliary field: A 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 3.2.31) needed for successful decoding, or both.

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

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

3.2.6 choice determinant: A 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: A 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: An encoding which is to be applied only if some specified condition is satisfied.

NOTE – The condition may be a bounds condition or a size range condition, or other more complex conditions.

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

NOTE – The ASN.1 types to which a contents constraint (using `CONTAINING/ENCODED BY`) can be applied are the bitstring and the octetstring types.

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

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

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

3.2.12 encoding class: The 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: Encoding classes with some common characteristics.

NOTE – Examples are the integer category, the boolean category, and the concatenation category.

3.2.14 encoding constructor: An 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): The (unique, for any given application) module that assigns encodings to ASN.1 types.

3.2.17 encoding object: The 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: A set of encoding objects.

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

3.2.19 encoding property: A piece of information used to define an encoding using the notation specified in clauses 23, 24 and 25.

3.2.20 encoding space: The number of bits (or octets, words or other units) used to encode an abstract value into a bit-field (see 9.21.5).

3.2.21 encoding structure: The 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: An 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: A reference to an encoding class, object, or object set that includes either the name of the EDM module in which that encoding class, object, or object set was defined, or (in the case of an implicitly generated encoding class) the name of the ASN.1 module in which it was generated. (See also 3.2.43.)

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.1/13.16).

3.2.25 generated encoding structure: An 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: A 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 handle value set: The specified set of all possible values of the identification handle that is exhibited by an encoding object.

3.2.28 identification handle: Part of an encoding which serves to distinguish the encodings produced by one encoding object (of a given class) from those produced by other encoding objects (of other classes).

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

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

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

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

3.2.32 negative integer value: A value less than zero.

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

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

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

3.2.36 positive integer value: A value greater than zero.

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

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

3.2.39 recursive definition (of a reference name): A reference name for which resolution of the reference name, or of the governor of the definition of the reference name, requires resolution of the original reference name.

NOTE – Recursive definition of an encoding class (including an encoding structure) or an encoding object is permitted (but see 17.1.4). Recursive definition of an encoding object set is forbidden by 18.1.3.

3.2.40 recursive instantiation (of a parameterized reference name): An instantiation of a reference name, where resolution of the actual parameters requires resolution of the original reference name.

NOTE – Recursive instantiation of an encoding class (including an encoding structure) or an encoding object is permitted (but see 17.1.4). Recursive instantiation of an encoding object set is forbidden by 18.1.3.

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

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

3.2.43 simple reference name: A reference to an encoding class, object, or object set that includes neither the name of the EDM module in which that encoding class, object, or object set was defined, nor (in the case of an implicitly generated encoding class) the name of the ASN.1 module in which it was generated.

NOTE – A simple reference name can only be used when the reference to the encoding class is unambiguous, otherwise a fully-qualified name (see 3.2.24) has to be used in the body of a module.

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

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

3.2.46 start pointer: An 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.

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

3.2.48 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.

3.2.49 transforms: Encoding objects of the class **#TRANSFORM** which specify that the encoding of the abstract values associated with some class (or of transform composites – see 3.2.50) is to be the encoding of different abstract values associated with the same or a different class (or of transform composites).

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

3.2.50 transform composites: An ordered list of elements that can itself be the source or the result of transforms.

NOTE – All the elements of a composite are required to have the same classification (see 9.18.2).

3.2.51 value encoding: The way in which an encoding space is used to represent an abstract value (see 9.21.5).

4 Abbreviations

For the purposes of this Recommendation | International Standard, 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 This Recommendation | International Standard employs the notational convention defined in Rec. ITU-T X.680 | ISO/IEC 8824-1, clause 5.

5.2 This Recommendation | International Standard employs the notation for information object classes defined in Rec. ITU-T X.681 | ISO/IEC 8824-2 as modified by Annex B.

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

6 Encoding conventions and notation

6.1 This Recommendation | International Standard 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 this Recommendation | International Standard, 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 this Recommendation | International Standard, 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 this Recommendation | International Standard, 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 this Recommendation | International Standard refers to the characters specified in ISO/IEC 10646, 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.6), 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 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)
`<`	LESS-THAN SIGN
`=`	(EQUALS SIGN)
`>`	GREATER-THAN SIGN
`{`	(LEFT CURLY BRACKET)
`|`	(VERTICAL LINE)
`}`	(RIGHT CURLY BRACKET)

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

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

8 ECN lexical items

In addition to the ASN.1 lexical items specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, clause 12, this Recommendation | International Standard uses lexical items specified in the following subclauses. The general rules specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.1, apply in this clause.

NOTE – Annex G lists all lexical items and all the productions used in this Recommendation | International Standard, identifying those that are defined in Rec. ITU-TX.680 | ISO/IEC 8824-1, Rec. ITU-T X.681 | ISO/IEC 8824-2 and Rec. ITU-T 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.4. In analyzing an instance of use of this notation, an "encodingobjectreference" is distinguished from an "identifier" by the context in which it appears.

8.2 Encoding object set references

Name of item - encodingobjectsetreference

An "encodingobjectsetreference" shall consist of the sequence of characters specified for a "typereference" in Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.2. 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.2. It shall not be one of the character sequences listed in 8.5

except in an EDM imports list (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.20, as modified by A.1) or in an "ExternalEncodingClassReference" (see the Note in 14.11).

8.4 Reserved word items

Names of reserved word items:

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

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

NOTE – The words (see Rec. ITU-T 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.14).

8.5 Reserved encoding class name items

Names of reserved encoding class name items:

#ALTERNATIVES	**#EXTERNAL**	**#PrintableString**
#BITS	**#GeneralizedTime**	**#REAL**
#BIT-STRING	**#GeneralString**	**#RELATIVE-OID**
#BMPString	**#GraphicString**	**#REPETITION**
#BOOL	**#IA5String**	**#SEQUENCE**
#BOOLEAN	**#INT**	**#SEQUENCE-OF**
#CHARACTER-STRING	**#INTEGER**	**#SET**
#CHARS	**#NUL**	**#SET-OF**
#CHOICE	**#NULL**	**#TAG**
#CONCATENATION	**#NumericString**	**#TeletexString**
#CONDITIONAL-INT	**#OBJECT-IDENTIFIER**	**#TIME**
#CONDITIONAL-REPETITION	**#ObjectDescriptor**	**#TIME-OF-DAY**
#DATE	**#OCTETS**	**#TRANSFORM**
#DATE-TIME	**#OCTET-STRING**	**#UniversalString**
#DURATION	**#OPEN-TYPE**	**#UTCTime**
#EMBEDDED-PDV	**#OPTIONAL**	**#UTF8String**
#ENCODINGS	**#OUTER**	**#VideotexString**
#ENUMERATED	**#PAD**	**#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 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 this ITU-T Recommendation | International Standard.

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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.13, as modified by A.1 of this Recommendation | International Standard).

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 group of categories (see 9.6) 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 components 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 groups of categories that contain 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 group of categories. Thus all classes that are in the integer or the boolean or the characterstring category are in the bit-field group of categories. Classes that are responsible for grouping or repeating encodings (for example classes in the alternatives or the repetition category) are in the encoding constructor group of categories. 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 group of categories. Encoding structures are defined using classes in the bit-field group of categories that are combined using classes in the encoding constructor group of categories, 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 group of categories).

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 group of categories. 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:

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

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

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

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

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

– The boolean, bitstring, characterstring, integer, null, objectidentifier, octetstring, opentype, pad, real, and time categories (categories for classes that are derived from the corresponding primitive classes).

– The encoding structure category (classes generated from ASN.1 type definitions, or by explicit definition of an encoding structure).

9.6.7 The following groups of categories are defined:

– The bit-field group of categories (classes that correspond to actual fields in an encoding such as those in the integer or boolean categories, together with any class in the encoding structure category). Classes in this group of categories are also referred to as bit-field classes.

– The encoding constructor group of categories (classes that are in the alternatives, concatenation, or repetition categories). Classes in this group of categories are also referred to as encoding constructor classes.

– The encoding procedure group of categories (classes not directly related to ASN.1 constructs, and which cannot be assigned new names – **#OUTER**, **#TRANSFORM**, **#CONDITIONAL-INT**, **#CONDITIONAL-REPETITION**). Classes in this group of categories are also referred to as encoding procedure classes.

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 9.17).

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 (**`UNIVERSAL`**, **`APPLICATION`**, **`PRIVATE`**, or context-specific) 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 encoding options encoding object, which contains an ordered list of encoding objects of the same class. It is an encoder's choice which encoding object from the list is to be applied, subject to the restriction that if only one encoding option can encode a given abstract value, that shall be used, and to the recommendation that the first available encoding in the list should be used.

NOTE – An encoding options encoding object could, for example, be used in the specification of short-form length encodings where these can encode a particular string length, using long-form length encodings where the short-form cannot be used. There is no current mechanism for the ECN specifier to require the use of the first available encoding object (if more than one can encode the abstract value), other than by comment.

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 9.21.5) 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 an ordered 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 Rec. ITU-T X.691 | ISO/IEC 8825-2, 10.3).

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 (consisting of a fixed set of bit positions) of all the encodings that it produces and distinguishes those encodings from the encodings produced by other encoding objects (of any class) that exhibit the same identification handle. Identification handles have a name and are 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, termination of repetitions, 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 component 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 Rec. ITU-T X.683 | ISO/IEC 8824-4 as modified by B.10 of this Recommendation | International Standard.

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 17.1.7).

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 component 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 components or of alternatives and the ordering of sets. The mechanism is similar in all these cases. Given an encoding class that is a "possible next class" and an encoding object applied to it, any encoding produced will contain, at some bit positions (the identification handle), a bit pattern that matches a bit pattern within a specified set of bit patterns (the handle value set) characterizing that class, but does not match any bit pattern characterizing any other "possible next class". All such encodings can be interpreted by a decoder as an encoding of a "possible next class", and the bit pattern found in the encoding 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 definition (see clause 11), the first **textually-present** ASN.1 tag notation in that definition generates an instance of the **`#TAG`** class, with the number of the ASN.1 tag associated with that instance of the **`#TAG`** class. Subsequent textually present instances of ASN.1 tag notation are not mapped into #TAG classes in the implicitly generated structure, but these tags and their values become properties of the element. 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 fields 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 field** – 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 17.5.15). (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.3.7.)

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 optionality category) that replace the optional component 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 of a **`#CONCATENATION`** (or similar structure). An example of this is given in D.3.1.5.

9.17 Mapping abstract values onto fields of encoding structures

There are six mechanisms provided for this.

9.17.1 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.)

9.17.2 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.)

9.17.3 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 encoding object (see 9.18). 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.)

9.17.4 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.)

9.17.5 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.)

9.17.6 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 Transforms and transform composites

9.18.1 Transforms are encoding objects of the class **`#TRANSFORM`**. They can be used to transform abstract values between different encoding classes, and 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.)

9.18.2 A transform can take a single value as its source and then produces a single value as its result. The following is a classification of the values that can be sources and results of transforms:

- an integer;
- a boolean;
- a characterstring;
- a bitstring;
- a single character;
- a single bit (source only, supporting the encoding of a bitstring – see 23.2).

9.18.3 Transform composites are an ordered list of elements, each of which is a single value and has the same classification (as listed in 9.18.2). (For example, an ordered list of single characters, or of single octets, or of integers.) They are only produced as the result of transforms, and can only be used as the source of a following transform.

9.18.4 If the classification is bitstring, the size of each bitstring value in the composite is the same, and is statically determined by the transform that produces the composite. (For example, an ordered list of single bits, or of six-bit units.)

9.18.5 There are transforms from the following abstract values to composites:

- characterstring to a single character composite;
- bitstring to a bitstring composite (all bitstring values of the composite are of the same size);
- octetstring to a bitstring composite (all bitstring values of the composite are of size 8 bits).

9.18.6 There are transforms from the following composites to abstract values:

- single character composites to characterstring values;
- bitstring composites to bitstring values;
- bitstring composites (with bitstring values of size 8 bits) to octetstring values.

9.18.7 All other transforms can take a value as their source and produce a new value (of the same or of a different classification). They can also take a transform composite as their source and produce a composite as its result, transforming each element of the source composite into an element of the result composite.

9.19 Contents of Encoding Definition Modules

9.19.1 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).

9.19.2 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.

9.19.3 The body of an EDM module contains:

"EncodingObjectAssignment" statements that define and name an encoding object for some encoding class (there are eight 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).

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

9.19.5 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).

9.20 Contents of the Encoding Link Module

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

9.20.2 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.

9.20.3 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.

9.20.4 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.1.17 for an example.)

9.21 Defining encodings for primitive encoding classes

9.21.1 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).

9.21.2 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 Rec. ITU-T X.681 | ISO/IEC 8824-2 as modified by Annex B of this Recommendation | International Standard.

9.21.3 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.

9.21.4 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 properties which collectively define an encoding.

9.21.5 The meaning of the encoding properties 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**.

9.21.6 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.

9.21.7 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.

9.21.8 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.

9.21.9 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.

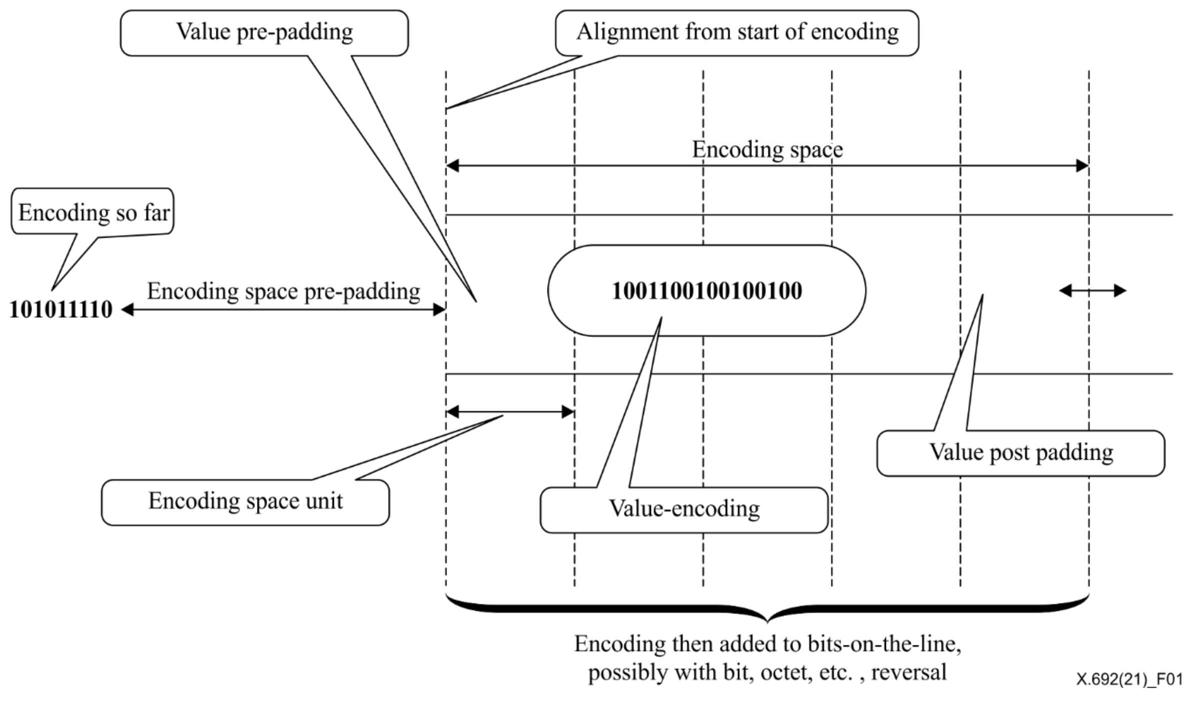
Value pre-padding
Alignment from start of encoding
Encoding space
Encoding so far
101011110
Encoding space pre-padding
1001100100100100
Encoding space unit
Value-encoding
Value post padding
Encoding then added to bits-on-the-line, possibly with bit, octet, etc. , reversal
X.692(21)_F01

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

9.21.10 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.

9.21.11 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** identifies the start of a field.

9.21.12 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.)

9.21.13 Encoding properties are available to support the specification of the encoding rules for all these steps.

9.21.14 In real cases, only some (or none!) of these encoding properties 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.)

9.22 Application of encodings

9.22.1 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.

9.22.2 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.

9.22.3 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.

9.22.4 There are two exceptions to 9.22.3. 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 also 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**".

9.23 Combined encoding object set

9.23.1 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**.

9.23.2 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.1.17 for an example of the application of a combined encoding object set.)

9.23.3 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.

9.24 Application point

9.24.1 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.

9.24.2 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.

9.24.3 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).

9.24.4 In a more complex case, there may be an **`#OPTIONAL`** class following a component class (and a **`#TAG`** class 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 tag, and finally to the component itself.

9.25 Conditional encodings

9.25.1 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`**.

9.25.2 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 clause 24, 23.7 and 23.14 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 18.1.7).

9.25.3 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.)

9.25.4 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 an ordered list of conditional encodings.

9.25.5 Encoding specification for the encoding classes in the octetstring 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 octetstring category is defined as an ordered list of **`#CONDITIONAL-REPETITION`** encoding objects. (See D.1.8.2 for an example.)

9.25.6 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 an ordered list of **`#CONDITIONAL-REPETITION`** encoding objects. (See D.1.7.3 for an example.)

9.25.7 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 permitted 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 an ordered list of these. (See D.1.9.2 for an example.)

9.26 Other conditions for applying encodings

9.26.1 There are a number of different conditions that can be tested in order to select an appropriate encoding. These include the actual value and the range of bounds.

9.26.2 It is also possible to require that all of a given list of conditions are to be satisfied.

9.26.3 A test for a condition uses either a single enumeration value (such as "**`bounded-without-negatives`**") which contains the entire test in the specification of the one enumeration, or a triple of enumerations.

9.26.4 If a triple is used, the first identifies (by an enumeration) the item that is being tested (for example "**`test-upper-bound`**"), the second is the nature of the test (for example "**`greater-than`**"), and the third provides an integer value for the test.

9.27 Encoding control for the open type

9.27.1 Open types frequently provide a means of extensibility using an identification field, with new values for the identification field and new types for the open type being added in successive versions (and often being available for vendor-specific extensions).

9.27.2 Both these features mean that a decoder may be asked to decode an open type when that particular implementation has no knowledge of the type that has been encoded into it.

9.27.3 The encoding support provided for the open type is the same as that for most other classes in the bitfield category, but with the added ability to specify that a different encoding object set is to be applied to the type which is to be encoded into the open type.

NOTE – This is in recognition that many protocols choose to use a different style of encoding (often based on a type-length-value approach) for the type contained in an open type, while retaining a more compact style of encoding for the fields of the message containing the open type.

9.27.4 The model used for decoding an open type recognizes that a decoder will not know what type fills the open type (table and relational constraints are not visible to either PER or to ECN), but that the application may be able to determine this from some other field in the protocol, or in a previous message, or (for vendor-specific additions) based on calling address.

9.27.5 The model is therefore that, having dealt with any specified pre-padding, and determined the encoding space and any value pre- and post-padding, the decoder will ask the application for the type which has been encoded. (In the case of tools, the application will almost certainly have pre-configured the tool with a list of the known types that might be present, and would simply return a pointer to one of these.) Decoding can now proceed normally.

9.27.6 The application may, however, say "unknown" (see 9.27.4), and the decoder then needs to know how to determine the end of this unknown encoding. This is satisfied by enabling the ECN specifier in this case to provide an encoding structure, and (optionally) an encoding object set to use with it, which is to be used by decoders for decoding unknown types in the open type. There is syntax provided in clause 23 for this purpose.

NOTE – An example of such an encoding structure could be one that specifies an encoding that is commonly known as a "Type, Length, Value" encoding, whose end can be determined without knowledge of the type being encoded.

9.28 Changes to ASN.1 Recommendations | International Standards

9.28.1 This Recommendation | International Standard 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.

9.28.2 There is also a need to extend the information object class notation to allow fields that are ordered 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.

9.28.3 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.

9.28.4 These modifications to other ASN.1 Recommendations | International Standards are specified in Annexes A to C, and are solely for the purposes of this Recommendation | International Standard.

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

10.1 Many of the productions within this Recommendation | International Standard 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 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 ELM (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 1 – 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.

NOTE 2 – Implicitly generated encoding structures can only be imported from the ASN.1 module which generates them.

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 explicitly generated encoding structure reference (which is automatically exported by the renames clause which generates it) 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 from that EDM module.

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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.5, 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 16.1.5)
encoding objects	**"EncodingObject"** (See clause 17.1.5)
encoding object sets	**"EncodingObjectSet"** (See clause 18.1)

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 that contains that type assignment. (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 – Subclause 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 is 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 below.

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	*Defined using* ***#SEQUENCE***
CHOICE	#CHOICE	#ALTERNATIVES
EMBEDDED PDV	#EMBEDDED-PDV	*Defined using* ***#SEQUENCE***
ENUMERATED	#ENUMERATED	#INT
EXTERNAL	#EXTERNAL	*Defined using* ***#SEQUENCE***
INTEGER	#INTEGER	#INT
NULL	#NULL	#NUL
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
TIME	#TIME	#TIME
DATE	#DATE	#TIME
TIME-OF-DAY	#TIME-OF-DAY	#TIME
DATE-TIME	#DATE-TIME	#TIME
DURATION	#DURATION	#TIME
GeneralizedTime	#GeneralizedTime	#CHARS
UTCTime	#UTCTime	#CHARS
ObjectDescriptor	#ObjectDescriptor	#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:

a) fully-defined in clause 11.3.4 below; or

b) referenced in those clauses as "See 11.3.2 b" and fully-defined in Rec. ITU-T X.680 | ISO/IEC 8824-1 (including Annex C) with all published amendments and technical corrigenda; or

c) referenced in those clauses as "See 11.3.2 c" and fully-defined in Rec. ITU-T X.681 | ISO/IEC 8824-2 with all published amendments and technical corrigenda.

d) referenced in those clauses as "See 11.3.2 d" and fully-defined in Rec. ITU-T 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 this Recommendation | International Standard.

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.)

e) Where an instance of ASN.1 tag notation is textually followed by one or more further instances of ASN.1 tag notation, the second and subsequent instances of tag notation are discarded.

NOTE – This is similar to the rules for implicit tagging in ASN.1, but applies for all tagging environments. Multiple tagging of the same type is still possible through the use of type reference names.

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 permitted alphabet and effective size constraints (see Rec. ITU-T X.691 | ISO/IEC 8825-2, 10.3).

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 override or not to override 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. (See also 11.3.4.1 e.)

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 "ObjectClassFieldType" (see Rec. ITU-T X.681 | ISO/IEC 8824-2, clause 14) 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 this Recommendation | International Standard. Full support for extensibility is expected to be provided in a later version of this Recommendation | International Standard.

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 are mapped to encoding structure fieldnames.

b) ASN.1 notation in column 1 of Table 2 are mapped to the built-in encoding classes in column 2 of Table 2.

NOTE 1 – The first textually present tag maps into a "**`[#TAG]`**" construction in the implicitly generated structure. The implicitly generated structure does not contain any "**`[#TAG]`**" constructions for subsequent textually present tags.

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 permitted alphabet constraints (see Rec. ITU-T X.691 | ISO/IEC 8825-2, 10.3) are mapped from the type definition to the corresponding field of the encoding structure.

f) The tag number of the first textually present tag maps to the **`#TAG`** class.

11.4.3 Three further implicitly generated structures are produced and exported from all ASN.1 modules. These structures have names **`#CHARACTER-STRING`**, **`#EMBEDDED-PDV`** and **`#EXTERNAL`**, and the structures that they de-reference to are the implicitly generated structures corresponding to the associated types for **`CHARACTER STRING`**, **`EMBEDDED PDV`** and **`EXTERNAL`**, specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 44.5, 36.5 and 37.5 respectively.

11.4.4 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 Recommendation | International Standard 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
        ELMModuleBody
        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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.1.

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 "ELMModuleBody" is:

```
ELMModuleBody ::=
    Imports ?
    EncodingApplicationList

EncodingApplicationList ::=
        EncodingApplication
        EncodingApplicationList ?
```

12.1.6 The production "Imports" (and its semantics) is defined in Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.1, 13.16, and 13.17, as modified by A.1 of this Recommendation | International Standard.

12.1.7 All reference names used in the "ELMModuleBody" shall be imported into the ELM.

NOTE – This is a stronger requirement than that imposed for ASN.1 modules. In ASN.1 modules external references can be used for types and values that have not been imported. In an ELM module (and in an EDM module) external references can only be used for encoding classes that have been referenced in an imports clause. The purpose of external references is solely to resolve ambiguities between imported names and built-in names, or between two identical names imported from different modules.

12.1.8 The "Imports" makes available within the ELM:

a) implicitly generated encoding structures from an ASN.1 module;

b) 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).

c) objects and encoding object sets from an EDM module.

12.1.9 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 contents 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 17.1.7 and 9.23.2).

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 application of encodings to encoding structures in EDMs (specified in 13.2.10) are always non-overlapping. The following subclauses 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 opentype 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 17.1.7).

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 class in the open type category. The encoding of the component of the open type depends on whether there is an **`ENCODED WITH`** present, and on the specification of the encoding object being applied (see 23.10.2).

e) 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.

f) An encoding class preceded by an encoding class 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 tagged class.

g) 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 17.1.7) 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 component.

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 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 – If the encoding object being applied to a class in the open type category contains an **ENCODED WITH**, this determines the encoding object set that is applied to the component, otherwise the combined encoding object set that is being applied to this class is applied to the component (see 23.10.2).

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 17.1.7) 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 encoding class in the tag category, 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 encoding class in the tag category).

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 the components of the constructions defined by those constructors exhibit identification handles (of a given name) to resolve alternatives, or optionality, or termination of a repetition, or order in a set-like concatenation. They may also require that the encoding objects applied to other encoding classes (following those constructions) exhibit the same identification handle, and that the handle value sets of all the involved encoding objects (exhibiting the same handle) be all disjoint. If these conditions are not satisfied, 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.1.

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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.1, as modified by A.1 of this Recommendation | International Standard.

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 except that of an implicitly generated structure. The "Symbol" in the "Exports" can reference any encoding class (except a built-in encoding class or an implicitly generated structure), 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. An implicitly generated encoding structure can never be exported from an 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.

NOTE – This is a stronger requirement than that imposed for ASN.1 modules. In ASN.1 modules, external references can be used for types and values that have not been imported. In an EDM module (and in an ELM module) external references can only be used for encoding classes that have been referenced in an imports clause. The purpose of external references is solely to resolve ambiguities between imported names and built-in names, or between two identical names imported from different modules.

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	Subclause 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.3.7.

15.1.2 The production "GlobalModuleReference" is defined in Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.1, 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 If a renames clause references more than one ASN.1 module and as a result of this two explicitly generated structures have the same simple name, then neither structure is available for explicit import into an ELM or an EDM module.

NOTE – These explicitly generated structures nonetheless exist, and are likely to be implicitly referenced by other explicitly generated structures that are exported without restriction.

15.1.6 The primary purpose of the renames clause is to make available the explicitly generated structures for import into other modules, particularly the ELM. However, this clause also makes these structures available for reference within the EDM module containing the renames clause except as specified in 15.1.7. If the simple name is ambiguous, then a fully-qualified name shall be used within the EDM module containing the renames clause, as specified in 15.1.9.

NOTE – Ambiguity can arise either because of clashes with the names of built-in classes, or because of clashes of simple names between structures generated from more than one ASN.1 module, or both.

15.1.7 When a renames clause produces an explicitly generated structure from an implicitly generated structure, that implicitly generated structure cannot be imported into this EDM module using an imports clause, and the implicitly generated structure is never available in this EDM module.

15.1.8 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 15.1.9.

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.

15.1.9 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. 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.

15.1.10 The "ExternalEncodingClassReference" notation shall not be used in an imports clause except where required by clause 15.1.9.

15.1.11 If a name which has been imported using an "ExternalEncodingClassRefererence" is used in the body of a module, then the simple "encodingclassreference" can be used unless an "ExternalEncodingClassReference" is required as specified in clause 15.1.9.

15.1.12 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 15.1.14.

NOTE – (Tutorial) If, in an EDM module, there are multiple structures with the same simple reference name (whether these names arise from an imports clause or from a renames clause, or from clashes with built-in classes, or from any combination of these), then a fully-qualified name is used except for references to a built-in class. For implicitly generated structures, the fully-qualified name always uses the ASN.1 module name. For structures generated by the renames clause in an EDM module, the fully-qualified name is used. This fully-qualified name in the body of this EDM always uses the ASN.1 module name referenced by the renames clause. For structures imported from another EDM module, the fully-qualified name uses the name of that EDM module. This is always unambiguous, as importation is not permitted if an EDM module generates multiple explicitly generated structures with the same simple reference name.

15.1.13 If "OptionalNameChanges" is "NameChanges", then 15.1.14 still applies, but the explicitly generated encoding structures are further modified as specified in 15.2.

15.1.14 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.

NOTE – The "<*OriginalClassName*>" is required to reference a built-in encoding class or an externally generated encoding structure produced by the renames clause in this module. In case of ambiguity, this will require the use of an external reference in "<*OriginalClassName*>".

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
        "."
        ComponentIdList

ComponentIdList ::=
        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 the identified component of that encoding structure.

15.3.4 The first "identifier" of the "ComponentIdList" shall be an "identifier" in a "NamedField" of the implicitly generated encoding structure identified by the "encodingclassreference" in the "ComponentReference", and identifies the entire definition of that component of the encoding structure. Each subsequent "identifier" of the "ComponentIdList" shall be an "identifier" in a "NamedField" of the implicitly generated encoding structure identified by the previous part of the "ComponentIdList", and identifies the entire definition of that component.

15.3.5 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".

15.3.6 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".

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

15.3.8 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) A category in the bit-field group of categories (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) A category in the encoding procedure group of categories (see 16.1.13).

NOTE – The term encoding constructor is used for any class in the alternatives, concatenation, and repetition categories. These are also called the encoding constructor group of categories.

16.1.4 The category of each built-in encoding class is specified in 16.1.14.

NOTE – If an encoding class is a tagged class (see 16.2.1), or has bounds (see 16.2.6), then the category of the class is the category of the class with the tag and the bounds removed.

16.1.5 The "EncodingClass" is:

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

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
                    |         #TIME
                    |         #DATE
                    |         #DATE-TIME
                    |         #TIME-OF-DAY
                    |         #DURATION
                    |         #GeneralizedTime
                    |         #UTCTime
                    |         #ObjectDescriptor
                    |         #BMPString
                    |         #GeneralString
                    |         #GraphicString
                    |         #IA5String
                    |         #NumericString
                    |         #PrintableString
                    |         #TeletexString
                    |         #UniversalString
                    |         #UTF8String
                    |         #VideotexString
                    |         #VisibleString
```

The categories of the classes that these built-in names reference (see 16.1.14) are all defined to be in the bit-field group of categories.

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. Their category is that of the class from which they are derived. 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. The third column of the table gives the category for each of the built-in classes that are not derived from other classes.

Built-in class	Derived from	Category
#ALTERNATIVES	(primitive)	**alternatives**
#BITS	(primitive)	**bitstring**
#BIT-STRING	**#BITS**	
#BOOL	(primitive)	**boolean**
#BOOLEAN	**#BOOL**	
#CHARACTER-STRING	(defined using **#SEQUENCE)**	
#CHARS	(primitive)	**characterstring**
#CHOICE	**#ALTERNATIVES**	
#CONCATENATION	(primitive)	**concatenation**
#CONDITIONAL-INT	(primitive)	**encoding procedure**
#CONDITIONAL-REPETITION	(primitive)	**encoding procedure**
#EMBEDDED-PDV	(defined using **#SEQUENCE)**	
#ENUMERATED	**#INT**	
#EXTERNAL	(defined using **#SEQUENCE)**	
#INT	(primitive)	**integer**
#INTEGER	**#INT**	
#NUL	(primitive)	**null**
#NULL	**#NUL**	
#OBJECT-IDENTIFIER	(primitive)	**objectidentifier**
#OCTETS	(primitive)	**octetstring**
#OCTET-STRING	**#OCTETS**	
#OPEN-TYPE	(primitive)	**opentype**
#OPTIONAL	(primitive)	**optionality**
#OUTER	(primitive)	**encoding procedure**
#PAD	(primitive)	**pad**
#REAL	(primitive)	**real**
#RELATIVE-OID	**#OBJECT-IDENTIFIER**	
#REPETITION	(primitive)	**repetition**
#SEQUENCE	**#CONCATENATION**	
#SEQUENCE-OF	**#REPETITION**	
#SET	**#CONCATENATION**	
#SET-OF	**#REPETITION**	
#TAG	(primitive)	**tag**
#TIME	(primitive)	**time**
#DATE	**#TIME**	
#TIME-OF-DAY	**#TIME**	
#DATE-TIME	**#TIME**	
#DURATION	**#TIME**	
#TRANSFORM	(primitive)	**encoding procedure**
#GeneralizedTime	**#CHARS**	
#UTCTime	**#CHARS**	
#ObjectDescriptor	**#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 ?
       "]"
       UntaggedStructure

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 encoding structure 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`**.)

NOTE – The syntax prohibits the specification of a tag class immediately following another tag class in the definition of an encoding structure, nor can such structures be produced by multiple textual tags in an ASN.1 type definition (see 11.3.4.1 e).

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

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
       |   #TIME
       |   #DATE
       |   #TIME-OF-DAY
       |   #DATE-TIME
       |   #DURATION
       |   #GeneralizedTime
       |   #UTCTime
       |   #ObjectDescriptor   Size?
       |   #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 represent 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
#TIME Time 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 permitted 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 19.1. 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
        "{"
        identifier ?
        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 There shall be no recursive definition (see 3.2.39) of an "encodingobjectreference", and there shall be no recursive instantiation (see 3.2.40) of an "encodingobjectreference" if these recursions lead to an infinite recursion in the definition of the encoding.

17.1.5 The "EncodingObject" is:

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

17.1.6 "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.

17.1.7 In this Recommendation | International Standard, "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, or shall be reference names that de-reference to the same encoding class name.

17.1.8 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 Rec. ITU-T X.681 | ISO/IEC 8824-2, 11.5 and 11.6, as modified by B.16 of this Recommendation | International Standard, 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 below. The syntax to be used for each encoding object is the "DefinedSyntax" for the corresponding category or 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 (possibly included as part of a parameterized encoding object set) are only useful for application to an encoding structure in an 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 category
boolean category
integer category
bitstring category
octetstring category
characterstring category
pad category
alternatives category
repetition category
concatenation category
optionality category
#CONDITIONAL-INT class
#CONDITIONAL-REPETITION class
tag category
#TRANSFORM class
#OUTER class
```

17.2.3 The information required (and the syntax to be used) to specify an encoding object of one of these categories or classes using the "DefinedSyntax" is specified by the definitions in clauses 23 to 25.

17.2.4 If a governor for a value of one of the fields appearing in the "DefinedSyntax" is needed for use in a dummy parameter list, then the notation "EncodingClassFieldType" (specified in B.17) 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 the "DefinedSyntax" construction by using a dummy parameter of the encoding object that is being defined or, in the case of **`flag-to-be-used`** or **`flag-to-be-set`**, by using a reference name that is textually present in the definition of a replacement structure. A **`REFERENCE`** that is used as a determinant shall not be the named component of a repetition.

17.2.6 The "DefinedSyntax" notation specifies whether the encoding object 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 is 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 Call E the encoding object (within the "CombinedEncodings") which is applied to the governor class. If the encoding object E exhibits an identification handle (with a given handle value set), then the encoding object being defined (see 17.1.5) exhibits the same identification handle as E (with the same handle value set); otherwise, it does not exhibit a 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 user-defined encoding structure or a built-in class in the bit-field group of categories (see 16.1.7).

17.4.3 The production "ValueMapping" is specified in 19.1.7, and shall be a mapping of values associated with the governing encoding class to the class identified by the "DefinedOrBuiltinEncodingClass". The governing encoding class shall be a class in the bit-field group of categories.

17.4.4 The "ValueMappingEncodingObjects" specifies the encoding of the "DefinedOrBuiltinEncodingClass". The 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 through the application of encodings specified in clause 13.

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 or effective size constraints on fields of the "DefinedOrBuiltinEncodingClass", and the specifications in clause 19 require values to be mapped to those fields that violate the specified bounds or effective size constraints, then such values are not mapped, and the encoding of such values is not possible. It is an ECN or application error if such values are submitted for encoding.

17.4.8 Call E the encoding object which is applied to the "DefinedOrBuiltinEncodingClass". If the encoding object E exhibits an identification handle (with a given handle value set), then the encoding object being defined (see 17.1.5) exhibits the same identification handle as E (with the same handle value set); otherwise, it does not exhibit a handle.

NOTE – The encoding object E may be either the "EncodingObject" in the "ValueMappingEncodingObjects", or a member of the "DefinedOrBuiltinEncodingObjectSet".

17.5 Encoding an encoding structure

17.5.1 The "EncodeStructure" is:

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

StructureEncoding ::=
                 STRUCTURED WITH
                    TagEncoding ?
                EncodingOrUseSet

TagEncoding ::= "[" EncodingOrUseSet "]"

EncodingOrUseSet ::=
                EncodingObject  |
                USE-SET
```

17.5.2 The "EncodeStructure" can be used to define an encoding only if the governing encoding class de-references to a construction defined using an encoding constructor in the alternatives, concatenation, or repetition categories, or to a construction defined using one of these categories preceded by a class in the tag category. This encoding constructor 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 class in the tag category that precedes the governor encoding constructor. 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 class 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 If the "ComponentEncodingList" is not empty, then the encoding object applied to the governing encoding constructor (whether from "StructureEncoding" or from "CombinedEncodings") shall not specify any replacement actions.

17.5.5 If the "EncodingOrUseSet" in the "StructureEncoding" 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., contains an encoding class in the optionality category).

17.5.10 If the "ComponentEncoding" for any component is not present in the "ComponentEncodingList", then the "CombinedEncodings" shall be present (but see also 17.5.6), and is required, on application to the component (see 13.2), to provide a complete encoding of that component (possibly including use of dummy parameters), otherwise it is an error in the ECN specification.

```
NonOptionalComponentEncodingSpec ::=
                identifier ?
                TagAndElementEncoding

OptionalComponentEncodingSpec ::=
                identifier
                TagAndElementEncoding
                OPTIONAL-ENCODING
                OptionalEncoding

TagAndElementEncoding ::=
                TagEncoding ?
                EncodingOrUseSet

OptionalEncoding ::= EncodingOrUseSet
```

17.5.11 The "identifier" shall be the "identifier" of the component of the governing encoding constructor. The "identifier" in "NonOptionalComponentEncodingSpec" shall be omitted if and only if the governing encoding constructor is a class in the repetition category for which there is no identifier on the repeated element.

17.5.12 "TagAndElementEncoding" in the "ComponentEncoding" shall provide a complete encoding for the component (including any class in the tag category that is prefixed to the element, but excluding any class in the optionality category that follows the element).

17.5.13 The "EncodingObject"s in the "EncodingOrUseSet"s in the "TagAndElementEncoding" shall be governed by the corresponding encoding classes in the component. If an "EncodingOrUseSet" is **USE-SET** then the encoding is obtained by applying the "CombinedEncodings" (which shall be present).

17.5.14 The "EncodingOrUseSet" in the "OptionalEncoding" shall completely encode the class in the optionality category of the component. If an "EncodingOrUseSet" is **USE-SET** then the encoding of the class in the optionality category is obtained by applying the "CombinedEncodings" (which shall be present).

17.5.15 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 a "ComponentIdList" (see 15.3.1 for the syntax of the "ComponentIdList" – the meaning of the "ComponentIdList" in this context is specified below).

17.5.16 If the governor is not a constructor in the repetition category, then the first (or only) "identifier" in the "ComponentIdList" shall be the "identifier" of a textually present "NamedType" (at some level of nesting – see 17.5.17) of the construction that is obtained by de-referencing the governor. It identifies the entire definition of that "NamedType" component, whether that definition is textually present or not.

17.5.17 If there is more than one such matching identifier, then the chosen matching identifier shall be determined by the first match in a scan (in textual order) of the outer-level identifiers, then by a scan (in textual order) of the second level identifiers, then by a scan (in textual order) of the third-level identifiers, and so on.

17.5.18 Each subsequent "identifier" of the "ComponentIdList" (if any) shall be an "identifier" in a "NamedType" of the structure identified by the previous part of the "ComponentIdList", and identifies the entire definition of that "NamedType" component, whether it is textually present or not in the definition of the structure identified by the previous part of the "ComponentIdList".

17.5.19 If the governor is a constructor in the repetition category, then the actual parameter for the **REFERENCE** shall be a "ComponentIdList" whose first "identifier" identifies a component that is textually present in the "EncodingStructure" in the "RepetitionStructure" obtained by de-referencing the repetition (see 17.5.17). Subclauses 17.5.17 and 17.5.18 then apply.

17.5.20 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 alternatives 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 or the use of **OUTER**, or where references are in support of **flag-to-be-used** or **flag-to-be-set** in the definition of an encoding object for a class in the repetition category which uses replacement.

17.5.21 Determination of whether the encoding object being defined (see 17.1.5) exhibits an identification handle shall be done as follows:

a) if the "TagEncoding" is present in "StructureEncoding", call E the encoding object which is applied to the encoding class in the tag category; or

b) if the "TagEncoding" is not present in "StructureEncoding", call E the encoding object which is applied to the governing encoding constructor (this may be either the "EncodingObject" in the "EncodingOrUseSet" in the "StructureEncoding", or may be a member of the "CombinedEncodings").

If the encoding object E exhibits an identification handle (with a given handle value set), then the encoding object being defined exhibits the same identification handle as E (with the same handle value set); otherwise, it does not exhibit a handle.

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 this specification.

NOTE 2 – The behaviour 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 If the "SpecForEncoding" and the "SpecForDecoders" exhibit the same identification handle with the same handle value set, then the encoding object being defined (see 17.1.5) exhibits that identification handle (with the same handle value set); otherwise, it does not exhibit a handle.

17.7 Encoding options

17.7.1 The "EncodingOptionsEncodingObject" is:

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

EncodingOptionsList ::= OrderedEncodingObjectList

AlternativesEncodingObject ::= EncodingObject
```

17.7.2 The "EncodingOptionsEncodingObject" specifies that the encoder may encode (subject to 17.7.6) 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 ordered list that is capable of encoding the abstract value to be encoded (see 17.7.6). The encoding options specification is provided only because it is necessary to reflect options provided in legacy protocols and to support different forms of length encoding for strings. All the encoding options 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 encoding options were encodings for alternatives of an instance of that class. The "AlternativesEncodingObject" shall not contain a **REPLACE** specification (see 23.1.1). The **DETERMINED BY** parameter shall be set to **handle**, and an identification handle shall be specified.

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

17.7.4 All "EncodingObject"s in the "EncodingOptionsList" shall exhibit that identification handle, and their handle value sets shall all be disjoint.

17.7.5 If the "AlternativesEncodingObject" exhibits an identification handle (with a given handle value set), then the encoding object being defined (see 17.1.5) exhibits the same identification handle (with the same handle value set); otherwise, it does not exhibit a handle.

NOTE – The identification handle exhibited by the "AlternativesEncodingObject" (if any) is unrelated to the identification handle exhibited by the "EncodingObject"s in the "EncodingOptionsList", even if they have the same name.

17.7.6 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 encoding options encoding object as the "SpecForDecoders" (see 17.6), where the "SpecForEncoding" is an encoding 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 13.1, as modified by A.1 of this Recommendation | International Standard. 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 this Recommendation | International Standard.

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 Rec. ITU-T X.660 | ISO/IEC 9834 series.

17.8.6 An identification handle (with a given handle value set) is exhibited by the encoding object being defined (see 17.1.5) 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 ::=
                    DefinedOrBuiltinEncodingObjectSet |
                    EncodingObjectSetSpec
```

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

18.1.3 There shall be no recursive definition (see 3.2.39) of an "encodingobjectsetreference", and there shall be no recursive instantiation (see 3.2.40) of an "encodingobjectsetreference".

18.1.4 "DefinedOrBuiltinEncodingObjectSet" is defined in 10.9.3.

18.1.5 The "EncodingObjectSetSpec" is:

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

EncodingObjects ::=
            DefinedEncodingObject  |
            DefinedEncodingObjectSet

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

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

18.1.7 Encoding objects forming an encoding object set shall all be of distinct encoding classes, and shall not be classes in the encoding procedure group of categories 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.

18.1.8 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 Rec. ITU-T X.690 | ISO/IEC 8825-1 and Rec. ITU-T 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 Recommendation | International Standard, 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(2) asn1(1) packed-encoding(3) basic(0) aligned(0)}`
`PER-BASIC-UNALIGNED`	`{joint-iso-itu-t(2) asn1(1) packed-encoding(3) basic(0) unaligned(1)}`
`PER-CANONICAL-ALIGNED`	`{joint-iso-itu-t(2) asn1(1) packed-encoding(3) canonical(1) aligned(0)}`
`PER-CANONICAL-UNALIGNED`	`{joint-iso-itu-t(2) packed-encoding(3) canonical(1) unaligned(1)}`
`BER`	`{joint-iso-itu-t(2) asn1(1) basic-encoding(1)}`
`CER`	`{joint-iso-itu-t(2) asn1(1) ber-derived(2) canonical-encoding(0)}`
`DER`	`{joint-iso-itu-t(2) 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), with appropriate de-referencing, to specify the corresponding BER or PER encodings.

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

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

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 components. If these encoding objects are applied to a **`#CONCATENATION`** with optional components, 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 opentype 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 by these encoding objects 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 as these encoding objects do not contain an **`ENCODED WITH`** (see 13.2.10.5 and 13.2.9 d).

19 Mapping values

19.1 General

19.1.1 This clause specifies the syntax for mapping values (and tag numbers) 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 structure to which the "DefinedOrBuiltinEncodingClass" in the "EncodingObjectAssignment" (see 17.1.1), of which it is a part, de-references is called the source governor or the source encoding class (depending on context). The structure to which the "DefinedOrBuiltinEncodingClass" in the "EncodeByValueMapping" itself de-references is called the target governor or the target encoding class (depending on context).

19.1.3 If the source governor has an initial class in the tag category, then the target governor shall have an initial class in the tag category and the tag number of the class in the source governor is mapped to the tag number of the class in the tag category in the target governor. If the class in the tag category in the target governor has an associated tag number, then it is an ECN specification error if this differs from the tag number being mapped from the source governor.

19.1.4 If the source governor does not have an initial class in the tag category, then the target governor is not required to have an initial class in the tag category, but if it does, then there shall be a tag number associated with that tag in the definition of the target governor.

19.1.5 The effect of the presence of an initial class in the tag category in the source or target governors is completely determined by 19.1.3 and 19.1.4, and the following text ignores the possible presence of such classes.

19.1.6 The encodings specified for values mapped to the target encoding class become the 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 (see 17.4.7).

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

19.1.7 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.)

19.1.8 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 Rec. ITU-T 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 Rec. ITU-T 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 Rec. ITU-T 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 14.1) 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.10 and 12.12)
boolean	**"BooleanValue"** (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 18.3)
characterstring	**"RestrictedCharacterStringValue"** (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 41.8)
integer	**"SignedNumber"** (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 19.1)
null	**"NullValue"** (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 24.3)
objectidentifier	**"DefinitiveIdentifier"** (see A.1)
octetstring	**"bstring" or "hstring"** (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.10 and 12.12)
real	**"RealValue"** (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 21.6)
time	**"TimeValue"** (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 38.3.2)

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 It is an ECN specification error if "MappedValue2" is a value which violates a bound or size constraint in the target.

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 until the source and target start with an encoding constructor. If this encoding constructor in the target is in the repetitions category, then de-referencing of the component of this repetition encoding constructor is performed until the component starts with an encoding constructor. References within the resulting structures are not resolved.

19.3.4 The effect of the possible presence of classes in the tag category on the initial de-referencing of "DefinedOrBuiltinEncodingClass" names in the source and target was fully specified in 19.1.3 to 19.1.5. It is an ECN specification error if further initial classes in the tag category are introduced by the application of 19.3.3.

19.3.5 After the application of 19.3.3, the source and the target encoding classes shall start with the same encoding constructor. This shall be either an encoding constructor in the concatenation category, or an encoding constructor in the repetitions category. If this encoding constructor is in the repetitions category, then its component in the target shall be a class in the concatenation category. For the purposes of this subclause 19.3, the resulting encoding structures are called the source and target encoding structures respectively.

19.3.6 The fieldnames of the (top-level) components of the encoding constructor produced by the application of 19.3.3 to the source are called the source fields.

NOTE – Source fields are restricted to the top-level fields of a concatenation or the component of a repetition. This restriction is imposed to ease implementation of ECN, and could be relaxed in the future.

19.3.7 The fieldnames of the components of the encoding constructor in the concatenation categories produced by the application of 19.3.3 to the target are called the potential target fields.

NOTE – The potential target fields may be either the components of a top-level concatenation, or the components of a concatenation that is the component of a repetition.

19.3.8 For every source field, there shall be a potential target field with the same fieldname (the matching target field).

NOTE – A component of a repetition class can only be mapped if it contains an identifier (matching one in the target). Use of mapping by matching fields would not be legal if the identifier was absent.

19.3.9 A matching target field shall be an optional element in a concatenation if and only if its source field is an optional element in a concatenation, and the presence or absence of the source field in an abstract value associated with the source encoding structure determines the presence or absence of the target field in the target encoding structure.

19.3.10 If the source field has an initial class in the tag category, then the matching target field shall have an initial class in the tag category and the tag number of the class in the source field is mapped to the tag number of the class in the tag category in the matching target field. If the class in the tag category in the matching target field has an associated tag number, then it is an ECN specification error if this differs from the tag number being mapped from the source field.

19.3.11 If the source field does not have an initial class in the tag category, then the matching target field is not required to have an initial class in the tag category, but if it does, then there shall be a tag number associated with that tag in the definition of the matching target field.

19.3.12 Apart from the presence or absence of classes in the tag category and optionality categories (as specified in 19.3.9 to 19.3.11), the matching target field and the source field shall have the same encoding class (see 17.1.7) or shall be defined using the same sequence of lexical items, ignoring comment and whitespace and bounds specifications.

19.3.13 All abstract values are mapped from each of the source fields to the matching target fields. 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.

19.3.14 If the source and target encoding constructors are classes in the repetition category, then the number of repetitions in the abstract value associated with the source encoding structure is mapped to the number of repetitions in the target encoding structure.

19.3.15 If a source field has an associated contents constraint, this is mapped as an associated contents constraint to the matching target field.

19.3.16 If, due to the presence of bounds or size constraints, there are values in the source field that are not present in the matching target field, then 17.4.7 shall apply.

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 "OrderedTransformList") 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
                 "{"
                 OrderedTransformList
                 "}"

OrderedTransformList ::= Transform "," +

Transform ::= EncodingObject
```

19.4.4 All the "EncodingObject"s in the "OrderedTransformList" 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 It is an ECN specification or application error if any "Transform" in the "OrderedTransformList" is not reversible for the abstract value being mapped.

NOTE – Clause 24 specifies, for each transform, the abstract values for which it is defined to be reversible.

19.4.7 If there are bounds or effective size constraints on the target encoding class, then 17.4.7 shall apply.

19.5 Mapping by abstract value ordering

19.5.1 This mapping enables abstract values associated with simple encoding classes to be distributed into the fields of complex encoding structures, and abstract values associated with complex encoding structures to be mapped to simple encoding classes such as **`#INT`**. It also allows the compaction of integer values or enumerations into a contiguous set of integer values (see D.1.4).

NOTE – The tag numbers associated with classes in the tag category are not abstract 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, or shall be a class in the concatenation category which has a single non-optional component.

19.5.4 The ordered set of values may be finite or infinite.

19.5.4.1 A finite set of ordered abstract values is defined for encoding classes in the following categories:

a) null;

b) boolean;

c) bounded integer;

d) real constrained to a finite number of values;

e) an encoding structure defined using the alternatives category, provided that all of the alternatives have a finite ordering defined;

f) an encoding structure defined using the concatenation category that has a single non-optional component, provided that the component has a finite ordering defined.

19.5.4.2 An infinite set of ordered abstract values is defined for encoding classes in the following categories:

a) integer, constrained to have a finite lower bound;

b) an encoding structure defined using the alternatives category, provided that all of the alternatives except the last are defined to have a finite set of ordered values, and the last alternative is defined to have an infinite set of ordered values;

c) an encoding structure defined using the concatenation category that has a single non-optional component, provided that the component is defined to have an infinite set of ordered 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 ordering of the abstract values associated with a class in the concatenation category that has a single non-optional component shall be the order determined by the ordering of the abstract values of its single component.

19.5.9 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.

19.5.10 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).

19.5.11 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.

19.5.12 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.

19.5.13 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 was the same as the source structure preceded by one or more instances of classes in the tag category.

19.5.14 Classes in the tag category may be present in the target structure, but are required to have an associated tag number specified in the structure definition. Their presence has no affect on the mapping of abstract values.

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, possibly with a preceding class in the tag category which is mapped according to 19.1.3 to 19.1.5.

19.6.4 The target encoding class may be any encoding structure of the concatenation category where all the components are optional, or of the alternatives category, and may contain classes in the tag category, but all fieldnames in the entire encoding structure shall be distinct, and all classes in the tag category in the target (except those mapped by 19.6.3) shall have a tag number in their definition and are otherwise ignored in the mapping.

19.6.5 Values shall be mapped only to fields defined at the top-level of the target structure that are classes in the integer category, possibly preceded by classes in the tag category (see 19.6.4), and possibly with bounds.

19.6.6 The "MappingByValueDistribution" is:

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

Distribution ::=
                 SelectedValues
                 TO
                 identifier

SelectedValues ::=
                 SelectedValue
            |    DistributionRange
            |    REMAINDER

DistributionRange ::=
                 DistributionRangeValue1
                 ".."
                 DistributionRangeValue2

SelectedValue ::= SignedNumber

DistributionRangeValue1 ::= SignedNumber
DistributionRangeValue2 ::= SignedNumber
```

19.6.7 "SignedNumber" is specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 19.1.

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

19.6.9 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".

19.6.10 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".

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

19.6.12 If there are bounds on the target field, then 17.4.7 shall apply.

19.6.13 If a value from the source is mapped into a field in the target whose presence depends on optionality or choice of alternatives or both, this is not an error, but the optionality and choice of alternatives in the target (when encoding such values) shall be such that the encoding of the target includes the target field.

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 (possibly preceded by classes in the tag category as specified in 19.1.3 to 19.1.5), mapping each integer value to a bitstring value (possibly preceded by classes in the tag category).

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, possibly preceded by classes in the tag category.

19.7.3 The destination encoding class shall be a class in the bitstring category, possibly preceded by classes in the tag category.

19.7.4 Classes in the tag category are mapped as specified in 19.1.3 to 19.1.5.

19.7.5 The "MappingIntToBits" is:

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

MappedIntToBits ::=
        SingleIntValMap      |
        IntValRangeMap
```

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

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

19.7.8 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.

19.7.9 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. It is an ECN or application error if such values are presented to an encoder.

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.

19.7.10 The "SingleIntValMap" is:

```
SingleIntValMap ::=
        IntValue
        TO
        BitValue

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

19.7.11 The "SignedNumber", "bstring", and "hstring" are specified in Rec. ITU-T X.680 | ISO/IEC 8824-1, 19.1, 12.10 and 12.12, respectively.

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

19.7.13 The "IntValRangeMap" is:

```
IntValRangeMap ::=
        IntRange
        TO
        BitRange

IntRange ::=
        IntRangeValue1
        ".."
        IntRangeValue2
```

BitRange ::=
```
        BitRangeValue1
        ".."
        BitRangeValue2
```

IntRangeValue1 ::= SignedNumber

IntRangeValue2 ::= SignedNumber

BitRangeValue1 ::=
```
        bstring |
        hstring
```

BitRangeValue2 ::=
```
        bstring |
        hstring
```

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

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

19.7.16 When interpreted as a positive integer encoding (see Rec. ITU-T 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".

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

19.7.18 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".)

19.7.19 It is an ECN specification error if any "BitRange" includes a value which violates a size constraint on the target.

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 Rec. ITU-T X.681 | ISO/IEC 8824-2 as modified by Annex B of this Recommendation | International Standard.

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 a class in the tag category.

20.3 The use of the modified information object class notation is solely for use within this Recommendation | International Standard.

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 values of some encoding properties, in conjunction with the values of other encoding properties, 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 properties 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 property 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.15).

20.7 The encoding properties specified in clauses 23 to 25 operate together in encoding property 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 properties.

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 property groups require an ordered 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 abstract values for 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 property of this type specifies the unit in which other encoding properties or determinant fields are counting.

21.1.4 The value of an encoding property of this type is restricted in all cases but one to the non-zero values. In these cases the encoding property specifies a number of bits. That number of bits determines the unit in which other encoding properties 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)} (-3..MAX)
```

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

21.2.3 An encoding property of this type specifies the size of the encoding space (see 9.21.5).

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 (since the encoding space needed for each component may be different), but is always that fixed number of repetitions, and it is an ECN specification or application error if an abstract value is to be encoded which does not have that 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`" (see 21.3) or "`RepetitionSpaceDetermination`" (see 21.7) is required.

NOTE – A value of type "`EncodingSpaceDetermination`" or "`RepetitionSpaceDetermination`" (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. In this case, a value of type "`EncodingSpaceDetermination`" (see 21.3) or "`RepetitionSpaceDetermination`" (see 21.7) is required (to provide a precise means of determining the size of the encoding space).

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. This value shall not be used if the abstract value to be encoded into the encoding space is an abstract value associated with a class in the concatenation (see 23.5.2.3) or repetition category (see 23.14.2.5).

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
      {field-to-be-set, field-to-be-used, container}
```

21.3.2 The default value for this type is always "`field-to-be-set`".

21.3.3 An encoding property of this type specifies the way in which the encoding space is determined when an encoding property of type "`EncodingSpaceSize`" (see 21.2) is set to "`variable-with-determinant`" or "`encoder-option-with-determinant`".

21.3.4 The value "`field-to-be-set`" requires the specification of a **`REFERENCE`** to a field that will be set by the encoder to carry length information, and used by a decoder. 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. If a field is set more than once through the use of "`field-to-be-set`" or "`flag-to-be-set`" (see 21.7), then it is an ECN specification or an application error if different values are produced by the different encoding procedures, and encoders shall not generate encodings in this case.

21.3.5 The value "`field-to-be-used`" requires the specification of a **`REFERENCE`** to a field whose value may be set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification) or may be set by some other encoder actions invoked by "`field-to-be-set`" or "`flag-to-be-set`". 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. This specification 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
      {field-to-be-set, field-to-be-used, not-needed}
```

21.4.2 The default value for this type is always "`field-to-be-set`".

21.4.3 An encoding property 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 "`field-to-be-set`" requires the specification of a **`REFERENCE`** to a field that will be set by the encoder to carry unused bits information, and used by a decoder. 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. If a field is set more than once through the use of "`field-to-be-set`" or "`flag-to-be-set`" (see 21.7), then it is an ECN specification or an application error if different values are produced by the different encoding procedures, and encoders shall not generate encodings in this case.

21.4.5 The value "`field-to-be-used`" requires the specification of a `REFERENCE` to a field whose value may be set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification) or may be set by some other encoder actions invoked by "`field-to-be-set`" or "`flag-to-be-set`". 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
      {field-to-be-set, field-to-be-used, container, handle, pointer}
```

21.5.2 The default value for this type is always "`field-to-be-set`".

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

21.5.4 The value "`field-to-be-set`" requires the specification of a `REFERENCE` to a field that will be set by the encoder to carry optionality information, and used by a decoder. The ECN specification will also include an encoding property that specifies how an encoder is to set the value of this field from a conceptual boolean value which is true if the optional component is present and false if the optional component is absent. If a field is set more than once through the use of "`field-to-be-set`" or "`flag-to-be-set`" (see 21.7), then it is an ECN specification or an application error if different values are produced by the different encoding procedures, and encoders shall not generate encodings in this case.

21.5.5 The value "`field-to-be-used`" requires the specification of a `REFERENCE` to a field whose value may be set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification) or may be set by some other encoder actions invoked by "`field-to-be-set`" or "`flag-to-be-set`". The specification will also include an encoding property that specifies how a decoder is to determine the presence or absence of the optional component 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 component, 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 component, then the decoder shall determine that this optional component is absent.

NOTE – This specification 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 component unless all earlier optional components 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 optionality 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 both by the encoding object for the optional component and by the encoding object applied to each possible alternative encoding class that can follow if this optional component is absent. Each possible alternative encoding class may be a component of the concatenation containing the optional component, or may be an encoding class following the concatenation. The handle value sets specified by all the involved encoding objects (exhibiting the same identification handle) shall all be disjoint.

NOTE – Every abstract value of a given component is required to have a handle value matching the specified handle value set (see 22.9.2.2).

21.5.8 If the end of any open container (or the end of the PDU) is detected at the time a decoder is attempting to detect the presence or absence of an optional component, then the decoder shall determine that the optional component is absent. Otherwise, the decoder shall determine that the component is present if and only if decoding the remaining parts of the encoding produces a value for the specified identification handle which matches the handle value set of the optional component. 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 component, but conforming encoders shall not generate such encodings.

21.5.9 The value "`pointer`" requires the specification of a start-of-encoding `REFERENCE` to another field. If that field is zero, then this component 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 {field-to-be-set, field-to-be-used, handle}
```

21.6.2 The default value for this type is always "`field-to-be-set`".

21.6.3 An encoding property 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 "`field-to-be-set`" requires the specification of a `REFERENCE` to a field that will be set by the encoder to carry information identifying an alternative, and used by a decoder. The specification will also include an encoding property 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 properties). If a field is set more than once through the use of "`field-to-be-set`" or "`flag-to-be-set`" (see 21.7), then it is an ECN specification or an application error if different values are produced by the different encoding procedures, and encoders shall not generate encodings in this case.

21.6.5 The value "`field-to-be-used`" requires the specification of a `REFERENCE` to a field whose value may be set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification) or may be set by some other encoder actions invoked by "`field-to-be-set`" or "`flag-to-be-set`". The specification will also include an encoding property 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 properties).

21.6.6 The value "`handle`" requires that an identification handle be specified. This identification handle shall be exhibited by the encoding objects applied to each of the alternatives in the construction defined by the class in the alternatives category. The handle value sets specified by those encoding objects shall all be disjoint. (Violation of this rule is an ECN specification error, and conforming encoders are required not to generate encodings where this rule is violated.)

21.6.7 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 handle value set 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 the handle value set of any alternative, then this is an encoding error.

NOTE – Every abstract value of a given alternative is required to have a handle value matching the handle value set of the alternative (see 22.9.2.2).

21.7 The `RepetitionSpaceDetermination` type

21.7.1 The "`RepetitionSpaceDetermination`" type is:

```
RepetitionSpaceDetermination ::= ENUMERATED
      {field-to-be-set, field-to-be-used, flag-to-be-set, flag-to-be-used,
       container, pattern, handle, not-needed}
```

21.7.2 The default value for this type is always "`field-to-be-set`".

21.7.3 An encoding property 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 property of type "`EncodingSpaceDetermination`" in the encoding of repetitions.

21.7.4 The value "`field-to-be-set`" requires the specification of a `REFERENCE` to a field that will be set by the encoder to carry information which identifies the size of the repetition space. 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. If a field is set more than once through the use of "`field-to-be-set`" or "`flag-to-be-set`", then it is an ECN specification or an application error if different values are produced by the different encoding procedures, and encoders shall not generate encodings in this case.

21.7.5 The value "`field-to-be-used`" requires the specification of a `REFERENCE` to a field whose value may be set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification) or may be set by some other encoder actions invoked by "`field-to-be-set`" or "`flag-to-be-set`". 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 "`flag-to-be-set`" requires the specification of a **`REFERENCE`** to a field that is part of the repeated element, and that will be set by the encoder to identify the last element of the repetition. The encoding specification determines how an encoder is to set the value of this field from a boolean value which is false if the element is the last in the repetition, and is true otherwise. If a field is set more than once through the use of "`flag-to-be-set`" or "`field-to-be-set`", then it is an ECN specification or an application error if different values are produced by the different encoding procedures, and encoders shall not generate encodings in this case.

21.7.7 The value "`flag-to-be-used`" requires the specification of a **`REFERENCE`** to a field that is part of the repeated element and whose value may be set from the abstract syntax (i.e., a corresponding field appears within the ASN.1 specification) or may be set by some other encoder actions invoked by "`flag-to-be-set`" or "`field-to-be-set`". The encoding specification determines how a decoder is to obtain a boolean value from the value of this field. The boolean value will be false if the element is the last element in the repetition, and true otherwise. A conforming encoder shall not produce encodings in which the decoder's transforms of this field do not correctly identify the last element of the repetition.

21.7.8 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 – This specification 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.

21.7.9 The value "`pattern`" specifies that some specified pattern of bits (see 21.10) will terminate the repetitions. In this case additional encoding properties 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.

21.7.10 The value "`handle`" requires that an identification handle be specified. This identification handle shall be exhibited both by the encoding object applied to the component being repeated, and by the encoding object applied to each possible (taking account of optionality) following encoding class. The handle value sets specified by those encoding objects shall all be disjoint.

NOTE – Every abstract value of a given component is required to have a handle value matching the handle value set of the component (see 22.9.2.2).

21.7.11 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 property 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 properties 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 property of this type is always "**`zero`**".

21.9.3 An encoding property 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 bits.

21.9.5 If the value is "**`one`**", then the padding is with one bits.

21.9.6 If the value is "**`pattern`**" then the bits are set according to the encoding property 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                INTEGER (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 property 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 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 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 value as a 32-bit value.

21.10.8 The "**`any-of-length`**" alternative specifies a size for the pattern. The actual value of the pattern is an encoder's option.

21.10.9 The "**`different:any`**" value is permitted only when there is another encoding property of type "**`Pattern`**" in the same encoding property group. In this case, either (but not both) of the encoding properties 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 property. 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 property.

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,
        test-lower-bound,
        test-upper-bound,
        test-range}
```

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

21.11.3 An encoding property 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 of the first five enumeration values of 21.11.1 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.

21.11.5 If the last three enumeration values of 21.11.1 are used, a value of the "`Comparison`" type (see 21.12) shall be provided, together with an integer `comparator` value. If the other enumeration values are used, these shall not be provided.

21.12 The `Comparison` type

21.12.1 The "`Comparison`" type is:

```
Comparison ::= ENUMERATED
      {equal-to,
       not-equal-to,
       greater-than,
       less-than,
       greater-than-or-equal-to,
       less-than-or-equal-to}
```

21.12.2 There is no default value for an encoding property of this type.

21.12.3 An encoding property of type "`Comparison`" is used to test an identified property of a class against an integer value (the `comparator`).

21.12.4 The predicate using a "`Comparison`" is satisfied for each enumeration value if and only if the identified property satisfies the following conditions:

a) `equal-to`: its value equals that of the specified integer `comparator` value.

b) `not-equal-to`: its value is different from that of the specified integer `comparator` value.

c) `greater-than`: its value is greater than that of the specified integer `comparator` value.

d) `less-than`: its value is less than that of the specified integer `comparator` value.

e) `greater-than-or-equal-to`: its value is greater than or equal to that of the specified integer `comparator` value.

f) **less-than-or-equal-to**: its value is less than or equal to that of the specified integer **comparator** value.

21.13 The SizeRangeCondition type

21.13.1 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,
        test-lower-bound,
        test-upper-bound,
        test-range}
```

21.13.2 The default value for an encoding property of this type is always "**no-ub-with-zero-lb**".

21.13.3 An encoding property 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.

21.13.4 The predicate is satisfied for each of the first five enumeration values of 21.13.1 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.

21.13.5 If the last three enumeration values of 21.13.1 are used, a value of the "**Comparison**" type (see 21.12) shall be provided, together with an integer **comparator** value. If the other enumeration values are used, these shall not be provided.

21.14 The ReversalSpecification type

21.14.1 The "**ReversalSpecification**" type is:

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

21.14.2 The default value for an encoding property of this type is always "**no-reversal**".

21.14.3 An encoding property 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.

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

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

21.14.6 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.

21.14.7 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`**".

21.14.8 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.

21.15 The `ResultSize` type

21.15.1 The "**`ResultSize`**" type is:

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

21.15.2 The default value for an encoding property of this type is always "**`variable`**".

21.15.3 An encoding property of this type specifies the size of the result in a **`#TRANSFORM`** class.

21.15.4 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.

21.15.5 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 this specification are defined for each transform in which values of this type are used.

21.15.6 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.

21.16 The `HandleValueSet` type

21.16.1 The "**`HandleValueSet`**" type is:

```
HandleValueSet ::= CHOICE {
      bits        BIT  STRING,
      octets      OCTET STRING,
      number      INTEGER (0..MAX),
      tag         ENUMERATED {any},
      range       SEQUENCE {
                        low   INTEGER(0..MAX),
                        high  INTEGER(0..MAX) },
      ranges      SET (SIZE(1..MAX)) OF SEQUENCE {
                        low   INTEGER(0..MAX),
                        high  INTEGER(0..MAX) }}
```

21.16.2 The "**`HandleValueSet`**" is used to specify the set of bit patterns (the handle value set) characterizing the encodings produced by an encoding object that exhibits an identification handle.

21.16.3 The value of an identification handle can be used to identify the presence or absence of optional components, the choice of alternatives, the ordering of sets, or the end of a repetition. There are requirements in such circumstances that the handle value sets of the encoding objects applied to the different alternatives or components be all disjoint (see 21.5.7, 21.6.6, 21.7.10, and 22.10.2.1), and requirements that all the possible values of the identification handle occurring in the encodings of any given alternative or component all match the specified handle value set of the encoding object applied to that alternative or component (see 22.9.2.2).

NOTE – The ECN specifier is required to specify the handle value set in all cases except where (for encodings of the tag class) the handle value set consists of a single value and depends on the tag number associated with that tag class, either directly through implicit generation from an ASN.1 tag, or by mapping from an implicitly generated structure.

21.16.4 The "**`bits`**", "**`octets`**" and "**`number`**" alternatives specify a handle value as a bitstring, octetstring or integer value respectively. It is an ECN specification error if this value cannot be encoded within the number of bits specified for the identification handle (see 22.9).

21.16.5 The "**`tag:any`**" alternative specifies a handle value determined by the number specified in an ECN encoding structure for a class in the tag category, or by the tag number mapped from an ASN.1 tag construction. It shall only be used when specifying the handle identification for the encoding of a class in the tag category.

21.16.6 The "**`range`**" alternative specifies a range of integer values, with **`high`** greater than or equal to **`low`**.

21.16.7 The "**`ranges`**" alternative specifies a set of ranges of integer values, each with **`high`** greater than or equal to **`low`**. One or more such ranges can be specified, and they shall not overlap.

21.17 The `IntegerMapping` type

21.17.1 The "`IntegerMapping`" type is:

```
IntegerMapping ::= SET OF SEQUENCE {
        source  SET OF INTEGER,
        result INTEGER} (CONSTRAINED BY {/* the intersection of the source
                                            components shall be empty */})
```

21.17.2 The "`IntegerMapping`" is used to specify explicitly an ints-to-ints transform.

22 Commonly used encoding property groups

This clause specifies groups of encoding properties that are commonly used in the defined syntax (see clause 20). The purpose of each group, the restrictions on both the values of encoding properties 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 properties, syntax and purpose

22.1.1.1 Full replacement specification uses the following encoding properties:

```
&#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 properties:

```
&#Replacement-structure                                              OPTIONAL,
&replacement-structure-encoding-object  &#Replacement-structure     OPTIONAL,
&#Head-end-structure                                                 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 properties:

```
          &#Replacement-structure                                     OPTIONAL,
          &replacement-structure-encoding-object  &#Replacement-structure    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 property groups specifies that either:

a) 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 component 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

b) 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

c) all its optional components (only for full replacement specification) are to be replaced ("**REPLACE OPTIONALS**"); or

d) all its non-optional components (only for full replacement specification) are to be replaced ("**REPLACE NON-OPTIONALS**"); or

e) 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 textual order of the original components.

22.1.1.11 In a full replacement specification, if the encoding object applied to the replacement structure exhibits an identification handle (with a given handle value set), then the encoding object whose defined syntax contains the full replacement specification exhibits the same identification handle (with the same handle value set), otherwise it does not exhibit a handle.

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, only the class reference name 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 be defined in a parameterized encoding object assignment in which the governor is 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 and shall have an "**`ENCODED BY`**" clause.

22.1.3 Encoder actions

22.1.3.1 If an encoding object of a class in the bit-field group of categories or in the tag category 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 numbers 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 properties, syntax and purpose

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

```
&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 properties 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 properties 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 up to the beginning of the container, see 22.2.1.4) is a multiple of the encoding property 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 property 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 property 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 properties, syntax and purpose

22.3.1.1 Start pointer specification uses the following encoding properties:

```
&start-pointer                      REFERENCE     OPTIONAL,
&start-pointer-unit                 Unit (ALL EXCEPT repetitions) DEFAULT bit,
&Start-pointer-encoder-transforms  #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 This specification 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 field referenced by the "**`START-POINTER`**" encoding property 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 field referenced by the "**`START-POINTER`**" encoding property 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 field referenced by the "**`START-POINTER`**" encoding property 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 It is an ECN specification or application error if any transform in the "**`ENCODER-TRANSFORMS`**" is not reversible for the abstract value to which it is applied. 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 properties, syntax and purpose

22.4.1.1 Encoding space specification uses the following encoding properties:

```
&encoding-space-size              EncodingSpaceSize
                                  DEFAULT self-delimiting-values,
&encoding-space-unit              Unit (ALL EXCEPT repetitions)
                                  DEFAULT bit,
&encoding-space-determination     EncodingSpaceDetermination
                                  DEFAULT field-to-be-set,
&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 this specification is:

```
EncodingSpaceSize ::= INTEGER
      { encoder-option-with-determinant(-3),
        variable-with-determinant(-2),
        self-delimiting-values(-1),
        fixed-to-max(0)} (-3..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
      {field-to-be-set, field-to-be-used, container} -- (see 21.3)
```

22.4.1.4 The purpose of this specification 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 properties 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 ("**`field-to-be-set`**") is assumed.

22.4.2.2 "**`USING`**" shall be specified if and only if "**`SIZE`**" is "**`variable-with-determinant`**" or "**`encoder-option-with-determinant`**".

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

22.4.2.4 It is an ECN specification or application error if any transform in the "**ENCODER-TRANSFORMS**" is not reversible for the abstract value to which it is applied. 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 "**field-to-be-used**". 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**" encoding property, if present, shall be a reference to 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 field referenced by the "**USING**" encoding property 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 This specification 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 properties 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 "**field-to-be-set**", 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 "**field-to-be-used**" 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) "**`field-to-be-set`**", 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 "**`field-to-be-used`**" 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 properties, syntax and purpose

22.5.1.1 Optionality determination uses the following encoding properties:

```
&optionality-determination        OptionalityDetermination
                                  DEFAULT field-to-be-set,
&optionality-reference            REFERENCE OPTIONAL,
&Encoder-transforms               #TRANSFORM ORDERED OPTIONAL,
&Decoder-transforms               #TRANSFORM ORDERED OPTIONAL,
&handle-id                        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
      {field-to-be-set, field-to-be-used, container, handle, pointer} -- (see 21.5)
```

22.5.1.4 The purpose of this specification is to specify rules that ensure that a decoder can correctly determine whether an encoder has encoded a value of an optional component. 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 component if required to do so by the application, unless such an encoding would be in violation of rules governing the presence of optional components.

NOTE – An example of violation of such a rule would be where the presence of an (absent) optional component was to be determined by the end of a container, and the application requested that later optional components in the same container be encoded.

22.5.1.6 This specification 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 ("**`field-to-be-set`**") 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 "**`DETERMINED BY`**" is "**`handle`**", then 21.5.7 applies.

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

22.5.2.7 It is an ECN specification or application error if any transform in the "**`ENCODER-TRANSFORMS`**" is not reversible for the abstract value to which it is applied. 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 "**`field-to-be-used`**". 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`**" encoding property, if present, shall be a reference to 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 field referenced by the "**`USING`**" encoding property 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 component 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 component 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 component to be encoded, and shall create a conceptual boolean value "**`element-is-present`**" set to "**`TRUE`**" if a value of the component is to be encoded, and to "**`FALSE`**" otherwise.

22.5.3.3 If "**`DETERMINED BY`**" is "**`field-to-be-set`**", 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 "**`field-to-be-used`**" 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 components in the "**`USING`**" container when the conceptual value "**`element-is-present`**" is false for this optional component.

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) "**`field-to-be-set`**", 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 "**`field-to-be-used`**" 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 specified identification handle. If the value matches the handle value set of the optional component, 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 component, otherwise the decoder expects an encoding of a value of the optional component and will diagnose an encoding error if one is not present.

22.6 Alternative determination

22.6.1 Encoding properties, syntax and purpose

22.6.1.1 Alternative determination uses the following encoding properties:

```
&alternative-determination      AlternativeDetermination
                                DEFAULT field-to-be-set,
&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
```

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 {field-to-be-set, field-to-be-used, handle} -- (see 21.6)
```

22.6.1.4 The purpose of this specification 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 ("`field-to-be-set`") 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 "`DETERMINED BY`" is "`handle`", then 21.6.6 applies.

22.6.2.5 "`ENCODER-TRANSFORMS`" shall be present only if "`DETERMINED BY`" is set to (or defaults to) "`field-to-be-set`". 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 It is an ECN specification or application error if any transform in the "`ENCODER-TRANSFORMS`" is not reversible for the abstract value to which it is applied.

22.6.2.7 "`DECODER-TRANSFORMS`" shall be present only if "`DETERMINED BY`" is set to "`field-to-be-used`". 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`" encoding property, if present, shall be a reference to 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 field referenced by the "`USING`" encoding property is required by a decoder but is absent (through the exercise of optionality).

22.6.2.9 This specification 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 "`field-to-be-set`", 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 "`field-to-be-used`" 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) "`field-to-be-set`", 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 "`field-to-be-used`" 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 identification handle. This value shall be compared to the handle value set of 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 properties, syntax and purpose

22.7.1.1 Repetition space specification uses the following encoding properties:

```
&repetition-space-size            EncodingSpaceSize
                                  DEFAULT self-delimiting-values,
&repetition-space-unit            Unit
                                  DEFAULT bit,
&repetition-space-determination   RepetitionSpaceDetermination
                                  DEFAULT field-to-be-set,
&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 bits '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 this specification is:

```
EncodingSpaceSize ::= INTEGER
      { encoder-option-with-determinant(-3),
        variable-with-determinant(-2),
        self-delimiting-values(-1),
        fixed-to-max(0)} (-3..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
      {field-to-be-set, field-to-be-used, flag-to-be-set, flag-to-be-used,
       container, pattern, handle, not-needed} -- (see 21.7)

Non-Null-Pattern ::= Pattern
      (ALL EXCEPT (bits:''B | octets:''H | char8:"" | char16:"" |
                   char32:"")) -- (see 21.10.2)
```

22.7.1.4 The purpose of this specification 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 properties 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 ("**field-to-be-set**") is assumed.

22.7.2.2 "**USING**" shall be specified if and only if "**SIZE**" is "**variable-with-determinant**" and "**DETERMINED BY**" is "**field-to-be-set**" or "**field-to-be-used**" or "**flag-to-be-set**" or "**flag-to-be-used**", or "**container**".

22.7.2.3 "**ENCODER-TRANSFORMS**" shall be present only if "**DETERMINED BY**" is set to (or defaults to) "**field-to-be-set**" or "**flag-to-be-set**". The first transform shall have a source which is integer if the "**DETERMINED BY**" is "**field-to-be-set**" and which is boolean if the "**DETERMINED BY**" is "**flag-to-be-set**". The last transform shall have a result which can be encoded by the class of the field referenced by "**USING**".

22.7.2.4 It is an ECN specification or application error if any transform in the "**ENCODER-TRANSFORMS**" is not reversible for the abstract value to which it is applied.

22.7.2.5 "**DECODER-TRANSFORMS**" shall be present only if "**DETERMINED BY**" is set to "**field-to-be-used**" or "**flag-to-be-used**". The first transform shall have a source which is the same as the category of the field referenced by "**USING**". The last transform shall have a result which is integer if the "**DETERMINED BY**" is "**field-to-be-used**" and which is boolean if the "**DETERMINED BY**" is "**flag-to-be-used**".

22.7.2.6 The "**USING**" encoding property, if present, for a "**field-to-be-set**" or a "**field-to-be-used**" shall be a reference to 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 repetition being encoded is present but the field referenced by the "**USING**" encoding property is absent (through the exercise of optionality).

22.7.2.7 The "**USING**" encoding property, if present, for a "**flag-to-be-set**" or a "**flag-to-be-used**" shall be a reference to a field that is present in the repeated element of a repetition. It is an application or an ECN specification error if, in an instance of encoding, the field referenced by the "**USING**" encoding property is absent (through the exercise of optionality) from any of the repeated elements.

NOTE – The requirement that the referenced field be present in an element of the repetition is satisfied if it is an identifier that is visible in accordance with 17.5 (encode structure), 19.3 (mapping by matching fields), 19.6 (mapping by value distribution), or if it is textually present in the definition of a replacement structure when "**REPLACE COMPONENT**" is used by an encoding object of a class in the repetition category.

22.7.2.8 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.

22.7.2.9 "**HANDLE**" shall be specified only if "**SIZE**" is "**variable-with-determinant**" and "**DETERMINED BY**" is "**handle**".

22.7.2.10 If "**DETERMINED BY**" is "**handle**", then 21.7.10 applies.

22.7.2.11 "**PATTERN**" shall be specified only if "**SIZE**" is "variable-with-determinant" and "**DETERMINED BY**" is "**pattern**".

22.7.2.12 "**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.

22.7.2.13 This specification 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 subclauses 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 "**field-to-be-set**", 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 "**field-to-be-used**" 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 "**flag-to-be-set**", then the encoder shall apply (for each repeated element) the transforms specified by "**ENCODER-TRANSFORMS**" (if any) to a boolean value which is true for all elements except the last and is false for the last element. The result of the "**ENCODER-TRANSFORMS**" shall be encoded in the "**USING**" reference.

22.7.3.10 If "**DETERMINED BY**" is "**flag-to-be-used**" then the encoder shall check (for each repeated element) that the value in the "**USING**" reference when transformed by the "**DECODER-TRANSFORMS**" (if any) is a boolean value which is true for all elements except the last, and is false for the last element. It is an application error if this condition is not met, and encoding shall not proceed.

22.7.3.11 If "**DETERMINED BY**" is "**handle**" there is no further action needed by the encoder.

22.7.3.12 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 subclauses 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) "`field-to-be-set`", 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 "`field-to-be-used`" then the decoder shall recover the value "s" by applying the "`DECODER-TRANSFORMS`" (if any) to the value of the "`USING`" reference.

22.7.4.6 If "`DETERMINED BY`" is "`flag-to-be-set`", then the decoder shall recover a boolean value by applying the reversal of the "`ENCODER-TRANSFORMS`" (if any) to the value of the "`USING`" reference. The element is the last of the repetition if and only if the boolean value is false.

22.7.4.7 If "`DETERMINED BY`" is "`flag-to-be-used`" then the decoder shall recover a boolean value by applying the "`DECODER-TRANSFORMS`" (if any) to the value of the "`USING`" reference. The element is the last of the repetition if and only if the boolean value is false.

22.7.4.8 If "`DETERMINED BY`" is "`handle`", then the decoder shall determine the value of the identification handle and attempt to decode the following encoding (in parallel) as either a further occurrence of the repetition or as a following encoding class, using the value of the identification handle to distinguish these alternatives. If decoding succeeds for more than one of these or for none of these, it is an encoding or a specification error.

22.7.4.9 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 properties, syntax, and purpose

22.8.1.1 Value padding and justification uses the following encoding properties:

```
&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 field-to-be-set,
&unused-bits-reference              REFERENCE OPTIONAL,
&Unused-bits-encoder-transforms     #TRANSFORM ORDERED OPTIONAL,
&Unused-bits-decoder-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 &Unused-bits-encoder-transforms]
                  [DECODER-TRANSFORMS &Unused-bits-decoder-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
            {field-to-be-set, field-to-be-used, not-needed} -- (see 21.4)
```

22.8.1.4 The purpose of this specification 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) "**field-to-be-set**". 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 It is an ECN specification or application error if any transform in the "**ENCODER-TRANSFORMS**" is not reversible for the abstract value to which it is applied.

22.8.2.5 "**DECODER-TRANSFORMS**" shall be present only if "**DETERMINED BY**" is set to "**field-to-be-used**". 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**" encoding property, if present, shall be a reference to 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 field referenced by the "**USING**" encoding property is absent (through the exercise of optionality).

22.8.2.7 This specification is considered set if the "**VALUE-PADDING**" 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 This specification 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 "**field-to-be-set**", 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 "**field-to-be-used**" 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) "**field-to-be-set**", 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 "**field-to-be-used**" 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 properties, syntax and purpose

22.9.1.1 Identification handle specification uses the following encoding properties:

```
&exhibited-handle                 PrintableString DEFAULT "default-handle",
&Handle-positions                 INTEGER (0..MAX) OPTIONAL,
&handle-value-set                 HandleValueSet DEFAULT tag:any
```

22.9.1.2 The syntax to be used for identification handle specification shall be:

```
[EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
      [AS &handle-value-set]]
```

22.9.1.3 The definition of the type used in identification handle specification is:

```
HandleValueSet ::= CHOICE {
      bits        BIT  STRING,
      octets      OCTET STRING,
      number      INTEGER (0..MAX),
      tag         ENUMERATED {any},
      range       SEQUENCE {
                        low   INTEGER(0..MAX),
                        high  INTEGER(0..MAX) },
      ranges      SET (SIZE(1..MAX)) OF SEQUENCE {
                        low   INTEGER(0..MAX),
                        high  INTEGER(0..MAX) }}  -- (see 21.16)
```

22.9.1.4 The purpose of this specification is to declare that an encoding object exhibits an identification handle and to specify its properties, which are:

a) the name of the handle;

b) the bit positions that form the handle; and

c) the possible bit patterns (for the bit positions forming the handle) occurring in the encodings produced by this encoding object (the handle value set).

22.9.1.5 The list of positions in "**AT**" shall be the positions of the bits forming the identification handle in the final encoding, after any pre-alignment has been applied, and 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.

NOTE – This means that a decoder needs to perform any bit-reversals specified in **#OUTER** for the entire PDU, but otherwise examines the bit-positions and their values without any consideration of possible bit-reversals that may be specified for particular encoding objects.

22.9.1.6 The list of positions in "**AT**" is a set of integer values (not necessarily contiguous, and not necessarily in ascending order in the ECN specification). These positions shall be ordered by encoders and decoders from the zero position (the first bit in that part of the encoding that is exhibiting the handle) upwards, and the bits in those positions form a conceptual handle field.

22.9.1.7 For a "`number`" value of "`HandleValueSet`" or the encoding of a tag number, the bit in the conceptual handle field nearest to the zero position is the high-order bit, and the "`number`" or tag number that specifies the "`HandleValueSet`" is right-justified within this field. If the "`number`" or tag number is too large for the field, this is an ECN specification error.

22.9.1.8 If the "`bitstring`" or "`octetstring`" alternatives of "`HandleValueSet`" are used, then their values shall have the same number of bits as those specified for the identification handle by "`AT`". The bit in the conceptual handle field nearest to the zero position is the leading bit of the "`bitstring`" or "`octetstring`" that specifies the "`HandleValueSet`".

22.9.1.9 The "`HandleValueSet`" shall not be specified as "`tag:any`" unless the specification is for an encoding object of the **`#TAG`** class. In this case the value of the identification handle is determined by either the tag number in the ECN specification or by the tag number mapped from an ASN.1 tag (as specified in clause 19), and need not be specified using "`HandleValueSet`". If, however, a value is specified by "`HandleValueSet`" and differs from that assigned in an ECN specification of a tag class or in an ASN.1 tag that maps to an ECN tag, that is an ECN specification error.

22.9.2 Specification constraints

22.9.2.1 In any ECN specification, all identification handles with the same name shall specify the same set of bit positions.

NOTE – There is no general requirement that the handle value sets of different encoding objects defined in an ECN specification be all disjoint, but disjoint handle value sets are required when the identification handle is used to resolve optionality, alternative selection, repetition termination, or ordering of set (see 21.5.7, 21.6.6, 21.7.10, and 22.10.2.1).

22.9.2.2 For an encoding object that exhibits an identification handle (with a given handle value set), the value of the identification handle occurring in each of the possible encodings produced by that encoding object (for all possible abstract values) shall be a member of the specified handle value set.

22.9.2.3 All encoding objects that exhibit the same identification handle shall either have no pre-alignment specification, or shall align to the same pre-alignment unit.

NOTE – This restriction is imposed so that decoders can move to the alignment position before looking for the handle when the decoding depends on a handle value.

22.9.2.4 This specification 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 value of the identification handle occurring in the encoding produced is a member of the specified handle value set, 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 properties, syntax and purpose

22.10.1.1 Concatenation specification uses the following encoding properties:

```
&concatenation-order          ENUMERATED {textual, tag, random}
                              DEFAULT textual,
&concatenation-alignment      ENUMERATED {none, aligned}
                              DEFAULT aligned,
&concatenation-handle         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 This specification determines the order in which the components 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 the encoding objects applied to all components shall exhibit that identification handle. The handle value sets of those encoding objects shall all be disjoint.

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 This specification 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 property taking its default value.

22.10.2.7 If (through the exercise of optionality) there is at least one abstract value of a concatenation that has no bits in its encoding, then the concatenation shall have no pre-alignment.

NOTE – This subclause will apply if a concatenation has no mandatory components, or if all its mandatory components can have (through the exercise of optionality) no bits in their encodings.

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 identification handle.

22.10.4.4 Decoding shall proceed until an abstract value for every component has been obtained, and a decoder 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 this Recommendation | International Standard, and such occurrences shall be treated as encoder errors.

22.11 Contained type encoding specification

22.11.1 Encoding properties, syntax and purpose

22.11.1.1 The contained type encoding specification uses the following encoding properties:

```
&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 this specification 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 This specification 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 This specification 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

22.11.3.1A 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 properties, syntax, and purpose

22.12.1.1 Bit reversal specification uses the following encoding property:

```
&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 21.14)
```

22.12.1.4 The purpose of this specification 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 This specification 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 This specification 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 property 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 bit-field 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 an encoding property 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-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,

      -- Alternative determination (see 22.6)
      &alternative-determination         AlternativeDetermination
                                         DEFAULT field-to-be-set,
      &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 DEFAULT "default-handle",
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value-set                  HandleValueSet DEFAULT tag:any

} 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]]
        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
              [AS &handle-value-set]]

}
```

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 the encoding object exhibits a specified identification handle (with a given handle value set).

23.1.2.2 If "**REPLACE STRUCTURE**" is set, then no other encoding property groups shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.1.2.3 An encoding object of this class does not exhibit an identification handle unless "**EXHIBITS HANDLE**" is set (even if the components of the defined construction exhibit an identification handle) or unless "**REPLACE STRUCTURE**" is set and the encoding object of the replacement structure exhibits an identification handle (see 22.1.1.11).

23.1.2.4 If "**EXHIBITS HANDLE**" is set, then the encoding object exhibits the specified identification handle.

23.1.3 Encoder actions

23.1.3.1 For any encoding property 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 property 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-unitUnit (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,
        &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 DEFAULT "default-handle",
        &Handle-positions                 INTEGER (0..MAX) OPTIONAL,
        &handle-value-set                 HandleValueSet DEFAULT tag:any,

        -- 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]
        [TRANSFORMS                  &Transforms]
        [REPETITION-ENCODINGS        &Bits-repetition-encodings]
        [REPETITION-ENCODING         &bits-repetition-encoding]
        [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
              [AS &handle-value-set]]
        [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 "**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 "**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 the encoding object exhibits a specified identification handle (with a given handle value set), and a specification of how to encode a contained type.

23.2.3.2 The **#CONDITIONAL-REPETITION** that is applied by this encoding object shall not specify "**REPLACE**" unless it is "**REPLACE STRUCTURE**".

23.2.3.3 If any of the **#CONDITIONAL-REPETITION** encoding objects contain a "**REPLACE STRUCTURE**" clause, then all of the **#CONDITIONAL-REPETITION** encoding objects shall contain a "**REPLACE STRUCTURE**" clause.

23.2.3.4 If there is a "**REPLACE STRUCTURE**" clause in the **#CONDITIONAL-REPETITION** encoding objects, then no other parameters shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.2.3.5 The first transform in "**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 3.2.42).

NOTE – This means that the final transform is required to be self-delimiting.

23.2.3.6 It is an ECN specification or application error if any transform in the "**TRANSFORMS**" is not reversible for the abstract value to which it is applied.

23.2.3.7 Exactly one of "**REPETITION-ENCODING**" and "**REPETITION-ENCODINGS**" shall be set.

23.2.3.8 If an encoding object in the "**REPETITION-ENCODINGS**" ordered list is defined using "**IF**" or "**IF-ALL**", then all preceding encoding objects in that list shall be defined using "**IF**" or "**IF-ALL**".

23.2.3.9 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.

23.2.3.10 If "**EXHIBITS HANDLE**" is set, then the encoding object exhibits 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.

23.2.3.11 If "**EXHIBITS HANDLE**" is set, then "**ALIGNED TO**" shall not be set in any of the "**REPETITION-ENCODING(S)**" specifications.

23.2.4 Encoder actions

23.2.4.1 For any encoding property 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 "**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 property 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 "**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 field-to-be-set,
      &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 field-to-be-set,
      &unused-bits-reference             REFERENCE OPTIONAL,
      &Unused-bits-encoder-transforms    #TRANSFORM ORDERED OPTIONAL,
      &Unused-bits-decoder-transforms    #TRANSFORM ORDERED OPTIONAL,

      -- Identification handle specification (see 22.9)
```

```
        &exhibited-handle                  PrintableString DEFAULT "default-handle",
        &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
        &handle-value-set                  HandleValueSet DEFAULT tag:any,

        -- 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 &Unused-bits-encoder-transforms]
                          [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
        [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
              [AS &handle-value-set]]
        [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 the encoding object exhibits a specified identification handle (with a given handle value set), and possible bit-reversal of the encoding space for the boolean.

23.3.2.2 If "**REPLACE**" is set, then no other encoding property 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 3.2.42).

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 3.2.42).

23.3.2.7 If there are any unused bits in the encoding space, then "**VALUE-PADDING**" shall be set.

23.3.3 Encoder actions

23.3.3.1 For any encoding property 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 "**field-to-be-set**" or "**field-to-be-used**" 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-PADDING**" shall be applied.

23.3.4 Decoder actions

23.3.4.1 For any encoding property 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,
        &Transforms                        #TRANSFORM ORDERED OPTIONAL,
        &Chars-repetition-encodings        #CONDITIONAL-REPETITION ORDERED OPTIONAL,
        &chars-repetition-encoding         #CONDITIONAL-REPETITION OPTIONAL,

        -- Identification handle specification (see 22.9)
        &exhibited-handle                  PrintableString DEFAULT "default-handle",
        &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
        &handle-value-set                  HandleValueSet DEFAULT tag:any

    } 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]
        [TRANSFORMS              &Transforms]
        [REPETITION-ENCODINGS    &Chars-repetition-encodings]
        [REPETITION-ENCODING     &chars-repetition-encoding]
        [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
            [AS &handle-value-set]]

    }
```

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 transform (specified by "**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 **#CONDITIONAL-REPETITION** 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 the encoding object exhibits a specified identification handle (with a given handle value set).

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 If any of the **#CONDITIONAL-REPETITION** encoding objects contain a "**REPLACE STRUCTURE**" clause, then all of the **#CONDITIONAL-REPETITION** encoding objects shall contain a "**REPLACE STRUCTURE**" clause.

23.4.3.4 If there is no "**REPLACE STRUCTURE**" clause in the **#CONDITIONAL-REPETITION** encoding objects, then "**TRANSFORMS**" shall be set. If there is a "**REPLACE STRUCTURE**" clause in the **#CONDITIONAL-REPETITION** encoding objects, then no other parameters shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.4.3.5 The first transform of "**TRANSFORMS**" 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 3.2.42).

NOTE – This means that the final transform is required to be self-delimiting.

23.4.3.6 It is an ECN specification or application error if any transform in the "**TRANSFORMS**" is not reversible for the abstract value to which it is applied.

23.4.3.7 Exactly one of "**REPETITION-ENCODING**" and "**REPETITION-ENCODINGS**" shall be set.

23.4.3.8 If an encoding object in the "**REPETITION-ENCODINGS**" ordered list is defined using "**IF**" or "**IF-ALL**", then all preceding encoding objects in that list shall be defined using "**IF**" or "**IF-ALL**".

23.4.3.9 If "**EXHIBITS HANDLE**" is set, then the encoding object exhibits 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.

23.4.3.10 If "**EXHIBITS HANDLE**" is set, then "**ALIGNED TO**" shall not be set in any of the "**REPETITION-ENCODING(S)**" specifications.

23.4.4 Encoder actions

23.4.4.1 For any encoding property 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 "**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 property 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 "**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 field-to-be-set,
        &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 field-to-be-set,
        &unused-bits-reference             REFERENCE OPTIONAL,
        &Unused-bits-encoder-transforms    #TRANSFORM ORDERED OPTIONAL,
        &Unused-bits-decoder-transforms    #TRANSFORM ORDERED OPTIONAL,

        -- Identification handle specification (see 22.9)
        &exhibited-handle                  PrintableString DEFAULT "default-handle",
        &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
        &handle-value-set                  HandleValueSet DEFAULT tag:any,

        -- 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 &Unused-bits-encoder-transforms]
                          [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
        [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
              [AS &handle-value-set]]
        [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 the encoding object exhibits a specified identification handle (with a given handle value set), and possible bit-reversal of the encoding space.

23.5.2.2 If "`REPLACE STRUCTURE`" is set, then no other encoding parameter groups shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

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 object exhibits the specified identification handle.

23.5.3 Encoder actions

23.5.3.1 For any encoding property 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 property 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**" ordered list is defined using "**IF**" or "**IF-ALL**", then all preceding encoding objects in that list shall be defined using "**IF**" or "**IF-ALL**".

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,
    &comparison                             Comparison OPTIONAL,
    &comparator                             INTEGER OPTIONAL,
    &Range-conditions                       RangeCondition ORDERED OPTIONAL,
    &Comparisons                            Comparison ORDERED OPTIONAL,
    &Comparators                            INTEGER ORDERED 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 field-to-be-set,
    &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 field-to-be-set,
    &unused-bits-reference             REFERENCE OPTIONAL,
    &Unused-bits-encoder-transforms    #TRANSFORM ORDERED OPTIONAL,
    &Unused-bits-decoder-transforms    #TRANSFORM ORDERED OPTIONAL,

    -- Identification handle specification (see 22.9)
    &exhibited-handle                  PrintableString DEFAULT "default-handle",
    &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
    &handle-value-set                  HandleValueSet DEFAULT tag:any,

    -- Bit reversal specification (see 22.12)
    &bit-reversal                      ReversalSpecification
                                       DEFAULT no-reversal

} WITH SYNTAX {
    [IF &range-condition [&comparison  &comparator]]
    [IF-ALL &Range-conditions [&Comparisons &Comparators]]
    [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 &Unused-bits-encoder-transforms]
                            [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
          [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
                [AS &handle-value-set]]
          [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 all of a set of conditions are to be satisfied (use of "**IF-ALL**"). It also allows the specification that there is no condition. The use of "**ELSE**", or omission of "**IF**", "**IF-ALL**" and "**ELSE**" specifies that there is no condition. "**IF-ALL**" shall be used with three lists if one or more of the size-range-conditions require a comparison, and shall be used with one list otherwise. When using three lists, size-range-conditions that do not require a comparison or comparator (if any) shall follow all those that require a comparison, and shall have no corresponding entry in the second and third lists. In using "**IF-ALL**" with three lists, the lists shall be interpreted as a list of predicates using the values in corresponding positions in the three lists.

NOTE – It is recommended that the three lists be formatted to provide a condition in each column.

EXAMPLE:

```
    IF-ALL     {test-lower-bound, test-range     , bounded-with-negatives }
          {greater-than    , less-than-or-equal-to }
          {-10             , 20                    }
```

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**", "**IF-ALL**" and "**ELSE**" shall be present.

23.7.2.5 If "**REPLACE**" is set, then no other encoding property groups shall be set.

23.7.2.6 It is an ECN specification or application error if any transform in the "**TRANSFORMS**" is not reversible for the abstract value to which it is applied. 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.

NOTE – The test for the "**IF**" and "**IF-ALL**" 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**" or "**IF-ALL**" 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`**" or "**`IF-ALL`**" 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`**".

NOTE – This means that the default value (which is set for consistency with other uses of this type) always has to be overridden.

23.7.2.10 If "**`EXHIBITS HANDLE`**" is set, then the encoding object exhibits the specified 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 property 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 Rec. ITU-T 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 "**`field-to-be-set`**" or "**`field-to-be-used`**" 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 property 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-unitUnit (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 field-to-be-set,
      &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 field-to-be-set,
      &unused-bits-reference            REFERENCE OPTIONAL,
      &Unused-bits-encoder-transforms   #TRANSFORM ORDERED OPTIONAL,
      &Unused-bits-decoder-transforms   #TRANSFORM ORDERED OPTIONAL,

      -- Identification handle specification (see 22.9)
      &exhibited-handle                 PrintableString DEFAULT "default-handle",
      &Handle-positions                 INTEGER (0..MAX) OPTIONAL,
      &handle-value-set                 HandleValueSet DEFAULT tag:any,
      -- 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]]
      [NULL-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 &Unused-bits-encoder-transforms]
                        [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
      [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
            [AS &handle-value-set]]
      [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 property groups shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.8.2.3 If the "**ENCODING-SPACE SIZE**" is positive, it shall be sufficient to hold the size of the "**NULL-PATTERN**" together with any bits added as a result of a "**VALUE-PADDING**" specification.

23.8.2.4 If there are unused bits in the encoding space, then "**VALUE-PADDING**" shall be set.

23.8.3 Encoder actions

23.8.3.1 For any encoding property 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 "**`NULL-PATTERN`**".

23.8.3.3 If "**`ENCODING-SPACE SIZE`**" is "**`variable-with-determinant`**" or "**`encoder-option-with-detereminant`**", 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 "**`field-to-be-set`**" or "**`field-to-be-used`**" imposes) if "**`ENCODING-SPACE SIZE`**" is set to "**`encoder-option-with-determinant`**".

23.8.3.5 If there are unused bits in the encoding space, then "**`VALUE-PADDING`**" shall be applied.

23.8.4 Decoder actions

23.8.4.1 For any encoding property 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-unitUnit (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,
      &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 DEFAULT "default-handle",
      &Handle-positions                 INTEGER (0..MAX) OPTIONAL,
      &handle-value-set                 HandleValueSet DEFAULT tag:any,

      -- 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]
        [TRANSFORMS              &Transforms]
        [REPETITION-ENCODINGS    &Octets-repetition-encodings]
        [REPETITION-ENCODING     &octets-repetition-encoding]
        [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
              [AS &handle-value-set]]
        [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 "**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 the encoding object exhibits a specified identification handle (with a given handle value set), a specification of how to encode a contained type.

23.9.3.2 The **#CONDITIONAL-REPETITION** that is applied by this encoding object shall not specify "**REPLACE**" unless it is "**REPLACE STRUCTURE**".

23.9.3.3 If any of the **#CONDITIONAL-REPETITION** encoding objects contain a "**REPLACE STRUCTURE**" clause, then all of the **#CONDITIONAL-REPETITION** encoding objects shall contain a "**REPLACE STRUCTURE**" clause.

23.9.3.4 If there is a "**REPLACE STRUCTURE**" clause in the **#CONDITIONAL-REPETITION** encoding objects, then no other parameters shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.9.3.5 The first transform of "**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 3.2.42).

23.9.3.6 It is an ECN specification or application error if any transform in the "**TRANSFORMS**" is not reversible for the abstract value to which it is applied.

23.9.3.7 Exactly one of "**REPETITION-ENCODING**" and "**REPETITION-ENCODINGS**" shall be set.

23.9.3.8 If an encoding object in the "**REPETITION-ENCODINGS**" ordered list is defined using "**IF**" or "**IF-ALL**", then all preceding encoding objects in that list shall be defined using "**IF**" or "**IF-ALL**".

23.9.3.9 If "**EXHIBITS HANDLE**" is set, then the encoding object exhibits the specified identification handle.

NOTE – This will in general require restrictions on the abstract values of the associated type.

23.9.3.10 If "**EXHIBITS HANDLE**" is set, then "**ALIGNED TO**" shall not be set in any of the "**REPETITION-ENCODING(S)**" specifications.

23.9.4 Encoder actions

23.9.4.1 For any encoding property 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 "**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 property 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 "**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 open type category

23.10.1 The defined syntax

The syntax for defining encoding objects for classes in the open type category is defined as:

```
#OPEN-TYPE ::= 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 field-to-be-set,
      &encoding-space-reference          REFERENCE OPTIONAL,
      &Encoder-transforms                #TRANSFORM ORDERED OPTIONAL,
      &Decoder-transforms                #TRANSFORM ORDERED OPTIONAL,

      -- Open-type encoding
      &Known-structure-encodings         #ENCODINGS OPTIONAL,
      &Unknown-structure                 OPTIONAL,
      &Unknown-structure-encodings       #ENCODINGS 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 field-to-be-set,
      &unused-bits-reference             REFERENCE OPTIONAL,
      &Unused-bits-encoder-transforms    #TRANSFORM ORDERED OPTIONAL,
      &Unused-bits-decoder-transforms    #TRANSFORM ORDERED 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]]
      [ENCODED WITH &Known-structure-encodings]
      [UNKNOWN IS &Unknown-structure
            [ENCODED WITH &Unknown-structure-encodings]]
      [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 &Unused-bits-encoder-transforms]
                        [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
      [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
            [AS &handle-value-set]]
      [BIT-REVERSAL &bit-reversal]

}
```

23.10.2 Model for the encoding of classes in the open type category

23.10.2.1 The model of open type encodings is:

a) The class in the open type category can be replaced by another structure to provide length delimitation if required.

b) The encoding object defined for this category applies the "**`ENCODED WITH`**" encoding object set to the type whose value is to be encoded for the open type. If there is no "**`ENCODED WITH`**", then the current combined encoding object set is used.

c) The decoder will request the application for identification of the type encoded into the open type. The application will either respond with identification of the type, which is then decoded, or will state that the type encoded in the open type cannot be determined (an "unknown" response).

d) If the response is "unknown" and the "**`UNKNOWN IS`**" is present, then the decoder will use the "**`UNKNOWN IS`**" structure and the "**`ENCODED WITH`**" within the "**`UNKNOWN IS`**" (if present) to determine the end of the encoding space.

e) If the response is "unknown" and the "**`UNKNOWN IS`**" is absent, then the encoding space size can be determined by the "**`ENCODING-SPACE`**" (see 23.10.3.3), and the decoder will return to the application all the bits contained in the defined encoding space except for value pre- and post-padding.

23.10.2.2 In the case of an unknown decoding, the decoder will pass the bits forming the unknown encoding to the application as the value of the open type.

23.10.3 Purpose and restrictions

23.10.3.1 This syntax is used to define the way an open type is encoded, and the means that a decoder uses to determine the end of the encoding of an unknown type in an open type.

23.10.3.2 If "**`REPLACE STRUCTURE`**" is set no other parameters shall be set.

23.10.3.3 If "**`ENCODING-SPACE SIZE`**" is "**`self-delimiting`**" then "**`UNKNOWN IS`**" shall be set.

23.10.4 Encoder actions

23.10.4.1 For any encoding property 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 23.10.4.3);

e) open-type encoding (see 23.10.4.2);

f) value padding and justification (see 23.10.4.5);

g) bit reversal.

23.10.4.2 The encoder shall encode the value of the type supplied by the application using the "**`ENCODED WITH`**" encoding object set if this is present, otherwise the current combined encoding object set shall be used.

23.10.4.3 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 23.10.4.5.

23.10.4.4 An encoder (as an encoder's option) may increase "s" (as determined in 23.10.4.3) 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`**".

23.10.4.5 If the number of unused bits is not zero, then "**`VALUE-JUSTIFICATION`**" shall be applied using either the set values or the default values.

23.10.5 Decoder actions

23.10.5.1 For any encoding property 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) open-type decoding (see 23.10.5.2).

23.10.5.2 For open type decoding, the decoder shall query the application for the type which has been encoded and shall decode a value of that type or of the "**UNKNOWN IS**" structure in accordance with the "**ENCODED WITH**" specifications in the "**UNKNOWN IS**".

23.10.5.3 If the decoding was of an unknown type, the bits forming the unknown encoding (without pre-padding bits and without value pre- and post-padding bits, if any) shall be passed to the application as the value of the open type.

23.11 Defining encoding objects for classes in the optionality category

23.11.1 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 field-to-be-set,
      &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]]

}
```

23.11.2 Purpose and restrictions

23.11.2.1 This syntax is used to define the encoding of a class in the optionality category.

23.11.2.2 If "**REPLACE STRUCTURE**" is set, then no other encoding property groups shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.11.3 Encoder actions

23.11.3.1 For any encoding property 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 23.11.3.2).

b) Pre-alignment and padding.

c) Start pointer.

d) Optionality determination.

23.11.3.2 If "**REPLACE STRUCTURE**" is set then the entire component (including any classes in the tag category, but excluding classes in the optionality category) is provided as the actual parameter for the replacement structure, which becomes a mandatory component.

23.11.4 Decoder actions

23.11.4.1 For any encoding property 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.

23.12 Defining encoding objects for classes in the pad category

23.12.1 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 field-to-be-set,
      &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 DEFAULT "default-handle",
      &Handle-positions                  INTEGER (0..MAX) OPTIONAL,
      &handle-value-set                  HandleValueSet DEFAULT tag:any,

      -- 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]]
      [PAD-PATTERN &pad-pattern]
      [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
            [AS &handle-value-set]]
      [BIT-REVERSAL &bit-reversal]

}
```

23.12.2 Purpose and restrictions

23.12.2.1 This syntax is used to define the encoding of a class in the pad category.

23.12.2.2 If "**ENCODING-SPACE SIZE**" is positive, "**PAD-PATTERN**" shall not be of zero length, and is replicated and truncated to fill the encoding space.

23.12.2.3 If "**REPLACE STRUCTURE**" is set, then no other encoding property group shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.12.3 Encoder actions

23.12.3.1 For any encoding property 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.

23.12.3.2 If "**ENCODING-SPACE SIZE**" is positive, the value shall be the "**PAD-PATTERN**", replicated and truncated to fill the encoding space.

23.12.3.3 "**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 "**PAD-PATTERN**" ("s", say), and the "**PAD-PATTERN**" shall then be replicated and truncated to fill that space (but see 23.12.3.4).

NOTE – This will be an empty encoding space if the "**PAD-PATTERN**" is null.

23.12.3.4 An encoder (as an encoder's option) may increase "s" (as determined in 23.12.3.3) in "**MULTIPLE OF**" units (subject to any restrictions that the range of values of any "**field-to-be-set**" or "**field-to-be-used**" imposes) if "**ENCODING-SPACE SIZE**" is set to "**encoder-option-with-determinant**".

23.12.4 Decoder actions

23.12.4.1 For any encoding property 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.

23.12.4.2 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.

23.13 Defining encoding objects for classes in the repetition category

23.13.1 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]

}
```

23.13.2 Purpose and restrictions

23.13.2.1 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.

23.13.2.2 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.

23.13.2.3 If an encoding object in the "**REPETITION-ENCODINGS**" ordered list is defined using "**IF**" or "**IF-ALL**", then all preceding encoding objects in that list shall be defined using "**IF**" or "**IF-ALL**".

23.13.3 Encoder actions

23.13.3.1 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.

23.13.4 Decoder actions

23.13.4.1 The decoder shall select and use the first **#CONDITIONAL-REPETITION** encoding object in "**ENCODING(S)**" whose conditions are satisfied.

23.14 Defining encoding objects for the #CONDITIONAL-REPETITION class

23.14.1 The defined syntax

The syntax for defining encoding objects for the **#CONDITIONAL- REPETITION** class is defined as:

```
#CONDITIONAL-REPETITION ::= ENCODING-CLASS {

      -- Condition (see 21.13)
      &size-range-condition             SizeRangeCondition  OPTIONAL,
      &comparison                       Comparison OPTIONAL,
      &comparator                       INTEGER OPTIONAL,
      &Size-range-conditions            SizeRangeCondition ORDERED OPTIONAL,
      &Comparisons                      Comparison ORDERED OPTIONAL,
      &Comparators                      INTEGER ORDERED 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-unitUnit (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 field-to-be-set,
      &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 bits '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 field-to-be-set,
      &unused-bits-reference            REFERENCE OPTIONAL,
      &Unused-bits-encoder-transforms   #TRANSFORM ORDERED OPTIONAL,
      &Unused-bits-decoder-transforms   #TRANSFORM ORDERED OPTIONAL,

      -- Identification handle specification (see 22.9)
      &exhibited-handle                 PrintableString DEFAULT "default-handle",
      &Handle-positions                 INTEGER (0..MAX) OPTIONAL,
      &handle-value-set                 HandleValueSet DEFAULT tag:any,

      -- Bit reversal specification (see 22.12)
      &bit-reversal                     ReversalSpecification
                                        DEFAULT no-reversal

} WITH SYNTAX {
      [IF &size-range-condition [&comparison  &comparator]]
      [IF-ALL &Size-range-conditions [&Comparisons &Comparators]]
      [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 &Unused-bits-encoder-transforms]
                          [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
        [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
              [AS &handle-value-set]]
        [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 repetition category subject to satisfaction of a condition based on the bounds of the repetition (use of "**IF**"). It also allows the specification that all of a set of conditions are to be satisfied (use of "**IF-ALL**"). It also allows the specification that there is no condition. The use of "**ELSE**", or omission of "**IF**", "**IF-ALL**" and "**ELSE**" specifies that there is no condition. "**IF-ALL**" shall be used with three lists if one or more of the size-range-conditions require a comparison, and shall be used with one list otherwise. When using three lists, size-range-conditions that do not require a comparison or comparator (if any) shall follow all those that require a comparison, and shall have no corresponding entry in the second and third lists. In using "**IF-ALL**" with three lists, the lists shall be interpreted as a list of predicates using the values in corresponding positions in the three lists.

NOTE – It is recommended that the three lists be formatted to provide a condition in each column (see the example in 23.7.2.2).

23.14.2.2 At most one of "**IF**", "**IF-ALL**" and "**ELSE**" shall be present.

23.14.2.3 If "**REPLACE STRUCTURE**" is set, then no other encoding property groups shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.14.2.4 If "**EXHIBITS HANDLE**" is set, then the encoding object exhibits the specified identification handle.

23.14.2.5 "**REPETITION-SPACE SIZE**" shall not be "**fixed-to-max**".

23.14.2.6 If the "**REPETITION-SPACE SIZE**" is "**self-delimiting-values**", and "**MULTIPLE OF**" is "**repetitions**", then the number of repetitions shall be constrained by bounds to a single value.

23.14.2.7 If there are any unused bits in the encoding space, then "**VALUE-PADDING**" shall be set.

23.14.3 Encoder actions

23.14.3.1 For any encoding property 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 23.14.3.4).

f) Value padding and justification.

g) Identification handle.

h) Bit reversal.

23.14.3.2 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.

23.14.3.3 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.

23.14.3.4 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 23.14.3.5).

23.14.3.5 An encoder (as an encoder's option) may increase "s" (as determined in 23.14.3.4) in "`MULTIPLE OF`" units (subject to any restrictions that the range of values of any "`field-to-be-set`" or "`field-to-be-used`" imposes) if "`ENCODING-SPACE SIZE`" is set to "`encoder-option-with-determinant`".

23.14.3.6 The repetition value is then placed in the encoding space, using "`VALUE-PADDING`" if there are any unused bits.

23.14.4 Decoder actions

23.14.4.1 For any encoding property 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 23.14.4.2).

23.14.4.2 Each repetition shall be extracted, and decoded in accordance with the encoding specification of the component of the repetition class.

23.15 Defining encoding objects for classes in the tag category

23.15.1 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 field-to-be-set,
      &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 field-to-be-set,
      &unused-bits-reference            REFERENCE OPTIONAL,
      &Unused-bits-encoder-transforms   #TRANSFORM ORDERED OPTIONAL,
      &Unused-bits-decoder-transforms   #TRANSFORM ORDERED OPTIONAL,

      -- Identification handle specification (see 22.9)
      &exhibited-handle                 PrintableString DEFAULT "default-handle",
      &Handle-positions                 INTEGER (0..MAX) OPTIONAL,
      &handle-value-set                 HandleValueSet DEFAULT tag:any,

      -- 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 &Unused-bits-encoder-transforms]
                        [DECODER-TRANSFORMS &Unused-bits-decoder-transforms]]]]
      [EXHIBITS HANDLE &exhibited-handle AT &Handle-positions
            [AS &handle-value-set]]
      [BIT-REVERSAL &bit-reversal]

}
```

23.15.2 Purpose and restrictions

23.15.2.1 This syntax is used to define the encoding of a class in the tag category.

23.15.2.2 If "**REPLACE STRUCTURE**" is set, then no other specifications shall be set. If the encoding object of the replacement structure exhibits a handle (with a given handle value set), the encoding object being defined exhibits the same identification handle (with the same handle value set – see 22.1.1.11).

23.15.2.3 The "**ENCODING-SPACE SIZE**" shall not be "**fixed-to-max**" or "**self-delimiting-values**".

NOTE – This means that the default value (which is set for consistency with other uses of this type) always has to be overridden.

23.15.3 Encoder actions

23.15.3.1 For any encoding property 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 23.15.3.3).

f) Value padding and justification.

g) Identification handle.

h) Bit reversal.

23.15.3.2 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.

23.15.3.3 The encoding shall be a positive integer encoding. The specification of a positive integer encoding is given in Rec. ITU-T X.690 | ISO/IEC 8825-1, 8.3.2 and 8.3.3.

23.15.3.4 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.

23.15.3.5 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.

23.15.3.6 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 23.15.3.7).

23.15.3.7 An encoder (as an encoder's option) may increase "s" (as determined in 23.15.3.6) in "**`MULTIPLE OF`**" units (subject to any restrictions that the range of values of any "**`field-to-be-set`**" or "**`field-to-be-used`**" imposes) if "**`ENCODING-SPACE SIZE`**" is set to "**`encoder-option-with-determinant`**".

23.15.4 Decoder actions

23.15.4.1 For any encoding property 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.

23.15.4.2 The decoder shall recover the tag number from the bits used to encode it, decoding from a positive integer encoding.

23.16 Defining encoding objects for classes in the other categories

In this version of this Recommendation | International Standard, there is no defined syntax for classes in the following categories:

```
                objectidentifier
                open-type
                real
                time
```

24 Defined syntax specification for the #TRANSFORM encoding class

24.1 Summary of encoding properties 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}},
                        mapping          IntegerMapping
                        } 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,

      -- chars-to-composite-char (see 24.14)
      -- There are no encoding properties for this transformation

      -- bits-to-composite-bits (see 24.15)
      &bits-to-composite-bits-unit      Unit (1..MAX) DEFAULT bit

      -- octets-to-composite-bits (see 24.16)
      -- There are no encoding properties for this transformation

      -- composite-char-to-chars (see 24.17)
      -- There are no encoding properties for this transformation

      -- composite-bits-to-bits (see 24.18)
      -- There are no encoding properties for this transformation

      -- composite-bits-to-octets (see 24.19)
      -- There are no encoding properties for this transformation

} 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]

          [CHARS-TO-COMPOSITE-CHAR]

          [BITS-TO-COMPOSITE-BITS
                [UNIT &bits-to-composite-bits-unit]]

          [OCTETS-TO-COMPOSITE-BITS]

          [COMPOSITE-CHAR-TO-CHARS]

          [COMPOSITE-BITS-TO-BITS]

          [COMPOSITE-BITS-TO-OCTETS]

    }
```

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). It also allows the specification of procedures that map a characterstring, octetstring or bitstring source into a transform composite, and a transform composite (whose values are a single character, a single octet, or bitstrings with a fixed unit size) into an abstract value (a characterstring, an octetstring, or a bitstring). 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 encoding classes in the bit-field group of categories (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 Subclauses 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 subclause 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 an integer, a boolean, a characterstring, a bitstring, a single character, or a single bit (source only). The source and result of a transform can also be a composite of these values. Transform composites can only be produced by transforms, and must be processed by another (the next) transform in a list of transforms. There are two groups of transforms: those designed to create composites from abstract values or to produce an abstract value from a composite; and those designed to transform single values. The latter can also transform composites of those values, producing a composite as the result which is the transform of every element in the source composite.

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.

24.2.8 The following subclauses specify conditions on the abstract values of the source which enable a transform to be defined as reversible. It is an ECN or application error if such values are supplied to a transform which is required to be reversible, and encoders shall not generate encodings for such values.

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 property:

```
&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}},
                 mapping          IntegerMapping
                 } OPTIONAL
```

24.3.2 The syntax for the int-to-int transform shall be:

```
[INT-TO-INT &int-to-int]
```

24.3.3 The definition of the type used in the int-to-int transform is:

```
IntegerMapping ::= SET OF SEQUENCE {
        source    SET OF INTEGER,
        result    INTEGER} (CONSTRAINED BY {/* the intersection of the source
                                                components shall be empty
                                                (see 21.17) */})
```

24.3.4 Both the source and result of this transform are integer or an integer composite. There are no bounds associated with the result unless this is the last transform in a mapping by transforms (see 19.4) (which means that neither the source nor the target can be a composite) and the target class of the mapping by transforms has bounds. In that case, it is an ECN specification or application error if the transform is applied to source integer values that do not map into the bounds of the target class.

24.3.5 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 an ordered list of transforms (this is permitted wherever transforms involving integers are allowed).

24.3.6 The values "**`increment:n`**", "**`decrement:n`**", "**`multiply:n`**", "**`negate:n`**" have their normal mathematical meaning.

24.3.7 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.

24.3.8 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".

24.3.9 The transform for the value "**`subtract:lower-bound`**" shall only be used as the first of an ordered list of transforms (and hence can never be used if the source is a composite). The source shall have a lower bound.

24.3.10 The transform for the value "**`mapping:integerMapping`**" is defined as follows. The original integer value is replaced with the value associated to the set of values to which it belongs. It is an ECN specification error if the intersection of the sets of values is not empty; it is an application error if the original integer does not belong to one of the value sets.

24.3.11 Each of these transforms is defined to be reversible if the source is a single value, not a composite, and if the condition on the abstract value (to which it is being applied) listed in Table 6 is satisfied. It is also defined to be reversible if the source is a composite and Table 6 specifies *Always reversible* as the condition.

NOTE – While an int-to-int transform with a composite input is formally reversible if Table 6 specifies *Always reversible* as the condition, it cannot in practice form part of a chain of reversible transforms since there is no such chain that starts with a non-composite input and produces a composite integer (with currently defined transforms).

Table 6 – Reversal of "INT-TO-INT" transforms

Transform	**Condition**
`increment:n`	*Always reversible*
`decrement:n`	*Always reversible*
`multiply:n`	*Always reversible*
`divide:n`	Value is a multiple of n
`negate:value`	*Always reversible*
`modulo:n`	*Never reversible*
`subtract:lower-bound`	*Always reversible*
`mapping:integerMapping`	*Source value sets, each containing only one value, and the result values are distinct.*

24.4 The bool-to-bool transform

24.4.1 The bool-to-bool transform uses the following encoding property:

```
&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 or a boolean composite.

24.4.4 If the source is a boolean, the result is a boolean. If the source is a boolean composite, the result is a boolean composite in which each element of the source has been transformed as specified in 24.4.5.

24.4.5 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.

24.4.6 This transform is defined to be reversible for all abstract values.

24.5 The bool-to-int transform

24.5.1 The bool-to-int transform uses the following encoding property:

```
&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 or a boolean composite and the result is integer or an integer composite. The integer result (and each element in the integer composite) has the value zero or one. The result has no associated bounds.

24.5.4 If the source is a boolean, the result is an integer. If the source is a boolean composite, the result is an integer composite in which each element of the source has been transformed as specified in 24.5.5.

24.5.5 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`.

24.5.6 This transform is defined to be reversible for all abstract values.

24.6 The int-to-bool transform

24.6.1 The int-to-bool transform uses the following encoding properties:

```
&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 or an integer composite and the result is boolean or a boolean composite.

24.6.4 Either one of "**`AS`**", "**`TRUE-IS`**" and "**`FALSE-IS`**" is set, or both "**`TRUE-IS`**" and "**`FALSE-IS`**" are set (and "**`AS`**" is not set), or none are 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`**" only is set, all of the integer values for "**`TRUE-IS`**" produce **`TRUE`** and all other integer values produce **`FALSE`**

24.6.7 If "**`FALSE-IS`**" only is set, all of the integer values for "**`FALSE-IS`**" produce **`FALSE`** and all other integer values produce **`TRUE`**.

24.6.8 If both "**`TRUE-IS`**" and "**`FALSE-IS`**" is set, then the integer values in "**`TRUE-IS`**" and "**`FALSE-IS`**" shall be disjoint. In this case, it is an ECN specification or application error if abstract values which are not included in either "**`TRUE-IS`**" or "**`FALSE-IS`**" are included in the source, and encoders shall not generate encodings for such values.

24.6.9 This transform is defined to be reversible if and only if both "**`TRUE-IS`**" and "**`FALSE-IS`**" are set, and they each specify a single integer value.

24.7 The int-to-chars transform

24.7.1 The int-to-chars transform uses the following encoding properties:

```
&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 definition of the type used in the int-to-chars transform is:

```
ResultSize ::= INTEGER {variable(-1), fixed-to-max(0)} (-1..MAX) --  (see 21.15)
```

24.7.4 The source for this transform is an integer or an integer composite, and the result is a characterstring or a characterstring composite.

24.7.5 If the source is an integer, the result is a characterstring. If the source is an integer composite, the result is a characterstring composite in which each element of the source has been transformed as specified in 24.7.6 to 24.7.13.

24.7.6 "**`SIZE`**", "**`PLUS-SIGN`**", and "**`PADDING`**" all have default values and can be omitted.

24.7.7 "**`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`**").

24.7.8 "**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.

24.7.9 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.

24.7.10 The most significant digit shall be at the leading end of the characterstring.

24.7.11 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.

24.7.12 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, and encoders shall not generate encodings for such abstract values.

24.7.13 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.

24.7.14 This transform is defined to be reversible for all abstract values.

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 properties:

```
&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-bits 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 definition of the types used in the int-to-bits transform are:

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

ResultSize ::= INTEGER {variable(-1), fixed-to-max(0)} (-1..MAX) --  (see 21.15)
```

24.8.4 The source for this transform is an integer or an integer composite and the result is a bitstring or a bitstring composite. There are no bounds associated with the result. The following clauses use the term resulting bitstring.

24.8.5 If the source is an integer, the result is the resulting bitstring. If the source is an integer composite, the result is a bitstring composite in which each element of the source has been transformed to the resulting bitstring as specified in 24.5.5.

24.8.6 "**AS**" and "**MULTIPLE OF**" have default values and need not be set.

24.8.7 "**SIZE**" has a default value and need not be set if the source is not a composite. It shall be set to a positive value if the source is a composite.

24.8.8 "**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.

NOTE – "**SIZE**" cannot be set to "**fixed-to-max**" if the source is a transform composite.

24.8.9 "**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 Rec. ITU-T X.690 | ISO/IEC 8825-1, 8.3.2 and 8.3.3.

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

24.8.11 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.

24.8.12 If "**`AS`**" is set to "**`positive-int`**", and the value to be transformed is negative, this is an ECN specification or an application error and encoders shall not encode such values.

24.8.13 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.

NOTE – This clause cannot apply if the source is composite.

24.8.14 If "**`SIZE`**" is a positive value, the size of the resulting bitstring shall be "**`MULTIPLE OF`**" multiplied by "**`SIZE`**".

24.8.15 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.

NOTE – This clause cannot apply if the source is composite.

24.8.16 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 and encoders shall not encode such values.

24.8.17 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.

24.8.18 This transform is defined to be reversible for all abstract values. This transform produces a self-delimiting bitstring if and only if "**`SIZE`**" is not "**`variable`**" and the source is not composite. A composite result is never self-delimiting.

24.9 The bits-to-int transform

24.9.1 The bits-to-int transform uses the following encoding property:

```
&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 a bitstring or a bitstring composite and the result is an integer or an integer composite. There are no bounds associated with the result.

24.9.4 If the source is a bitstring, the result is an integer. If the source is a bitstring composite, the result is an integer composite in which each integer is the result of the specification in 24.9.5.

24.9.5 The integer value shall be produced by interpreting the bits as 2's-complement or as a positive integer encoding, as specified in Rec. ITU-T 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.

24.9.6 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 properties:

```
&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
```

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 definition of the types used in the char-to-bits transform are:

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

ResultSize ::= INTEGER {variable(-1), fixed-to-max(0)} (-1..MAX) --  (see 21.15)
```

24.10.4 The source for this transform is a single character from either:

a) the specification of an encoding for the characterstring category (see 23.4.2.1); or

b) a single character composite;

and the result is a bitstring in case a) and a bitstring composite in case b).

24.10.5 The source for this transform is a single character or a single character composite. If the source is a single character, the result is a bitstring. If the source is a single character composite, the result is a bitstring composite.

24.10.6 Where the source is a composite, the resulting composite is determined by applying the following specification to all elements of the source composite to form the result composite. It is an ECN specification error if this transform is applied to a composite with "**AS**" set to "**mapped**" and the size of the bitstrings in the "**BITS-LIST**" are not all the same.

24.10.7 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.

NOTE – This can only be the case if the class to which the transform is applied is part of an implicitly or explicitly generated structure. This clause can never apply to a composite, whose elements never have effective permitted alphabet constraints.

24.10.8 "**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.

24.10.9 ECN supports only characters in the ISO/IEC 10646 character set. Where ASN.1 types such as "GeneralString" are in use, characters outside of this character set can in theory appear. Such characters are not supported by this transform.

24.10.10 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 24.10.10.1 to 24.10.10.5.

24.10.10.1 "**CHAR-LIST**" and "**BITS-LIST**" are respectively an ordered list of single characters and of bitstring values. (These parameters are ignored if "**AS**" is not set to "**mapped**".)

24.10.10.2 There shall be an equal number of values in each list, and all character values in "**CHAR-LIST**" shall be distinct.

24.10.10.3 The transform of a character in "**CHAR-LIST**" is the bitstring specified in the corresponding position in "**BITS-LIST**".

24.10.10.4 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.

24.10.10.5 In this case ("**AS**" set to "**mapped**"), the transform is defined to be reversible (for all abstract values) if and only if the set of all bitstring values in "**BITS-LIST**" are distinct, otherwise it shall not be used where a reversible transform is required. The result is self-delimiting if the bitstring values in "**BITS-LIST**" are self-delimiting (see 3.2.42). A composite result is never self-delimiting.

24.10.11 If "**AS**" is "**iso10646**", the transform is specified in 24.10.11.1 to 24.10.11.5.

24.10.11.1 The character is first converted to an integer with the numerical value specified in ISO/IEC 10646.

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

24.10.11.2 If the character is from a character string that has an associated effective permitted alphabet constraint (see 24.10.7), then the integer has effective size constraints just sufficient to contain the numerical values of all characters in the effective permitted alphabet.

24.10.11.3 If there is no effective permitted alphabet constraint, then the integer has an associated effective size constraint of 0..32767.

24.10.11.4 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.)

24.10.11.5 In this case ("`AS`" set to "`iso10646`"), the transform is defined to be reversible for all abstract values. It produces a self-delimiting string of bits if and only if "`SIZE`" is not "variable". A composite result is never self-delimiting.

24.10.12 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 24.10.12.1 to 24.10.12.4.

24.10.12.1 All characters in the effective permitted alphabet are placed in canonical order using their ISO/IEC 10646 value, lowest value first. The first in the list is then assigned the integer value zero, the next one, and so on.

24.10.12.2 If the effective permitted alphabet contains "n" characters, then the integer has an effective size constraint of 0..n-1.

24.10.12.3 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 this Recommendation | International Standard.

24.10.12.4 In this case ("`AS`" set to "`compact`"), the transform is defined to be reversible for all abstract values. It produces a self-delimiting string of bits if and only if "`SIZE`" is not "`variable`". A composite result is never self-delimiting.

24.11 The bits-to-char transform

24.11.1 The bits-to-char transform uses the following encoding properties:

```
&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 a bitstring or a bitstring composite. If the source is a bitstring, the result is a single character. If the source is a bitstring composite, the result is a single character composite.

24.11.4 If the source is a bitstring composite, then the resulting single character composite is an ordered list of single characters resulting from the transformation of each of the elements of the bitstring composite.

24.11.5 If "`AS`" is "`iso10646`", then the bitstring shall be interpreted as a positive integer encoding which contains the ISO/IEC 10646 numerical value of a character. It is an ECN specification error if the integer value exceeds 32767.

24.11.6 If "**AS**" is "**mapped**", then the transform is specified by the values of "**CHAR-LIST**" and "**BITS-LIST**". The transform is defined in 24.11.6.1 to 24.11.6.5.

24.11.6.1 "**CHAR-LIST**" and "**BITS-LIST**" are respectively an ordered list of single characters and of bitstring values. (These parameters are ignored if "**AS**" is not set to "**mapped**".)

24.11.6.2 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.

24.11.6.3 The transform of a bitstring in the "**BITS-LIST**" is the character specified in the corresponding position in the "**CHAR-LIST**".

24.11.6.4 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.

24.11.6.5 The transform is defined to be reversible for all abstract values.

24.12 The bit-to-bits transform

24.12.1 The bit-to-bits transform uses the following encoding properties:

```
&bit-to-bits-one            Non-Null-Pattern DEFAULT bits:'1'B,
&bit-to-bits-zero           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 definition of the type used in the bit-to-bits transform is:

```
Non-Null-Pattern ::= Pattern
      (ALL EXCEPT (bits:''B | octets:''H | char8:"" | char16:"" |
                   char32:"")) -- (see 21.10.2)
```

24.12.4 The source for this transform is a single bit from either:

a) the specification of an encoding for the bitstring category (see 23.2); or

b) a bitstring composite with a unit of 1 bit.

The result is a bitstring in case a) and a bitstring composite in case b).

24.12.5 The bitstring composite in case b) shall be the ordered sequence of bitstrings produced by the following transformations applied to each element of the source bitstring composite. It is an ECN specification error if the "**ZERO-PATTERN**" and the "**ONE-PATTERN**" have different sizes.

24.12.6 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.

24.12.7 The "**any-of-length**" alternative shall not be used for either "**ZERO-PATTERN**" or "**ONE-PATTERN**".

24.12.8 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**".

24.12.9 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.

24.12.10 This transform is defined to be reversible for all abstract values and the result is self-delimiting unless the transform is applied to a composite. A composite result is never self-delimiting.

24.13 The bits-to-bits transform

24.13.1 The bits-to-bits transform uses the following encoding properties:

```
&Source-values              BIT STRING ORDERED,
&Result-values              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 either a bitstring or a bitstring composite. If the source is a bitstring the result is a bitstring. If the source is a bitstring composite the result is a bitstring composite.

24.13.4 If the source is a bitstring composite, then the resulting bitstring composite is the ordered list of bitstrings obtained by applying the following specification to each bitstring in the source.

24.13.5 "**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.

24.13.6 The transform is specified by the values of "**SOURCE-LIST**" and "**RESULT-LIST**".

24.13.7 There shall be an equal number of bitstring values in each list, and all bitstring values in "**SOURCE-LIST**" shall be distinct.

24.13.8 The transform of a bitstring in "**SOURCE-LIST**" is the bitstring specified in the corresponding position in "**RESULT-LIST**".

24.13.9 If this transform is applied to a composite, all bitstrings in the "**RESULT-LIST**" shall have the same size.

24.13.10 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.

24.13.11 The transform is defined to be reversible (for all abstract values) if and only if the set of all bitstring values in "**RESULT-LIST**" are distinct, otherwise it shall not be used where a reversible transform is required. The result is self-delimiting if the bitstring values in "**RESULT-LIST**" are distinct and self-delimiting (see 3.2.42) and the transform is applied to a bitstring. A composite result is never self-delimiting.

24.14 The chars-to-composite-char transform

24.14.1 The chars-to-composite-char transform converts a characterstring to a single character composite.

24.14.2 The syntax for the chars-to-composite-char transform shall be:

```
[CHARS-TO-COMPOSITE-CHAR]
```

24.14.3 The source of this transform is a characterstring and the result is a single character composite.

24.14.4 The single character composite is an ordered list of the characters in the source characterstring.

24.14.5 This transform is defined to be reversible for all abstract values.

24.15 The bits-to-composite-bits transform

24.15.1 The bits-to-composite-bits transform converts a bitstring to a bitstring composite, where each bitstring element has the same (known) size.

24.15.2 The bits-to-composite-bits transform uses the following encoding properties:

```
&bits-to-composite-bits-unit       Unit (1..MAX) DEFAULT bit
```

24.15.3 The syntax for the bits-to-composite-bits transform shall be:

```
[BITS-TO-COMPOSITE-BITS
      [UNIT &bits-to-composite-bits-unit]]
```

24.15.4 The definition of the type used in the bits-to-composite-bits transform is:

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

24.15.5 The source of this transform is a bitstring and the result is a bitstring composite of size "**UNIT**".

24.15.6 The bitstring composite of size "`UNIT`" is an ordered list of bitstrings each of which is of size "`UNIT`". The first bitstring in the composite is the first "`UNIT`" bits from the source bitstring. The second is the next "`UNIT`" bits, and so on. If the source bitstring is not a multiple of "`UNIT`" bits, this is an ECN specification or application error.

24.15.7 This transform is defined to be reversible for all abstract values.

24.16 The octets-to-composite-bits transform

24.16.1 The octets-to-composite-bits transform converts an octetstring to a bitstring composite of size 8 bits.

24.16.2 The syntax for the octets-to-composite-bits transform shall be:

```
[OCTETS-TO-COMPOSITE-BITS]
```

24.16.3 The source of this transform is an octetstring and the result is a bitstring composite of size 8 bits.

24.16.4 The bitstring composite of size 8 is an ordered list of the bitstrings corresponding to the octets in the source octetstring.

24.16.5 This transform is defined to be reversible for all abstract values.

24.17 The composite-char-to-chars transform

24.17.1 The composite-char-to-chars transform converts a single character composite to a characterstring.

24.17.2 The syntax for the composite-char-to-chars transform shall be:

```
[COMPOSITE-CHAR-TO-CHARS]
```

24.17.3 The source of this transform is a single character composite and the result is a characterstring.

24.17.4 The characterstring is formed from the ordered list of characters present in the (source) single character composite.

24.17.5 This transform is defined to be reversible for all abstract values.

24.18 The composite-bits-to-bits transform

24.18.1 The composite-bits-to-bits transform converts a bitstring composite of a known unit size to a bitstring.

24.18.2 The syntax for the composite-bits-to-bits transform shall be:

```
[COMPOSITE-BITS-TO-BITS]
```

24.18.3 The source of this transform is a bitstring composite and the result is a bitstring.

24.18.4 The bitstring is formed from the ordered list of bitstrings present in the (source) bitstring composite.

24.18.5 This transform is defined to be reversible for all abstract values. The result bitstring is not self-delimiting.

NOTE – This transform is reversible because the units used in its generation are specified in the transform that produced the bitstring composite, and are associated with that composite.

24.19 The composite-bits-to-octets transform

24.19.1 The composite-bits-to-octets transform converts a bitstring composite of unit size 8 to an octetstring. It is an ECN specification error if this is applied to a bitstring composite that has a unit size which is not 8.

24.19.2 The syntax for the composite-bits-to-octets transform shall be:

```
[COMPOSITE-BITS-TO-OCTETS]
```

24.19.3 The source of this transform is a bitstring composite and the result is an octetstring.

24.19.4 The octetstring is formed from the ordered list of bitstrings present in the (source) bitstring composite.

24.19.5 This transform is defined to be reversible for all abstract values.

25 Complete encodings and the **#OUTER** class

If there is no encoding object of the **#OUTER** class in the combined encoding object set being applied to a type in the ELM, then the encoder and decoder shall assume an encoding object of this class in which all encoding properties have their default values.

25.1 Encoding properties, syntax and purpose for the **#OUTER** class

25.1.1 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 different:any)
                                        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]

}
```

25.1.2 The definition of the types used in the **#OUTER** specification are:

```
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)

Non-Null-Pattern ::= Pattern
      (ALL EXCEPT (bits:''B | octets:''H | char8:"" | char16:"" |
                  char32:"")) -- (see 21.10.2)
```

25.1.3 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.

25.1.4 Three independent specifications can be made (see 25.1.5 to 25.1.7).

25.1.5 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.

25.1.6 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.

25.1.7 The "**ADDED BITS DECODING**" specification is applicable only to decoders, 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 this Recommendation | International Standard is expected to give enhanced support for extensibility.

25.1.8 "`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.

25.2 Encoder actions for `#OUTER`

25.2.1 If "`ALIGNMENT`" is "`unchanged`", then the alignment point used in encoding a contained type shall be the alignment point used in encoding the container.

25.2.2 If "`ALIGNMENT`" is "`reset`", then the alignment point used in encoding a contained type shall be the start of the encoding of that type.

25.2.3 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.

25.2.4 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.

25.2.5 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`".

25.2.6 The encoder shall ignore "`ADDED BITS DECODING`".

25.3 Decoder actions for `#OUTER`

25.3.1 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`".

25.3.2 If "`ALIGNMENT`" is "`unchanged`", then the alignment point used in encoding a contained type shall be the alignment point used in encoding the container.

25.3.3 If "`ALIGNMENT`" is "`reset`", then the alignment point used in encoding a contained type shall be the start of the encoding of that type.

25.3.4 The decoder shall determine the bits added by "`PADDING`" (if any), and shall silently ignore the added bits, no matter what their value.

25.3.5 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

Addendum to Rec. ITU-T X.680 | ISO/IEC 8824-1

(This annex forms an integral part of this Recommendation | International Standard.)

This annex specifies the modifications that are to be applied when productions and/or clauses from Rec. ITU-T X.680 | ISO/IEC 8824-1 are referenced in this Recommendation | International Standard.

A.1 Exports and imports clauses

The productions "AssignedIdentifier", "Symbol" and "Reference" of 13.1, as well as subclauses 13.13 and 13.16, of Rec. ITU-T X.680 | ISO/IEC 8824-1 are modified as follows:

```
13.1    AssignedIdentifier ::= DefinitiveIdentifier    |
        empty

        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.

where "DefinitiveIdentifier is defined as:

```
DefinitiveIdentifier ::=
          "{" DefinitiveObjIdComponentList "}"
     |    empty
```

13.13 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.

13.16 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 17.1, is modified as follows:

```
Type ::=
                BuiltinType
    |           ReferencedType
    |           ConstrainedType
    |           REFERENCE
```

A.3 Notation for character string values

The production "CharsDefn" of Rec. ITU-T X.680 | ISO/IEC 8824-1, 41.8, 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 Rec. ITU-T X.680 | ISO/IEC 8824-1, 42.1).

Annex B

Addendum to Rec. ITU-T X.681 | ISO/IEC 8824-2

(This annex forms an integral part of this Recommendation | International Standard.)

This annex specifies the modifications that are to be applied when productions and/or clauses from Rec. ITU-T X.681 | ISO/IEC 8824-2 are referenced in this Recommendation | International Standard.

B.1 Definitions

The following definitions are added to Rec. ITU-T X.681 | ISO/IEC 8824-2, 3.4:

encoding class field: A field which contains an arbitrary encoding class.

encoding class field type: A type specified by reference to some type field of an encoding object class.

encoding object field: A 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 is some (specific) encoding class field of the same encoding object.

encoding object set field: A field which contains a set of encoding objects of some specified encoding class.

fixed-type ordered value list field: A field which contains an ordered (possibly empty) list of values of some specified type.

ordered encoding object list field: A field which contains an ordered non-empty list of encoding objects of some specified encoding class.

reference field: A field which contains a reference to an encoding structure field (see also 17.5.15).

B.2 Additional lexical items

NOTE – This modification is introduced for the sole purpose of clause 23.

The following definitions are added to Rec. ITU-T X.681 | ISO/IEC 8824-2, clause 7:

B.2.1 Ordered value list field references

Name of item – orderedvaluelistfieldreference

An "orderedvaluelistfieldreference" shall consist of an ampersand ("&") immediately followed by a sequence of characters as specified for a "typereference" in Rec. ITU-T X.680 | ISO/IEC 8824-1, 12.2.

B.2.2 Ordered encoding object list field references

Name of item – orderedencodingobjectlistfieldreference

An "orderedencodingobjectlistfieldreference" shall consist of an ampersand ("&") immediately followed by a sequence of characters as specified for an "objectsetreference" in Rec. ITU-T X.681 | ISO/IEC 8824-2, 7.3.

B.2.3 Encoding class field references

Name of item – encodingclassfieldreference

An "encodingclassfieldreference" shall consist of an ampersand ("&") 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 Rec. ITU-T X.681 | ISO/IEC 8824-2, 9.3.

B.4 FieldSpec additions

NOTE – This modification is introduced for the sole purpose of clause 23.

Rec. ITU-T X.681 | ISO/IEC 8824-2, 9.4, is modified as follows:

```
FieldSpec ::=
                              FixedTypeValueFieldSpec
                        |     FixedTypeValueSetFieldSpec
                        |     FixedTypeOrderedValueListFieldSpec
                        |     FixedClassEncodingObjectFieldSpec
                        |     VariableClassEncodingObjectFieldSpec
                        |     FixedClassEncodingObjectSetFieldSpec
                        |     FixedClassOrderedEncodingObjectListFieldSpec
                        |     EncodingClassFieldSpec
```

B.5 Fixed-type ordered value list field spec

NOTE – This modification is introduced for the sole purpose of clause 23.

A "FixedTypeOrderedValueListFieldSpec" specifies that the field is a fixed-type ordered value list field (see B.1 of this Recommendation | International Standard):

```
FixedTypeOrderedValueListFieldSpec ::=
        orderedvaluelistfieldreference
        DefinedType
        ORDERED
        FixedTypeOrderedValueListOptionalitySpec ?
```

FixedTypeOrderedValueListOptionalitySpec ::= OPTIONAL | DEFAULT OrderedValueList

The name of the field is "orderedvaluelistfieldreference". The "DefinedType" references the type of values contained in the field. The "FixedTypeOrderedValueListOptionalitySpec", if present, specifies that the field may be unspecified in an encoding object definition, or, in the "DEFAULT" case, that omission produces the following "OrderedValueList" (see Rec. ITU-T X.680 | ISO/IEC 8824-1, 26.3), 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 this Recommendation | International Standard):

```
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 17.1.5 of this Recommendation | International Standard) 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 this Recommendation | International Standard):

```
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 this Recommendation | International Standard):

```
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 ordered encoding object list field spec

NOTE – This modification is introduced for the sole purpose of clause 23.

A "FixedClassOrderedEncodingObjectListFieldSpec" specifies that the field is a fixed-class ordered encoding object list field (see B.1 of this Recommendation | International Standard):

```
FixedClassOrderedEncodingObjectListFieldSpec ::=
        orderedencodingobjectlistfieldreference
        DefinedOrBuiltinEncodingClass
        ORDERED
        OrderedEncodingObjectListOptionalitySpec?
```

OrderedEncodingObjectListOptionalitySpec ::= OPTIONAL | DEFAULT OrderedEncodingObjectList

The name of the field is "orderedencodingobjectlistfieldreference". The "DefinedOrBuiltinEncodingClass" references the class of the encoding objects contained in the field. The "OrderedEncodingObjectListOptionalitySpec", if present, specifies that the field may be unspecified in an encoding object definition, or, in the **DEFAULT** case, that omission produces the following "OrderedEncodingObjectList" (see B.11 of this Recommendation | International Standard), 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 this Recommendation | International Standard):

```
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 Ordered value list notation

OrderedValueList ::= "{" Value "," + "}"

The "OrderedValueList" is an ordered list of one or more values of the governing type. It is used when the application applies semantics to the order of values in the list.

NOTE – A value list can only be specified by in-line notation (which is governed by a type field, a fixed-type value set field, or a fixed-type ordered value list field).

B.12 Ordered encoding object list notation

OrderedEncodingObjectList ::= "{" EncodingObject "," + "}"

The "OrderedEncodingObjectList" 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 encoding objects in the list.

Example: A list of **#TRANSFORM** encoding objects is applied in the stated order.

NOTE – The following restrictions arise from normative text and BNF productions: An ordered encoding object list can only be specified by in-line notation (which is governed by an ordered encoding object list field); encoding objects within that list can be specified using either a reference name or in-line notation; the governor cannot be **#ENCODINGS**.

B.13 Primitive field names

Rec. ITU-T 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
        |           orderedvaluelistfieldreference
```

B.14 Additional reserved words

Rec. ITU-T X.681 | ISO/IEC 8824-2, 10.6 and 10.7, are 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
OPTIONS
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.

10.7 A "Literal" specifies the actual inclusion of that "Literal", which is required to be a "word", at that position in the defined syntax.

B.15 Definition of encoding objects

The restriction imposed by Rec. ITU-T 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.16 Additions to "Setting"

Rec. ITU-T X.681 | ISO/IEC 8824-2, 11.7, is modified as follows:

11.7 A "Setting" specifies the setting of some field within an encoding object being defined:

```
Setting ::=
                  Value
        |         ValueSet
        |         OrderedValueList
        |         EncodingObject
        |         EncodingObjectSet
        |         OrderedEncodingObjectList
        |         DefinedOrBuiltinEncodingClass
        |         OUTER
```

If the field is:

a) a value field, the "Value" alternative;

b) a fixed-type value set field, the "ValueSet" alternative;

c) a fixed-type ordered value list field, the "OrderedValueList" alternative;

d) an encoding object field, the "EncodingObject" alternative;

e) an encoding object set field, the "EncodingObjectSet" alternative;

f) an ordered encoding object list field, the "OrderedEncodingObjectList" 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 Rec. ITU-T X.681 | ISO/IEC 8824-2, 9.5 to 9.12, and 11.8 to 11.9.

B.17 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.17.2 to B.17.4 below specify the type that is referenced.

B.17.1 The notation for an encoding class field type shall be "EncodingClassFieldType":

```
EncodingClassFieldType ::=
                  DefinedOrBuiltinEncodingClass
                  "."
                  FieldName
```

where the "FieldName" is as specified in Rec. ITU-T X.681 | ISO/IEC 8824-2, 9.14, relative to the encoding class identified by the "DefinedOrBuiltinEncodingClass".

B.17.2 For a fixed-type value, a fixed-type value set field, or a fixed-type ordered 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.17.3 This notation is not permitted if the field is an encoding object, an encoding object set or an ordered encoding object list field.

B.17.4 The notation for defining a value of this type shall be "FixedTypeFieldVal" as defined in Rec. ITU-T X.681 | ISO/IEC 8824-2, 14.6.

Annex C

Addendum to Rec. ITU-T X.683 | ISO/IEC 8824-4

(This annex forms an integral part of this Recommendation | International Standard)

This annex specifies the modifications that need to be applied when productions and/or clauses from Rec. ITU-T X.683 | ISO/IEC 8824-4 are referenced in this Recommendation | International Standard.

C.1 Parameterized assignments

Clauses 8.1 and 8.3 of Rec. ITU-T X.683 | ISO/IEC 8824-4 are modified as follows:

8.1 There are parameterized assignment statements corresponding to each of the assignment statements specified in this Recommendation | International Standard. The "ParameterizedAssignment" construct is:

```
ParameterizedAssignment ::=
     ParameterizedEncodingObjectAssignment
|    ParameterizedEncodingClassAssignment
|    ParameterizedEncodingObjectSetAssignment
```

8.3 **ParameterList ::= "{<" Parameter "," + ">}"**

```
Governor ::=
                 EncodingClassFieldType
      |          REFERENCE
      |          DefinedOrBuiltinEncodingClass
      |          #ENCODINGS
      |          Type
```

A "DummyReference" in "Parameter" may stand for:

a) an encoding class, in which case there shall be no "ParamGovernor";

a) an ASN.1 value, value set, or fixed-type ordered value list, in which case the "ParamGovernor" shall be present as a "Governor" that is a type extracted from an encoding class ("EncodingClassFieldType");

b) an "identifier", in which case the "ParamGovernor" shall be present as a "Governor" that is **REFERENCE**;

c) an encoding object, or an ordered encoding object list, in which case the "ParamGovernor" shall be present as a "Governor" that is an encoding class ("DefinedOrBuiltinEncodingClass");

d) 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 Rec. ITU-T X.683 | ISO/IEC 8824-4, 8.2:

```
ParameterizedEncodingClassAssignment ::=
     encodingclassreference
     ParameterList
     "::="
     EncodingClass

ParameterizedEncodingObjectAssignment ::=
     encodingobjectreference
     ParameterList
     DefinedOrBuiltinEncodingClass
     "::="
     EncodingObject

ParameterizedEncodingObjectSetAssignment ::=
     encodingobjectsetreference
     ParameterList
     #ENCODINGS
     "::="
     EncodingObjectSet
```

Rec. ITU-T 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 Rec. ITU-T X.683 | ISO/IEC 8824-4, 9.1, is modified as follows:

ParameterizedReference ::=
```
        Reference
|       Reference "{<"   ">}"
```

The following productions are added to Rec. ITU-T 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

Rec. ITU-T X.683 | ISO/IEC 8824-4, 9.5, is modified as follows:

9.5 The "ActualParameterList" is:

ActualParameterList ::=
```
          "{<" ActualParameter "," + ">}"
```

ActualParameter ::=
```
                    Value
          |         ValueSet
          |         OrderedValueList
          |         DefinedOrBuiltinEncodingClass
          |         EncodingObject
          |         EncodingObjectSet
          |         OrderedEncodingObjectList
          |         ComponentIdList
          |         STRUCTURE
          |         OUTER
```

If the corresponding dummy parameter is:

a) a value: the "Value" alternative shall be used;

b) a value set: the "ValueSet" alternative shall be used;

c) a fixed-type ordered value list: the "OrderedValueList" alternative shall be used;

d) an encoding class: the "DefinedOrBuiltinEncodingClass" alternative shall be used;

e) an encoding object: the "EncodingObject" alternative shall be used;

f) an encoding object set: the "EncodingObjectSet" alternative shall be used;

g) an ordered encoding object list: the "OrderedEncodingObjectList" alternative shall be used;

h) a reference: the "identifier", **`STRUCTURE`** or **`OUTER`** alternative shall be used.

`STRUCTURE` shall only be used when the actual parameter is used as specified in 17.5.15. **`OUTER`** can be used whenever a reference is required to identify a container, and identifies the container of the entire encoding.

Annex D

Examples

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

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

– General examples, which show the look-and-feel of ECN definitions (D.1).

– Specialization examples, which show how to modify some parts of a standardized encoding. Each example has a description of the requirements for the encoding and a description of the selected solution and possible alternative solutions (D.2).

– Explicitly generated structure examples, which show the use of explicitly generated structures when the same specialized encoding is used several times (D.3).

– A legacy protocol example which shows three ways of handling the problem of a traditional "more-bit" approach to sequence-of termination (D.4).

– A second legacy protocol example, which shows how to construct ECN definitions for a protocol whose message encodings have been specified using a tabular notation (D.5).

D.1 General examples

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

D.1.1 An encoding object for a boolean type

D.1.1.1 The ASN.1 assignment is:

```
Married ::= BOOLEAN
```

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

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

marriedEncoding-1 #Married ::= booleanEncoding
```

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

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

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

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

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

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

D.1.2 An encoding object for an integer type

D.1.2.1 The ASN.1 assignments are:

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

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

D.1.2.2 The encoding object assignments are:

```
evenPositiveIntegerEncoding #EvenPositiveInteger ::= {
      USE #NonNegativeInt
      MAPPING TRANSFORMS {{INT-TO-INT divide:2}}
      WITH PER-BASIC-UNALIGNED}

#NonNegativeInt ::= #INT(0..MAX)

evenNegativeIntegerEncoding #EvenNegativeInteger ::= {
      USE #NonPositiveInt
      MAPPING TRANSFORMS {{INT-TO-INT divide:2
      -- Note: -1 / 2 = 0 - see clause 24.3.7 -- }}
      WITH PER-BASIC-UNALIGNED}

#NonPositiveInt ::= #INT(MIN..0)
```

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

D.1.3 Another encoding object for an integer type

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

D.1.3.2 The ASN.1 assignment is:

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

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

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

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

D.1.4.1 The ASN.1 assignment is:

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

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

```
integerWithHoleEncoding #IntegerWithHole ::= {
      USE #IntFrom0To1280
      MAPPING ORDERED VALUES
      WITH PER-BASIC-UNALIGNED}

#IntFrom0To1280 ::= #INT (0..1280)
```

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

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

D.1.5.1 The ASN.1 assignments are:

```
PositiveInteger ::= INTEGER (1..MAX)

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

D.1.5.2 The encoding object assignments are:

```
positiveIntegerEncoding #PositiveInteger ::=
      integerEncoding

negativeIntegerEncoding #NegativeInteger ::=
      integerEncoding
```

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

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

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

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

D.1.6 Positive integers encoded in BCD

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

D.1.6.2 The ASN.1 assignment is:

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

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

```
positiveIntegerBCDEncoding #PositiveIntegerBCD ::= {
      USE #CHARS
      MAPPING TRANSFORMS{{
```

```
                INT-TO-CHARS
                -- We convert to characters (e.g., integer 42
                     -- becomes character string "42") and encode the characters
                     -- with the encoding object "numeric-chars-to-bcdEncoding"
                          SIZE variable
                          PLUS-SIGN FALSE}}
                     WITH numeric-chars-to-bcdEncoding }

numeric-chars-to-bcdEncoding #CHARS ::= {
     ALIGNED TO NEXT nibble
          TRANSFORMS {{
          CHAR-TO-BITS
          -- We convert each character to a bitstring
          --(e.g., character "4" becomes '0100'B and "2" becomes
          -- '0010'B)
          AS mapped
               CHAR-LIST { "0","1","2","3",
                           "4","5","6","7",
                           "8","9"}
               BITS-LIST { '0000'B, '0001'B, '0010'B, '0011'B,
                           '0100'B, '0101'B, '0110'B, '0111'B,
                           '1000'B, '1001'B }}}
     REPETITION-ENCODING {
          REPETITION-SPACE
          -- We determine the concatenation of the bitstrings for the
          -- characters and add a terminator (e.g.,
          -- '0100'B + '0010'B becomes '0100 0010 1111'B)
               SIZE variable-with-determinant
               DETERMINED BY pattern
               PATTERN bits:'1111'B}}
```

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

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

D.1.7 An encoding object of class `#BITS`

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

D.1.7.2 The ASN.1 assignment is:

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

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

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

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

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

D.1.8 An encoding object for an octetstring type

D.1.8.1 The ASN.1 assignment is:

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

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

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

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

D.1.9 An encoding object for a character string type

D.1.9.1 The ASN.1 assignment is:

```
Password ::= PrintableString
```

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

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

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

D.1.10 Mapping character values to bit values

D.1.10.1 The ASN.1 assignment is:

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

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

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

#IntFrom0To2 ::= #INT (0..2)
```

where "integerEncoding" is defined in D.1.5.5.

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

D.1.11 An encoding object for a sequence type

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

D.1.11.2 The ASN.1 assignment is:

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

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

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

parameterizedPresenceEncoding {< REFERENCE:reference >}  #OPTIONAL ::= {
      PRESENCE
            DETERMINED BY field-to-be-used
            USING reference}
```

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

D.1.12 An encoding object for a choice type

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

D.1.12.2 The ASN.1 assignment is:

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

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

```
choiceEncoding-1 #Choice ::= {
      ENCODE STRUCTURE {
            boolean     [tagEncoding] USE-SET,
            integer     [tagEncoding] USE-SET,
            string      [tagEncoding] USE-SET
            STRUCTURED WITH {
                  ALTERNATIVE
                        DETERMINED BY handle
                              HANDLE "Tag"}}
                  WITH PER-BASIC-UNALIGNED}

tagEncoding #TAG ::= {
      ENCODING-SPACE
            SIZE 3
```

```
                    MULTIPLE OF bit
        EXHIBITS HANDLE "Tag" AT {0 | 1 | 2}}
```

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

```
MyEncodings #ENCODINGS ::= { tagEncoding } COMPLETED BY PER-BASIC-UNALIGNED

choiceEncoding-2 #Choice ::= {
      ENCODE STRUCTURE {
            STRUCTURED WITH {
                  ALTERNATIVE
                        DETERMINED BY handle
                              HANDLE "Tag"}}
            WITH MyEncodings}
```

D.1.13 Encoding a bitstring containing another encoding

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

D.1.13.2 The ASN.1 assignment are:

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

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

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

```
sequence2Encoding #Sequence2 ::= {
      ENCODE STRUCTURE {
            b     { REPETITION-ENCODING {
                        REPETITION-SPACE
                              SIZE 8
                                    MULTIPLE OF bit}
                  CONTENTS-ENCODING {sequence3Encoding}
      COMPLETED BY PER-BASIC-UNALIGNED}}
      WITH PER-BASIC-UNALIGNED}

sequence3Encoding #Sequence3 ::= {
      ENCODE STRUCTURE {
            STRUCTURED WITH sequence3StructureEncoding
            }
      WITH PER-BASIC-UNALIGNED }

sequence3StructureEncoding #CONCATENATION ::= {
      ENCODING-SPACE
            MULTIPLE OF octet
            VALUE-PADDING
                  JUSTIFIED left:0
                  POST-PADDING zero
                  UNUSED BITS
                        DETERMINED BY not-needed }
```

D.1.14 An encoding object set

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

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

D.1.15 ASN.1 definitions

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

```
Example1-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-
module1(2)}
      DEFINITIONS AUTOMATIC TAGS ::=
      BEGIN

MyPDU ::= CHOICE {
      marriedMessage         Married,
      altitudeMessage        Altitude
      -- etc.
      }

Married ::= BOOLEAN

Altitude ::= INTEGER (0..65535)

-- etc.

END
```

D.1.16 EDM definitions

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

```
Example1-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module1(3)}
      ENCODING-DEFINITIONS ::=
      BEGIN

EXPORTS Example1Encodings;

IMPORTS #Married, #Altitude, #EvenPositiveInteger, #EvenNegativeInteger,
      #IntegerWithHole, #PositiveInteger, #NegativeInteger, #PositiveIntegerBCD,
      #Fax, #BinaryFile, #Password, #CharacterStringToBit, #Sequence1, #Choice
      FROM Example1-ASN1-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5)
                              asn1-module1(2) };

Example1Encodings #ENCODINGS ::= {
      marriedEncoding-1|
            -- etc
      sequence3Encoding}

      -- etc


END
```

D.1.17 ELM definitions

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

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

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

ENCODE #MyPDU WITH Example1Encodings
      COMPLETED BY PER-BASIC-UNALIGNED

END
```

D.2 Specialization examples

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

– There are some special semantics associated with message components that make it possible to remove some of the PER-generated auxiliary fields.

– The user wants different encodings for PER auxiliary fields that are generated by default, such as variable-width determinant fields.

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

D.2.1.1 The ASN.1 assignment is:

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

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

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

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

```
normallySmallValuesEncoding-1 #NormallySmallValues ::= {
      USE   #NormallySmallValuesStruct-1
      MAPPING DISTRIBUTION {
            0..63       TO small,
            REMAINDER   TO large }
      WITH PER-BASIC-UNALIGNED}
```

D.2.1.4 The encoding structure assignment is:

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

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

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

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

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

```
normallySmallValuesEncoding-2 #NormallySmallValues ::= {
      USE   #NormallySmallValuesStruct-2
      MAPPING ORDERED VALUES
      WITH  NormallySmallValuesTag-encoding-plus-PER}

normallySmallValuesTag-encoding    #TAG ::= {
      ENCODING-SPACE
            SIZE 1}

NormallySmallValuesTag-encoding-plus-PER #ENCODINGS ::=
      {normallySmallValuesTag-encoding}
      COMPLETED BY PER-BASIC-UNALIGNED
```

D.2.2.3 The encoding structure assignment is:

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

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

D.2.3 Compression of non-continuous value ranges

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

D.2.3.2 The ASN.1 assignment is:

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

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

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

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

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

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

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

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

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

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

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

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

D.2.5.1 The ASN.1 assignment is:

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

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

D.2.5.3 The encoding object assignment is:

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

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

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

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

D.2.6.1 The ASN.1 assignment is:

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

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

D.2.6.3 The encoding object assignment is:

```
conditionalPresenceOnValueEncoding #ConditionalPresenceOnValue ::= {
      ENCODE STRUCTURE {
            c     USE-SET OPTIONAL-ENCODING is-c-present{<  a  >},
            d     USE-SET OPTIONAL-ENCODING is-d-present{<  a  >}}
      WITH PER-BASIC-UNALIGNED}

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

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

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

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

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

D.2.7.1 The ASN.1 assignment is:

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

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

D.2.7.3 The encoding object assignment is:

```
conditionalPresenceOnExternalConditionEncoding
      #ConditionalPresenceOnExternalCondition ::= {
      ENCODE STRUCTURE {
            a     USE-SET OPTIONAL-ENCODING is-a-present}
      WITH PER-BASIC-UNALIGNED}

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

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

D.2.8 A variable length list

D.2.8.1 The ASN.1 assignment is:

```
EnclosingStructureForList ::= SEQUENCE {
      list  VariableLengthList}

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

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

D.2.8.3 The encoding object assignment is:

```
enclosingStructureForListEncoding #EnclosingStructureForList ::= {
      USE   #EnclosingStructureForListStruct
      MAPPING FIELDS WITH {
            ENCODE STRUCTURE {
                  aux-length  list-lengthEncoding,
                  list  {
                        ENCODE STRUCTURE {
                              STRUCTURED WITH {
                              REPETITION-ENCODING {
                                    REPETITION-SPACE
                                          SIZE variable-with-determinant
                                                MULTIPLE OF repetitions
                                          DETERMINED BY field-to-be-set
                                          USING aux-length}}}
                        WITH PER-BASIC-UNALIGNED }}
            WITH PER-BASIC-UNALIGNED}}
-- First mapping: use of an encoding structure with an explicit length
-- determinant.

list-lengthEncoding #AuxVariableListLength ::= {
      USE   #AuxVariableListLengthStruct       -- See D.2.8.4.
      MAPPING ORDERED VALUES
      WITH PER-BASIC-UNALIGNED}
-- Second mapping: list length is encoded as a choice between
-- a short form "normally" and a long form "sometimes".
```

D.2.8.4 The encoding structure assignments are:

```
#EnclosingStructureForListStruct ::= #CONCATENATION {
      aux-length  #AuxVariableListLength,
      list  #VariableLengthList}

#AuxVariableListLength ::= #INT (0..1023)

#AuxVariableListLengthStruct ::= #ALTERNATIVES {
      normally    #INT (0..31),
      sometimes   #INT (32..1023)}
```

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

D.2.9 Equal length lists

D.2.9.1 The ASN.1 assignment is:

```
EqualLengthLists ::= SEQUENCE {
      list1 List1,
      list2 List2}
      (CONSTRAINED BY {
            -- "list1" and "list2" always have the same number of elements. --
      })

List1 ::= SEQUENCE (SIZE (0..1023)) OF BOOLEAN

List2 ::= SEQUENCE (SIZE (0..1023)) OF INTEGER (1..2)
```

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

D.2.9.3 The encoding object assignments are:

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

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

```
list1Encoding {< REFERENCE : length >} #List1 ::= {
      ENCODE STRUCTURE { USE-SET
            STRUCTURED WITH list-with-determinantEncoding {< length >}}
      WITH PER-BASIC-UNALIGNED}

list2Encoding {< REFERENCE : length >} #List2 ::= {
      ENCODE STRUCTURE { USE-SET
            STRUCTURED WITH list-with-determinantEncoding {< length >}}
      WITH PER-BASIC-UNALIGNED}

list-with-determinantEncoding {< REFERENCE : length-determinant >} #REPETITION ::=
{
      REPETITION-ENCODING {
            REPETITION-SPACE
                  SIZE variable-with-determinant
                        MULTIPLE OF repetitions
                  DETERMINED BY field-to-be-set
                  USING length-determinant}}
```

D.2.9.4 The encoding structure assignments are:

```
#EqualLengthListsStruct ::= #CONCATENATION {
      aux-length  #AuxListLength,
      list1       #List1,
      list2       #List2}

#AuxListLength ::= #INT (0..1023)
```

D.2.10 Uneven choice alternative probabilities

D.2.10.1 The ASN.1 assignment is:

```
EnclosingStructureForChoice ::= SEQUENCE {
      choice            UnevenChoiceProbability }

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

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

D.2.10.3 The encoding structure assignments are:

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

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

D.2.10.4 The encoding object assignments are:

```
enclosingStructureForChoiceEncoding #EnclosingStructureForChoice ::= {
      USE   #EnclosingStructureForChoiceStruct
      MAPPING FIELDS
      WITH {
            ENCODE STRUCTURE {
                  aux-selector     auxSelectorEncoding,
                  choice {
                        ENCODE STRUCTURE  {
                              STRUCTURED WITH  {
                                    ALTERNATIVE
                                          DETERMINED BY field-to-be-set
                                          USING aux-selector}}
                        WITH PER-BASIC-UNALIGNED }}
      WITH PER-BASIC-UNALIGNED} }
      -- First mapping: inserts an explicit auxiliary alternative
      -- determinant.
      -- This encoding object specifies that an auxiliary determinant is used
      -- as an alternative determinant.

auxSelectorEncoding #AuxSelector ::= {
      USE   #BITS
      -- ECN Huffman
       -- RANGE (0..7)
       -- (0..1) IS 60%
       -- (2..4) IS 30%
       -- (5..7) IS 10%
       -- End Definition
       -- Mappings produced by "ECN Public Domain Software for Huffman encodings,
       -- version 1"
       -- (see E.8)
      MAPPING TO BITS {
      0 ..  1 TO '10'B   .. '11'B,
      2 ..  4 TO '001'B .. '011'B,
      5  TO '0001'B,
      6 ..  7 TO '00000'B .. '00001'B}
      WITH bitStringEncoding }
      -- Second mapping: Map determinant indexes to bitstrings

bitStringEncoding #BITS ::= {
      REPETITION-ENCODING {
            REPETITION-SPACE }}
```

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

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

D.2.11 A version 1 message

D.2.11.1 ASN.1 assignment:

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

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

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

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

D.2.11.3 The encoding object assignments are:

```
version1MessageEncoding  #Version1Message ::= {
      ENCODE-DECODE
            {ENCODE WITH PER-BASIC-UNALIGNED }
      DECODE AS IF decodingSpecification}

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

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

D.2.12 The encoding object set

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

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

D.2.13 ASN.1 definitions

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

```
Example2-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-
module2(5)}
      DEFINITIONS AUTOMATIC TAGS ::=
      BEGIN

ExampleMessages ::= CHOICE {
      normallySmallValues                  NormallySmallValues,
      sparseEvenlyDistributedValueSet      SparseEvenlyDistributedValueSet
      -- etc.
      }
```

```
NormallySmallValues ::= INTEGER (0..1000)

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

      -- etc.

END
```

D.2.14 EDM definitions

```
Example2-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module2(6)}
      ENCODING-DEFINITIONS ::=
      BEGIN

EXPORTS Example2Encodings;

IMPORTS #NormallySmallValues, #SparseEvenlyDistributedValueSet,
      #SparseUnevenlyDistributedValueSet, #ConditionalPresenceOnValue,
      #ConditionalPresenceOnExternalCondition,
      #EnclosingStructureForList, #EqualLengthLists, #EnclosingStructureForChoice,
      #Version1Message, #List1, #List2, #VariableLength,#UnevenChoiceProbability
      FROM Example2-ASN1-Module
      {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module2(5)};

Example2Encodings #ENCODINGS ::= {
      normallySmallValuesEncoding-1    |
       -- etc.
      version1MessageEncoding}
       -- etc.


END
```

D.2.15 ELM definitions

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

```
Example2-ELM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) elm-module2(4)}
      LINK-DEFINITIONS ::=
      BEGIN

IMPORTS
      Example2Encodings FROM Example2-EDM
            {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module2(6)}
      #ExampleMessages FROM Example2-ASN1-Module
            {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module2(5)};

ENCODE #ExampleMessages WITH Example2Encodings
      COMPLETED BY PER-BASIC-UNALIGNED


END
```

D.3 Explicitly generated structure examples

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

The examples are presented using the following format:

– The "ASN.1 type assignment". This gives the original ASN.1 type definition.

– The requirement. This lists the required changes from the encodings provided by PER basic unaligned.

– Modification of the implicitly generated encoding structure to produce a new encoding structure.

– The encoding class and encoding object assignments.

D.3.1 Sequence with optional components defined by a pointer

D.3.1.1 The ASN.1 assignment is:

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

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

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

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

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

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

#Pointer ::= #INTEGER

#Integer-with-pointer-concat {< #Element >} ::= #CONCATENATION {
      element     #Element OPTIONAL-ENCODING #Integer-optionality }

#Integer-optionality ::= #OPTIONAL

integer-optionality-encoding{< REFERENCE: start-pointer >}
      #Integer-optionality ::= {
      ALIGNED TO ANY octet
      START-POINTER start-pointer
      PRESENCE DETERMINED BY pointer}

integer-with-pointer-concat-encoding
      {< #Element, REFERENCE:pointer, #ENCODINGS:EncodingObjectSet >}
      #Integer-with-pointer-concat{< #Element >} ::= {
      ENCODE STRUCTURE {
            element USE-SET OPTIONAL-ENCODING
                  integer-optionality-encoding{< pointer >}
            STRUCTURED WITH concat}
      WITH EncodingObjectSet}

concat #CONCATENATION ::= {
      ENCODING-SPACE }
```

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

D.3.2.1 The ASN.1 assignment is:

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

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

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

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

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

where:

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

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

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

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

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

where:

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

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

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

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

```
if-component-present-encoding {<REFERENCE:presence-bit>} #Presence-determinant ::=
{
      PRESENCE
            DETERMINED BY field-to-be-set
            USING presence-bit}

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

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

D.3.3.1 The ASN.1 assignments are:

```
Octet3 ::= OCTET STRING (CONTAINING Sequence3)

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

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

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

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

```
#Octets3 ::= #OCTET-STRING

octets3-encoding #Octets3 ::= {
      REPETITION-ENCODING {
            REPLACE STRUCTURE
            WITH #Octets-with-length
                  ENCODED BY octets-with-length-encoding}}

#Octets-with-length{< #Element >} ::= #CONCATENATION {
      length      #INT(0..2047),
      octets      #Element}

octets-with-length-encoding{< #Element >} #Octets-with-length{< #Element >} ::= {
      ENCODE STRUCTURE {
            octets octets-encoding{< length >}}
      WITH PER-BASIC-UNALIGNED}

octets-encoding{< REFERENCE:length >} #OCTETS ::= {
      REPETITION-ENCODING {
            REPETITION-SPACE
                  SIZE variable-with-determinant
                        MULTIPLE OF octet
                  DETERMINED BY field-to-be-set
                  USING length} }
```

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

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

      ENCODE STRUCTURE {
            STRUCTURED WITH sequence3Structure-encoding }
      WITH Sequence3-encodings
            COMPLETED BY PER-BASIC-UNALIGNED }

Sequence3-encodings #ENCODINGS ::= {
      sequence3-optional-encoding |
      tag-4-bits-encoding }
```

```
#Sequence3-optional ::= #OPTIONAL

sequence3-optional-encoding  #Sequence3-optional ::= {
      PRESENCE
            DETERMINED BY container
            USING OUTER}

#TAG-4-bits ::= #TAG

tag-4-bits-encoding  #TAG-4-bits   ::= {
      ENCODING-SPACE
            SIZE 4}
```

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

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

D.3.4 Sequence-of type with a count

D.3.4.1 The ASN.1 assignment is:

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

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

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

```
#SequenceOf ::= #REPETITION

sequenceOf-encoding #SequenceOf ::= {
      REPETITION-ENCODING {
            REPLACE STRUCTURE
            WITH #SequenceOf-with-count
            ENCODED BY sequenceOf-with-count-encoding}}

#SequenceOf-with-count{< #Element >} ::= #CONCATENATION {
      count #INT(0..63),
      elements    #Element }

sequenceOf-with-count-encoding{< #Element >}
      #SequenceOf-with-count{< #Element >} ::= {
      ENCODE STRUCTURE {
            elements {
                  ENCODE STRUCTURE {
                        STRUCTURED WITH elements-encoding{< count >}}
                  WITH PER-BASIC-UNALIGNED}}
      WITH PER-BASIC-UNALIGNED}

elements-encoding{< REFERENCE:count >} #REPETITION ::= {
      REPETITION-ENCODING {
            REPETITION-SPACE
                  SIZE variable-with-determinant
      MULTIPLE OF repetitions
            DETERMINED BY field-to-be-set
                  USING count}}
```

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

D.3.5 Encoding object sets

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

```
Example3Encodings-1    #ENCODINGS ::= {
      sequence1-encoding }

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

D.3.6 ASN.1 definitions

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

```
Example3-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-
module3(9)}
      DEFINITIONS
      AUTOMATIC TAGS ::=
      BEGIN

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

-- etc.

END
```

D.3.7 EDM definitions

```
Example3-EDM {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module3(10)}

ENCODING-DEFINITIONS ::=

BEGIN

EXPORTS Example3Encodings-1, Example3Encodings-2;

RENAMES
            #OPTIONAL AS #Sequence2-optional
                  IN #Sequence2
            #OCTET-STRING AS #Octets3
                  IN ALL
            #OPTIONAL AS #Sequence3-optional
                  IN #Sequence3
            #TAG AS #TAG-4-bits
                  IN #Sequence3
            FROM Example3-ASN1-Module
                  { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module3(9)};


Example3Encodings-1 #ENCODINGS    ::= {
      sequence1-encoding }

Example3Encodings-2 #ENCODINGS    ::= {
      concat |
      -- etc.
      sequenceOf-encoding }

      -- etc.

END
```

D.3.8 ELM definitions

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

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

LINK-DEFINITIONS ::=

BEGIN


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


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


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

END
```

D.4 A more-bit encoding example

D.4.1 Description of the problem

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

D.4.2.2 The implicitly generated encoding structure is:

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

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

```
    more-bit-encoding {< REFERENCE:more-bit >} #SEQUENCE-OF ::= {
        REPETITION-ENCODING {
            REPETITION-SPACE
                SIZE variable-with-determinant
                    DETERMINED BY flag-to-be-set
                USING more-bit
                ENCODER-TRANSFORMS { { BOOL-TO-BOOL AS logical:not } } } }
```

D.4.2.4 This encoding object is also used in D.4.3 and D.4.4, as it provides the fundamental description of the encoding needed for the repetition.

D.4.2.5 With the first (simple but dirty!) approach, we can now define our encoding object for **#ProfileIndication** by using **ENCODE STRUCTURE**, and apply that encoding object in the ELM, completing the example. The encoding object is defined as:

```
    profileIndicationEncoding #ProfileIndication ::= {
        ENCODE STRUCTURE {
            STRUCTURED WITH more-bit-encoding {< more-bit >} }
        WITH PER-BASIC-UNALIGNED }
```

D.4.3 Use of value mappings to provide the more-bit determinant

D.4.3.1 In this approach, we hide the encoding structure in an ECN definition of a user-defined encoding structure, and use value mapping by matching fields to enable an encoding of the user-defined encoding structure to encode a simplified ASN.1 type definition.

D.4.3.2 The ASN.1 type definition is now:

```
    ProfileIndication2 ::= SEQUENCE OF
        protocol-Profile-ID  INTEGER (0..31)
```

D.4.3.3 This has an implicitly-generated encoding structure (to which we apply our encodings in the ELM) of:

```
    #ProfileIndication2 ::=  #SEQUENCE-OF {
        protocol-Profile-ID  #INTEGER (0..31) }
```

D.4.3.4 We define an encoding structure for the encoding we require, similar to the ASN.1 we wrote in the first approach (see D.4.2.1), except that we use **#PAD** for the reserved bits:

```
    #ProfileIndicationStruct ::= #SEQUENCE-OF {
        #SEQUENCE {
        more-bit-field              #BOOLEAN,
        reserved                    #PAD,
        protocol-Profile-ID         #INTEGER (0..31) } }
```

D.4.3.5 We now need an encoding object for the two-bit **#PAD**, before we can complete the encoding:

```
    pad-encoding #PAD ::= {
        ENCODING-SPACE SIZE 2
        PAD-PATTERN bits:'00'B }
```

NOTE – Subclause 23.12.4.2 specifies that decoders should accept any value for **#PAD** bits, which is what we require here, so we do not need a differential encode/decode.

D.4.3.6 We define an encoding object for our structure, much as in the first approach (see D.4.2.5):

```
    profileIndicationStructEncoding #ProfileIndicationStruct ::= {
        ENCODE STRUCTURE {
            STRUCTURED WITH more-bit-encoding {< more-bit-field >} }
        WITH {pad-encoding} COMPLETED BY PER-BASIC-UNALIGNED }
```

D.4.3.7 Finally, we use value mapping from the implicitly generated structure to our explicitly generated structure to define our final encoding:

```
profileIndication2Encoding #ProfileIndication2 ::= {
      USE #ProfileIndicationStruct
      MAPPING FIELDS WITH profileIndicationStructEncoding }
```

D.4.4 Use of the replacement mechanism to provide the more-bit determinant

D.4.4.1 In our final approach, we define a generic sequence-of encoding that can apply to any sequence of. For this we need a parameterised encoding structure:

```
#SequenceOfStruct {< #Component >} ::=
      #SEQUENCE {
            more-bit-field         #BOOLEAN,
            reserved               #PAD,
            sequence-of-component  #Component }
```

D.4.4.2 We define our sequence-of encoding to perform a replacement of the component with this structure, specifying more-bit-encoding and using the defined pad-encoding:

```
sequence-of-encoding #SEQUENCE-OF ::= {
      REPETITION-ENCODING {
            REPLACE COMPONENT WITH #SequenceOfStruct
            REPETITION-SPACE
                  SIZE variable-with-determinant
                        DETERMINED BY flag-to-be-set
                        USING more-bit-field
                        ENCODER-TRANSFORMS { { BOOL-TO-BOOL AS logical:not } } } }
```

D.4.4.3 When this is applied in the ELM, "**COMPLETED BY PER-BASIC-UNALIGNED**" is used as the combined encoding object set to complete the encoding, giving the desired effect.

D.5 Legacy protocol specified with tabular notation

D.5.1 Introduction

D.5.1.1 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 "bits and bytes" pictures and 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.5.1.2 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.5.1.3 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.5.1.4 The fields "b", "c" and "d" are all required to start on an octet boundary.

D.5.1.5 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.5.1.6 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.5.1.7 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.5.1.8 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.5.1.9 The ASN.1 module is:

```
LegacyProtocol-ASN1-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-
module4(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.5.1.10 The EDM module in D.5.7 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.5.2 Encoding definition for the top-level message structure

D.5.2.1 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.5.2.2 The encoding object "**legacyProtocolMessagesEncoding**" is:

```
    legacyProtocolMessagesEncoding #LegacyProtocolMessages ::= {
          ENCODE STRUCTURE {
                message-id {
                      ENCODING {
                            ENCODING-SPACE
                                  SIZE 8}},
                messages {
                      ENCODE STRUCTURE {
                            STRUCTURED WITH {
                                  ALTERNATIVE
                                        DETERMINED BY field-to-be-used
                                        USING message-id}}
                      WITH PER-BASIC-UNALIGNED}}
          WITH PER-BASIC-UNALIGNED}
```

D.5.3 Encoding definition for a message structure

D.5.3.1 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.5.3.2 The encoding object for "Message1" is:

```
    message1Encoding #Message1 ::= {
          ENCODE STRUCTURE {
                b     b-encoding
```

```
                    OPTIONAL-ENCODING {
                         PRESENCE
                              DETERMINED BY field-to-be-used
                              USING b-flag},
               c    octet-aligned-seq-of-with-ext-determinant{< c-len >},
               d    octet-aligned-seq-of-until-end-of-container
                    OPTIONAL-ENCODING USE-SET}
          WITH PER-BASIC-UNALIGNED}
```

D.5.4 Encoding for the sequence type "B"

D.5.4.1 Padding of one bit is inserted between the fields "**b2**" and "**b3**" ("**aux-reserved**"). The encoding of "**B**" is octet-aligned.

D.5.4.2 The encoding for "**B**" is:

```
     b-encoding #B ::= {
          ENCODE STRUCTURE {
               -- Components
               b3 {
                    ENCODING {
                         ALIGNED TO NEXT nibble
                         ENCODING-SPACE
                              SIZE 2
                                   MULTIPLE OF bit }}
               -- Structure
               STRUCTURED WITH {
                    ALIGNED TO NEXT octet
                    ENCODING-SPACE
                         SIZE self-delimiting-values
                              MULTIPLE OF bit }}
          -- The rest
          WITH PER-BASIC-UNALIGNED}
```

D.5.5 Encoding for an octet-aligned sequence-of type with a length determinant

D.5.5.1 One of the sequence-of types used in the legacy protocol has an explicit length determinant.

D.5.5.2 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 ::= {
          REPETITION-ENCODING {
               ALIGNED TO NEXT octet
               REPETITION-SPACE
                    SIZE variable-with-determinant
                         MULTIPLE OF repetitions
                    DETERMINED BY field-to-be-used
                    USING len}}
```

D.5.6 Encoding for an octet-aligned sequence-of type which continues to the end of the PDU

D.5.6.1 The encoding is octet-aligned. The number of elements is determined by the end of the PDU.

D.5.6.2 The encoding object is:

```
     octet-aligned-seq-of-until-end-of-container #REPETITION ::= {
          REPETITION-ENCODING {
               ALIGNED TO NEXT octet
               REPETITION-SPACE
                    SIZE variable-with-determinant
                    DETERMINED BY container
                    USING OUTER}}
```

D.5.7 EDM definitions

The EDM definitions are:

```
LegacyProtocol-EDM-Module {joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-
module4(13)}

ENCODING-DEFINITIONS ::=

BEGIN

EXPORTS LegacyProtocolEncodings;

IMPORTS #B, #LegacyProtocolMessages, #Message1
      FROM  LegacyProtocol-ASN1-Module
      { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module4(11) };

LegacyProtocolEncodings #ENCODINGS ::= {
      legacyProtocolMessagesEncoding     |
      message1Encoding }

-- etc.

END
```

D.5.8 ELM definitions

The ELM for the legacy protocol is:

```
LegacyProtocol-ELM-Module { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) elm-
module4(12) }

LINK-DEFINITIONS ::=

BEGIN

IMPORTS
      LegacyProtocolEncodings FROM LegacyProtocol-EDM-Module
            { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) edm-module4(13) }
      #LegacyProtocolMessages FROM LegacyProtocol-ASN1-Module
            { joint-iso-itu-t(2) asn1(1) ecn(4) examples(5) asn1-module4(11) };

ENCODE #LegacyProtocolMessages WITH LegacyProtocolEncodings
      COMPLETED BY PER-BASIC-UNALIGNED

END
```

Annex E

Support for Huffman encodings

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

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 is freely available from ITU website at http://www.itu.int/rec/T-REC-X.692-200203-S!AnnE, and from ISO website at http://standards.iso.org/ittf/PubliclyAvailableStandards/c034390_ISO_8825-3_2003(E)_Annex_E.zip).

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:

ECN Huffman

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 1000, 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-case 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 '0000000000000000'B ..
    '0000000110101001'B
           }
WITH my-self-delim-bits-encoding }
```

Annex F

Additional information on the Encoding Control Notation (ECN)

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

Additional information and links on the Encoding Control Notation can be found on the following Web site:

- http://www.itu.int/itu-t/asn1/ecn

Annex G

Summary of the ECN notation

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

G.1 Terminal symbols

The following terminal symbols are used in this Recommendation | International Standard

G.1.1 The following items are defined in clause 8:

anystringexceptnonecend	**IF**
encodingobjectreference	**IMPORTS**
encodingobjectsetreference	**IN**
encodingclassreference	**LINK-DEFINITIONS**
"::="	**MAPPING**
".."	**MAX**
"{"	**MIN**
"}"	**MINUS-INFINITY**
"("	**NON-ECN-BEGIN**
")"	**NON-ECN-END**
","	**NULL**
"."	**OPTIONAL-ENCODING**
"|"	**OPTIONS**
ALL	**ORDERED**
AS	**OUTER**
BEGIN	**PER-BASIC-ALIGNED**
BER	**PER-BASIC-UNALIGNED**
BITS	**PER-CANONICAL-ALIGNED**
BY	**PER-CANONICAL-UNALIGNED**
CER	**PLUS-INFINITY**
COMPLETED	**REFERENCE**
DECODE	**REMAINDER**
DER	**RENAMES**
DISTRIBUTION	**SIZE**
ENCODE	**STRUCTURE**
ENCODE-DECODE	**STRUCTURED**
ENCODING-CLASS	**TO**
ENCODING-DEFINITIONS	**TRANSFORMS**
END	**TRUE**
EXCEPT	**UNION**
EXPORTS	**USE**
FALSE	**USE-SET**
FIELDS	**VALUES**
FROM	**WITH**
GENERATES	

G.1.2 The following item is defined in Annex A:

REFERENCE

G.1.3 The following items are defined in Rec. ITU-T X.680 | ISO/IEC 8824-1:

bstring	**typereference**	**FALSE**
cstring	**"-"**	**FROM**
hstring	**";"**	**IMPORTS**
identifier	**":"**	**MINUS-INFINITY**
modulereference	**ALL**	**NULL**
number	**EXCEPT**	**PLUS-INFINITY**
realnumber	**EXPORTS**	**TRUE**

G.1.4 The following items are defined in Rec. ITU-T X.681 | ISO/IEC 8824-2 :

word

valuefieldreference
valuesetfieldreference

G.1.5 The following items are defined in Rec. ITU-T X.683 | ISO/IEC 8824-4:

```
"{<"
">}"
```

G.2 Productions

G.2.1 The following productions are used in this Recommendation | International Standard, with the items defined in G.1 as terminal symbols:

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

ELMModuleBody ::=
```
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 ::= NameChange NameChanges ?

NameChange ::=
```
        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
        "."
        ComponentIdList
```

ComponentIdList ::=
```
        identifier "." +
```

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

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

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

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

EncodingObjectSet ::=

```
        DefinedOrBuiltinEncodingObjectSet |
        EncodingObjectSetSpec
```

EncodingStructure ::=

```
        TaggedStructure          |
        UntaggedStructure
```

TaggedStructure ::=

```
        "["
        TagClass
        TagValue ?
        "]"
        UntaggedStructure
```

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
        "{"
        identifier ?
        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
        |               EncodingOptionsEncodingObject
        |               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

EncodingOptionsEncodingObject ::=

```
        "{"
        OPTIONS
        EncodingOptionsList
        WITH
        AlternativesEncodingObject
        "}"
```

EncodingOptionsList ::= OrderedEncodingObjectList

AlternativesEncodingObject ::= EncodingObject

NonECNEncodingObject::=

```
        NON-ECN-BEGIN
        AssignedIdentifier
        anystringexceptnonecnend
        NON-ECN-END
```

EncodeStructure ::=

```
        "{"
        ENCODE STRUCTURE
        "{"
        ComponentEncodingList
        StructureEncoding ?
        "}"
        CombinedEncodings ?
        "}"
```

StructureEncoding ::=

```
        STRUCTURED WITH
        TagEncoding ?
        EncodingOrUseSet
```

ComponentEncodingList ::=

```
        ComponentEncoding "," *
```

ComponentEncoding ::=

```
                     NonOptionalComponentEncodingSpec
        |            OptionalComponentEncodingSpec
```

NonOptionalComponentEncodingSpec ::=

```
        identifier ?
        TagAndElementEncoding
```

OptionalComponentEncodingSpec ::=

```
        identifier
        TagAndElementEncoding
        OPTIONAL-ENCODING
        OptionalEncoding
```

TagAndElementEncoding ::=

```
        TagEncoding ?
        EncodingOrUseSet
```

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
    |           #TIME
    |           #DATE
    |           #DATE-TIME
    |           #TIME-OF-DAY
    |           #DURATION
    |           #GeneralizedTime
    |           #UTCTime
    |           #ObjectDescriptor
    |           #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
        |               #TIME
        |               #DATE
        |               #DATE-TIME
        |               #TIME-OF-DAY
        |               #DURATION
        | #GeneralizedTime
        | #UTCTime
        | #ObjectDescriptor     Size?
        | #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
        "{"
        OrderedTransformList
        "}"
```

OrderedTransformList ::= 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 in Rec. ITU-T 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? ";" |
       EXPORTS ALL ";"

SymbolsExported ::= SymbolList

Imports ::= IMPORTS SymbolsImported? ";"

SymbolsImported ::= SymbolsFromModuleList

SymbolsFromModuleList ::=
            SymbolsFromModule             |
            SymbolsFromModuleList  SymbolsFromModule

SymbolsFromModule ::=
            SymbolList
       FROM
       GlobalModuleReference SelectionOption

SelectionOption ::=
       empty
       |    WITH "SUCCESSORS"
       |    WITH "DESCENDANTS"

GlobalModuleReference ::=
            modulereference AssignedIdentifier

AssignedIdentifier ::=
                        DefinitiveIdentifier
                  |     empty

SymbolList ::=
            Symbol                  |
            SymbolList "," Symbol

Symbol ::=
                        Reference
                  |     ParameterizedReference
                  |     BuiltinEncodingClassReference

Reference ::=
                        encodingclassreference
                  |     ExternalEncodingClassReference
                  |     encodingobjectreference
                  |     encodingobjectsetreference

Value ::=
            BuiltinValue
```

```
        ~~|    ReferencedValue~~
        ~~|    ObjectClassFieldValue~~

BuiltinValue ::=
        BitStringValue
        |    BooleanValue
        |    CharacterStringValue
        |    ChoiceValue
        ~~|    EmbeddedPDVValue~~
        |    EnumeratedValue
        ~~|    ExternalValue~~
        ~~|    InstanceOfValue~~
        |    IntegerValue
        |    NullValue
        |    ObjectIdentifierValue
        |    OctetStringValue
        |    RealValue
        |    RelativeOIDValue
        ~~|    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
        |    AbsoluteCharReference

Quadruple  ::= "{"  Group  ","  Plane  ","  Row  ","  Cell "}"
Group           ::= number
Plane           ::= number
Row             ::= number
Cell            ::= number

Tuple ::= "{" TableColumn "," TableRow "}"
TableColumn      ::= number
TableRow   ::= number

AbsoluteCharReference ::=
       ModuleIdentifier
       "."
        valuereference
```

```
~~UnrestrictedCharacterStringValue ::= SequenceValue~~

ChoiceValue ::= identifier ":" Value

~~EmbeddedPDVValue ::= SequenceValue~~

EnumeratedValue ::= identifier

~~ExternalValue ::= SequenceValue~~

IntegerValue ::=
           SignedNumber
       ~~|   identifier~~

SignedNumber ::=
       number          |
       "-" number

NullValue ::= NULL

ObjectIdentifierValue ::=
                       "{" ObjIdComponentsList "}"
                 ~~|     "{" DefinedValue ObjIdComponentsList "}"~~

ObjIdComponentsList ::=
       ObjIdComponents                   |
       ObjIdComponents ObjIdComponentsList

ObjIdComponents ::=
       NameForm  |
       NumberForm      |
       NameAndNumberForm

NameForm ::= identifier

NumberForm ::=
           number
       ~~|   DefinedValue~~

NameAndNumberForm ::= identifier "(" NumberForm ")"

OctetStringValue ::=
       bstring   |
       hstring

RealValue ::=
           NumericRealValue
       |   SpecialRealValue

NumericRealValue ::=
           ~~0~~
       ~~|~~   realnumber
       |   "-" realnumber
       |   SequenceValue

SpecialRealValue ::=
           PLUS-INFINITY
       |   MINUS-INFINITY

RelativeOIDValue ::= "{" RelativeOidComponentsList "}"

RelativeOidComponentsList ::=
           RelativeOidComponents
|          RelativeOidComponents RelativeOidComponentsList

RelativeOidComponents ::=
           NumberForm
|          NameAndNumberForm
~~|          DefinedValue~~

SequenceValue ::=
       "{" ComponentValueList "}" |
       "{"  "}"

ComponentValueList ::=
                 NamedValue
            |    ComponentValueList "," NamedValue

NamedValue ::=
       identifier 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
       ~~|   IElems IntersectionMark IntersectionElements~~

IntersectionElements ::= Elements ~~| Elems 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 Rec. ITU-T 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
         |      OrderedValueList
         |      EncodingObject
         |      EncodingObjectSet
         |      OrderedEncodingObjectList
         |      DefinedOrBuiltinEncodingClass
         |      OUTER

OrderedValueList ::= "{" Value "," + "}"

OrderedEncodingObjectList ::= "{" EncodingObject "," + "}"

InstanceOfValue ::= Value

EncodingClassFieldType ::=
           DefinedEncodingClass
           "."
           FieldName

FieldName ::= PrimitiveFieldName "." +

PrimitiveFieldName ::=
                valuefieldreference
         |      valuesetfieldreference
         |      orderedvaluelistfieldreference
```

G.2.4 The following productions are defined Rec. ITU-T X.683 | ISO/IEC 8824-4, as modified by Annex C, with the items defined in G.1 as terminal symbols:

```
ParameterizedAssignment ::=
           ParameterizedEncodingObjectAssignment
|          ParameterizedEncodingClassAssignment
|          ParameterizedEncodingObjectSetAssignment

ParameterizedEncodingObjectAssignment ::=
           encodingobjectreference
           ParameterList
           DefinedOrBuiltinEncodingClass
           "::="
           EncodingObject

ParameterizedEncodingClassAssignment ::=
           encodingclassreference
           ParameterList
           "::="
           EncodingClass

ParameterizedEncodingObjectSetAssignment ::=
           encodingobjectsetreference
           ParameterList
           #ENCODINGS
           "::="
           EncodingObjectSet

ParameterList ::= "{<" Parameter "," + ">}"

Parameter ::=
           ParamGovernor ":" DummyReference
       |   DummyReference
ParamGovernor ::=
           Governor
       ~~|   DummyGovernor~~

Governor ::=
                 EncodingClassFieldType
           |     REFERENCE
           |     DefinedOrBuiltinEncodingClass
           |     #ENCODINGS
           |     Type

~~DummyGovernor ::= DummyReference~~

DummyReference ::=
                       encodingclassreference
                 |     valuereference
                 |     typereference
                 |     identifier
                 |     encodingobjectreference
                 |     encodingobjectsetreference

ParameterizedReference ::=
                 Reference
           |     Reference "{<"  ">}"

ParameterizedEncodingObject ::=
       SimpleDefinedEncodingObject
       ActualParameterList

SimpleDefinedEncodingObject ::=
                 ExternalEncodingObjectReference
           |     encodingobjectreference

ParameterizedEncodingObjectSet ::=
           SimpleDefinedEncodingObjectSet
           ActualParameterList

SimpleDefinedEncodingObjectSet ::=
                 ExternalEncodingObjectSetReference
           |     encodingobjectsetreference

ParameterizedEncodingClass ::=
           SimpleDefinedEncodingClass
           ActualParameterList

SimpleDefinedEncodingClass ::=
                 ExternalEncodingClassReference
           |     encodingclassreference

ActualParameterList ::= "{<" ActualParameter "," + ">}"

ActualParameter ::=
                       Value
                 |     ValueSet
                 |     OrderedValueList
                 |     DefinedOrBuiltinEncodingClass
                 |     EncodingObject
                 |     EncodingObjectSet
                 |     OrderedEncodingObjectList
                 |     identifier
                 |     STRUCTURE
                 |     OU
```

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	**Data networks, open system communications and security**
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2021